Distributed RDF stream processing and reasoning

Xiangnan Ren

Ecole Doctorale Mathématiques et STIC

Thèse présentée pour obtenir le grade de docteur
Discipline: Informatique

UNIVERSITÉ PARIS-EST

Traitement et Raisonnement Distribués des Flux RDF

Xiangnan REN

Atos, Innovation Lab
Université de Paris-Est
Department of Computer Science

Rapporteurs:	Prof. Dr. Jeff Z. Pan
	Prof. Dr. Pascal Molli
Examinateurs:	Assoc. Prof. Dr. Olivier Curé
	Assoc. Prof. Dr. Zakia Kazi-Aoul
Date de Soutenance:	19 Novembth, 2018

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

1. Acknowledgment

Abstract

Real-time processing of data streams emanating from sensors is becoming a common task in industrial scenarios. In an Internet of Things (IoT) context, data are emitted from heterogeneous stream sources, *i.e.*, coming from different domains and data models. This requires that IoT applications efficiently handle data integration mechanisms. The processing of RDF data streams hence became an important research field. This trend enables a wide range of innovative applications where the real-time and reasoning aspects are pervasive. The key implementation goal of such application consists in efficiently handling massive incoming data streams and supporting advanced data analytics services like anomaly detection.

However, a modern RSP engine has to address volume and velocity characteristics encountered in the Big Data era. In an on-going industrial project, we found out that a 24/7 available stream processing engine usually faces massive data volume, dynamically changing data structure and workload characteristics. These facts impact the engine's performance and reliability. To address these issues, we propose Strider, a hybrid adaptive distributed RDF Stream Processing engine that optimizes logical query plan according to the state of data streams. Strider has been designed to guarantee important industrial properties such as scalability, high availability, fault-tolerant, high throughput and acceptable latency. These guarantees are obtained by designing the engine's architecture with state-of-the-art Apache components such as Spark and Kafka.

Moreover, an increasing number of processing jobs executed over RSP engines are requiring reasoning mechanisms. It usually comes at the cost of finding a trade-off between data throughput, latency and the computational cost of expressive inferences. Therefore, we extend Strider to support real-time RDFS+ (*i.e.*, RDFS + `sameAs`) reasoning capability. We combine Strider with a query rewriting approach for SPARQL that benefits from an intelligent encoding of knowledge base. The system is evaluated along different dimensions and over multiple datasets to emphasize its performance.

Finally, we have stepped further to exploratory RDF stream reasoning with a

fragment of Answer Set Programming. This part of our research work is mainly motivated by the fact that more and more streaming applications require more expressive and complex reasoning tasks. The main challenge is to cope with the large volume and high-velocity dimensions in a scalable and inference-enabled manner. Recent efforts in this area still missing the aspect of system scalability for stream reasoning. Thus, we aim to explore the ability of modern distributed computing frameworks to process highly expressive knowledge inference queries over Big Data streams. To do so, we consider queries expressed as a positive fragment of LARS (a temporal logic framework based on Answer Set Programming) and propose solutions to process such queries, based on the two main execution models adopted by major parallel and distributed execution frameworks: Bulk Synchronous Parallel (BSP) and Record-at-A-Time (RAT). We implement our solution named BigSR and conduct a series of evaluations. Our experiments show that BigSR achieves high throughput beyond million-triples per second using a rather small cluster of machines.

Résumé de Thèse

Le traitement en temps réel des flux de données émanant des capteurs est devenu une tâche courante dans de nombreux scénarios industriels. Dans le contexte de l'Internet des objets (IoT), les données sont émises par des sources de flux hétérogènes, c'est-à-dire provenant de domaines et de modèles de données différents. Cela impose aux applications de l'IoT de gérer efficacement l'intégration de données à partir de ressources diverses. Le traitement des flux RDF est dès lors devenu un domaine de recherche important. Cette démarche basé sur des technologies du Web Sémantique supporte actuellement de nombreuses applications innovantes où les notions de temps réel et de raisonnement sont prépondérantes. La recherche présentée dans ce manuscript s'attaque à ce type d'application. En particulier, elle a pour objectif de gérer efficacement les flux de données massifs entrants et à avoir des services avancés d'analyse de données, *e.g.*, la détection d'anomalie.

Cependant, un moteur de RDF Stream Processing (RSP) moderne doit prendre en compte les caractéristiques de volume et de vitesse rencontrées à l'ère du Big Data. Dans un projet industriel d'envergure, nous avons découvert qu'un moteur de traitement de flux disponible 24/7 est généralement confronté à un volume de données massives, avec des changements dynamiques de la structure des données et les caractéristiques de la charge du système. Ces faits ont un impact sur les performances et la fiabilité du moteur. Pour résoudre ces problèmes, nous proposons Strider, un moteur de traitement de flux RDF distribué, hybride et adaptatif qui optimise le plan de requête logique selon l'état des flux de données. Strider a été conçu pour garantir d'importantes propriétés industrielles telles que l'évolutivité, la haute disponibilité, la tolérance aux pannes, le haut débit et une latence acceptable. Ces garanties sont obtenues en concevant l'architecture du moteur avec des composants actuellement incontournables du Big Data: Apache Spark et Apache Kafka.

De plus, un nombre croissant de traitements exécutés sur des moteurs RSP nécessitent des mécanismes de raisonnement. Ils se traduisent généralement par un compromis entre le débit de données, la latence et le coût computationnel des inférences. Par conséquent, nous avons étendu Strider pour prendre en charge la

capacité de raisonnement en temps réel avec un support d'expressivité d'ontologies en RDFS + (*i.e.*, RDFS + `sameAs`). Nous combinons Strider avec une approche de réécriture de requêtes pour SPARQL qui bénéficie d'un encodage intelligent pour les bases de connaissances. Le système est évalué selon différentes dimensions et sur plusieurs jeux de données, pour mettre en évidence ses performances.

Enfin, nous avons exploré le raisonnement du flux RDF dans un contexte d'ontologies exprimés avec un fragment d'ASP (Answer Set Programming). La considération de cette problématique de recherche est principalement motivée par le fait que de plus en plus d'applications de streaming nécessitent des tâches de raisonnement plus expressives et complexes. Le défi principal consiste à gérer les dimensions de débit et de latence avec des méthologies efficaces. Les efforts récents dans ce domaine ne considèrent pas l'aspect de passage à l'échelle du système pour le raisonnement des flux. Ainsi, nous visons à explorer la capacité des systèmes distribuées modernes à traiter des requêtes d'inférence hautement expressive sur des flux de données volumineux. Nous considérons les requêtes exprimées dans un fragment positif de LARS (un cadre logique temporel basé sur Answer Set Programming) et proposons des solutions pour traiter ces requêtes, basées sur les deux principaux modèles d'exécution adoptés par les principaux systèmes distribuées: Bulk Synchronous Parallel (BSP) et Record-at-A-Time (RAT). Nous mettons en œuvre notre solution nommée BigSR et effectuons une série d'évaluations. Nos expériences montrent que BigSR atteint un débit élevé au-delà du million de triplets par seconde en utilisant un petit groupe de machines.

Contents

List of Figures

List of Tables

2. Introduction

2.1. Motivation

Nowadays, we are in the era of rapid data generation and rapid data consumption. Processing data from real-time data streams and sensors devices is becoming ubiquitous. Applications like GPS (Global Positioning System), Social Network, Traffic Monitoring services, Financial Transaction System and Building Management System are continuously producing and consuming massive timely information.

Under this trend, the service of Internet of Things (IoT) is gaining in popularity. Real-time processing of data streams emanating from sensors is becoming a common task in industrial scenarios. Such applications usually require low latency and high throughput. Moreover, integrating these information sources, deriving valuable information and knowledge from data stream also enable a new wide range of real-time applications. To achieve this, the RDF data model could be applied and provides two major advantages: supporting data integration and reasoning over the represented data and knowledge. In 2006, W3C Semantic Sensor Network Incubator Group first introduced the semantic annotation to data stream [1]. The idea is to make data stream available based on the Linked Data principles [2]:

- Use URIs as names for things
- Use HTTP URIs for name lookup
- Use URI to provide useful information
- Include links to other URIs for discovering more things.

The advantage of such a data stream model would be simplifying the data integration from heterogeneous data sources. *I.e.*, , it does not only connect different sources of streaming data, but also provides an easy way to bridge data stream and static data. In a traditional relational database setting, a regular way to handle such a request is done by converting the input data stream into a certain relation and then to execute an ad hoc query. Since data conversion and disk-based storage could be quite costly,

this approach normally assumes that the data does not change frequently. Moreover, it does not meet the real-time aspect.

On the other hand, Data Stream Management Systems (DSMS) might be a better choice. However, existing DSMS mainly focuses on relational data stream processing. For heterogeneous data stream integration, DSMS potentially bring an expensive overhead for data transformation. Moreover, due to the schema-free nature of RDF data, the data structure of input data stream is not predictable in general. *I.e.*, , in real world scenarios, we are frequently facing dynamically changing data and workload characteristics, *e.g.*, a sensor could emit different types of messages based on the user requests. These changes impact the execution performance of continuous queries executed over data streams and the stability of the system.

In addition, the scenarios like [3, 4, 5, 6, 7, 8, 9] require the streaming service to have reasoning capabilities for advanced data analytics. An inference service generally produces, preferably in a sound and complet manner, implicit data from explicit data and knowledge. Such a service is usually costly on a computation point of view and not natively supported by standard data management systems, *e.g.*, RDBMS.

In the last decade, the RDF stream processing (RSP) /reasoning community has contributed to the effort of tackling the above mentioned issues. Most of work either ignore the fact that the amount of data could be massive, or the ability of reasoning over RDF data stream. A common way to cope with performance issue or system scalability is to adopt a distributed approach. However, using distributed techniques for RDF stream processing/reasoning is still an emerging trend. In this thesis, we tackle the above-mentioned issues by developing a scalable, high performance stream processing system for real-time RDF stream processing and reasoning.

As the first step of our work, we start by investigating the related work of RDF stream processing and reasoning. In particular, we mainly concentrate on the state of the art for RSP benchmarks. A deep insight into RSP benchmark design gives us a preliminary view of basic conceptions and system design. After that, we shift to distributed stream processing over RDF data stream. For this part, our work covers distributed stream processing and SPARQL query optimization. Our approach aims at designing an adaptive, scalable, high performance RSP engine in distributed settings, namely, Strider. Then, we extend Strider to support online inference, *i.e.*, RDFS and RDFS extended with `sameAs`. Finally, we proposed BigSR, a technical demonstrator for expressive real-time RDF stream reasoning. The main goal of BigSR is to verify the feasibility of applying modern Big Data techniques for real-

time expressive and complex RDF stream reasoning (*e.g.*, temporal Datalog and Answer Set Programming).

2.2. Use Case

This thesis is partially sponsored by the French national environmental project, Waves FUI # 17 [10]. The initial motivation of Waves is awareness of water as a finite resource. Waves aims to reduce water losses and provides an efficient solution for industrial water resource management. A robust water management system should be able to monitor water production and consummation for various regions, provides data analytic services like anomaly detection in real-time.

In the Waves context, we are processing data streams emanating from sensors distributed over the potable water distribution network of a resource management international company. For France alone, this company distributes water to over 12 million clients through a network of more than 100.000 kilometers equipped with thousands (and growing) sensors. Our system's main objective is to automatically detect anomalies, *e.g.*, water leaks, from analyzed data streams. Obviously, the promptness and accuracy of our anomaly discoveries potentially impacts ecological (loss of cleaned up water) as well as economical (price of clients' consumed water) aspects.

In Waves scenarios, real-time data streams are emitted from various sensors located in France. The sensor observation covers water flow, temperature and chlorine level, etc. Each type of sensors provides its proper data format (*i.e.*, CSV - Comma-Separated Values) of observation. We build an ontology for data conversion from relational CSV data to RDF data. Moreover, this ontology is also served as the input of our knowledge base for reasoning tasks. *E.g.*, , some representative use cases in Waves are *returning the difference of water flow in each sector for every 5 seconds ? Which sector has potential water leak anomaly in last 15 minutes, and which category of this leak belongs to?* Efficiently handling such requirements is recognized as a difficult problem for DSMS, but it opens up many interesting topics for stream processing/reasoning and query optimization.

We regard Waves as the cornerstone of this thesis, and the results of this thesis also play an essential role for Waves. Note that even though Waves is originally motivated by the use case of water resource management, its final goal is to provide a platform for general RDF stream processing and reasoning in IoT context.

2.3. Contributions

The main contributions of this thesis are:

2.3.1. Survey of RSP performance evaluations

We conduct a survey and experiments to give an insight and comparisons of RSP engine. Based on existing RSP benchmark, we have proposed some new metrics RSP engine performance evaluation.

2.3.2. Hybrid Adaptive Distributed RDF Stream Processing engine.

We design a production-ready RSP engine for large scale RDF data streams processing which is based on the state-of-the-art distributed computing frameworks. Strider integrates two forms of adaptation. In the first one, for each execution of a continuous query, the system decides, based on incoming stream volumes, to use either a query compile-time (rule-based) or query run-time (cost-based) optimization approach. The second one concerns the run-time approach and decides when the query plan is optimized (either at the previous query window or at the current one). An evaluation of Strider over real-world and synthetic data sets is provided.

2.3.3. Massive RDF stream reasoning (RDF++ and `sameAs`) in the cloud.

We extend Strider to support reasoning services over RDFS plus the sameAs property with an intelligent knowledge base encoding and query rewriting techniques. It thus minimizes the reasoning cost, and guarantee high throughput and acceptable latency.

2.3.4. BigSR: An empirical study for real-time expressive RDF stream reasoning on modern Big Data platforms.

We build a connection between recent theoretical work on RDF stream reasoning to state of the art Big Data technologies (*e.g.*, Apache Spark). We also try to combine stream reasoning (with complex temporal logics and recursion) with distributed computing. Then, we implement a reusable prototype to support a positive fragment of the LARS framework on two distributed systems, namely Apache Spark and Apache Flink. We also identify the pros and cons of BSP (Bulk Synchrounous Parallel) and RAT (Recort At a Time) for different scenarios, respectively. Finally, we conduct a series of evaluation and experimentation on various datasets, and

through our experiments, we highlight an interesting inference expressiveness and scalability trade-off.

2.4. Publications

The related publications of this thesis are listed as follows:

- (2018) **Xiangnan Ren**, Olivier Curé, Hubert Naacke, Guohui Xiao
  BigSR: real-time expressive RDF stream reasoning on modern Big Data platforms, IEEE Big Data (Chapter 9)

- (2018) **Xiangnan Ren**, Olivier Curé, Hubert Naacke, Guohui Xiao
  RDF Stream Reasoning via Answer Set Programming on Modern Big Data Platform, 2018, Poster & Demo@ISWC (Chapter 9)

- (2017) **Xiangnan Ren**, Olivier Curé, Li Ke, Jérémy Lhez, Badre Belabbess, Tendry Randriamalala, Yufan Zheng, Gabriel Képéklian: Strider: An Adaptive, Inference-enabled Distributed RDF Stream Processing Engine, Demo@VLDB (Chapter 8)

- (2017) **Xiangnan Ren**, Olivier Curé, Li Ke, Jérémy Lhez, Badre Belabbess, Tendry Randriamalala, Yufan Zheng, Gabriel Képéklian: Strider: An Adaptive, Inference-enabled Distributed RDF Stream Processing Engine, BDA (Chapter 8)

- (2017) **Xiangnan Ren**, Olivier Curé, Hubert Naacke, Li Ke
  StriderR: Massive and distributed RDF graph stream reasoning. IEEE BigData (Chapter 8)

- (2017) **Xiangnan Ren**, Olivier Curé Strider: A Hybrid Adaptive Distributed RDF Stream Processing Engine. ISWC (Chapter 7)

- (2017) Jérémy Lhez, **Xiangnan Ren**, Badre Belabbess, Olivier Curé
  A Compressed, Inference-Enabled Encoding Scheme for RDF Stream Processing, ESWC (Chapter 7)

- (2016) **Xiangnan Ren**, Olivier Curé, Houda Khrouf, Zakia Kazi-Aoul, Yousra Chabchoub (Chapter 5)
  Apache Spark and Apache Kafka at the Rescue of Distributed RDF Stream Processing Engines, Posters & Demos@ISWC

- (2016) **Xiangnan Ren**, Houda Khrouf, Zakia Kazi-Aoul, Yousra Chabchoub, Olivier Curé (Chapter 5)
  On Measuring Performances of C-SPARQL and CQELS. SR+SWIT@ISWC

- (2016) **Xiangnan Ren**
  Towards a distributed, scalable and real-time RDF Stream Processing engine, DoctoralConsortium@ISWC (Chapter 5)

2.5. Thesis Outline

This thesis is organized as follows: Chapter 3 gives general background knowledge on semantic web, RDF data management, SPARQL query processing, distributed computing framework, and Datalog/ASP. Chapter 4 covers the state of the art for RDF stream processing and reasoning. Chapter 6 introduces a high level view of Strider architecture. Chapter 7 explores the details of the query execution and optimization in Strider. Chapter 8 presents the reasoning techniques used in Strider. The empirical study of applying distributed stream processing techniques on expressive RDF stream reasoning is given in Chapter 9. Finally, Chapter 10 concludes this thesis and points out future work.

3. Background Knowledge

This chapter provides the background concepts for RDF, SPARQL and stream processing. In 3.1, we provide the fundamentals of RDF and SPARQL query processing. In Section 3.2, we discuss available techniques for the physical storage of RDF data. For semantic web knowledge base and reasoning in Section 3.3, we introduce the basic concepts of knowledge base, ontology and reasoning techniques. After that, Section 3.4 presents the relevant notations about data stream models, the semantics of continuous query and the execution mechanisms of stream processing engine. Finally, Section 3.6 summarizes the main features of existing distributed stream processing engines.

3.1. RDF and SPARQL

Data on the Web is frequently represented using RDF, a schema-free graph data model. Assuming disjoint infinite sets I (RDF IRI references), B (blank nodes) and L (literals), a triple (s, p, o) ∈ (I ∪ B) x I x (I ∪ B ∪ L) is called an RDF triple with s, p and o respectively being the subject, predicate and object. IB = I ∪ B, IBL = I ∪ B ∪ L are the respective unions. We denote the respective unions as IB = I ∪ B AND IBL = I ∪ B ∪ L. It models the statement "s has property p with value o ". From graph theory aspect, the model of RDF triple can be considered as a directed edge e starts from vertex s to vertex o, and e is labeled by p. Hence, a set of triples $\{t_1, ...t_n\}$ form a directed graph G. Figure G_1 shows the RDF graph G_1, where $G_1 = \{(A, cityname, B), (A, isCapitalOf, C), (C, countryname, D), (A, hasPopulation, B)\}$.

From our example, we see that the nature of schema-free gives RDF a flexible way to represent the knowledge. *E.g.*, , it is easy to add more triples that describes A, or associate A to other entities. Such schema-less nature facilitates the data integration from heterogeneous resources.

We now recall the definitions given in [11]. Assume that V is an infinite set of variables and that it is disjoint with I, B and L. SPARQL is the W3C declarative query language recommendation for the RDF format. We can recursively define a

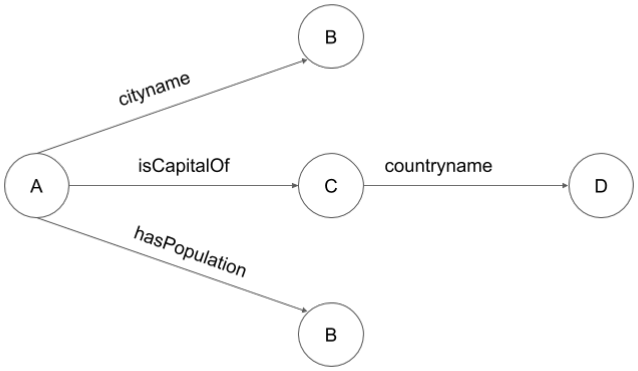


Figure 3.1.: Graph representation of G_1

SPARQL [12] triple pattern (tp) as follows: (i) a triple $tp \in$ (IB $\cup$ V) x (I $\cup$ V) x (IB $\cup$ V $\cup$ L) is a SPARQL triple pattern, (ii) if tp_1 and tp_2 are triple patterns, then ($tp_1.tp_2$) represents a group of triple patterns that must all match, (tp_1 `OPTIONAL` tp_2) where tp_2 is a set of patterns that may extend the solution induced by tp_1, and (tp_1 `UNION` tp_2), denoting pattern alternatives, are triple patterns and (iii) if tp is a triple pattern and C is a built-in condition, then, (tp `FILTER` C) is a triple pattern enabling to restrict the solutions of a triple pattern match according to the expression C. A set of tp is denoted a Basic Graph Pattern (BGP). The SPARQL syntax follows the select-from-where approach of SQL queries. *E.g.*, , the query below returns all capital cities and their belonging countries.

```
SELECT ?s  ?n
WHERE {
?s  isCapitalOf  ?c;
?c  countryname  ?n.  }
```

The semantics of SPARQL query is defined by mapping. A mapping μ is a partial function $\mu : V \rightarrow IBL$. $dom(\mu)$ is called the domain of μ. Two solution mappings μ_1 and μ_2 are compatible, $\mu_1 \sim \mu_2$, if and only if $\forall ?v \in dom(\mu_1) \cap dom(\mu_2)$, $\mu_1(?v) = \mu_2(?v)$. The union of two compatible mappings $\mu_1 \cap \mu_2$ is also a mapping.

The answer to a triple pattern tp for graph G is a bag of mappings $\Omega_{tp} = \{\mu \mid dom(\mu) = vars(tp), \mu(tp) \in G\}$. Most commonly used operators on sets of mappings, *e.g.*, join($\bowtie$) and union$\cap$ can be described as follow:

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1 \wedge \mu_2 \in \Omega_2 \wedge \mu_1 \sim \mu_2\}$$
$$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu_1 \in \Omega_1 \vee \mu_2 \in \Omega_2\}$$

As previously-defined, a basic graph pattern (BGP) $bgp = \{tp_1, ..., tp_k\}$ is a set of triple patterns. We have $\Omega_{bgp} = \Omega_{tp_1} \bowtie ...\Omega_{tp_k}$. Considering the given example in this section, bgp={(?s isCapitalOf ?c), (?c countryname ?n)}. $\Omega_{bgp} = \Omega_{tp_1} \bowtie \Omega_{tp_2}$.

In addition to BGP operator, SPARQL contains other operators like OPTIONAL and FILTER. Define the semantics of graph pattern expressions as function $[[.]]_D$. $[[.]]_D$ takes a graph pattern expression P (*i.e.*, triple pattern or BGP pattern), an RDF dataset D and G an RDF graph in D as parameters and returns a set of mappings. The evaluation of graph pattern P is defined as follow:

$$P \; is \; a \; BGP, \; [[P]]_G^D = [[P]]_G$$
$$P = (P_1 \; AND \; P_2), \; [[P]]_G^D = [[P_1]]_G^D \bowtie [[P_2]]_G^D$$
$$P = (P_1 \; OPTIONAL \; P_2), \; [[P]]_G^D = [[P_1]]_G^D \; \bar{\bowtie} \; [[P_2]]_G^D$$
$$P = (P_1 \; UNION \; P_2), \; [[P]]_G^D = [[P_1]]_G^D \cup [[P_2]]_G^D$$

For a given mapping μ and a built-in condition R, we say μ satisfies R, denoted by $\mu \vDash R$. The filter expression $(P \; FILTER \; R)$, we have $[[(P \; FILTER \; R)]]_G^D = \{\mu \in [[P]]_G^D \mid \mu \vDash R\}$.

Undirected Connected Graph. We now introduce the notion of Undirected Connected Graph (UCG) [13].

A graph $g \in \mathcal{G}$, where g is represented as an ordered pair $g := (\mathcal{V}, \mathcal{E})$. $\mathcal{V}$ is the distinct set of triple patterns and $\mathcal{E}$ is the distinct set of triple pattern pairs. $\mathcal{V}$ and $\mathcal{E}$ correspond to the set of vertices and the set of edges in g, respectively. We call the subject, predicate and object are components of triple pattern. A triple pattern pair with a shared component refers to the join relation between two triple patterns. *E.g.*, , the ordered pair e = ((?s isCapitalOf ?c), (?c countryname ?n)) means the relation obtained by the binary join of (?s isCapitalOf ?c) and (?c countryname ?n) on variable *c*.

Here, we use a more complex query for a better illustration. Let us consider query Q_1 with a BGP operator $bgp_1 = \{tp_1, tp_2, tp_3, tp_4\}$, Figure 3.2 gives a visual

presentation of the relation of triple patterns in Q_1. As tp_1, tp_2 and tp_3 are joined with the common variable $?s$, tp_1 and tp_4 are joined with $?o1$. More precisely, each RDF triple obtained by its corresponding triple pattern is a RDF graph node, and each node is connected with the others by the common variable in the triple patterns.

```
 SELECT ?s  ?o4
 WHERE {
(tp1)   ?s  isCapitalOf  ?o1.
(tp2)   ?s  cityname  ?o2.                    (Q1)
(tp3)   ?s  hasPopulation ?o3.
(tp4)   ?o1 countryname ?o4.  }
```

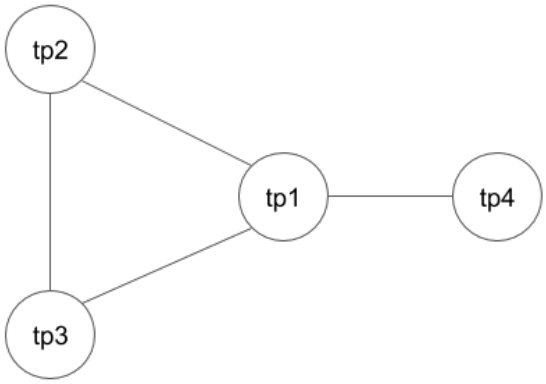


Figure 3.2.: Undirected Connected Graph of Q_1

The representation of BGP operator in UCG fashion helps us to construct the logical plan used in query execution, more details will be given in Chapter 7.

3.2. Storage of RDF Data

Before we shift the discussion of RDF stream model, we briefly cover the available approaches to store static RDF data. Compared to RDF stream processing engine, the development of RDF storage system is more mature. Some optimization technique such as data indexing and query optimization are widely-used in recent RDF stores. A deep insight into these systems gives us some clues to design our own RDF stream processing engines.

In general, RDF data storage can be categorized by either centralized or distributed. Moreover, based on these two categories, the storage can be distinguished by either relational database (*i.e.*, SQL) or non-relational database (*i.e.*, NoSQL, graph database).

Centralized RDF store. The famous centralized RDF systems like Jena [14], Sesame [15], Virtuoso [16], RDF-3X [17], RDFox [18] use RDBMS as the foundation of data storage. Since domain of relational database has been well developed since 1970, storing or querying RDF data over relational database can benefit from the mature technique (*e.g.*, vertical partitioning [], indexing and compression) of RDBMS. *E.g.*, , Hexasotre creates six indexes consist the permutations of subject, predicate and object in RDF triple. In RDF-3X and RDFox generate a set of indexes for all triple permutations, which leads a total of 15 distinct indexes for fast data access. In additionally, RDFox encodes each RDF triple in 12 bytes integer which enables billions triples to be stored in main memory.

Distributed RDF store. To enhance the system scalability, distributed approach is gaining more and more attention for the implementation of RDF data store. Most of distributed RDF stores possess shared nothing architecture. According to the summary in [19], we introduce the distributed RDF stores in three categories:

- **Standalone** distributed RDF stores are primarily dedicated and optimized for RDF data processing. *E.g.*, , RDF-Trinity [20] relies on the main system architecture of Trinity [21]. Trinity generates the query plan through its proxy and delivers to all the Trinity machine. Each Trinity machine holds a partition of data, and performs the query execution of the subplan of the original query. The query evaluation in the Trinity cluster is coordinated by the proxy. Besides, RDF-Trinity models RDF data in key-value, adjacent list format. The query evaluation is done by exploring the RDF graph from one RDF node to another. TriAD [22] uses Message Passing Interface for asynchronous query evaluation in distributed environment. It accelerates the distributed join execution through fully parallel, asynchronous message passing.

- **Federation**. The federation-based RDF stores use centralized RDF stores for subquery evaluation. Based on that, they add a layer of communication and scheduling to coordinate the query evaluation in the cluster. Partout [23] and DREAM [24] are two representative systems in this category. Partout and DREAM use RDF-3X to support the computation of the SPARQL query fragment. In addition, DREAM replicates a copy of the whole input dataset to

avoid cluster shuffling.

- **Built on top of Big Data framework**. The systems in this category, [25], [26], [27], [28], [29], [19] *e.g.*, are built on top of available Big Data computing framework like Hadoop[30] or Spark [31]. [25], [26], [28] use Hadoop as the computing layer. The system keep the data in Hadoop Distributed File System, and compile the query execution plan into Hadoop MapReduce job. PigSPARQL first stores the data in vertical partitioning schema, then the query execution plan is compiled into the built-in operators of Pig to support the query evaluation.

  S2X is based on GraphX [32]. Basically, GraphX shares the same data abstraction (*i.e.*, Resilient Distributed Dataset) as Spark for large scale graph computation. S2X first converts original RDF data into *property graph* of GraphX, the query execution is done by graph exploring algorithm. S2RDF is another Spark-based RDF store. Different from S2X, S2RDF relies on SQL approach for SPARQL query execution. The system performs a pre-processing to create a so-called Extended Vertical Partitioning tables, to pre-compute the semi-join of triple patterns for accelerating the join execution in Spark.

3.3. Semantic Web Knowledge Base (KB) and Reasoning

One important aspect that differentiates RDF from other non-relational data structure, is the ability to reason over the represented information through knowledge bases. The knowledge is derived from a set of ontology, which is usually expressed by RDF Schema (RDFS) [33] or Web Ontology Language (OWL) [34] fragments. An ontology formalizes the description of classification networks and dedicates the structure of knowledge for various domains. We consider that a KB consists of an ontology, aka Terminological Box (Tbox), and a fact base, aka Assertional Box (Abox). The least expressive ontology language of the Semantic Web is RDFS. It allows to describe groups of related re- sources (concepts) and their relationships (proper- ties). RDFS entailment can be computed using 14 rules. But practical inferences can be computed with a subset of them. The one we are using is ρ df which has been defined and theoretically investigated in [35]. In a nutshell, *ρdf* considers inferences using `rdfs:subClassOf`, `rdfs:subPropertyOf` as well as `rdfs:range` and `rdfs:domain` properties.

An RDF property is defined as a relation between subject and object resources. RDFS allows to describe this relation in terms of the classes of resources to which

they apply by specifying the class of the subject (*i.e.*, the domain) and the class of the object (*i.e.*, the range) of the corresponding predicate. The corresponding `rdfs:range` and `rdfs:domain` properties allow to state that respectively the subject and the object of a given `rdf:Property` should be an instance of a given `rdfs:Class`. The property `rdfs:subClassOf` is used to state that a given class (*i.e.*, `rdfs:Class`) is a subclass of another class. Similarly, using the `rdfs:subPropertyOf` property , one can state that any pair of resources (*i.e.*, subject and object) related by a given property is also related by another property. *E.g.*, , Figure 3.3 visualizes a fragment of LUBM [36] ontology,

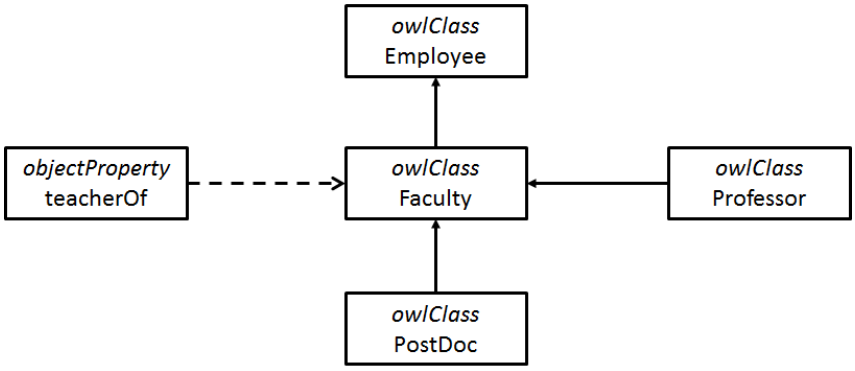


Figure 3.3.: A fragment of visual representation of LUBM ontology

The concept hierarchy is limited to:

$$Postdoc \sqsubseteq Faculty$$
$$Professor \sqsubseteq Faculty$$
$$Faculty \sqsubseteq Employee$$
$$\exists teacherOf \top \sqsubseteq Faculty$$

where $\sqsubseteq$ denotes the subsumption of concepts. *I.e.*, *PostDoc* and *Professor* are the subclasses of *Faculty*. Faculty is a subclass of *Employee*. The domain of object property $teacherOf$ is the concept $Faculty$.

Other ontology languages, OWL and its fragments, of the Semantic Web stack extend RDFS expressiveness, *e.g.*, by supporting properties such as `sameAs` or `owl:TransitiveProperty`. Note that we mainly concentrate RDFS and owl:sameAs in this thesis.

There are two main approaches used to support inferences in KBs. The first

approach consists in materializing all derivable triples before evaluating any queries. It implies a possibly long loading time due to running reasoning services during a data preprocessing phase. This generally drastically increases the size of the buffered data and imposes specific dynamic inference strategies when data is updated. Besides, data materialization also potentially increases the complexity for query evaluation (*e.g.*, ., longer processing to scan the input data structure). These behaviors can seriously impact query performance. The second approach consists in reformulating each submitted query into an ex- tended one including semantic relationships from the ontologies. Thus, query rewriting avoids costly data preprocessing, storage extension and complex update strategies but induces slow query response times since all the reasoning tasks are part of a complex query preprocessing step.

In a streaming context, due to the possibly long life- time of continuous queries, the cost of query rewriting can be amortized. On the other hand, materialization tasks have to be performed on each incoming streams, possibly on rather similar sets of data, which implies a high processing cost, i.e., lower throughput and higher latency. More details will be given in Chapter 8.

3.4. Stream Model and Continuous Query Processing

In this section, we formalize the basic conceptions of RDF Stream Processing (RSP), *i.e.*, the RDF stream model and the query semantics in a continuous context.

3.4.1. RDF Stream Model

A temporal annotation of a single RDF triple can be either time-point-based [37] or time-interval-based [38].

Time-interval-based. Considering an RDF stream S as a sequence of elements <(s,p,o), [start, end]>. (s,p,o) is an RDF triple, [start, end] is a closed time interval which assign a temporal annotation to (s, p, o), *i.e.*, (s, p, o) is valid from instant *start* to instant *end*. *E.g.*, , (car1, hasSpeed, 100, [10, 12]) means that car1 has speed 100 km/h from 10 to 12.

Time-point-based. Instead of using the time interval to assign the temporal annotation, time-point-based approach labels an RDF triple by a time point t, <(s,p,o), [t]>, (s, p, o) is valid at t. Practically, time-point-based can be considered as a special case of time-interval-based, where *start* = *end*. The equivalent time-point-based representation of (car1, hasSpeed, 100, [10, 12]) could be expressed as (car1, hasSpeed, 100, 10), (car1, hasSpeed, 100, 11), (car1, hasSpeed, 100, 12).

Intuitively, time-point-based seems to be more redundant that time-interval-based. Using time interval to assign the temporal annotation could be more expressive than using a single time-point, since a certain time-point is a special case of the time interval. Moreover, system like ETALIS [39], EP-SPARQL [40] use time interval to handle complex event processing. However, time-point-based has an advantage for the applications like data-driven, reactive and low latency system. It allows data stream to be generated instantaneously. Instead of buffering and waiting the expiration of the time interval, the system can process the incoming data stream as soon as possible. In the above-mentioned example, for time-interval-based approach, the computing layer should wait until time point 12 is expired. On the other hand, time-point-based approach allows the system to continuously generate the result at each time point 10, 11, 12.

3.4.2. Continuous SPARQL Query Processing

To continuously process SPARQL query over RDF data stream, the first step is to extend standard SPARQL language to continuous and temporal annotation.

CQL [41] pioneers the stream processing over relational data stream. It adopts window operator to temporally store incoming data stream and continuously launch the query execution over the buffered data stream. To formalize the semantic of continuous query, CQL categorizes the streaming operators into three categories: Stream-to-Relation, Relation-to-Relation and Relation-to-Stream. For the sake of better illustration, we use C-SPARQL as an example, it inherits the main spirit of CQL, and extends SPARQL grammar to handle RDF data stream.

(1) Stream-to-Relation (S2R) operator produces a relation from input data stream. Window operator fall into this category. C-SPARQL inherits the main spirits of CQL, We use C-SPARQL to illustrate the window conception in RDF stream processing.

In C-SPARQL, the system identifies each data stream by an associated, unique IRI. The IRI signifies *where the stream source comes from.* It represents an IP address for accessing streaming data. The syntax of window operator is defined as follow:

$$
\begin{array}{lll}
FromClause & ::= & \text{`}From\text{`}[\text{`}Named\text{`}]\text{`}Stream\text{`}StreamIRI\ \text{`}[RANGE]\text{`} \\
Window & ::= & TimeBasedWindow \mid TripleBasedWindow \\
TimeBasedWindow & ::= & Number\ TimeUnit\ WindowOverlap \\
Timeunit & ::= & (\text{`}ms\text{`}|\text{`}s\text{`}|\text{`}m\text{`}|\text{`}h\text{`}|\text{`}d\text{`}) \\
WindowOverlap & ::= & \text{`}STEP\text{`}\ Number\ TimeUnit \mid \text{`}TUMBLING\text{`} \\
TripleBasedWindow & ::= & \text{`}TRIPLES\text{`}\ Number
\end{array}
$$

The window operator extracts most recent data from input streams. The extraction could be time-based (all triples valid within a given time interval) or triple-based (a given number of triples). Time-based sliding window progressively slides along the timeline, `RANGE` defines the size of window buffer, `STEP` indicates the frequency to update the window. *E.g.*, , `RANGE 10s STEP 5s` means that *buffer the input data stream of the last 10 seconds, and update the window for every 5 seconds.*

(2) Relation-to-Relation operator (R2R) produces a relation from one (*e.g.*, projection, selection) or several input relations (*e.g.*, join, union). R2R operator computes over the instantaneous relation within a given time interval or a time instant. *E.g.*, , considering the following C-SPARQL query:

```
SELECT ?sensor ?value
FROM STREAM <http://example.stream.org/temperature>
[RANGE 10s STEP 5s]
WHERE {
(tp1)  ?sensor          hasValue         ?observation;
(tp2)  ?observation     numericalValue   ?value. }
```

The above query can be interpreted as: *for the last 10 seconds, what is the observed value and which sensor it belongs to?* The projection (SELECT ?sensor ?value) and the join ($tp_1 \bowtie tp_2$) are applied to the sliding window.

(3) Relation-to-Stream operator (R2S) produces an output stream S from a relation R. Define R a temporal relation, s is an RDF triple and t is a time point. Considering a time interval $T = [t_1, t_3]$ which consists of 3 time points t_1, t_2, t_3. A sliding window W possesses range of 2 time points and sliding step of 1 time point. An input relation R contains three RDF triples s_1, s_2, s_3 which are assigned with t_1, t_2, and t_3, respectively. In CQL, three R2S operators are introduced:

- **Istream (for "Insert Stream")** is applied to R, whenever triple s is in $R(t) - R(t-1)$. If at a same time point t, s happens to be inserted and deleted, Istream does not perform the insert operation. Phrasing differently, if an RDF triple S is valid for $R(t)$ and $R(t-1)$, Istream operator does not output s in $R(t)$. Recall the above-mentioned example, at t_3, output stream s only produces $s3$.

- **Dstream (for "Delete Stream")** is applied to R at t, whenever triple s is in $R(t-1)$ but not in $R(t)$. If an RDF triple s is invalid for $R(t)$ but valid for $R(t-1)$, Dstream outputs s in $R(t)$. Recall the previous example, at t_3, S produces s_1.

- **Rstream** maintains the entire current state of its input relation and outputs all of the RDF triples as insertions at each time step. *I.e.*, , at t_2, S outputs s_1, s_2. At t_3, S outputs s_2, s_3 even s_2 has already been produced at t_2.

Notably, C-SPARQL implements Rstream as the R2S operator.

3.4.3. Execution Semantics for Stream Processing

To simply the further explanations in following chapters, we brief the main internal execution models of streaming system in this section. In SECRET [42], authors formalize the execution mechanism of stream processing engines. The evaluation over input data stream can be regarded as a loop:

$$Tick \rightarrow Report \rightarrow Content \rightarrow Scope$$

Tick refers to what drives a stream processing engine to take action on input data stream. *Report* defines the temporal conditions that the elements in the window are visible or not (for query evaluation and result reporting). *Content* captures the elements of input stream that are satisfied the given temporal condition of the window. Finally, *Scope* maps an application time value to a time interval which the input query should be evaluated. In this section, we mainly discus *Tick* model in SECRET.

Tick is a part of streaming engine's internal execution model. It basically indicates the system *how to react* or *how to perform an action* when the window state change. Considering an input data stream S and a continuous query Q, there are three fashion that system *ticks*:

- **Data-Driven.** The system eagerly triggers the evaluation of Q when the arrival of new item in S is detected. CQELS [43], another famous RSP engine possesses this mechanism for the implementation. CQELS immediately launches the query execution when a new triple is arrived.

- **Time-Driven.** Where a the update frequency of time-based window triggers a query execution. *E.g.*, , a parameterized C-SPARQL time-based window `RANGE 10s STEP 5s` triggers the query execution for every 5 seconds over the elements of the last 10 seconds.

- **Batch-Driven.** The new batch of data stream arrival, or the update frequency of time-based window triggers a query execution. In particular, Time-Driven can be regarded as a special case of Bath-Driven.

3.5. Datalog and Answer Set Programming (ASP)

3.5.1. Foundations of Datalog/ASP

A Datalog program is a finite set of rules. A *rule* is an expression of the form

$$R_1(u_1) \leftarrow R_2(u_2), ..., R_n(u_n) \qquad (r)$$

Where, $R_1, ..., R_n$ are relation names and $u_1, ..., u_n$ are *terms* which can be *constants*, *variables* or *functions*. Each expression $R_i(u_i), i \geqslant$ is called an *atom*, and the relation name R_i is the predicate of $R_i(u_i)$.

We call the expression $R_1(u_1)$ is the *head* of r, and $R_2(u_2), ..., R_n(u_n)$ form the *body* of r. For a relation R_i occurs only in the body of rules, R_i is called as *extensional* relation. The comma between each extensional relation in the body is a logical conjunction. Whereas a relation R_i occurs in the head of a rule, R_i is called an intensional relation. The evaluation of rule r is the procedure to compute an instantiation

$$R_1(v(u_1)) \leftarrow R_2(v(u_2)), ..., R_n(v(u_n))$$

of rule r with a valuation v by replacing each variable x by $v(x)$. *I.e.*, a assignment of all variables in the body derives a *fact* for the intensional relation in the head.

Example: Transitive Closure. The following program (P_1) computes all connected vertices by some path in a given undirected graph:

$$T(X,Y) \leftarrow R(X,Y) \qquad (r_1)$$
$$T(X,Z) \leftarrow R(X,Y), T(Y,Z) \qquad (r_2)$$

P_1 consists of two rules r_1, r_2. r_1 is an *exit rule* which is used for the initialization of the recursion. Relation R is an extensional relation which represents the edge of the graph. r_1 derives T fat from each R fact. We call rule r_2 is a recursive rule since relation T appears in both the head and the body of rules. Program P_1 outputs relation $T(X,Y)$ as the result, where $\forall x \in X, \forall y \in Y$, it exists at least one path from x to y.

Answer Set Programming (ASP) can be regarded as an extension of Datalog. In addition to the fragment of Datalog program, an ASP program also supports the logical operators like term of function symbols and disjunction of atoms. A complete introduction to ASP can be found here [44].

3.5.2. LARS Framework for RDF Streams

In this section, we mainly introduce LARS [45], a theoretical framework for temporal Datalog/ASP evaluation.

Assume an atom set $\mathcal{A} = \mathcal{A}^{\mathcal{I}} \cup \mathcal{A}^{\mathcal{E}}$, where $\mathcal{A}^{\mathcal{I}}$ is a set of *intensional* atoms and $\mathcal{A}^{\mathcal{E}}$ is a set of *extensional* atoms disjoint from $\mathcal{A}^{\mathcal{I}}$. In the rest part of this chapter, a *term* starting with a capital letter refers to a variable, otherwise it is a constant.

Definition 1 (Stream)**.** *A stream is a pair $S = (T, v)$, where T is a timeline interval in $\mathbb{N}$, and $v : \mathbb{N} \to 2^{\mathcal{A}^{\mathcal{E}}}$ is an evaluation function such that $v(t) = \varnothing$ for $t \in \mathbb{N} \backslash T$.*

A stream $S' = (T, v')$ is a sub-stream of $S = (T, v)$, if $T' \subseteq T$, and $v'(t') \subseteq v(t')$ for all $t' \in T'$.

The vocabulary of RDF contains three disjoint sets of symbols: IRIs $\mathbf{I}$, blank nodes $\mathbf{B}$ and RDF literals $\mathbf{L}$. An *RDF term* is an element of $\mathbf{C} = \mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$, and and *RDF triple* is an element of $\mathbf{C} \times \mathbf{I} \times \mathbf{C}$. By convention, an RDF triple (s, p, o) can be also written as a fact $s(o)$, if $p =$ `rdf:type`, or $p(s, o)$, otherwise. An *RDF graph* is a finite set of RDF triples. Then, an RDF stream is a stream restricting $\mathcal{A}^{\mathcal{E}}$ to the set of all RDF triples, i.e., at each time points, $v(t)$ evaluates to an RDF graph.

Definition 2 (RDF Stream)**.** *A RDF stream is a stream $S = (T, v)$ such that $v(t) \subseteq \mathbf{C} \times \mathbf{I} \times \mathbf{C}$ for every $t \in T$.*

Definition 3 (Window function)**.** *A window function w takes a stream $S = (T, v)$ as input and returns a sub-stream S', where $S' = (T', v')$. where $T' \subseteq T$, $\forall t' \in T'$, $v'(t') \subseteq v(t')$. S' selects the most recent atoms in the n time points.*

Definition 4 (Time-based Window)**.** *Consider a stream $S = (T, v)$, $T = [t_{min}, t_{max}]$ and a pair $(l, d) \in \mathbb{N} \cup \{\infty\}$. A time-based window $w_\iota(S, t, l, d)$ returns the sub-stream S' of S that contains all the elements of the last l time units, and w slides with step size d.*

LARS distinguishes two types of streams - S and $S^\star$. S represents the currently considered window S, and $S^\star$ is called fixed input stream. To meet the real-time feature, we consider that S as the type of input stream, *i.e.*, we do not assume that the system is capable of loading the stream $S = (T, v)$, $T = [t_{min}, t_{max}]$ from t_{min} to t_{max} directly.

Definition 5 (Window operators)**.** *Let w be a window function. The window operator $\boxplus^w$ signifies that the evaluation should occur on the delivered stream by window function w.*

We consider the set $\mathcal{A}^+$ of extended atoms by the grammar: $a \mid \boxplus^w \Diamond\alpha$, where $a \in \mathcal{A}$ and $t \in \mathbb{N}$ is a time point. The formula $\Diamond\alpha$ means $\Diamond\alpha$ holds in the current window S, if α holds at some time point in S. The window operator $\boxplus^w$ signifies that the evaluation should occur on the delivered stream by window function w.

Definition 6 (Rule and program)**.** *An expression of the form $\alpha \leftarrow \beta_1, \ldots, \beta_n$ is called a LARS rule. where α is an atom and $\beta_1, \ldots \beta_n$ are extended atoms. A (positive plain) LARS program $\mathcal{P}$ is a set of LARS rules.*

The semantics of LARS programs is given by the notion of *answer stream*. For a positive LARS program, its answer stream is unique.

3.6. Distributed Stream Processing Engines (DSPEs)

A stream processing engine can be either self-contained or built on top of an existing framework. For the purpose of high performance, consistency, fault tolerance and easy-to-use, Instead of building a streaming service from scratch, we use available distributed stream processing framework as the computing layer. Such systems are better designed and operated upon when implemented on top of robust, state-of-the-art engines. This section lists the recent Distributed Stream Processing Engines (DSPEs) of general use cases.

Some engineering concepts for DSPEs. Before we introduce the DSPEs, we first illustrate some related engineering concepts:

- **Streaming Models in DSPEs.** At the physical level, a computation model in DSPEs has two principle classes: Bulk Synchronous Parallel (BSP) and Record-at-a-time (RAT) [46]. Recall the definitions given in previous parts of this section, a stream processing engine uses the concept of Tick to drive the system in taking actions over input streams, *i.e.*, Data-Driven, Time-Driven and Batch-Driven. In general, the physical BSP is associated to Time-Driven and/or Batch-Driven models. *E.g.*, Spark Streaming and Google DataFlow with FlumeJava [47] adopt this approach by creating a micro-batch of a certain duration T. That is data are accumulated and processed through the entire Directed Acyclic Graph (DAG) within each batch. The RAT model is usually associated to the logical Data-Driven model (although Time-Driven and Batch-Driven are possible) and prominent examples are Storm [48] and Flink [49]. The RAT/Data-Driven model provides lower latency than BSP/Time-Driven/Batch-Driven model for typical computation. On the other hand, the RAT model requires state maintenance for all operators with record-level-granularity. This behavior obstructs system throughput and brings much higher latencies when recovering after a system failure [46]. For complex tasks involving lots of aggregations and iterations, the RAT model could be less efficient, since it introduces and an overhead for the launch of frequent tasks. A more detailed comparison between BSP on Spark and RAT on Flink will be given in Chapter 9.

- **Message Delivery Guarantee.** A stream processing engine can be abstracted as *producer-consumer* model. Producer generates data stream continuously, and consumer receives input data stream and performs the next computations. (1) If the consumer always receives at least once the message from producer (*i.e.*, duplicated message delivery may exist), we say that the engine enables to guarantee **at-least-once** semantic. (2) If the consumer at most receives once the message from producer (*i.e.*, message may be lost during the transmission), we say that engine has **at-most-once** semantic. (3) If the consumer always receives one and only once the message from producer, we thus say that the engine possesses **exactly-once** semantic.

- **Backpressure.** When the consumer is unable to process the messages delivered from the producer, the data will be accumulated in consumer's buffer or causes

memory leak in the consumer. Worse, the operations of downstream services may also be affected. Backpressure is thus a mechanism which is designed to contend this over-pressure. If input stream is too fast to be consumed, the consumer will send a notification to the producer to slow down the stream generation.

We now use Table 3.1 to summarize the main differences of above-mentioned DSPEs.

	Storm	**Spark**	**Flink**
API	Low-level	High-level	Compositional
Streaming Model	RAT	BSP	RAT
Exactly-once	No	Yes	Yes
Back Pressure	Yes	Yes	Yes
Latency	Low	Medium	Low
Throughput	Low	High	High
Fault Tolerance	Yes	Yes	Yes
State Management	Yes	Yes	Yes

Table 3.1.: Comparisons of different DSPEs

Storm is one of the first production-ready DSPEs. Storm use *Topology*, a DAG to describe the application workflow. The main data structure used in Storm is called *tuple*. Tuple is a serializable key-value pair for message passing between different vertices of the DAG. The two types of elements in a topology are *Spout* and *Bolt*. Spout is the vertex with in-degree equals to 0 in Storm's topology, it represents the data stream source which emit *tuple* to downstream operators. Bolt consumes data from upstream operators and emits the evaluated results to downstream operators.

Once the topology of a streaming application is defined, Storm is able to handle the distributed of tasks to computing nodes in the cluster (Figure 3.4).

Spark & Spark Streaming Spark is a MapReduce-like cluster-computing framework that proposes a parallelized fault-tolerant collection of elements called Resilient Distributed Dataset (RDD) [31]. An RDD is divided into multiple partitions across different cluster nodes such that operations can be performed in parallel. Spark enables parallel computations on unreliable machines and automatically handles locality-aware scheduling, fault-tolerant and load balancing tasks. The computation is described as a DAG of operators and is partitioned into different *stages*.

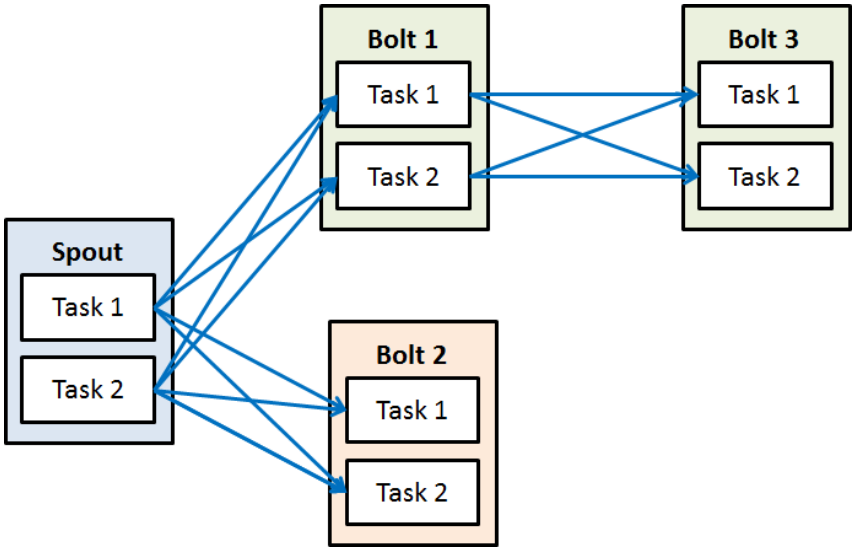


Figure 3.4.: Storm Topology Architecture

Spark Streaming extends RDD to Discretized Stream (DStream) [50] and thus enables to support near real-time data processing by creating *micro-batches* of duration T. DStream represents a sequence of RDDs where each RDD is assigned a timestamp. Similar to Spark, Spark Streaming describes the computing logics as a template of RDD DAG. Each batch generates an instance according to this template for later job execution (Figure 3.5). The micro-batch execution model provides Spark Streaming second/sub-second latency and high throughput. To achieve continuous SPARQL query processing on Spark Streaming, we bind the SPARQL operators to the corresponding Spark SQL relational operators. Moreover, the data processing is based on DataFrame (DF), an API abstraction derived from RDD.

Flink is another distributed computing framework that integrates stream processing and batch processing. Flink handles stream processing in a similar way as Storm. The logic of computation is compiled into Flink's *Streaming Topology*, *i.e.*, a DAG. The vertex of the DAG represents the operator for data stream transformation, data stream flows from one operator to another. Flink cluster consists of a *Job Manager* and several *Task Managers*. The workflow of the application is mapped into Flink's Streaming Topology which corresponds to a Flink *Job*. A job contains group of tasks, each task is managed by a Task Manager that locates on a node of the cluster. A Task Manager locally manages a group of parallel tasks. A runtime example is given in Figure 4.3.

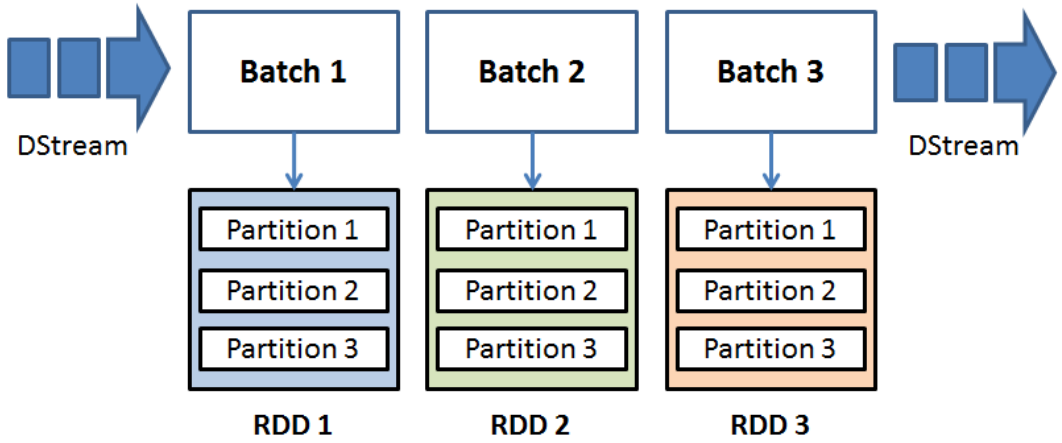


Figure 3.5.: Discretized Stream Processing on Spark Streaming

Comparing to Storm, Flink enables to achieve exactly-once semantic. Besides, Flink uses a variant of Chandy-Lamport algorithm for distributed lightweight snapshot drawing. Therefore, the state maintenance and data checkpoint in Flink are more efficient than Storm.

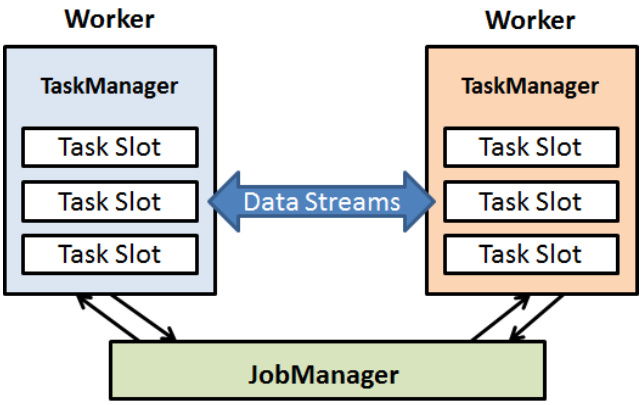


Figure 3.6.: Flink Runtime Environment

4. Related Work

RDF stream processing was introduced [51] in order to bridge the gap between stream and RDF data processing. The two main consensus motivations of materializing data stream as RDF graph nodes are: (i) facilitate data integration from heterogeneous stream sources; (ii) enable stream reasoning for advanced data analytic. The design and benchmarking of a RDF Stream Processing (*cf.* Section 3.4.1) system can be quite challenging:

- The stream processing model does not have a uniform paradigm so far. Normally, a streaming system usually possesses its own streaming model and query language. This makes it difficult to design a streaming system, since no uniform standards can be followed. Besides, benchmark across systems should consider the diversity of execution mechanism between different streaming system.
- Due to fast generation rates and schema free natures of RDF data streams, a continuous SPARQL query usually involves intensive join tasks which may rapidly become a performance bottleneck, thus requiring dedicated optimization technique.
- Compared to continuous SPARQL query processing, RDF stream reasoning involves more complexity to support real-time inference, *e.g.*, query rewriting, data materialization, recursion and fine-grained timestamp manipulation.

In this chapter, we present how recent contributions address above-mentioned challenges, the presentation is organized as follow: Section 4.1 gives an overview of existing RSP benchmarks. Section 4.2 showcases the implementations of the most-known RSP engines. Finally, Section 4.3 gives a survey of state-of-the-art systems that are tailored for enhancing reasoning ability and language expressiveness over RDF data stream.

4.1. RSP Benchmarks

Linear Road [52] is one of the earliest benchmark for stream processing system. It is the first benchmark which formalizes the performance metrics, evaluation methodologies and infrastructure of DSMSs. Linear Road simulates a context of expressway toll system in a city: the city contains expressways, vehicle will be charged with tolls based on the traffic congestion and accident occurrence. The benchmark generates both dynamic data (*e.g.*, vehicle positions, accident information, *etc.*) and static data (*e.g.*, vehicle information, toll system, *etc.*). Instead of interlinking dynamic data and static data, Linear Road separates them into two independent parts, and the benchmark considers query latency and maximum query load as two primary performance metrics for DSMSs.

SRBench [53] is one of the first available RSP benchmarks, comes with 17 queries on LinkedSensorData. The datasets consists of weather observations about hurricanes and blizzards in the United States (from 2001 to 2009). SRBench is mainly design for the purpose of functionality test, *i.e.*, , the support of different operators like aggregations, property path, etc. SRBench does not include any RSP engine performance evaluation.

LSBench [54] covers functionality, correctness and performance evaluation. It uses a customized data generator and provides insights into some performance aspects of RSP engines. However, there is no consideration of important performance metrics such as stream rate, window size and number of streams. Besides, the memory consumption has not been included in their experiments.

CSRBench [55] is another RSP benchmark for correctness evaluation of RSP engine's output. The infrastructure of CSRBench is based on SRBench. The correctness-check in CSRBench is based offline oracle verification. The system sinks the engine output on disk and compare them to the expected query answer. Notably, CSRBench distinguishes the different execution mechanisms for correctness validation (*cf.* Section 3.4.3).

CityBench [56] is a recent RSP benchmark based on smart city data and real application scenarios. It provides a consistent and relevant plan to evaluate performance. Only the number of concurrent queries and the number of streams have been considered to evaluate the execution time and memory consumption, whereas other important factors such as window size and stream rate are missing.

YABench [57] extends CSRBench. YABench inherits the same tested dataset and queries from CSRBench, but it provides more metrics such as memory and CPU

usage, output accuracy per window, etc.

4.2. RSP Systems

We divide the introduction of RSP systems into two parts: centralized and distributed design. This section starts with a summary of the most popular centralized RSP engines, then a quick introduction of distributed RSP engines will also be given.

Table 4.1 gives a comparison of existing popular centralized RSP systems. Due to [58], the implementation of RSP systems can be broadly categorized as *BlackBox* and *WhiteBox*.

	C-SPARQL	**CQELS**	**EP-SPARQL**	**SPARQLstream**
Architecture	BlackBox	WhiteBox	BlackBox	BlackBox
Input	RDF Stream	RDF Stream	RDF Stream	RDF Stream
Window	TiW & TpW	TiW & TpW	SEQ	TiW & TpW
R2S	RStream	IStream	RStream	IStream
Tick	TD/BD	DD	DD	TD/BD
Static Data	Yes	Yes	Limited	Yes
Underlying	Jena/Sesame	Native	Prolog	SNEE
Reasoning	RDFS	No	CEP	RDFS

Table 4.1.: Comparisons of RSP engines.
TiW: Time-Based Window. **TpW:** Triple-Based Window
TD: Time-Driven. **BD:** Batch-Driven. **DD:** Data-Driven.

The BlackBox approach delegates the query processing to existing systems (*e.g.*, SPARQL engine), and achieves continuously SPARQL query processing by adding a modular for data flow management. An intuitive advantage of BlackBox is that it requires less efforts for implementation. However, the defects are: (1) each sub-component has its own required data structure, therefore, data conversions between different sub-components are expensive. *E.g.*, , based on our experience, C-SPARQL could spend 40% of query execution time to convert the output of ESPER to the input of Jena. (2) Using SPARQL endpoint for query processing may suffer from poor performance. Since standard SPARQL engine is tailored for querying static RDF data, some negligible overheads for static data processing becomes critical in streaming context, *e.g.*, engine initialization, data pre-processing.

Different from BlackBox approach, WhiteBox relies less on available systems and has more control of data access for different sub-components. It thus increases the difficulty of implementation. On the contrary, WhiteBox approach makes it possible to internally optimize the data processing in each sub-component. This is very import to improve the system performance and scalability.

In the rest of this section, we provide an extensive discussion on the state-of-the-art RSP systems.

C-SPARQL

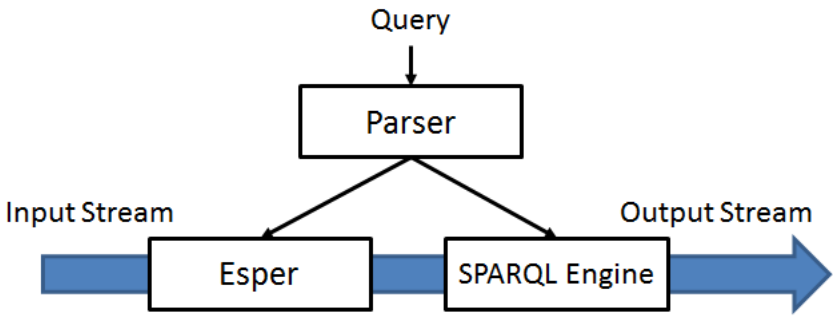


Figure 4.1.: C-SPARQL Architecture

A high-level architecture view of C-SPARQL is given in Figure 4.1. C-SPARQL uses Esper [59] to implement window operators (*i.e.*, S2R), and integrates Jena/Sesame for query processing (*i.e.*, R2R). Recall the syntax defined in 3.4.2, C-SPARQL first divides an input continuous query into two parts, *i.e.*, the dynamic sub-query involves S2R operators, and a static sub-query which refers to a standard SPARQL query. Esper buffers input data stream and delivers it to a SPARQL engine. After that, the SPARQL engine triggers the query execution periodically and output the query results.

CQELS

CQELS is developed in WhiteBox fashion to have a low-level data access. CQELS is the first RSP system which emphasizes the optimization of continuous SPARQL query processing. Figure 4.2 shows the architecture of CQELS. CQELS supports data-driven query execution following the content-change policy, in which query execution is triggered immediately at the arrival of new elements in the window.

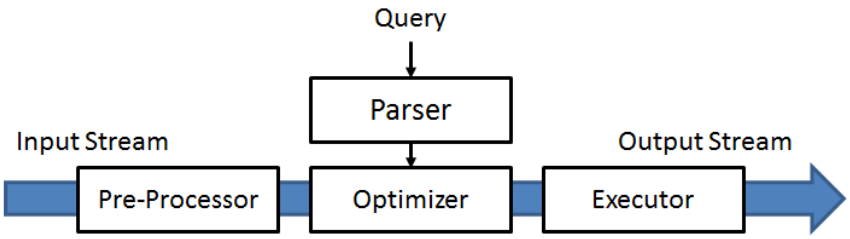


Figure 4.2.: CQELS Architecture

When new data is injected into CQELS, The pre-processor encodes every RDF triple into integer that omits the effect of index. The index is maintained as ring for fast lookup purpose. Next, encoded data stream will be injected into query optimizer. The optimizer pre-computes all the possible query execution plan and dynamically choose the plan with lowest cost (based on some heuristics). The algebra optimization in CQLES is relatively coarse-grained, *i.e.*, it adjusts the join order among different windowing operators. Eventually, CQELS uses multi-way hash join to accelerate the query processing.

Note that even if the `SLIDE` keyword is supported in CQLES syntax, it does not have any effect on the engine behavior. The frequency of query execution depends on the arrival of new data in the stream.

Etalis/EP-SPARQL

Etalis [39] is a rule-based complex event processing engine. EP-SPARQL inserts a compiler layer on the top of Etalis which compiles continuous SPARQL query into logic rules. EP-SPARQL bases on WhiteBox implementation, the query evaluation is done by a Prolog engine. In addition, EP-SPARQL converts standard RDF triple into Prolog atom, *i.e.*, (s, p, o) → triple(s, p ,o), and each atom is annotated by a time interval (*cf.*Section 3.4.1).

EP-SPARQL handles temporal annotation by *Sequence* operator.The execution mechanism of EP-SPARQL bases on Data-Driven, or Event-Driven. More precisely, EP-SPARQL implements Event-Driven Backward Chaining (EDBC) rules [60] for timely and incrementally query evaluation.

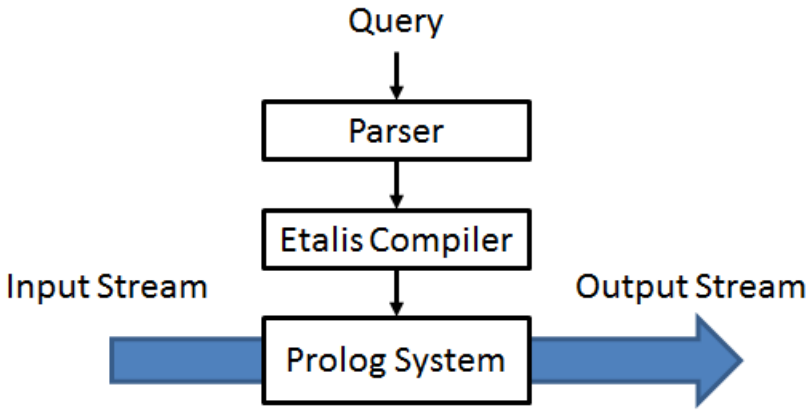


Figure 4.3.: C-SPARQL Architecture

SPARQLstream

SPARQLstream [61] is designed for federated querying from heterogeneous data sources. As shown in Figure 4.4, SPARQLstream rewrites input SPARQLstream query into SNEEql [62] based on a given ontology. The rewriting in SPARQLstream is achieved by using S2O [61] mapping, where S2O is an extension of R2O mapping that supports continuous query processing.

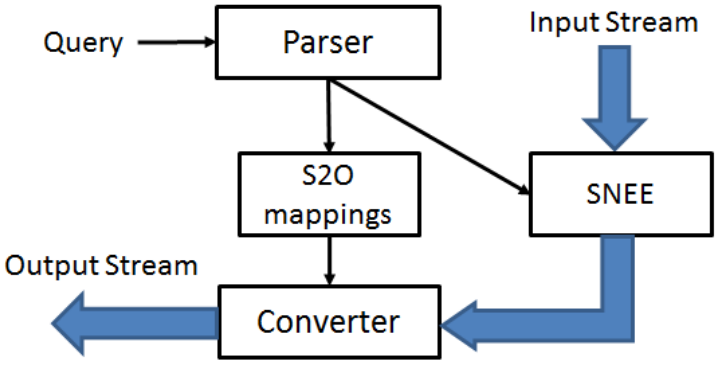


Figure 4.4.: C-SPARQL Architecture

The input query is divided into sub-queries, each sub-query is evaluated by the SNEE engine with the associated stream sources. After that sub-queries are evaluated, SPARQLstream deliver the intermediate results to a converter to generate output RDF stream.

Distributed RSP systems

Distributed RDF stream processing is still an emerging area. Current implementation of distributed RSP engines still stays in the scientific stage for the purpose of technical demonstration. In this section, we introduce two distributed RSP systems: CQLES-Cloud and Katts.

CQELS-Cloud. The centralized CQELS engine uses single thread for query evaluation (*cf.* Section 7.3 for more details). Consequently, CQELS does not scale up by adding more computing resources. To tackle this issue, and to cope with the use case that incoming data stream is massive, a distributed version of CQLES is proposed, namely CQLES-Cloud [63]. CQELS-Cloud is the first RSP system which mainly focuses on the engine elasticity and scalability. The whole system is based on Apache Storm. Firstly, CQELS-Cloud compresses the incoming RDF streams by dictionary encoding in order to reduce the data size and the communication in the computing cluster. Instead of evaluating the continuous SPARQL query in a centralized, blocking way, CQELS-Cloud maps query logical plan into Storm topology for parallel query evaluation. Practically, CQLES-Cloud inherits the main spirit of optimization techniques in CQLES, *e.g.*, dictionary encoding, multi-way hash joins.

Katts is another distributed RSP engine based on Storm. To reduce the network communication in Storm cluster, Katts [64] uses METIS [65] to optimize the partitioning of Storm topology. The implementation of Katts[64] is relatively primitive, it is more or less a platform for algorithm testing but not an RSP engine. The main goal of Katts' design is to verify the efficiency of graph partitioning algorithm for cluster communication reduction.

4.3. Datalog, Answer Set Programming, and RDF Stream Reasoning

To obtain more power of expressiveness for RSP, a natural way is to add the reasoning capability to existing RSP engine. In [66], motivated by IoT use cases of smart city, the authors first proposes the study to combine reasoning techniques with data streams. Although the original idea of RDF stream reasoning is proposed since a decade, the development of reasoning over RDF stream is still in its infancy. Systems like C-SPARQL or StriderR[67] allow reasoning of RDFS++ and owl:sameAs over RDF streams, however, they still have limit expressiveness on temporal logical operators, *e.g.*, the combination or even nesting of window operators. Moreover, the

support of recursion is also missing. In this section, we consider related work in the context of RDF stream reasoning and Datalog engines.

RDF Stream Reasoner. TrOWL[1] [68, 69] is one of the first system which pioneers RDF stream reasoning. The stream reasoner in TrOW is based on a truth maintenance system. TrOWL is thus capable to approximately compute and maintain the justification of inferred results on-the-fly. TrOWL provides justifications for atomic concept subsumption, atomic class assertion and atomic object property assertion.

The goal of the open source ELK reasoner [70] is to support the OWL 2 EL profile. It is part of the ConDOR project[2] which investigates novel "consequence driven" reasoning procedures. Even though ELK is not originally tailored for RDF stream reasoning, we consider that LiteMat could rely on ELK's TBox classification facilities. However, once LiteMat's encoding are performed, we do not need rely on any other reasoners, *e.g.*, [68, 71].

Both StreamRule [72] and its recent parallelized (single machine but multi-threading) version StreamRule^P [73] use an RSP engine for data stream pre-filtering and Clingo [74] as the ASP solver. The expressiveness of BSP implementation in BigSR can fully cover StreamRule and StreamRule^P, since the implementation in these two reasoners stay on positive stratified Datalog program. Moreover, evaluation of StreamRule/StreamRule^P showcases that the average throughput is around thousand-triples/second (1.x-2.x compared to C-SPARQL and CQELS) with second-level delay. A centralized approach limits StreamRule/StreamRule^P to remain at the same performance level as existing centralized RSP engines.

Laser [75] and Ticker [76] are both stream processing systems based on the LARS framework but do not concentrate on scalability. Ticker concentrates on incremental model maintenance and sacrifices performance by relying on an external ASP engine (Clingo). Laser also proposes an incremental model based on time interval annotations which can prevent unnecessary re-computations. Although Laser claims to represent a trade-off between expressiveness and data throughput, it cannot scale the way BigSR enables to. This is mainly due to Laser's inability to distribute stream processing.

[77] is one of the most recent work concentrating on RDF stream reasoning with a distributed approach. The paper aims to check the result consistency with massive, complex RDF data stream. The authors propose two approaches to achieve this goal: (1) to computes the closure of Negative Inclusions (NIs) of DL-Lite ontologies, then

[1] http://trowl.org/
[2] http://www.cs.ox.ac.uk/projects/ConDOR/

register NIs as streaming queries; (2) to compiles the ontology as Strom/Heron's topology to evenly distribute the workload. The idea proposed in [77] could potentially improve and complement our current research.

Other Datalog Solvers. Logiblox [78] is a single-machine commercial transactional and analytical system. Its query language, namely LogiQL [79], is a unified and declarative query language based on Datalog equipped with incremental maintenance. RDFox [18] is a centralized, main-memory RDF store with support for parallel Datalog reasoning and incremental materialization maintenance. None of these systems consider stream processing.

Myria [80] and BigDatalog [81] are both distributed datalog engines that perform on shared-nothing architectures. The former is implemented on its parallel processing framework and interacts with PostgreSQL [82] databases for write and read operations. Much of the effort in the datalog engine of Myria has been concentrated on distributing rule processing in a fault-tolerant manner. BigDatalog implements a parallel semi-Naïve datalog evaluation on top of Spark. Neither Myria nor BigDatalog support stream processing.

5. RSP Performance Evaluation

5.1. Introduction

This chapter starts by introducing the general approaches for RSP benchmarking. The main goals of this chapter are: (i) to get familiar with the architectures and the execution mechanisms of existing RSP systems; (ii) and to identify the proper performance metrics for the design of RSP benchmarks. This chapter can be considered as the cornerstone for the rest of our works, *i.e.*, design, implement and evaluate our native RSP engines for scalable RDF stream processing in a distributed setting.

This chapter consists of 4 sections. In Section 5.2, we first choose C-SPARQL and CQELS, two well-know RSP systems as our evaluation baseline. We discuss about the execution mechanisms on C-SPARQL and CQELS, which illustrate the connections between different execution mechanisms to the corresponding evaluation approaches. Then, we introduce a deeper insight of available RSP benchmark by briefly covering their the pros and cons. In Section 5.3, we present our own infrastructure which includes data stream generator, continuous SPARQL queries and performance metrics monitoring for the experiments. After that, Section 5.4 gives the experiments results with some formal discussions. Finally, we conclude the work of this chapter in Section 5.5.

5.2. C-SPARQL, CQELS and RSP Benchmarks

In this section, we first recall some basic features of C-SPARQL, CQELS, and the involved RSP benchmarks for performance evaluation.

C-SPARQL and CQELS represent, at the time of writing this thesis, certainly the two most popular RSP engines. Each of the mature engines proposes its own continuous query language extensions to query time-annotated triples, and employs a specific RSP mechanism. Since C-SPARQL and CQELS are two centralized RSP systems, to simplify the discussion, we distinguish two kinds of RSP mechanisms in

this section, *i.e.*, time-driven and data-driven for C-SPARQL and CQELS, respectively. The time-driven mechanism periodically executes SPARQL queries within a time-based or triple-based window. Whereas, the data-driven mechanism executes SPARQL queries immediately after the arrival of new data streams.

Other RSP benchmarks, such as SRBench, CSRBench and YABench are not considered performance evaluations. They are thus not in the scope of the discussion for this chapter. As introduced in 4.1, although LSBench and CityBench cover performance evaluation, they still miss some important performance metrics for the evaluation baselines, *e.g.*, stream rate, window size, and proper measurement of memory usage.

Note that we do not propose a new benchmark for RSP engines. This chapter aims to deeply understand the performance of C-SPARQL and CQELS, which also helps us to design our own RSP engines.

5.3. Evaluation Plan

In our experiments, we resolved to use our own data generator for two main reasons: first, to be able to control the size of the generated data streams and, second, to control the data content in order to check the results correctness. In particular, we use both streaming and static data related to the Waves' use case, *i.e.*, , the domain of water resource management. The logical data model is presented in Figure 5.1. The dynamic data describes sensors observations and their metadata, e.g., the message, the observation and the assigned tags. A message basically contains an observation, and we set a fixed number of tags (hasTag predicate) for each observation. For each 50 *flow* observations, we include a *chlorine* observation. The static data provide detailed information about each sensor, namely the label, the manufacturer ID, and the sector ID to which it belongs to in the potable water network.

We define a set of queries $Q = \{Q_1, Q_2, Q_3, Q_4, Q_5, Q_6\}$ of increasing complexity, where $Q_1, ..., Q_5$ operate over streaming data, and Q_6 integrates static data as background knowledge. These queries involve different SPARQL operators (e.g., FILTER, UNION, etc.) and are sorted in ascending order based on the execution complexity (*e.g.*, complex queries involve more query operators). Only the time-based window is addressed in all these queries. As for the last query Q_6, we compare the behavior of RSP engines when varying the size of static data. Details and pseudo code of the predefined queries are available on Github[1]. They can be summarized as

[1] `https://github.com/renxiangnan/Reference_Stream_Reasoning_2016/wiki`

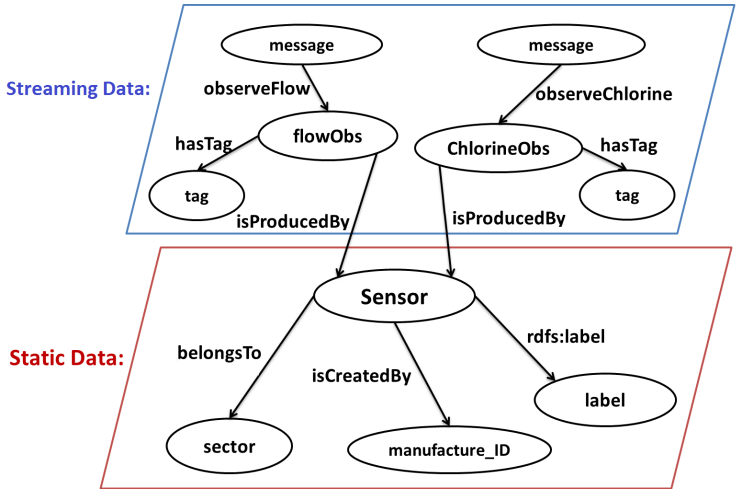


Figure 5.1.: Dynamic and static data in Water Resource Management Context

follows:

- Q_1: Which observation involves chlorine value?
- Q_2: How many tags are assigned to each chlorine observation?
- Q_3: Which observation ID has an identification ending with "00" or "50"?
- Q_4: Which chlorine observation possesses three tags?
- Q_5: Which observation has an identification that ends with "00" or "10" and how many tags assigned to this observation?
- Q_6: What is the sector, manufacturer, and assigned label of each chlorine observation?

5.3.1. Performance metrics

Let us denote the input parameters by X={stream rate, number of triples, window size, number of streams, static data size}, and the set of output metrics by Y={execution time, memory consumption}. We next detail each of these parameters.

- X: (1) stream rate. The time-driven mechanism consists in executing periodically the query with a frequency step specified in the query. This frequency, specified in `STEP` clause, can be time-based (e.g., every 10 seconds) or tuple-based (e.g., every 10 triples). The query is periodically performed over the most recent items. The

keyword `RANGE` defines the size of these temporary items. Just like the frequency step, the window size can be time or tuple-based. In case of time-based window, the execution time and memory consumption are closely dependent on stream rate. Increasing the stream rate makes the engines, such as C-SPARQL, process more data for each execution. The frequency step indicates the interval between two successive executions of the same query. Therefore, input stream rate should not exceed the engine's processing capacity, otherwise the system has to store an always growing amount of data.

Example has already been given in related work. **- X: (2) number of triples.** The stream rate is not an appropriate factor to be considered for the data-driven mechanism because the query execution and the data injection are performed in parallel. In another words, it is not feasible to precisely control the input stream rate. In this context, we need to once feed the system with a fixed number of triples, and that is why we define an additional parameter called *number of triples* N. A bigger N generates a smaller error rate, but N should remain under a given threshold to respect the processing limitations of the RSP engines. *E.g.*, in CQELS, based on our experience, if N is too large, the query evaluation will be blocked or the system even get crashed. Such a behavior signifies that the data flow management is not well designed in CQELS, in fact it seems that an important feature likes back pressure is still missing.

- X: (3) window size. We use *window size* as a performance metric for RSP engines. Note that the window size (`RANGE`) is closely related to the volume of the queried triples for each execution of the query. According to our preliminary experiments, the window size has marginal impact on the performance of CQELS. Thus, we do not consider this metric when evaluating CQELS.

- X: (4) number of streams, (5) static data size. The capacity to handle complex queries with multi-stream sources or static background information is an important criterion to evaluate RSP engines. LSBench and CityBench have already proposed these metrics.

- Y: execution time and memory consumption. As the machine conditions have uncontrollable varying factors, we evaluate the execution time, for a given query, as the average value of n iterations. Since C-SPARQL and CQELS have two different execution mechanisms (time-driven and data-driven), we adapt the definition of execution time to each context. As a consequence, the execution time represents for C-SPARQL the average execution time over several query executions, while it represents for CQELS the global query execution time for processing N triples.

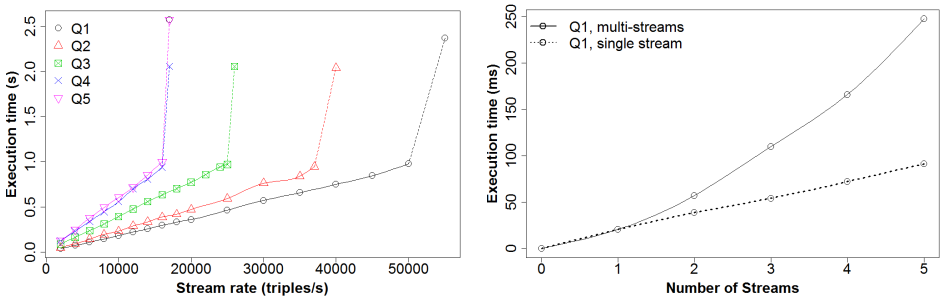
Time-driven and Data-Driven does not determine the output. It actually depends on S2R operator, example is already given in related work.

5.4. Experiments

All experiments are performed on a laptop equipped with Intel Core i5 quad-core processor (2.70 GHz), 8GB RAM, the maximum heap size is set to 2 GB, running Windows 7, Java version JDK/JRE 1.8. The formal evaluation is done after a 1-to-2-minutes warm-up period with relatively low stream rate.

5.4.1. Time-driven: C-SPARQL

We conducted our experiments over C-SPARQL by testing the previously defined queries. We measure the average value of twenty iterations for query execution time and memory consumption.



(a) (b)

Figure 5.2.: Impact of stream rate and number of streams on the execution time of C-SPARQL.

Execution Time We evaluate query execution time by varying stream rate, number of streams, window size (time-based) and static data size.

In Figure 5.2 (a), one can see that the five curves exhibit approximately a linear trend (up to a given threshold concerning the stream rate). For each query, the linear trend can be maintained only when the stream rate is under a given threshold. For all five queries, C-SPARQL normally operates when its execution time is smaller than one second, which is also the query preset `STEP` value. Let us denote by $Rate_{max}(triples/s)$ the maximum stream rate that can be accepted by C-SPARQL for a given query. $Rate_{max}$ represents the maximum number of triples that can be processed per unit time. Table 5.1 shows the $Rate_{max}$ for each query.

Query	Q_1	Q_2	Q_3	Q_4	Q_5
$Rate_{max}$ (triples/s)	≈ 55000	≈ 40000	≈ 25000	$\approx 16000^{+}$	≈ 16000

Table 5.1.: $Rate_{max}$ for the considered queries in C-SPARQL.

As shown in Figure 5.2 (a), if the stream rate exceeds the corresponding $Rate_{max}$, the results provided by C-SPARQL are erroneous. The reason behind is that C-SPARQL does not have enough time to process both current and incoming data. Indeed, newly incoming data streams are jammed in memory, and the system will enforce C-SPARQL to start the next execution which causes errors. Thus, $Rate_{max}$ represents the maximum number of triples under which C-SPARQL delivers correct results.

In some cases, queries require data from multiple streams. In Figure 5.2 (b), we focus on C-SPARQL's behavior by varying the **number of streams** where the stream rate is set to 1,000 triples/s (i.e. the dotted line in Figure5.2 (a)). This figure reports the execution time of Q_1 for different number of streams. The dotted line represents the execution time of Q_1 on a single equivalent (i.e. same workload) stream with a rate $Stream\ Rate_{single} = Number\ of\ Streams \times Stream\ Rate_{multi}$, where $Stream\ Rate_{single}$ and $Stream\ Rate_{multi}$ denote the stream rate for respectively single and multi streams. The curve of the query execution time increases as a convex function over the number of streams. C-SPARQL has a substantial delay by the increasing number of streams. Indeed, it has to repeat the query execution for each stream [83], then executes the join operation among the intermediate results from different stream sources. This action requires important computing resources, so we can deduce that C-SPARQL is more efficient to process single stream than multi-streams. In addition, according to our experiments, we find that the query execution time linearly increases with the growth of the size of **time-based window** and **static data**. C-SPARQL has a constant overhead for delay when increasing these two metrics.

In Figure 5.2 (b), we present the impact of time-based **Window Size** on execution time. Here, we fix stream rate at 1,000 triples/s while increasing the window size from 2s to 10s using the `STEP` clause. Figure 5.2 (b) shows that there is a linear relation between window size and execution time for the five queries.

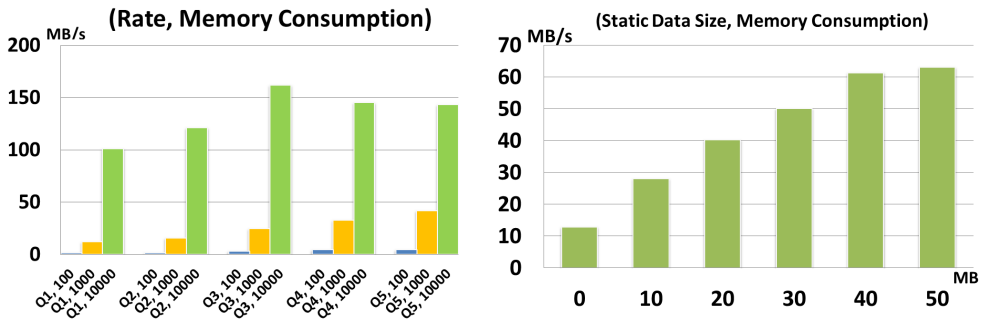
Query Q_6 is specially designed for testing **Static Data**. We recorded the query execution time by varying the size of static data from 10MB to 50MB. Stream rate is fixed at 1,000 triples/s. The curve displayed in Figure 5.4 (b) illustrates the linear increase trend of execution time over static data size. This emphasizes

Figure 5.3.: The real-time monitoring of memory consumption of Q_1

that C-SPARQL holds a stable performance while varying the size of background data. The execution time of Q_6 is close to one second when 50 MB static data were added. 50 MB is approximately the largest size of static data which can be handled by C-SPARQL for Q_6. Experimentation with over 50 MB static data injection emphasizes that C-SPARQL then spends more than one second (i.e. `STEP` value) to finish its current execution. This will make C-SPARQL unreliable, as the correctness of output will seriously drop.

Memory Consumption We used VisualVM to monitor the Memory Consumption of C-SPARQL.

Since the Java Virtual Machine executes the GC lazily (in order to leave the maximum available CPU resources to the application), using the maximum memory allocated during execution is not an appropriate way to measure the memory consumption. Practically, the processing of a simple query, while allocating far



(a) (b)

Figure 5.4.: The impact of (a) stream rate and (b) static data size on memory consumption in C-SPARQL.

less memory on each execution, can also reach the maximum allocated heap as the processing of a complex query. Thus, instead, we define a new evaluation metric called Memory Consumption Rate (MCR). Measuring the amount (in megabytes) of allocated and released memory by GC per unit of time comprehensively describes MCR. $MCR(\text{MB/s}) = \frac{\overline{Max}-\overline{Min}}{\overline{Period}}$, $\overline{Max}$ and $\overline{Min}$ refer to the average maximum and minimum memory consumption, respectively. $\overline{Period}$ is the average duration of two consecutive maximum memory observed instances. $\overline{Max}$, $\overline{Min}$, $\overline{Period}$ are computed over 10 observed periods. MCR signifies the memory changes in heap per second. A higher MCR shows a more frequent activity of GC (Figure 5.3). It intuitively shows how many bytes have been released and reallocated by GC per unit time. Figure 5.4 (a) shows the impact of stream rate on MCR. For each query, the period decreases and MCR increases with the growth of Stream Rate. Query Q_3 has the highest MCR. This can be explained by the aggregate operator which produces more intermediate results during query execution. Note that MCR is not a general criterion for measuring memory consumption. In some use cases, we could not observe periodical activity on the GC. The main goal of using MCR is to give a comprehensive description of memory management on C-SPARQL.

Figure 5.4 (b) displays the increase of memory consumption rate over the growth of **static data size**. For query Q_6, memory peak varies marginally while increasing static data size, but the minimum consumed memory is directly impacted. One possible explanation is that C-SPARQL produces additional objects to process static data, and keeps these objects as long-term in memory.

5.4.2. Data-driven: CQELS

This section focuses on the performance evaluation of CQELS. The variant parameters are number of triples, number of streams, and static data size. Q_4 and Q_5 are not included in this evaluation since CQELS does not support the timestamp function (*i.e.*, function that performs basic temporal filtering on the streamed triples).

Execution Time Since CQELS uses a so-called probing sequence (for multi-way hash joins) to support its query evaluation, getting the running time for each query execution is not experimentally feasible. Thus, we evaluate the global execution time of N triples for CQELS. More precisely, we keep the same strategy as LSBench, i.e. inject a finite sequence of stream into the system which contains N triples. N should be big enough to get more accurate results ($N \geqslant 10^5$ [54]).

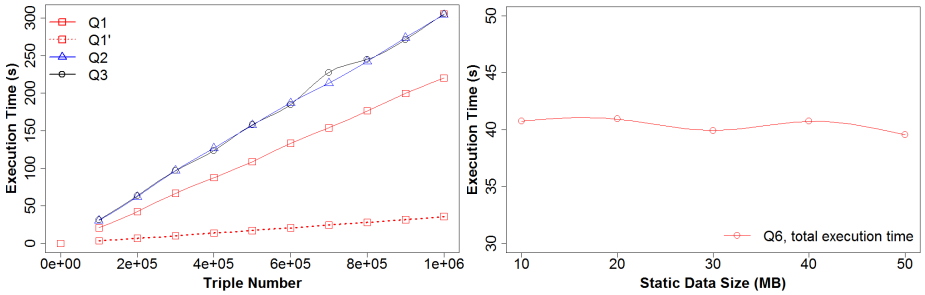


Figure 5.5.: The impact of number of triples and static data size on query execution time in CQELS.

Figure 5.5 (a) shows the impact of **number of triples** on execution time. N should also be controlled within a certain range to prevent the engine from crashing (c.f. "Memory Consumption" part of CQELS). Queries Q_1, Q_2, Q_3 contain chain patterns (join occurs on subject-object position) that select chlorine observation: $\{T_1$*: ?observation ex:observeChlorine ?chlorineObs .* T_2*: ?chlorineObs ex:hasTag ?tag .* $\}$. Pattern T_1 returns all results by matching the predicate "observeChlorine", then T_2 filters among all selected observations in T_1 those which have been assigned tags. In Figure 5.5 (a), note that there is no significant difference between Q_2 and Q_3. Based on Q_2, the query Q_3 adds a "FILTER" operator to restrict that preselected observations which have an ID ending by "00" or "50". This additional filter in Q_3 slightly influences the engine performance, which lets suggest that CQELS is very efficient at processing "FILTER" operator. As the dotted line Q'_1, it represents Q_1 without the pattern T_2. Its corresponding execution time is reduced to one-six times

compared with Q_1. Indeed, the pattern T_2 plays a key role in term of execution time. Without T_2, CQELS will return the results immediately if T_1 is verified, but pattern T_2 makes the engine wait till T_2 is verified.

CQELS supports queries with **multi-streams**. It allows to assign the triple patterns which are only relative to the corresponding stream source. CQELS requires that the associated stream source (*i.e.*, URI) for each triple pattern must be explicitly indicated.

This property gives the engine some advantages to process complex queries. Each triple just needs to be verified in its attached stream source. However, C-SPARQL has to repeat verification on all presenting streams for the whole query syntax, and this behavior leads to a waste of computing resources. Due to data-driven mechanism, serious mismatches occur in output for a multi-streams query, especially when the query requests synchronization among the triples. Asynchronous streams are illustrated in our GitHub[2].

Suppose that we have two streams, S_1 and S_2, sent sequentially (due to the data-driven approach adopted by CQELS) into the engine. If the window size defined on S_1 is not large enough, ?*observation* in pattern T_2 will not be matched with ?*observation* in T_1. This problem can be solved by defining a larger window size in T_1 with a small number of streams. In our experiments, we carry out the multi-streams test by constructing two streams on Q_1, Q_2 and Q_3. For Q_1, with two streams, CQELS spent approximately 26s to process ($2\times$) 10^5 triples, that is just 30% more than the single stream case. To conclude, CQELS gains some advantages in term of execution time to process queries with multi-streams. However, the output may also be influenced by the asynchronous behavior in multi-stream context. Note that C-SPARQL does not suffer from the streams synchronization since it follows batch-oriented approach.

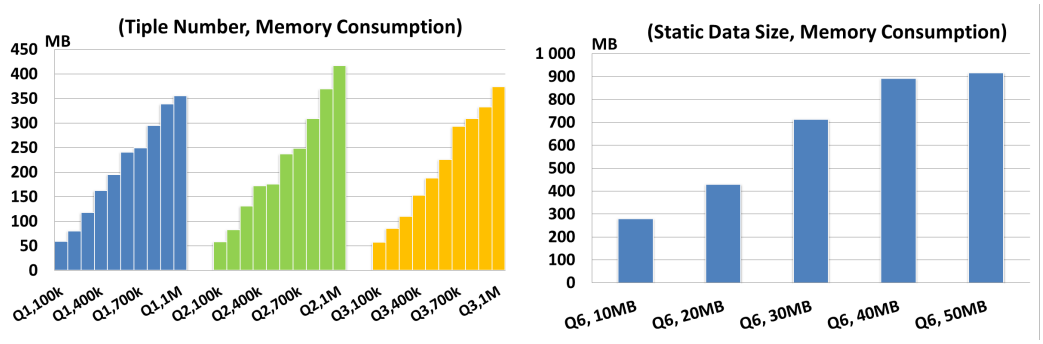
In Figure 5.5 (b), the curve gives the total execution time(s) for 1.260.000 triples. The execution time for N triples slightly changes while increasing the size of **Static Data** from 10MB to 50MB. The result shows that CQELS is efficient for processing static data of a large size.

Memory Consumption As we directly send N triples into the system at once, CQELS's memory consumption does not behave as C-SPARQL (which follows a periodic pattern). Generally, the memory consumption on CQELS keeps growing by increasing the number N of triples. As mentioned in the previous section, N should not exceed a given threshold. If N is very large, the memory consumption will reach

[2] `https://github.com/renxiangnan/Reference_Stream_Reasoning_2016/wiki`

its limit. In this situation, latency on query execution will increase substantially. Furthermore, since serious mismatch occurs on multi-streams query, X = Number of Stream is not considered as a metric for memory consumption. We evaluate the peak of memory consumption (MC) during query execution. The trend increases over time, where MC reaches the peak just before the end of query execution.

Figure 5.6 (a) shows that the memory consumption of Q_1, Q_2 and Q_3 is very close when varying the **number of triples**, i.e., the complexity of queries are not reflected by their memory consumption. CQELS manages efficiently the memory for complex queries. In Figure 5.6 (b), the memory consumption of Q_6 is proportional to the size of **static data**. According to the evaluation, we found that a lower maximum allocated heap size (e.g., 512MB) causes a substantial delay on CQELS. The consumed memory keeps growing to the limited heap size, i.e. the GC could clear the unused objects in a timely manner. This behavior is possibly caused by the built-dictionary for URI encoding [43].



(a) (b)

Figure 5.6.: Impact of the number of triples and the static data size on memory consumption in CQELS.

5.5. Result Discussion & Conclusion

As we generate different streaming modes for time-driven (C-SPARQL) and data-driven (CQELS) engines, the memory consumption is not comparable between them. This section mainly derives a discussion on query execution time based on observed results. It is about a simple comparison between C-SPARQL and CQELS.

It is not obvious to compare the performance of different RSP engines, since each of them has a specific execution strategy. According to [54] and our experiments, we list the following conditions to support a fair cross-engines performance comparison: (i) the engine results should be correct, at least *comparable* [54]. We remind that the untypical behavior of C-SPARQL occurs when the incoming stream rate exceeds the threshold. Even if the engine still produces results, it is meaningless to measure the execution time; (ii) The execution time for different RSP engines should associate the same workload. As C-SPARQL uses a batch mechanism, it is easy to control the workload of the window operator. However, the data-driven eager mechanism practically makes infeasible the workload control. Therefore, we choose $t = \frac{T}{N}$, the average execution time per triple to support our comparison. T is the total execution time for N triples. Note that t marginally changes when varying the metrics defined in section 5.3.1; (iii) The engine warming up is also recommended. We inject the "warming up" stream (with a relatively low stream rate) into the system before the formal evaluation.

	Average execution time per triple (millisecond)			
RSP engine	Q_1	Q_2	Q_3	Q_6 (50MB static data)
CSPARQL	0.018	0.025	0.040	0.952
CQELS	0.169	0.239	0.243	0.032

Table 5.2.: Execution time (in seconds) of Q_1, Q_2, Q_3 and Q_6.

Table 5.2 shows that C-SPARQL outperforms CQELS when dealing with queries Q_1, Q_2 and Q_3. This can be explained by the chain query pattern existing in Q_1, Q_2 and Q_3, which forces CQELS to repeat the verification on matching condition for the whole window. This behavior significantly hinder the engine performance. For Q_6, CQELS is almost 27 times faster than C-SPARQL. It shows its high efficiency to process queries with static data.

Finally, we summarize our experiment over three aspects:

- 1) **Functionality support**. Since C-SPARQL uses the Sesame/Jena APIs during query processing, it supports most of the SPARQL 1.1 grammar. In contrast, as CQELS is implemented in a native way, it supports less operations than C-SPARQL, e.g., timestamp function, property path, etc.
- 2) **Output correctness**. As mentioned in section 5.4.2, CQELS suffers from a serious output mismatch in the multi-stream context. This is due to the

eager execution mechanism and asynchronous streams. C-SPARQL behaves normally with multi-stream queries since it is characterized by a time-driven mechanism. As a matter of fact, real use cases often require concurrency of join from different stream sources. In this context, C-SPARQL takes the advantages of correctness and completeness of output results.

- 3) **Performance**. C-SPARQL shows stability with complex queries. However, in practical applications, input stream rate should be controlled at a low level to guarantee C-SPARQL's output correctness. Besides, C-SPARQL has scalability problem when dealing with static data. CQELS takes advantage from its dictionary encoding technique and dynamic routing policy, and thus, is efficient for simple queries and is scalable with static data.

Yahoo Benchmark for Distributed Streaming Systems Right after the end of this work, Yahoo published their benchmark for the performance evaluation of distributed stream processing systems [3]. The originally published Yahoo benchmark covered the evaluation of Spark Streaming, Storm and Flink. The benchmark mainly considers system throughput and latency as the two primary performance impact factors. In our case, *i.e.*, distributed RDF Stream Processing, throughput refers that how many RDF triples can be processed by the system per unit time (*e.g.*, triples/second). Latency means how long does the RSP engine consumes between the arrival of an input and the generation of its output (*i.e.*, execution time as previously-mentioned).

This chapter focuses on the performance evaluation of two state-of the-art RSP engines with our native RSP performance benchmark proposals. We propose some new performance metrics and designed a specific evaluation plan. In particular, we take into account the specific implementation of each RSP engine. We performed many experiments to evaluate the impact of *Stream Rate*, *Number of Triples*, *Window Size*, *Number of Streams* and *Static Data Size* on *Execution Time* and *Memory Consumption*. Several queries with different complexities have been considered. The main result of this complete study is that each RSP engine has its own advantage and is adapted to a particular context and use case, e.g., C-SPARQL excels on complex and multi-stream queries while CQELS stands out on queries requiring static data.

Based on the experience of this work, we have accumulated a lot of experience for RSP engine performance evaluation. Since the benchmark for distributed RSP engine is still missing, we refer to Yahoo's benchmark. Yahoo's DSPE benchmark

[3] `https://github.com/yahoo/streaming-benchmarks`

omits the impact fact like window size, number of streams, stream rate, etc. And the memory consumption is also not in the scope of the consideration, since this metric is practically difficult to measure in a precise way. In the rest part of this thesis, all the experiments based on Yahoo's benchmark, *i.e.*, we regard system throughput and latency as the main performance metrics. Be aware that the reason why we abandoned existing RSP performance benchmarking systems [54, 56] is that, none of them is tailored for massive data stream. This limitation is contrary to our original intention of using distributed stream processing framework to cope with massive RDF stream.

6. Strider Architecture

6.1. Motivation

Querying over RDF data streams can be quite challenging. Due to fast generation rates and the schema free nature of RDF data streams, a continuous SPARQL query usually involves intensive join tasks which may rapidly become a performance bottleneck. Existing centralized RSP systems like C-SPARQL, CQELS and ETALIS are not capable of handling massive incoming data streams, as they do not benefit from task parallelism and the scalability of a computing cluster. Besides, most streaming systems are operating 24/7 with patterns (*e.g.*, number of temperature or flow observations), *i.e.*, stream graph structures, that may change overtime (in terms of graph shapes and sizes). This can potentially have a performance impact on query processing since in most available distributed RDF streaming systems, *e.g.*, CQELSCloud and Katts, the logical query plan is determined at compile time. Such a behavior can hardly promise long-term efficiency and reliability, since there is no single query plan that is always optimal for a given query.

In this chapter, we provide a high-level view of Strider's architecture, the core stream management component of the Waves project. Strider is not just a tailored RDF stream processing engine for IoT usage purpose, but also aims to handle general use cases for RDF stream processing.

The key implementation goal of Strider consists in efficiently handling massive incoming data streams and supporting advanced data analytics services like anomaly detection. Strider has been designed to guarantee important industrial properties such as scalability, high availability, fault-tolerant, high throughput and acceptable latency. These guarantees are obtained by designing the engine's architecture with state-of-the-art Apache components such as Spark and Kafka. To cope with the above-stated problems, Strider comes with a capability of optimizing logical query plan according to the state of data streams.

In Section 6.2, we detail the principal components of Strider and its properly defined grammar for continuous SPARQL query processing.

6.2. System Architecture

6.2.1. Syntax

Listing 6.2 introduces a running example that we will use throughout this chapter and Chapter 7. The example corresponds to a query encountered in the Waves project, *i.e.*, query Q_8 continuously processes the messages of various types of sensor observations. The semantic of this query can be interpreted as: *return all the observation ID of each sensor which measures water flow, temperature, and chlorine level.*

We introduce new lexical rules for continuous SPARQL queries which are tailored to a micro-batch approach. The query syntax we use in Strider is defined as follow:

$$
\begin{array}{lll}
StreamingClause & ::= & `Streaming'(`Window'|`Slide'|`Batch') \\
Window & ::= & (SlidingWindow) \\
Timeunit & ::= & (`Minutes'|`Seconds'|`Milliseconds') \\
RegisterClause & ::= & `Register'(`QueryId'|`Sparql')
\end{array}
$$

```
STREAMING { WINDOW [10 Seconds] SLIDE [10 Seconds] BATCH [5 Seconds] }
REGISTER { QUERYID [Q8] SPARQL [
  prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  prefix ssn: <http://purl.oclc.org/NET/ssnx/ssn/>
  prefix cuahsi: <http://www.cuahsi.org/waterML/>
  SELECT ?s ?o1 ?o2 ?o3
  WHERE {   ?s ssn:hasValue ?o1 (tp1); ssn:hasValue ?o2 (tp2);
               ssn:hasValue ?o3 (tp3).
            ?o1 rdf:type cuahsi:flow (tp4).
            ?o2 rdf:type cuahsi:temperature (tp5).
            ?o3 rdf:type cuahsi:chlorine    (tp6). }] }
```

Listing 6.1: Strider's query example (Q_8)

The Current implementation of Strider covers the most commonly used operator of SPARQL 1.1, we list all supported operators as follow:

- Query types: Select, Construct, Ask
- Algebra operators: Projection, Inner-Join, Optional (Left-Join), BGP, Union, Filter, Distinct, GroupBy

- Expressions (sub-operators of algebra): LogicalAnd, LogicalOr, Bound, LogicalNot, Equals, NotEquals, GreaterThan, GreaterThanOrEqual, LessThan, LessThanOrEqual, NodeValue, ExprVar

- Aggregations: Sum, Max, Min, Avg, Count

The `STREAMING` keyword initializes the application context of Spark Streaming and the windowing operator. More precisely, `WINDOW` and `SLIDE` respectively indicate the size and sliding parameter of a time-based window. The novelty comes from the `BATCH` clause which specifies the micro-batch interval of discretized stream for Spark Streaming. Here, a sliding window consists of one or multiple micro-batches.

The `REGISTER` clause is used to register standard SPARQL queries. Each query is identified by an identifier. The system allows to register several queries simultaneously in a thread pool. This is the motivation of thread pool: Thus by sharing the same application context and cluster resources, Strider launches all registered continuous SPARQL queries asynchronously by different threads.

Note that we do not expose the stream URI for the users as the other RSP engines do. Based on our experience, we find that exposing such parameters to the users increases the difficulties of system deployment and tuning. Such inconvenience may not be significant for a centralized system since the system setup is far less complicated.

6.2.2. Architecture Overview

Strider contains two principle modules: (1) data flow management. In order to ensure high throughput, fault-tolerance, and easy-to-use features, Strider uses Apache Kafka to manage input data flow. The incoming RDF streams are categorized into different *message topics,* which practically represent different types of RDF events. (2) Computing core. Strider core is based on the Spark programming framework. Spark Streaming receives, maintains messages emitted from Kafka in parallel, and generates data processing pipeline.

Figure 6.1 gives a high-level overview of the system's architecture. The upper part of the figure provides details on the application's data flow management. In a nutshell, data sources (IoT sensors) are sending messages to a publish-subscribe layer. This layer emits messages for the streaming layer which executes registered queries. The sensor's metadata are converted into RDF events for data integration purposes. We use Kafka to design the system's data flow management. Kafka is connected

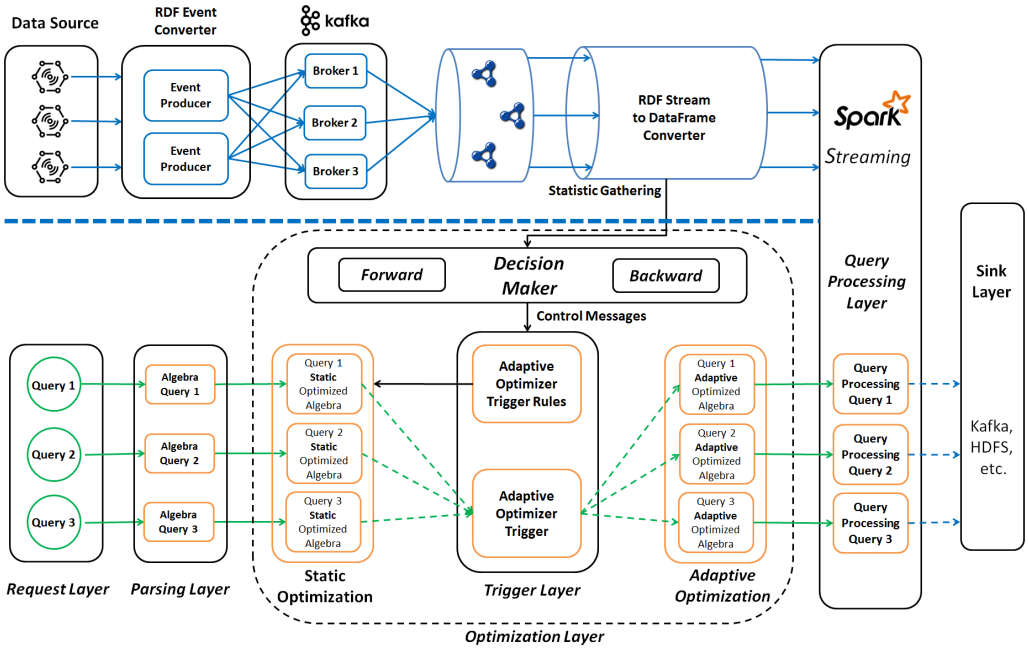


Figure 6.1.: Strider Architecture. Blue arrows and green arrows refer respectively to the dataflow and the processes

to Spark Streaming using a *Direct Approach*[1] to guarantee exactly-once semantics and parallel data feeding. The input RDF event streams are then continuously transformed to DataFrames.

```
STREAMING { WINDOW [10 Seconds] SLIDE [10 Seconds] BATCH [5 Seconds] }
REGISTER { QUERYID [Q8] SPARQL [
  prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  prefix ssn: <http://purl.oclc.org/NET/ssnx/ssn/>
  prefix cuahsi: <http://www.cuahsi.org/waterML/>
  SELECT ?s ?o1 ?o2 ?o3
  WHERE {   ?s ssn:hasValue ?o1 (tp1); ssn:hasValue ?o2 (tp2);
               ssn:hasValue ?o3 (tp3).
            ?o1 rdf:type cuahsi:flow (tp4).
            ?o2 rdf:type cuahsi:temperature (tp5).
            ?o3 rdf:type cuahsi:chlorine    (tp6). }] }
```

Listing 6.2: Strider's query example (Q_8)

The use case of waves is also mentioned at the beginning of this chapter. I think it is repeated to mention again the work flow here. Since previous paragraphs already

[1] https://spark.apache.org/docs/latest/streaming-kafka-integration.html

described how the data flow from one component to the others.

In this chapter, we describe the main components and the workflow of Strider. We implement Strider by using Spark Streaming as the underlying engine, and we use Kafka data flow management. The details of each component will be given in the next chapter.

7. Hybrid Adaptive Continuous SPARQL Query Processing in Strider

In this chapter, we detail continuous SPARQL query processing in Strider. The content of this chapter is organized as follows: Section 7.1 illustrates how Strider handles basic SPARQL query processing on Spark. Section 7.2 gives a deep insight of adaptive continuous SPARQL query processing in Strider. Section 7.3 showcases the results of the experiments that we have conducted on an Amazon Web Service (AWS) cluster. Finally, Section 7.4 concludes the work in this chapter.

7.1. RDF to RDBMS Mapping

We now briefly cover the mapping from RDF to RDBMS in Strider. In general, to enable SPARQL query processing on Spark, Strider:

- parses a query with Jena ARQ and obtains a query algebra tree in the Parsing layer.
- reconstructs the algebra tree into a new Abstract Syntax Tree (AST) based on the Visitor model.
- pushes obtained AST into the algebra Optimization layer.
- traverses the AST, binds the SPARQL operators to the corresponding Spark SQL relational operators for query evaluation.

Converting RDF data into a relational database model is a common way for RDF data management. The advantage of this approach is to benefit from mature techniques developed in RDBMS systems for RDF data processing. Figure 7.1 gives a SPARQL query example and its translation in SQL (considering that all triples are stored in a single RDBMS table denoted database) in Listing 6.2.

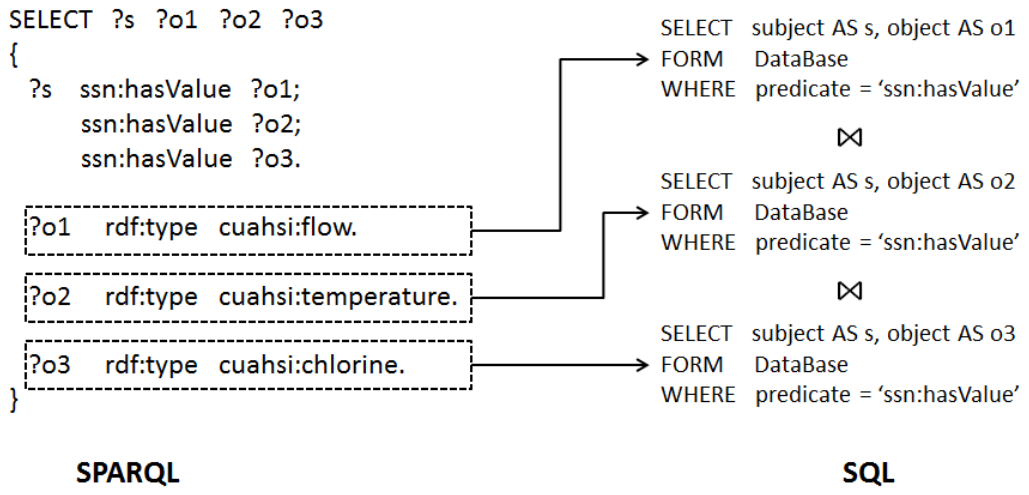


Figure 7.1.: Conversion from RDF to RDBMS

An intuitive observation shows that SPARQL query processing with RDBMS as back-end involves intensive self-joins [84]. The inner-join operator between two relations in a distributed environment refers to a shuffle operation, which is a main performance bottleneck. Therefore, the join order of triple patterns becomes the critical factor for query processing.

7.2. Hybrid Adaptive Query Processing

7.2.1. Query processing outline & trigger layer

Strider possesses a hybrid SPARQL query optimization strategy, two optimization components are proposed, *i.e.*, static and adaptive, which are respectively based on heuristic rules and (stream-based) statistics (Figure 7.2).

The first trigger layer decides whether the query processing adopts a static or an adaptive approach. Once the system moves to adaptive query processing, the second trigger layer determines whether backward (B-AQP) and forward (F-AQP) should be applied in AQP. They mainly differ on when, *i.e.*, at the previous or current window, the query plan is computed.

Before providing a detailed explanation, we briefly present the interactions between the main components of the query optimizer. In general, the system will first estimate the size of the input stream in the trigger layer. If the data size is considered small

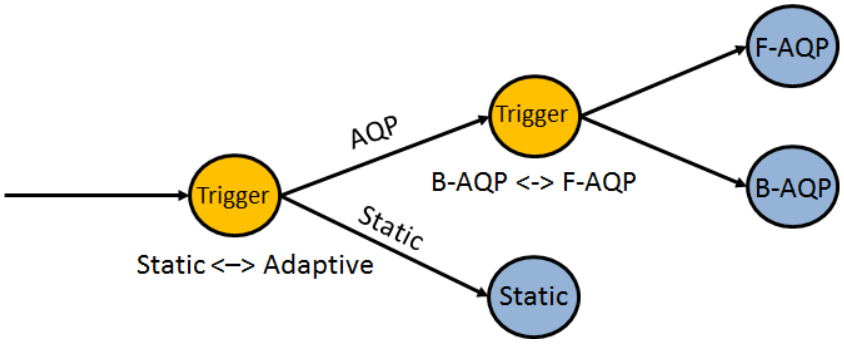


Figure 7.2.: Strider Hybrid Query Optimization

(the default threshold of data volume is empirically set at 100 megabytes. However, this parameter depends on query complexity and cluster resources.), Strider will takte the already-calculated static query plan since there will be no performance gain by triggering the adaptive optimization. When the adaptive optimization is triggered, the system will decide whether backward or forward for adaptive query processing.

Strider's optimizer is tailored for Basic Graph Pattern (BGP) reconstruction at run-time. The system thus varies the optimal join ordering of triple patterns based on the collected statistics. Fundamentally, both static and adaptive optimizations are processed using a graph $G^U = (V, E)$, denoted Undirected Connected Graph (UCG) [13] where vertices represent triple patterns and edges symbolize joins between triple patterns. Naturally, for a given query q and its query graph $G^Q(q)$, $G^U(q) \subseteq G^Q(q)$. A UCG showcases the structure of a BGP and the join possibilities among its triple patterns. That query representation is considered to be more expressive [85] than the classical RDF query graph. The weight of UCG's vertices and edges correspond to the selectivity of triple patterns and join, respectively. Once the weights of an UCG are initialized, the query planner automatically generates an optimal logical plan and triggers a query execution.

Strider's static optimization retains the philosophy of [86]. Basically, static optimization implies a heuristics-based query optimization. It ignores data statistics and leads to a static query planner. In this case, unpredictable changes in data stream structures may incur a bad query plan. The static optimization layer aims at giving a basic performance guarantee. The predefined heuristic rules set empirically assign the weights for UCG vertices and edges. Next, the query planner determines the shortest traversal path in the current UCG and generates the logical plan for query

execution. The obtained logical plan represents the query execution pipeline which is permanently kept by the system.

The Trigger layer supports the transition between the stages of static optimization and adaptive optimization. In a nutshell, that layer is dedicated to notify the system whether it is necessary to proceed an adaptive optimization. Our adaptation strategy requires collecting statistical information and generating an execution logical plan. The overhead coming with such actions is not negligible in a distributed environment. The Strider prototype provides a set of straightforward trigger rules, *i.e.*, the adaptive algebra optimization is triggered by a configurable workload threshold. The threshold refers to two factors: (1) the input number of RDF events/triples; (2) the fraction of the estimated input data size and the allocated executors' heap memory.

7.2.2. Query plan generation

This section explains how we collect statistics and construct query plan. Then, we give an insight into the AQP optimization, which is essentially a cardinality-based optimization.

Both compile-time (static) and run-time (adaptive) query plan generation are base on UCG. Once an input query is initialized, we parse the query and reconstruct its corresponding UCG graph. This step is done only once per query, i.e. at query compiled time. Figure 7.3 displays the corresponding UCG of query in Listing 6.2 .

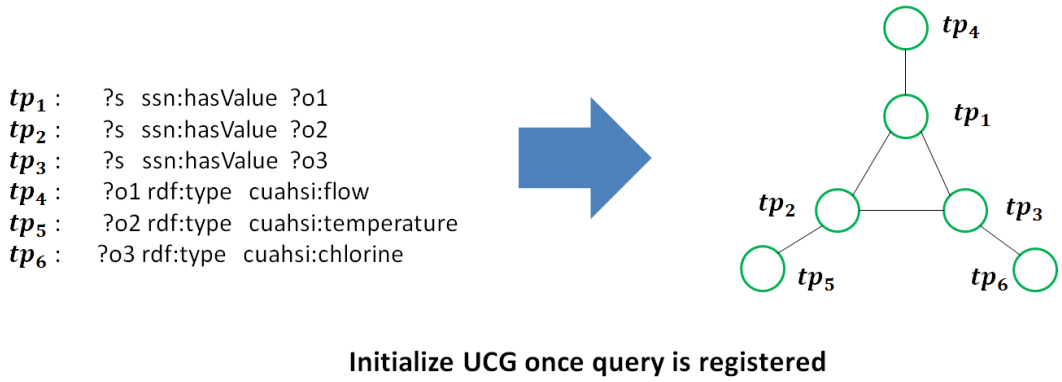


Figure 7.3.: UCG creation

Unlike systems based on a greedy and left-deep tree generation, *e.g.*, [13, 43], Strider makes a full usage of CPU computing resources and benefits from parallel hardware settings. It thus creates query logical plans in the form of general (bushy)

directed trees. Hence, the nodes with the same height in a query plan p_n can be asynchronously computed in a non-blocking way (in the case where computing resources are allowed). Coming back to our Listing 6.2 example, Figure 7.5 refines the procedure of query processing (F-AQP) at w_n, $n \in N$. If w_n contains multiple RDDs (micro-batches), the system performs the union all RDDs and generates a new combined RDD. Note that the union operator has a very low-cost in Spark. Afterward, the impending query plan optimization follows three steps: (a) UCG (weight) initialization; (b) UCG path cover finding; (c) query plan generation.

UCG weight initialization is briefly described in Algorithm 1 and Figure 7.6 (step (a), step (b)). Since triple patterns are located at the bottom of a query tree, the query evaluation is performed in a bottom-up fashion and starts with the selection of triple patterns $\sigma(tp_i)$, $1 \leqslant i \leqslant I$ (with I the number of triple patterns in the query's BGP). The cardinality $Card(tp_i)$ and the already-computed intermediate result of tp_i are cached main memory. The cardinality (*i.e.*, statistic gathering) is obtained by *count* action, which returns the number of rows for each triple pattern (Figure 7.4).

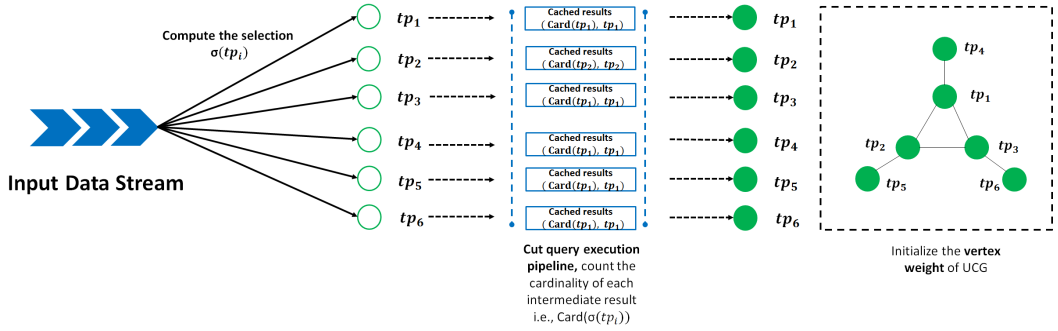


Figure 7.4.: UCG weight initialization

The system computes $\sigma(tp_i)$ asynchronously for each i and temporally caches the corresponding results ($R^{\sigma}(tp_i)$) in memory. $Card(tp_i)$, *i.e.*, the cardinality of $R^{\sigma}(tp_i)$, is computed by a Spark count action. Thence, we can directly assign the weight of vertices in $G^U(Q)$. Note that the estimation of $Card(tp_i)$ is exact.

Once all vertices are set up, the system predicts the weight of edges (*i.e.*, joined patterns) in $G^U(q)$. We categorize two types of joins (edges): (i) star join, includes two sub-types, *i.e.*, star join without bounded object and star join with bounded object; (ii) non-star join. To estimate the cardinality of join patterns, we make a

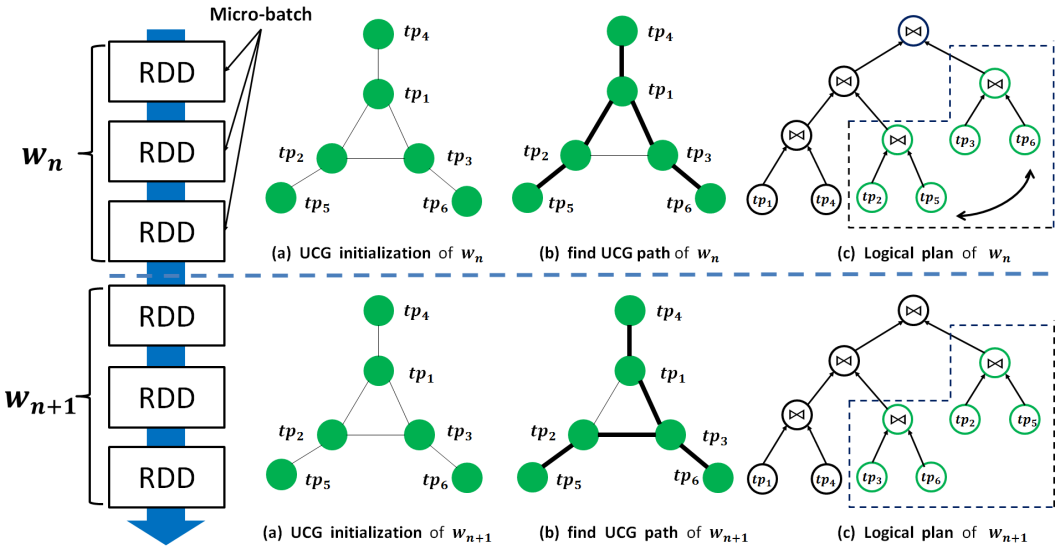


Figure 7.5.: Dynamic Query Plan Generation for Q_8

trade-off between accuracy and complexity. The main idea is inspired by a research conducted in [13, 85, 87]. However, we infer the weight of an edge from its connected vertices, *i.e.*, no data pre-processing is required. The algorithm begins by iteratively traversing $G^U(q)$ and identifies each vertex $v \in V$ and each edge $e \in E$. Then we can decompose $G^U(q)$ into the disjoint star-shaped joins and their interconnected chains (Figure 7.6, step (b)). The weight of an edge in a star join shape is estimated by the function getStarJoinWeight. The function first estimates the upper bound of each star join output cardinality (*e.g.*, , $Card(tp_1 \bowtie tp_2 \bowtie tp_3)$), then assigns the weight edge by edge. Every time the weight of the current edge e is assigned, we mark e as visited. This process repeats until no more star join can be found. Then, the weight of unvisited non-star join shapes is estimated by the function getNonStarJoinweight. It lookups the two vertices of the current edge, and chooses the one with smaller weight to estimate the edge cardinality. The previous processes are repeated until all the edges have been visited in $G^U(q)$.

UCG path cover finding & Query plan generation. Figure 7.6 step (c) introduces path cover finding and query plan generation. The system starts by finding the path cover in $G^U(q)$ right after $G^U(q)$ is prepared. Intuitively, we search the undirected path cover which links all the vertices of $G^U(q)$ with a minimum total edge weight. The path searching is achieved by applying Floyd–Warshall algorithm [88] iteratively. The extracted path $Card(G^U(q)) \subseteq G^U(q)$, is regarded as

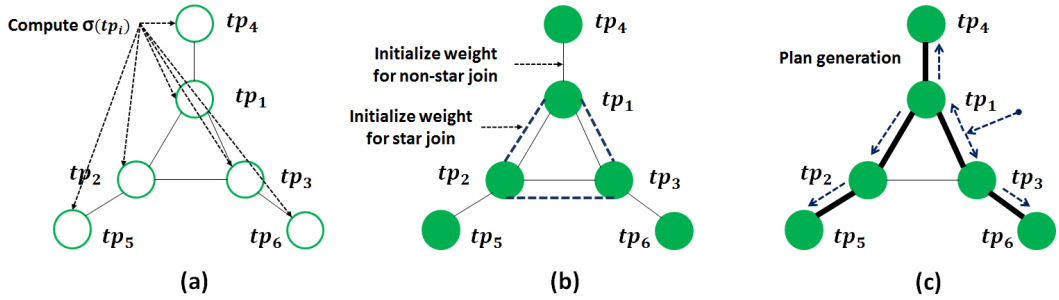


Figure 7.6.: Initialized UCG weight, find path cover and generate query plan

the candidate for the logical plan generation. Finally, we construct p_n, the logical plan of $G^U(q)$ at w_n, in a top-down manner (Figure 7.6, step (c)). Note that path finding and plan generation are both computed on the driver node and are not expensive operations (around 2 - 4 milliseconds in our case).

Algorithm 1 UCG weight initialization

Input : query q, $G^U(q) = (V, E) \subseteq G^Q(q)$, current buffered window w_n
Output : $G^U(q)$ with weight-assigned
while $\exists v$ *unvisited* $\in V$ **do**
  mark v as visited, $R^\sigma(v) \leftarrow$ `compute` (v) `buffer` $(v, R^\sigma(v)) \wedge v$.weight $\leftarrow Card(v)$
end
while $\exists e$ *unvisited* $\in E$ **do**
  mark e as visited **if** $(\exists$ *star join* $S_J) \wedge e \cap S_J \neq \varnothing$ **then**
    locate each $S_J \in G^U(q)$
    foreach $\forall e_S \in S_J$ **do**
      mark e_S as visited e_S.weight $\leftarrow$ `getStarJoinWeight`$(S_J, e_S.vertices)$
    end
  end
  else e.weight $\leftarrow$ `getNonStarJoinWeight`(S_J);
end

7.2.3. B-AQP & F-AQP

We propose a dual AQP strategy, namely, **backward** (B-AQP) and **forward** (F-AQP). B/F-AQP depict two philosophies for AQP, Figure 7.7 roughly illustrates how B/F-AQP switching is decided at run-time, *i.e.*, this is the responsibility of the Decision Maker component. Generally, B-AQP and F-AQP are using similar techniques for query plan generation. Compared to F-AQP, B-AQP delays the process for query plan generation.

Our B-AQP strategy is inspired by [46]'s pre-scheduling. Backward implies gathering, feeding back the statistics to the optimizer on the current window, then the optimizer constructs the query plan for the next window. That is the system computes the query plan p_{n+1} of a window w_{n+1} using the statistics of a previous window w_n. Strider possesses a time-driven execution mechanism, the query execution is triggered periodically with a fixed update frequency s (*i.e.*, sliding window size). Between two consecutive window w_n and w_{n+1}, there is a computing barrier to reconstruct the query plan for w_{n+1} based on the collected statistics from a previous window w_n. Suppose the query execution of w_n consumes a time t_n (*e.g.*, in seconds), then for all $t_n < s$, the idle duration $\delta_n = s - t_n$ allows to re-optimize the query plan. But δ_n should be larger than a configurable threshold ϵ. For $\delta_n < \epsilon$, the system may not have enough time to (i) collect the statistic information of w_n and (ii) to construct a query plan for w_{n+1}. This potentially expresses a change of incoming steams and a degradation of query execution performance. Hence, the system decides to switch to the F-AQP approach.

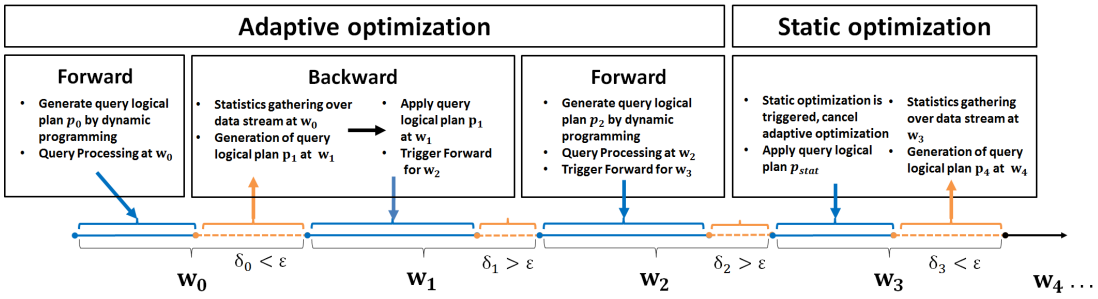


Figure 7.7.: Decision Maker of Adaptation Strategy

F-AQP applies a Dynamic Programming strategy to find the optimal logical query plan for the current window w_n. The main purpose of F-AQP is to adjust the system state as soon as possible. The engine executes a query, collects statistics and computes

the logical query plan simultaneously. Here, the statistics are obtained by counting intermediate query results, which causes data shuffling and DAG interruption, *i.e.*, the system has to temporally cut the query execution pipeline. In Spark, such suspending operation is called an *action*, which immediately triggers a job submission in Spark application. However, frequent job submission may bring some side effects. The rationale is, for a master-slave based distributed computing framework (*e.g.*, Spark, Storm) uses a master node (*i.e.*, driver) to schedule jobs. The driver locally computes and optimizes each submitted DAG and returns the control messages to each worker node for parallel processing. Although the "count" action itself is not expensive, the induced side effects (*e.g.*, driver job-scheduling/submission, communication of control message between driver and workers) will potentially impact the system's stability. For instance, based on our experience, F-AQP's frequent job submission and intermediate data persistence/unpersistence put a great pressure on the JVM's Garbage Collector (GC), *e.g.*, untypical GC pauses are observed from time to time in our experiment.

Decision Maker. Through experimentations of different Strider configurations, we understood the complementarity of both the B-AQP and F-AQP approaches. Real performance gains can be obtained by switching from one approach to another. This is mainly due to their properties which are summarized as below:

- **B-AQP.** For B-AQP, no overhead of dynamic programming for run-time query plan generation is involved. However, B-AQP generate approximate optimal query plan through previously-collected statistic, which could be inaccurate for query plan generation.
- **F-AQP.** For F-AQP, the system takes extra overhead to collect statistics and generate query execution plan at run-time. This overhead can not be ignored, however, it obtains the precise statistic information for query plan generation.

We designed a decision maker to automatically select the most adapted strategy for each query execution. The decision maker takes into account two parameters: a configurable switching threshold $\epsilon \in \ ]0, 1[$; $\gamma_n = \frac{t_n}{s}$, the fraction of query execution time t over windowing update frequency s. For the query execution at w_n, if $\gamma_n < \epsilon$, the system updates the query plan from p_n to p_{n+1} for the next execution. Otherwise, the system recomputes p_{n+1} by DP at w_{n+1} (see Algorithm 2). We empirically set $\epsilon = 0.7$ by default.

The decision maker plays a key role for maintaining the stability of the system's performance. Our experiment (Sec. 7.3.2) shows that, the combination of F/B-AQP

Algorithm 2 B-AQP and F-AQP Switching in Decision Maker

Input : query q, switching threshold ϵ, sliding window $W = \{w_n\}_{n \in N}$, update frequency s of W

foreach $w_n \in W$ **do**
  $t_n \leftarrow$ `getRuntime` { `execute` (q) } // executionTime
  $\lambda_n \leftarrow$ `getAdaptiveStrategy` (ϵ, t_n, s) // adaptiveStrategy
  if λ_n == *Backward* **then**
    update query plan p_n of q at w_n
    $p_{n+1} \leftarrow$ `update` (p_n)
  end
  if λ_n == *Forward* **then** Recompute p_{n+1} at w_{n+1};
end

through decision maker is able to prevent the sudden performance declining during a long running time.

7.3. Experiments

Strider is written in Scala, the code source can be found here[1]. To enable SPARQL query processing on Spark, Strider parses a query with Jena ARQ and obtains a query algebra tree in the Parsing layer. The system reconstructs the algebra tree into a new Abstract Syntax Tree (AST) based on the Visitor model. Basically, the AST represents the logical plan of a query execution. Once the AST is created, it is pushed into the algebra Optimization layer. By traversing the AST, we bind the SPARQL operators to the corresponding Spark SQL relational operators for query evaluation.

We use a series of micro-benchmarks to measure the performance of Strider, including the system's adaptivity. We first focus on continuous SPARQL query processing with stable stream structure. *I.e.*, , in the ideal case, incoming data streams maintain invariant structure, the proportion of variant types of RDF triples does not change over time. We next demonstrate the efficiency of Strider's AQP by feeding the system structurally unstable RDF streams, *i.e.*, the structure of input stream varies over time. To that end, we first present the experimental setup and then provide results.

[1] https://github.com/renxiangnan/strider

7.3.1. Experimental Setup

We test and deploy our engine on an Amazon EC2/EMR cluster of 9 computing nodes with resource management handled by Yarn. The system holds 3 nodes of m4.xlarge for data flow management (*i.e.*, Kafka broker and Zookeeper [89]). Each node has 4 CPU virtual cores of 2.4 GHz Intel Xeon E5-2676, 16 GB RAM and 750 MB/s bandwidth. We use Apache Spark 2.0.2, Scala 2.11.7 and Java 8 as baselines for our evaluation. The Spark (Streaming) cluster is configured with 6 nodes (1 master, 5 workers) of type c4.xlarge. Each one has 4 CPU virtual cores of 2.9 GHz Intel Xeon E5-2666, 7.5 GB RAM and 750 MB/s. The experiments of Strider on local mode, C-SPARQL and CQELS are all performed on a single instance of type c4.xlarge.

Datasets & Queries. We evaluated our system using two datasets that are built around real world streaming use cases: *SRBench* and *Waves*.

We have already introduced SRBench in 4.1, we thus only give some further information about Waves dataset. Waves dataset describes different water measurements captured by sensors. Values of flow, water pressure and chlorine levels are examples of these measurements. The value annotation uses three popular ontologies: SSN, CUAHSI-HIS and QUDT. Each sensor observes and records at least one physical phenomenon or a chemical property, and thus generates RDF data stream through Kafka producer. Our micro-benchmark contains 9 queries, denoted from Q_1 to Q_9[2]. The road map of our evaluation is designed as follow: (1) injection of structurally stable stream for experiment of Q_1 to Q_6. Q_1 to Q_3 are tested by SRBench datasets. Here, a comparison between Strider and the state of the art RSP systems *e.g.*, C-SPARQL and CQELS are also provided. Then we perform Q_4 to Q_6 based on Waves dataset. (2) Injection of structurally unstable stream. We generate RDF streams by varying the proportion of different types of Kafka messages (*i.e.*, sensor observations). For this part of the evaluation, queries Q_7 to Q_9 are considered.

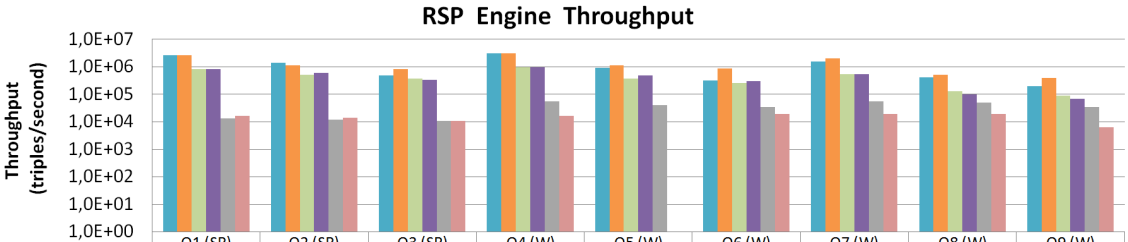
In accordance with the discussion at the end of Chapter 5, we choose system throughput and query latency as two primary performance metrics. Be aware that, Throughput indicates how many data can be processed in a unit of time. Throughput is denoted as "triples per second" in our case. Latency means how long does the RSP engine consumes between the arrival of an input and the generation of its output. We did not record the latency of C-SPARQL, CQELS and Strider in local mode for two reasons: (1) given the scalability limitation of C-SPARQL, we have to control input stream rate within a low level to ensure the engine can run normally [54]. (2)

[2]Check the wiki of our github page for more details of the queries and datasets

due to its design, based on a so-called eager execution mechanism and *DStream* R2S operator, the latency measure in CQELS is unfeasible [54].

Performance tuning on Spark is quite difficult. Inappropriate cluster configuration may seriously hinder engine performance. So far we can only empirically configure Spark cluster and tune the cluster settings step by step. We briefly list some important performance settings based on our experience. First of all, we apply some basic optimization techniques. *e.g.*, using Kryo serializer to reduce the time for task/data serialization. Besides, we generally considered adjustments of Spark configuration along three control factors to achieve better performance. The first factor is the size of micro-batch intervals. Smaller batch sizes can better meet real-time requirements. However, it also brings frequent job submissions and job scheduling. The performance of a BSP system like Spark is sensitive to the chosen size of batch intervals. The second factor is GC tuning. Set appropriately, the GC strategy (*e.g.*, using Concurrent Mark-Sweep) and storage/shuffle fraction may efficiently reduce GC pressure. The third factor is the parallelism level. This includes the partition number of Kafka messages, the partition number of RDD for shuffling, and the upper/lower bound for concurrent job submissions, *etc.*.

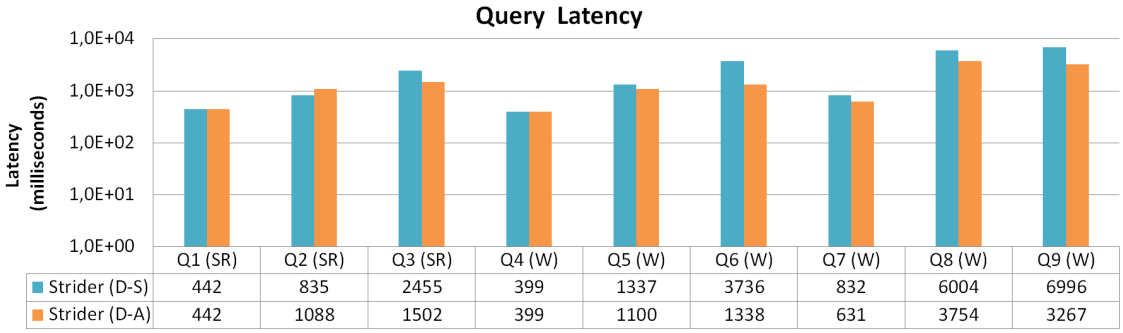
7.3.2. Evaluation Result



	Q1 (SR)	Q2 (SR)	Q3 (SR)	Q4 (W)	Q5 (W)	Q6 (W)	Q7 (W)	Q8 (W)	Q9 (W)
Strider (D-S)	2620005	1431808	495380	3116147	939506	325328	1555542	422591	194225
Strider (D-A)	2620005	1164471	838531	3116147	1148896	886769	2005617	524181	387742
Strider (L-S)	837860	510231	382972	965986	370904	264665	538524	127276	91710
Strider (L-A)	837860	608228	329683	965986	481945	310039	538524	98573	68345
C-SPARQL	13142	12008	10934	56179	40469	34882	54917	49985	33988
CQELS	16817	14211	10637	16541	0	19574	19534	19472	6527

Figure 7.8.: RSP engine throughput (triples/second). **D/L-S:** Distributed/Local mode Static Optimization. **D/L-A:** Distributed/Local mode Adaptive Optimization. **SR:** Queries for SRBench dataset. **W:** Queries for Waves dataset.

In Figure 7.8, we observe that Strider generally achieves million/sub-million-level throughput under our test suite. Note that both Q_1 and Q_4 have only one join, *i.e.*, optimization is not needed. Most tested queries scale well in Strider. Adaptive optimization generates query plans based on the workload statistics. In total, it provides a more efficient query plan than static optimization. But the gain of AQP for the simple queries that have less join tasks (*e.g.*, Q_1, Q_5) becomes insubstantial. We also found out that, even if Strider runs on a single machine, it still provides up to 60x gain on throughput compared to C-SPARQL and CQELS. Figure 7.9 shows Strider attains a second/sub-second delay. Obviously, for queries with 2 triple patterns in the query's BGP, we can observe the same latency between static and adaptive optimizations, Q_1 and Q_4. Query Q_2 is the only query where the latency of the adaptive approach is higher than the static one. This is due to the very simple structure of the BGP (2 joins in the BGP). In this situation, the overhead of DP covers the gain from AQP. For all other queries, the static latency is higher than the adaptive one. This is justified by more complex BGP structures (more than 5 triple patterns per BGP) or some union of BGPs.

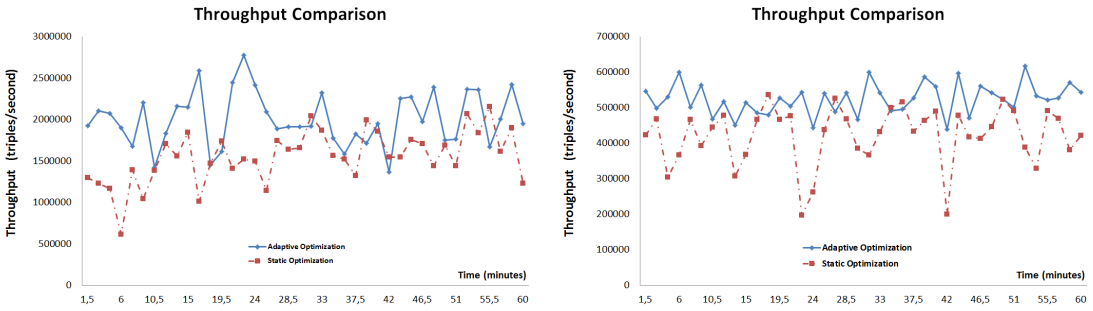


	Q1 (SR)	Q2 (SR)	Q3 (SR)	Q4 (W)	Q5 (W)	Q6 (W)	Q7 (W)	Q8 (W)	Q9 (W)
Strider (D-S)	442	835	2455	399	1337	3736	832	6004	6996
Strider (D-A)	442	1088	1502	399	1100	1338	631	3754	3267

Figure 7.9.: Query latency (milliseconds) for Strider (in distributed mode)

On the contrary, the average throughput of C-SPARQL and CQELS is maintained in the range of 6.000 and 50.000 triples/second. The centralized designs of C-SPARQL and CQELS limit the scalability of the systems. Beyond the implementation of query processing, the reliability of data flow management on C-SPARQL and CQELS could also cause negative impact on system robustness. Due to the lack of some important features for streaming system (*e.g.*, back pressure, checkpoint and failure recovery) once input stream rate reaches to certain scale, C-SPARQL and CQELS start behaving abnormally, *e.g.*, data loss, exponential increasing latency or query process

interruption [54, 90]. Moreover, we have also observed that CQELS' performance is insensitive to the changing of computing resources. We tested CQELS on different EC2 instance types, *i.e.*, with 2, 4 and 8 cores, and the results evaluation variations were negligible.



(a) (b)

Figure 7.10.: Record of throughput on Strider. (a)-throughput for q_7; (b)-throughput for q_8

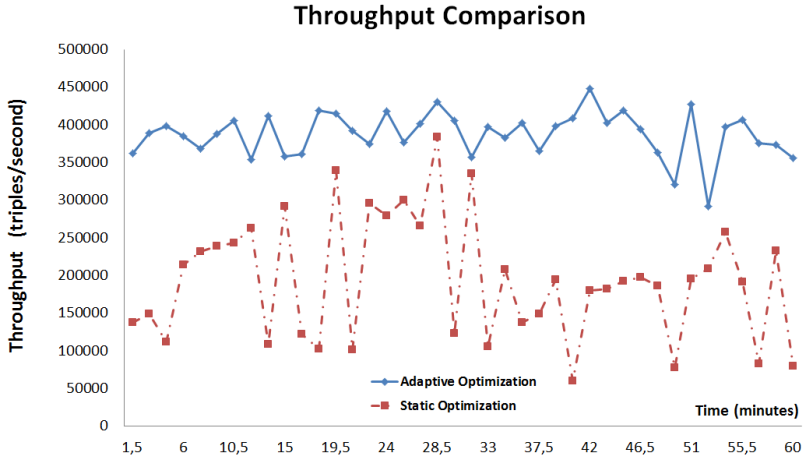
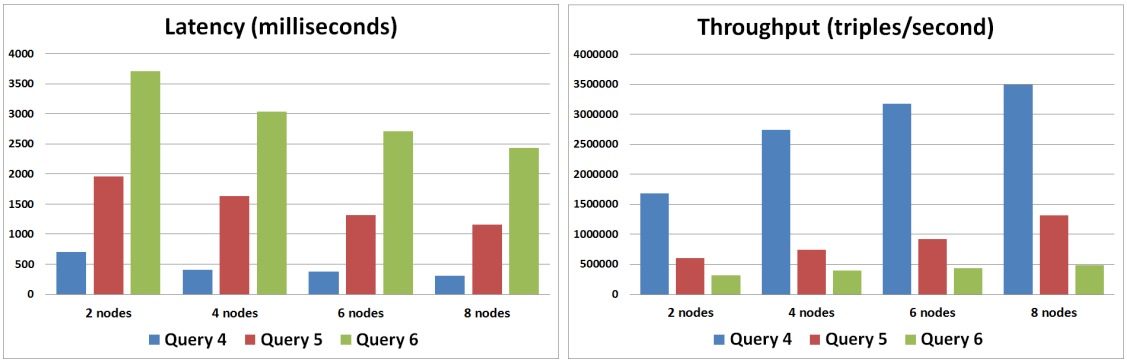


Figure 7.11.: Throughput for q_9 on Strider



(a) (b)

Figure 7.12.: Scalability evaluation of Q_4, Q_5, Q_6 on Strider. (a)-throughput; (b)-latency for

Figure 7.10 and Figure 7.11 concern the monitoring of Strider's throughput for Q_7 to Q_9. We recorded the changes of throughput over a continuous period of time (one hour). The source stream produces the messages with different types of sensor observations. The stream is generated by mixing temperature, flow and chlorine-level measurement with random proportions. The red and blue curves denote query with respectively static and adaptive logical plan optimization. For Q_7 and Q_8 (Figure 7.10), except when some serious throughput drops have been observed in 7.12b, static and adaptive planners return a close throughput trend. For a more complex query Q_9 (Figure 7.11), which contains 9 triple patterns and 8 join operators. Altering logical plans on Q_9 causes significant impact on engine performance. Consequently, our adaptive strategy is capable to handle the structurally unstable RDF stream. Thus the engine can avoid a sharp performance degradation. Figur 7.12 reports a group of scalability test. Strider scales well when the number of machines varies from 2 nodes to 8 nodes.

Through this experiment, we identified some shortcomings in Strider that will be addressed in future work: (1) the data preparation on Spark Streaming is relatively expensive. It costs around 0.8 to 1 second to initialize before triggering the query execution in our experiment. (2) Strider has a more substantial throughput decreasing with an increasing number of join tasks. In order to alleviate this effect, possible solutions are to enlarge the cluster scale or to choose a more powerful driver node. (3) Strider does not support well high concurrent requests, although this is not at the moment one of our system design goals. *E.g.*, some use cases demand to process a big amount of concurrent queries. Even through Strider allows to perform multiple

queries asynchronously, it could be less efficient.

7.4. Conclusion

In this chapter, we present the details of continuous SPARQL query processing in Strider. Strider is built on top of Spark Streaming and Kafka to support high performance query evaluation and thus possesses the characteristics of a production-ready RSP. Strider comes with a set of hybrid AQP strategies: *i.e.*, static heuristic rule-based optimization, forward and backward adaptive query processing. We insert the trigger into the optimizer to attain the automatic strategy switching at query runtime. Moreover, with its micro-batch approach, Strider fills a gap in the current state of RSP ecosystem which solely focuses on record-at-a-time. Through our micro-benchmark based on real-word datasets, Strider provides a million/sub-million-level throughput and second/sub-second latency, a major breakthrough in distributed RSPs. And we also demonstrate the system reliability which is capable to handle the structurally instable RDF streams.

8. Distributed RDF Stream Reasoning in Strider with Litemat

8.1. Introduction

In this chapter, we focus on the extension of Strider, namely Strider^{R}, that integrates a set of inferencce services in Strider for RDFS and `sameAs`, sometimes denoted as RDFS++ stream reasoning in the cloud. Strider^{R} extends an existing reasoning technique *i.e.*, LiteMat to adapt to large volumes of semantically annotated data streams.

The main goal amounts to producing sound and complete answers from a set of continuous queries. This problem is quite important for many Big data applications in domains such as science, finance, information technology, social networks and Internet of Things (IoT) in general. For instance, in the Waves project, we are dealing with "real-time" anomaly detection in large water distribution networks. By working with domain experts, we found out that such detections can only be performed using reasoning services over data streams. Such inferences are performed over knowledge bases (KB) about the sensors used in water networks, *e.g.*, sensor characteristics together with their measure types, locations, geographical profiles, events occurring nearby, etc..

Tackling this issue implies to find a trade-off between high data throughput and low latency on the one hand and reasoning over semantically annotated data streams on the other hand. This is notoriously hard and even though it is currently getting some attention, it still remains an open problem.

Existing RSP engines are either not scalable (*i.e.*, they do not distribute data and/or processing) or do not support expressive reasoning services. The velocity aspect of Big Data implies the emergence of almost real time applications which are generally expressed via the processing of data streams. Alongside this frequent system design movement, cognitive aspects, such as the ability to reason about represented data and knowledge, are becoming prominent.

We present Strider^{R} that addresses these two dimensions, *i.e.*, it infers data

necessary for the computation of sound and complete answer sets of continuous queries. Our work deals with the hardest problems in designing such a system: guaranteeing high throughput and acceptable latency with reasoning services performed over expressive Knowledge Bases.

Strider^{R} combines Strider RSP engine with a reasoning approach. As previously introduced in Chapter 7, Strider is capable of processing and adaptively optimizing continuous SPARQL queries. Nevertheless, it was not originally designed to perform inferences. Hence, a main goal of this work is to integrate stream reasoning services that can support the main RDFS inferences together with the `owl:sameAs` property (henceforth denoted `sameAs`).

Intuitively, this property enables to define aliases between RDF resources. This is frequently used when a domain's (meta) data is described in a collaborative way, *i.e.*, a given object has been described with different identifiers (possibly by different persons) and are later reconciled by stating their equivalence. Reasoning with the `sameAs` property is motivated by the popularity of `sameAs` across many datasets, including several domains of the Linked Open Data (LOD). For instance, the `sameAs` constructor is frequently encountered to practically define or maintain ontologies. In [91], the authors measured the frequency of `sameAs` triples in an important repository of LOD. That property was involved in more than 58 million triples over 1,202 unique domain names with the most popular domains being biology, *e.g.*, Bio2rdf and Uniprot (respectively 26 and 6 million `sameAs` triples), and general domains *e.g.*, DBpedia (4.3 million `sameAs` triples).

Moreover, the knowledge management of LOD, estimated to more than 100 billion triples, clearly amounts to big data issues. In our Waves running example, we also found out that, due to the cooperative ontology building, many `sameAs` triples were necessary to re-conciliate ontology designs. We discovered several of these situations in the context of the IoT Waves project. For instance, we found out that sensors or locations in water distribution networks could be given different identifiers.

These `sameAs` triples are generally persisted in RDF stores[92] but data streams are providing dynamic data streams about these resources. Such metadata are needed to perform valuable inferences. In the Waves project, they correspond to the topology of the network, characteristics of the network's sensors, etc. We consider that the presence of static metadata can be generalized to many domains, *e.g.*, life science, finance, social, cultural, and is hence important when designing a solution that reasons over their data streams. Strider^{R} thus needs to reason over both static KBs, *i.e.*, a set of facts together with some ontologies, and dynamic data streams,

i.e., a set of facts which once annotated with ontology concepts and properties can be considered as an ephemeral extension of the KB fact base. Apart from `sameAs` inferences, the most prevalent reasoning services in a streaming context are related to ontology concept and property hierarchies. We are addressing these inferences tasks via a trade-off between the query rewriting and materialization approaches.

The main contributions of this chapter are:

- (i) to combine a scalable, production-ready RSP engine that supports reasoning services over RDFS plus the `sameAs` property.
- (ii) to minimize the reasoning cost, and thus to guarantee high throughput and acceptable latency.
- (iii) to propose a thorough evaluation of the system and thus to highlight its relevance.

The chapter is organized as follows. Section 8.2 provides an overview of the system's architecture. In Section 8.3, we detail a running example. Then Sections 8.4 and 8.5 provide reasoning approaches with respectively concept/property hierarchies and `sameAs` individuals. Section 8.6 evaluates Strider$^{\mathrm{R}}$ and demonstrates its relevancy. Finally, we conclude this chapter in Section 8.7.

8.2. Strider$^{\mathrm{R}}$ Overview

This section gives a high-level overview of the Strider$^{\mathrm{R}}$ system. Its architecture has been designed to support the distribution the processing of RDF data streams and to provide guarantees on fault tolerance, high throughput and low latency. Moreover, Strider$^{\mathrm{R}}$ aims to integrate efficient reasoning services into an optimized continuous query processing solution.

Figure 8.1 shows 3 vertical "columns" or groups of functions: (a), (b), and (c). On the middle and the right, (a) and (b) are off-line pre-processing functions. Note that both (a) and (b) are based on LiteMat [93] for knowledge base encoding and query rewriting. On the left, (c) is the on-line stream processing pipeline. We detail the three groups below, in the order they participate to the whole workflow:

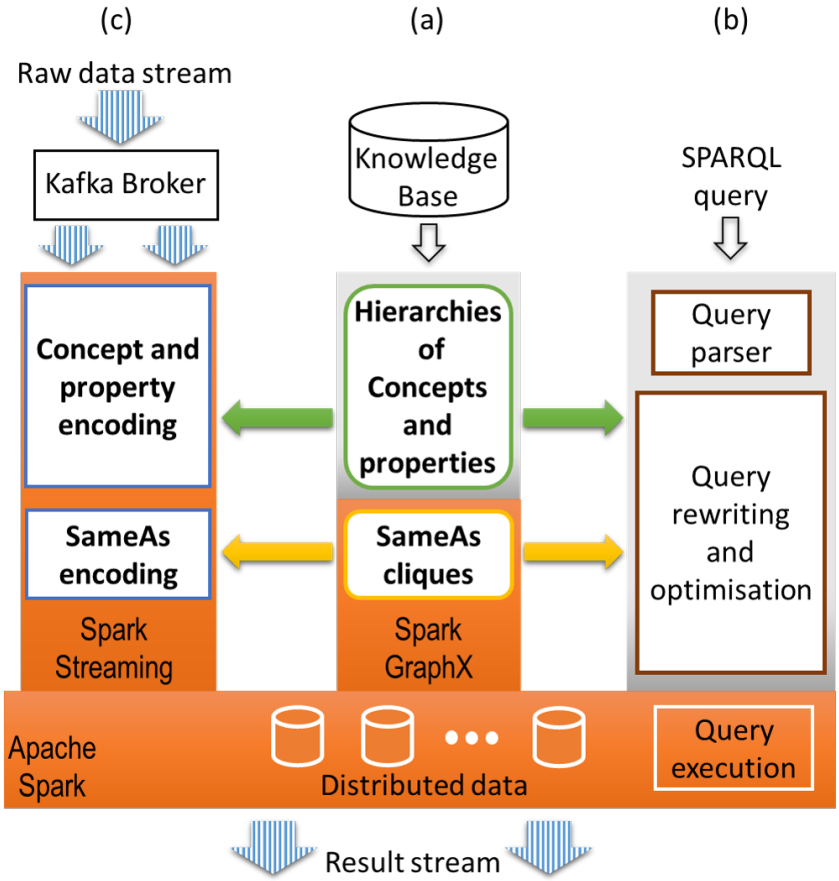


Figure 8.1.: Strider^{R} Functional Architecture

(a) Off-line KB encoding consists in reading the static knowledge base to get the classification of concepts and properties, both organized into a hierarchy. The knowledge base also contains `sameAs` predicates from which `sameAs` cliques are detected. This step generates the identifiers for each concept, property and cliques of `sameAs` individuals that is later used in steps (b) and (c). Note that we use the GraphX library of Apache Spark to efficiently process clique detection in parallel.

(b) Off-line query preparation. Once a SPARQL query is registered into the system, it is parsed then rewritten into a plan composed of basic RDF processing operations. The plan is extended with dedicated operations to support the reasoning over properties and concepts, using the semantic identifiers generated at step (a). The plan also relies on the `sameAs` cliques information to support the `sameAs` reasoning for various use cases.

(c) On-line stream semantic encoding. The data stream is encoded based on the hierarchical codes generated from the static KB at step (a). Each concept and property is replaced by an identifier that allows for fast reasoning over concept and property hierarchies. The stream is also completed with `sameAs` clique membership information. For the purpose of ensuring high throughput and fault-tolerance, we use Apache Kafka to manage the data flow. The incoming raw data are assigned to so-called Kafka *topics*. The Kafka broker distributes the topics and the corresponding data over a cluster of machines to enable parallelism of upstream/downstream operations. Then the distributed streams seamlessly enter the Spark Streaming layer which encodes them in parallel.

Continuous query processing. The logical plan obtained at step (b) is pushed into the query execution layer (*i.e.*, the base layer of Figure 8.1 on which the three "groups" (*i.e.*, a,b and c) of previously defined functions rely). To achieve continuous SPARQL query processing on Spark Streaming, we bind the SPARQL operators to the corresponding Spark SQL relational operators that access a distributed compressed in-memory representation of the data stream (through the DataFrame and the RDD APIs provided by the Spark platform). Note that, Strider$^{\mathrm{R}}$ is capable of adjusting the query execution plan at-runtime via its adaptive optimization component. Concerning the computing core, the query processing pipeline is implemented using the Apache Spark parallel computing framework. Spark Streaming continuously receives data from Kafka, and performs continuous SPARQL queries executions in parallel.

Figure 8.1 also serves as a map to better outline our main contributions:

- The green arrows highlight the contributions about **reasoning over concepts and properties** presented in Section 8.4: generating hierarchies of concepts and properties (Section 8.4), concept and property encoding and the corresponding query rewriting method (Section 8.4.2).
- The yellow arrows highlight the contributions about **reasoning over `sameAs` facts** presented in Section 8.5: `sameAs` clique detection (Section 8.5.1) and two alternative methods (Sections 8.5.2 and 8.5.3) for `sameAs` encoding and query rewriting.

8.3. Running Example of Continuous Reasoning Query

In this section, we present a running example that will be used all along the remaining of the chapter. The data sets provided by our Waves use case partner are proprietary

and we do not have the permission to distribute them. So a first issue concerns the selection of a benchmark/data sets which could support researchers and interested developers to replay our experimentation. Two characteristics prevent us from using well-established RSP benchmarks such as SRBnech, LSBench, and CityBench: their lack of support for the considered reasoning tasks and their inability to cope with massive RDF streams. We thus selected a benchmark with which the Semantic Web community is confident with, namely the Lehigh University Benchmark (henceforth LUBM)[36], and extended it in two directions. First, we created a stream generator based on the triples contained in the LUBM Abox. Second, we extended the LUBM generator with the ability to create individuals related by the `sameAs` property. Intuitively, novel individuals are generated and stated as being equivalent to some other LUBM individuals. This generator is configurable and one can decide how many sameAs cliques and how many individuals per clique are created.

The LUBM Tbox has not been extended and in Figure 8.2 we provide an extract of it. It contains a subset of the property hierarchy (*i.e.*, memberOf, worksOf and headOf) as well as a subset of the concept hierarchy. We will emphasize in Section 8.4 on the encoding of this extract of the Tbox. This figure also presents elements of the Abox,*i.e.*, RDF triples concerning individuals. That extract highlights the creation of individuals related by `sameAs` property, thus creating individual cliques. We have three cliques in this figure: (pDoc1, pDoc2, pDoc3), (pDoc4,pDoc5,pDoc6) and (pDoc7, pDoc8 and pDoc9). This example will be used in Section 8.5 when detailing inferences concerned with the `sameAs` property.

As shown in Q4, we have extended the standard SPARQL query language with some clauses for a continuous query processing (more details in Appendix B).

```
STREAMING { WINDOW [10 Seconds]  SLIDE [10 Seconds] BATCH [5 Seconds] }
REGISTER { QUERYID [Q1]  REASONING [U,SM]
SPARQL [ PREFIX rdf: <http://...ns#>
         PREFIX lubm: <http://..owl#>
         SELECT ?o ?n WHERE {
             ?x  rdf:type lubm:Professor;
             ?x  lubm:memberOf ?o;
             ?x  lubm:name ?n. }  }  ]
}
```

Listing 8.1: Query Q4 involving concept hierarchy inference

In Listing 8.2, we present the SPARQL part of query Q6 (*i.e.*, the streaming and register clauses are not presented since they do not provide any new information).

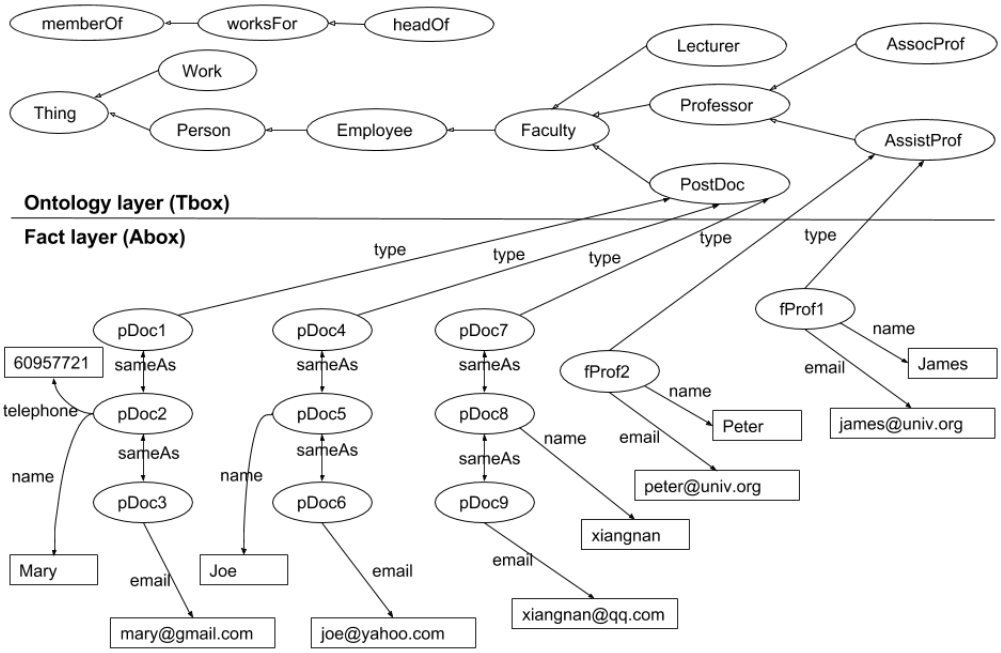


Figure 8.2.: LUBM's Tbox and Abox running example

This query retrieves names and email addresses of resources typed as PostDoc. It requires `sameAs` inferences since several individuals typed with a PostDoc concept belong to `sameAs` cliques (namely pDoc1 to pDoc9).

```
PREFIX rdf: <http://...ns#>
PREFIX lubm: <http://..owl#>
SELECT ?n ?e
WHERE {
           ?x  rdf:type  lubm:PostDoc;
           ?x  lubm:name  ?n;
           ?x  lubm:emailAddress  ?e. }
```

Listing 8.2: Query Q6 involving sameAs inference

8.4. Reasoning over concept and property hierarchies

In the following, we consider the approaches for reasoning over concept and property hierarchies. We first present the classical approach consisting in the standard query rewriting. Then, we present an extension of LiteMat reasoner, which compared to the standard approach, provides better performances in most queries.

In both approaches, encoding the elements of the Tbox, *i.e.*, concept and property hierarchies, is needed upfront to any data stream processing. The KB Encoding component encodes concepts, properties and instances of registered static KBs. This aims to provide a more compact (*i.e.*, replacing string-based IRIs or literals with integer identifiers) representation of the Tbox and Abox as well as supporting more efficient comparison operations. In the general case, each concept and property is mapped to an arbitrary unique integer identifier. We will emphasize that our LiteMat approach produces a semantic encoding scheme that supports important reasoning services. In the following, we consider inferences pertaining to the ρdf subset of RDFS (the `sameAs` property is considered in Sectio 8.5) and the input ontology is considered to be the union of (supposedly aligned) ontologies necessary to operate over one's application domain.

8.4.1. Standard rewriting: add UNION Clauses

The standard rewriting approach to perform inferences over concept and property hierarchies on SPARQL queries consists in a reformulation according to an analysis of the Tbox. Intuitively, for a query Q with BGP B. For all triples $t \in B$, the system searches in the Tbox if the property (resp. concept) in t has sub-properties (resp. sub-concepts). In the affirmative, a set of UNION clauses is appended to Q, thus producing a new query Q'. A new UNION clause contains a rewriting of B where the property (resp. concept) is replaced with a sub-property (resp. sub-concept). A UNION clause will be added for each direct and indirect sub-properties (resp. sub-concepts) and their combinations if the BGP contains several of them.

In Listing 8.3, we provide an example of this rewriting for Q4 and the extract of the LUBM ontology (see Figure 8.2). We display only six (out of the twelve, three properties times four concepts) UNION clauses present in the rewriting. Note that for each UNION clause, two joins are required.

```
SELECT ?o ?n
WHERE {{ ?x rdf:type lubm:Professor;
            memberOf ?o;
            lubm:name ?n. }
 UNION { ?x rdf:type lubm:Professor;
            worksFor ?o;
            lubm:name ?n. }
 UNION { ?x rdf:type lubm:Professor;
            headOf ?o;
            lubm:name ?n. }
 UNION { ?x rdf:type lubm:AssistantProfessor;
            memberOf ?o;
```

```
            lubm:name ?n. }
 UNION { ?x rdf:type lubm:AssistantProfessor;
            worksFor ?o;
            lubm:name ?n. }
  UNION { ?x rdf:type lubm:AssistantProfessor;
            headOf ?o;
            lubm:name ?n. }  ...  }
```

Listing 8.3: Strider's query example (Q_4)

This approach guarantees the completeness of the query result set but comes at the high cost of executing a potentially very large queries (due to an exponential increase of original query). Those constraints are not compatible with executions in a streaming environment. In the next section, we present a much more efficient approach.

8.4.2. LiteMat adapted to stream reasoning

In the following, we dedicate two subsections to our encoding scheme: one for the static KB and one for the streaming (dynamic) data. Then a rewriting dedicated to this encoding scheme is detailed.

Static encoding

Inferences drawn from properties such as `rdfs:subClassOf` and `rdfs:subPropertyOf` in Litemat, are addressed by attributing numerical identifiers to ontology terms, *i.e.*, concepts and properties. Moreover, LiteMat also supports `rdfs:domain` and `rdfs:range` using a set of addittional data structures. The compression principle of this term encoding lies in the fact that subsumption relationships are represented within the encoding of each term. This is performed by prefixing the encoding of a term with the encoding of its direct parent (a workaround using an additional data structure is proposed to support multiple inheritance). The generation of the identifiers is performed at the bit level. More precisely, the concept (resp. property) encoding are performed in a top-down manner, *i.e.*, starting from the top concept of the hierarchy (the classification is performed by a state-of-the-art reasoner, *e.g.*, HermiT[71], and hence supports all OWL2 logical concept subsumptions), such that the prefix of any given sub-concept (resp. sub-property) corresponds to its super-concept (resp. super-property). Intuitively, for the entity hierarchy (*i.e.*, concept or property), we start from a top entity and assign it the value 1 (see the raw id of `owl:Thing` in Table 8.1) and process its N direct sub-entities. These sub-entities

will be encoded over $\lceil log_2(N+1) \rceil$ bits and their identifiers will be incremented by 1. This approach is performed recursively until all entities in the TBox are assigned an identifier. It is guaranteed at the end of this first phase that, for 2 entities A and B with $B \sqsubseteq A$, the prefix of idB matches with the encoding idA. Note that for the property hierarchy, a reasoner is not needed to access the direct sub-properties. Moreover, we distinguish between object and datatype properties by assigning different starting identifiers, respectively '01' and '10'. Finally, some RDF and OWL properties, *e.g.*, `rdf:type` are assigned identifiers in the '00' range.

In a second step, to guarantee a total order among the identifiers of a given concept or property hierarchy, the lengths of these identifiers have to be normalized. This is performed by computing the size of the longest branch in each hierarchy and by encoding each identifier on this length of bits (*i.e.*, filling '0' on the right most bit positions).

These normalized identifiers are stored as integer values in our dictionary. The characteristics of this encoding scheme ensures that from any concept (resp. property) element, all its direct and indirect sub-elements can be computed with only two bit shift operations and are comprised into a discrete interval of integer values, namely its lower and upper bound (resp. LB,UB). Table 8.1 presents the identifiers of Figure 8.2's LUBM concept hierarchy extract. The first step of the encoding generates raw ids (column 1). We can observe that the `Faculty`'s prefix 110101 corresponds to the `Employee`'s identifier, and his hence one of its direct sub-concept. Moreover, `Employee` is a direct sub-concept of `Person` and indirect sub-concept of `owl:Thing`. These raw ids are normalized to produce column 2 of Table 8.1. Finally, integer values contained in the id column are stored in the dictionary.

The encoding scheme of individuals of static KBs can take two forms. It depends on whether an individual is involved in a triple with a `sameAs` property or not. The encoding scheme for `sameAs` cliques is detailed in Section 8.5.1. For non-sameAs individuals, we apply a simple method which attributes a unique integer identifier (starting from 1) to each individual. In [93], we provided an efficient distributed method to perform this encoding.

Dynamic partial encoding

In the previous section, we detailed the generation of dictionaries for the elements of static KBs, *i.e.*, their concepts, properties and individuals. These maps are being used to encode, on the fly, incoming data streams. That is concept and property IRIs of a data stream are being replaced with their respective integer identifier. The same

Raw ids	Normalized ids	id	Term
1	1000000000000	4096	`owl:Thing`
1001	1001000000000	4608	Schedule
1010	1010000000000	5120	Organization
1011	1011000000000	5632	Publication
1100	1100000000000	6144	Work
1101	1101000000000	6656	Person
110101	1101010000000	6784	Employee
110101001	1101010010000	6800	Faculty
11010100101	1101010010100	6804	Lecturer
11010100110	1101010011000	6808	PostDoc
11010100111	1101010011100	6812	Professor
1101010011101	1101010011101	6813	AssistantProf.
1101010011110	1101010011110	6814	AssociateProf.

Table 8.1.: Encoding for an extract of the concept hierarchy of the LUBM ontology

transformation is processed for individual IRIs or literals. This approach permits to drastically compress the streams without incurring high compression costs.

In practical use cases, some entries of data streams may not correspond to an entry in one of the dictionaries. For instance, due to their infinite nature, numerical values, *e.g.*, sensor measures in the IoT, can not possibly all be stored in the individual dictionary. Other cases are possible where a stream emits a message where concepts and/or properties are not present in our dictionaries. Note that such situations prevent the system from performing any reasoning tasks upon the missing ontology elements.

When facing the absence of a dictionary entry, we are opting for a partial stream encoding. Intuitively, this means that we are not trying to create a new identifier on the fly but rather decide to leave the original data as is, *i.e.*, as an IRI, literal or blank node. After some experimentations, we found out that this is good trade-off between maintaining ever growing, distributed dictionaries and favoring an increase of incoming data streams rate.

Figure8.3 provides some details on how this partial encoding is implemented into Strider$^{\mathrm{R}}$. Intuitively, it uses the Discretized Stream (DStream) abstraction of Apache Spark Streaming where each RDD is composed of a set of RDF triples. For each RDD, a transformation is performed which takes a IRI/literal based representation to a partially encoded form. This transformation lookups into the TBox and Abox dictionaries precomputed from static KBs. The bottom right table of the figure

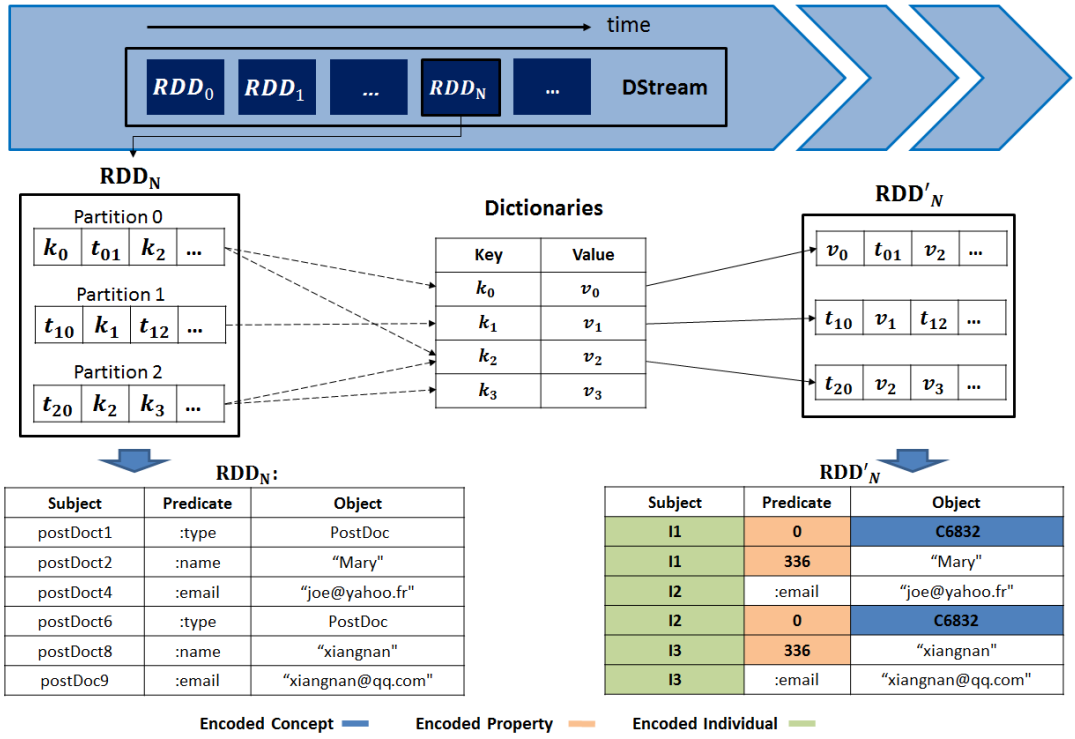


Figure 8.3.: Parallel partial encoding over DStream

emphasizes that some triple elements are encoded while some other are not. The dictionaries are broad-casted to all the machines in the cluster. The encoding for each partition of data is thus performed locally.

We briefly summarize important advantages of the partial encoding of RDF streams: (i) an efficient parallel encoding to meet real-time request; (ii) no extra overhead for dictionary generation.

Query rewriting: FILTER clauses and UDF

The query rewriting in Strider^{R} is done in the Inference Layer. Intuitively, the system parses a given SPARQL query Q and rewrites it into Q'. This rewriting concerns inferences pertaining to concept and property hierarchies. For sameAs individuals, no specific rewriting is necessary due to our data streams encoding. Due to space limitations, we do not present the rewriting of SPARQL queries into Spark SQL Scala programs but the interested reader can find details on our github page.

In Section 8.4.2, we highlighted that to each concept and property corresponds to a unique integer identifier. Moreover, one characteristic of our encoding method

guarantees that all sub-concept (resp. sub-property) identifiers of a given concept (resp. property) are included into an interval of integer values, denoted lower bound (LB) and upper bound (UB) of that ontology element.

We now concentrate on the query rewriting for concepts. In order to speed up the rewriting, we take advantage of the following context: since we are only considering that data streams are representing elements of the Abox, concepts are necessarily at the object position of a triple pattern and the property must be `rdf:type`. Intuitively, if a concept has at least one sub-concept then it is replaced in the triple pattern by a novel variable and a SPARQL `FILTER` clause is added to the query's BGP. That filter imposes that the new variable is included between the LB and UB values (which have been previously computed at encoding-time and stored in the dictionary) of that concept.

The overall approach is quite similar for the rewriting concerning the property hierarchy but no specific context applies, *i.e.*, all triple patterns have to be considered. For each triple pattern, we check whether the property has some sub-properties. If it is the case then the property is replaced by a new variable in the triple pattern and a SPARQL `FILTER` clause is introduced in the BGP. That filter clause restricts the new variable to be included in the LB and UB of that property.

As a concrete example of this rewriting, we are using query Q4 of our benchmark since it requires inferences over both the concept and property hierarchies.

The rewriting Q4' of Q4 contains two `FILTER` clauses, one for the `Professor` concept and one for the `memberOf` property (LB() and UB() functions respectively return the LB and UB of their parameter). Given p, the parameter submitted to LB and UB, these functions respectively return the identifier of p and an identifier computed using two bit shift operations on p. So, the computation of the UB function is quite fast. Note the introduction of the $?p$ and $?m$ variables, respectively replacing the `Professor` concept and `memberOf` property. The lower and upper bounds for $?p$ correspond to the identifier of the Professor and AssociateProfessor, respectively (equal to 6812 and 6814 in Table 8.1).

```
SELECT ?o ?n
WHERE { ?x  rdf:type ?p;  ?x  ?m  ?o;  ?x  lubm:name  ?n.
        FILTER (?p>=LB(Professor)  &&  ?p<UB(Professor)).
        FILTER (?m>=LB(memberOf)  &&  ?m<UB(memberOf)).}
```

Listing 8.4: LiteMat query rewriting for query Q4

Finally, this rewriting is much more compact and efficient than the classical reformulation which would require twelve `UNION` clauses and twenty four joins[1].

8.5. Reasoning with the `sameAs` property

This section concerns inferences performed in the presence of triples containing the `sameAs` property. In Section 8.5.1 a distributed, parallelized approach to encode `sameAs` cliques and a naive approach to materialize inferred triples are proposed. We emphasize that this approach is not adapted to a streaming context. Therefore, the challenge is to support `sameAs` reasoning efficiently while answering queries over streaming data. In Strider^{R}, we address this challenge and propose two solutions. The first one (Section 8.5.2) aims for efficiency, the second one (Section 8.5.3) aims to handle reasoning applied to a "provenance awareness" scenario which is not supported by the first solution.

8.5.1. SameAs clique encoding

Consider, in a KB, a set of `sameAs` triples that represents a graph denoted G_{sa}. The nodes V of G_{sa} are set of individuals (either at the subject or object position of a triple) of the `sameAs` triples. Let $x \in V$ denote such a node. For convenience, every node $x \in V$ is uniquely identified by an integer value $Id(x) \in [1, |V|]$.

Let C be the set of connected components that partition G_{sa}. Although a component may not be fully connected, it represents a `sameAs` clique because of the semantic equivalence (*i.e.*, transitivity and symmetry) of the `sameAs` property. Let C_i denote the connected component containing the node identified by i.

In Strider^{R}, we assume that the `sameAs` triples are in the static KB. We detect the C_i using a parallel algorithm to compute connected components [94].

The principle of that algorithm is to propagate through the graph a numeric value representing a component id, such that every connected component will end up with a component id assigned to its members.

Initially, each node x is assigned with $Id(x)$. Then for each node x, the group that comprises x and its neighbors is considered and the minimum number among the group members is assigned to all the group members. The algorithm ends when no more update occurs at any group, *i.e.*, for any group (or connected component) all the members share the same component id. About the computational cost of clique detection, note that it is computed in a distributed and parallel manner (using the

[1]Rewriting available on our github page

GraphX library of the Apache Spark engine). Hence it is able to scale to very large static KBs.

Once the connected components are detected, we define the $Cl(x)$ mapping that associates the IRI of x with its clique id. Table 8.2 summarizes the notations used in this section.

Notation	Description
G_{sa}	The `sameAs` graph of individuals
$Id(x)$	The integer ID of individual x
C_i	The clique st. i is the minimal ID among the members
$Cl(x)$	The clique ID of individual x.
$IRI(i)$	The IRI of ID i (or set of IRIs if i is in a clique)
S	The average size of a clique

Table 8.2.: Notations used for `sameAs` reasoning

In order to reason over `sameAs`, an obvious solution is to materialize all inferences. However that is not tractable because the number of inferred triples is far too high in general. Consider a triple $t = (x, p, y)$ where x (resp. y) belongs to the `sameAs` clique C_x (resp. C_y). Let S be the average size of a clique. The number of triples inferred from t is $2 \times S^2$. Therefore, the number of triples inferred from the entire dataset D (of size $|D|$) can grow up to $2 \times |D| \times S^2$ in the worst case. For instance, from Figure 8.2 we obtain Figure 8.4(a) where all dashed edges correspond to a materialization of all inferences induced by `sameAs` explicit triples. We can easily witness the increase of represented triples. This naïve approach is generally not adopted in RDF database systems storing relatively static datasets due to its ineffectiveness. This is even more relevant in a dynamic, streaming context. First, it may not be feasible to generate all materialization within the time constraint of a window execution. Second, the execution of a continuous query over such stream sets would be inefficient.

8.5.2. Representative-based (RB) reasoning

Based on the detected cliques, the principle of the representative-based (RB) reasoning is to fuse the dataset such that all the individuals that belong to the same clique appear as a single (representative) individual. As a consequence, the fused graph implicitly carries the `sameAs` semantics. Then, a regular evaluation of any query on the fused graph is guaranteed to comply with the `sameAs` semantics.

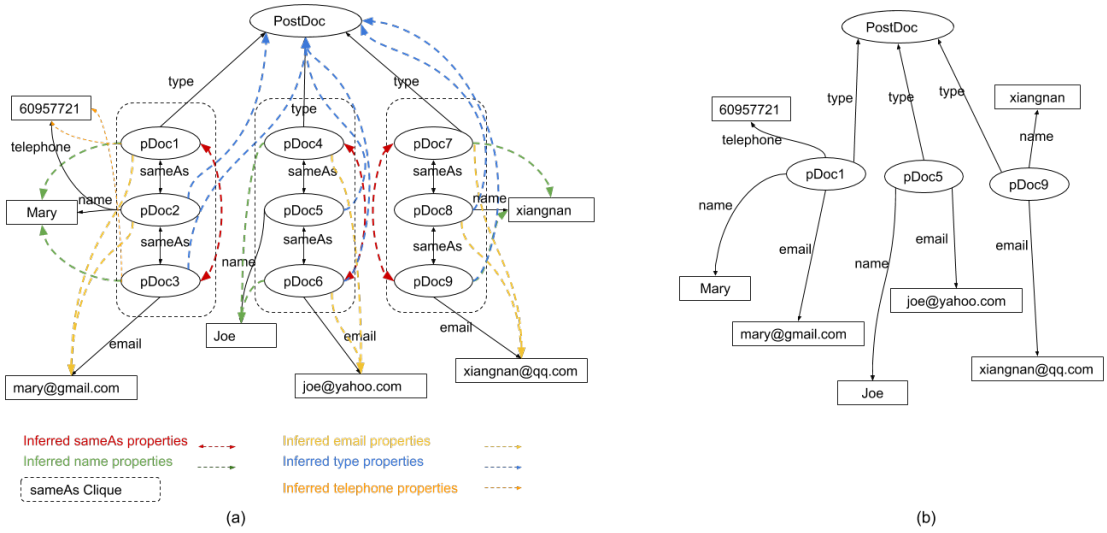


Figure 8.4.: `sameAs` representation solutions

Stream encoding

Stream encoding according to `sameAs` individuals consists of two steps:

1. First, select a single individual per clique C_i, which acts as the clique *representative*. The representative can be any member, as long as there is only one representative per clique. Without loss of generality, we assume that the representative for C_i is the member whose node number equals to i. Therefore, given the IRI x of any individual, its representative is numbered $Cl(x)$. In Figure 8.4(a) shows three cliques in dotted boxes, and $pDoc1$ individual can serve as the representative of the clique $(pDoc1, pDoc2, pDoc3)$.

2. Second, encode the input stream: replace every (x, p, y) triple by its corresponding representative-based triple: $(Cl(x), p, Cl(y))$. Fig. 8.4(b) shows the result of the encoding where individuals $pDoc1$, $pDoc5$ and $pDoc9$ are the so-called representatives of the cliques.

This approach has many advantages, especially in a data streaming context:

- (i) The inferred graph is more compact without loss of information from the original graph.

- (ii) The dictionary data structure that implements $Cl(x)$ is light. The dictionary size equals to the size of G_{sa} which is in practice very small compared to the

number of triples to process in a streaming window. The computing overhead of encoding the input stream is negligible.

- (iii) Clique updates (*e.g.*, removing or adding an individual from a clique) does not imply to update the input stream since the data streams are ephemeral. This assumes that an update of the static KB is taken into account starting from the next window that only contains data produced after the update. For instance, consider the clique named C_1 with 3 members ($pDoc1, pDoc2, pDoc3$). At time t, the KB is updated: $pDoc4$ is declared to be `sameAs` $pDoc2$ thus $pDoc4$ joins C_1. The data already streamed before t are not updated, *i.e.*, the triples mentioning $pDoc4$ are not updated. Whereas, in the window following t, $pDoc4$ will be translated to $Cl(pDoc4)$.

Query processing

Based on the above encoding, a standard query processing is performed where variable bindings concern both standard individuals and `sameAs` representative.

Note that because the `sameAs` reasoning is fully supported by the representative-based encoding, we can simplify the query by removing the `sameAs` triple patterns that it may contain.

To evaluate a filter clause that refers to an IRI value, *e.g.*, `FILTER {?x like '*w3c.org*'}`, we rewrite it into an expression that refers back to the IRI value(s) instead of the encoded identifier. Let define $IRI(x)$ as the IRI (or the set of IRIs in case of a clique) associated with encoded ID x. Let $f(x)$ be a `FILTER` condition on variable $?x$, $f(x)$ is then rewritten into $\{\exists e \in IRI(x) | f(e)\}$.

A final step decodes the bindings: each encoded value is translated to its respective IRI or literal value. If the encoded value is a clique number, then it translates to the IRI of the clique representative.

8.5.3. SAM reasoning

SAM stands for SAM for `sameAs` Materialization and aims to handle reasoning in the case of origin-preserving (or provenance-awareness) scenario which are not supported by the RB solution introduced above (Section 8.5.2).

SAM reasoning targets the use cases that require to make the distinction between the *original* dataset triples and the *inferred* triples. That distinction is necessary for a user investigating which part (or domain) of the dataset contributes to the query, *i.e.*, brings some piece of knowledge, when the IRIs within a clique have different

domains. In this section, we begin by motivating SAM using a concrete example then we detail a method to evaluate queries in this setting.

Motivation

We briefly sketch an example showing the limitations of the RB approach and the need for the proposed SAM approach. For instance, consider the dataset of Figure 8.2. Suppose all the triples about email addresses come from domain1 (*e.g.*, mail.univ.edu), then the IRIs $pDoc3$, $pDoc6$, and $pDoc9$ are in that domain. Similarly, suppose all telephone numbers come from domain2 (*e.g.*, phone.com), then $pDoc2$ is in domain2. Let consider a query searching the IRI and the email of a person named Mary. We could write that query Q as follows:

```
Q: SELECT ?x, ?y
     WHERE {
           ?x   name  "Mary" .
           ?x   email   ?y.  }
```

The result is $?x = pDoc2$ and $?y = mary@gmail.com$. Based on that result and on the clique membership information, we know that the possible bindings for $?x$ are also $pDoc1$ or $pDoc3$. However the result does not inform us that $pDoc3$ as well as its $domain1$ were originally concerning the `email` triple. To get this provenance information, we could write the query as Q':

```
Q': SELECT ?x1, ?y
      WHERE {
           ?x  name  "Mary" .
           ?x  sameAs  ?x1 .
           ?x1  email  ?y . }
```

However, through the RB approach, the result is still $?x1 = pDoc2$ and $?y = mary@gmail.com$ because $pDoc2$ is the representative of $pDoc3$. The goal of the SAM approach is to make Q' return the binding $?x1 = pDoc3$ instead of $?x1 = pDoc2$. Doing this way, we will get that the IRI $pDoc3$ and $domain1$ directly relate to the email information within the dataset. To sum up, the RB approach (8.5.2) does not support the origin-preserving use case, because a query result only binds to individuals that are clique representatives or not member of a clique at all. The result lacks information about which IRI originally exists in the triples that match the query.

To overcome this drawback, we propose the SAM approach that keeps track of the individuals that match the query even if they are part of a `sameAs` clique. The principle of the SAM approach is to **explicitly handle the `sameAs` equivalence** such that the equivalent individuals that match a query are preserved in the query result. From a logical point of view, this means to manage explicit `sameAs` information both in the dataset and in the query.

- `sameAs` in the dataset: complete the input stream with explicit information representing the `sameAs` equivalences between IRIs.
- `sameAs` in the query: complete the query with triple patterns explicitly expressing the `sameAs` matching.

lue

Materialize `sameAs` data streams

A general method consists in completing the input stream with `sameAs` information. We devise an efficient solution that guarantees to materialize only the necessary triples and prevents from exploding the size of the data stream. Moreover, the IRIs are encoded to get a more concise representation to save on query execution time.

We now detail the steps of our solution. Let W be the current streaming window. The idea is to express each clique that has at least one member in W by a minimal set of triples. Let $\mathcal{C}$ be the set of cliques used in W and a clique $C_i \in \mathcal{C}$. For each member x of C_i such that $Id(x) \neq i$ (the identifier $Id(x)$ is defined in Table 8.2), add the triple $< i$ `sameAs` $Id(x) >$ into W. For instance, consider the three `sameAs` cliques of Figure 8.4(a). Let denote C_1 the clique containing $(pDoc1, pDoc2, pDoc3)$, the minimal Id in C_1 is $Id(pDoc1) = 1$. Suppose the input stream window contains:

```
pDoc1 type PostDoc.
pDoc2 name "Mary".
pDoc3 emailAddress "mary@gmail.com".
```

While applying the SAM approach, the window is completed with only two triples:

```
1 sameAs Id(pDoc2)
1 sameAs Id(pDoc3)
```

Let S be the average clique size. Notice that only $S - 1$ edges of C_i out of S^2 (those with subject i) are added into the stream. The added `sameAs` edges represent

a directed star centered at i the member of C_i with minimal Id. As explained below in Section 8.5.3, that light materialization is sufficient to fully enable the `sameAs` reasoning during query processing.

Cost analysis. We analyze the materialization cost in terms of data size. The total amount of materialized triples in W is $|\mathcal{C}| \times (S-1)$. The space overhead of SAM is indeed far smaller than a full materialization of every triple inferred from the `sameAs` reasoning which would add $2 \times |W| \times S^2$ triples (Table 8.2). Moreover, our solution materializes S times less triples compared to materializing all the clique edges. This low memory footprint makes our solution more scalable.

Query rewriting: add `sameAs` patterns

Consider a BGP query represented by a graph of triple patterns where nodes are variables, IRIs or literals. In a query, a join node is a variable that connects at least two triple patterns. The query rewriting method consists in extending a BGP query with `sameAs` patterns that could match the materialized stream. The principle is to"inject" the `sameAs` semantics into each join appearing in the query, *i.e.*, to decompose a **direct** join on one variable into an **indirect** join through a path of two `sameAs` triple patterns. Consequently, each join is decomposed into three join operations. Intuitively, the join nodes of the BGP are split to be replaced by a star of `sameAs` triple patterns such that the join"traverses" the star center.

The shape of the added `sameAs` patterns is a star because it has to match the stars `sameAs` triples that have been materialized into the stream. We consistently adopt a star-shaped representation of the `sameAs` information both in the materialized stream and in the query pattern. This guarantees any `sameAs` relation within a clique to be expressed by a path of length two (the diameter of a star). Besides, a star triple pattern is guaranteed to match any path of length two. Therefore, our proposed rewriting is guaranteed to match any `sameAs` path within the stream, *i.e.*, the rewritten query is semantically equivalent to the initial one.

We next detail the query rewriting algorithm. Let V be the set of join variables of a query. For each $v \in V$, (i) *Split the join variable*: replace each occurrence of v in the query by a distinct variable. Let $v_1, \cdots, v_n$ denote the variables replacing the n occurrences of v. (ii) *Express the indirect join*: For each v_i add the ($?v$ `sameAs` $?v_i$) triple pattern.

For example, consider the following query Q6 and its graphical representation shown in Figure 8.5:

```
SELECT ?x
WHERE {
        ?x  type  PostDoc.
        ?x  name  ?n.
        ?x  emailAddress  ?y. }
```

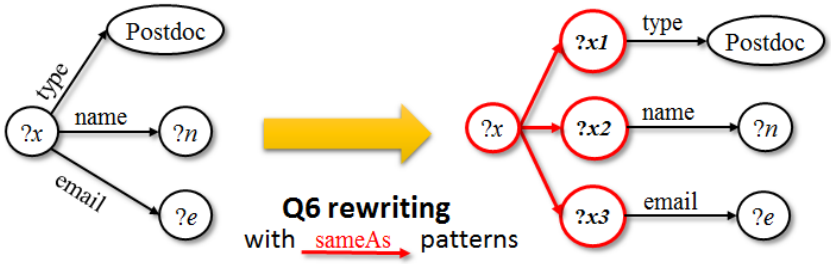


Figure 8.5.: SAM rewriting for the Q_6 query

The $?x$ join variable is split into $?x_1, ?x_2, ?x_3$ and these new variables are connected through `sameAs` patterns. The rewritten equivalent query is:

```
SELECT ?x, ?x1, ?x2, ?x3
WHERE {
?x1 type PostDoc.     ?x sameAs ?x1.
?x2 name ?n.          ?x sameAs ?x2.
?x3 emailAddress ?y.  ?x sameAs ?x3. }
```

Another example, on Figure 8.6, shows the rewriting case of a join variable that appears both as an object and as a subject position.

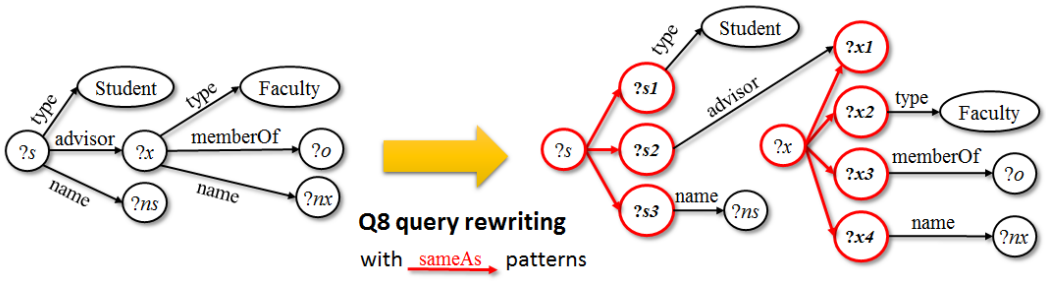


Figure 8.6.: SAM rewriting for Q_8 query

Query evaluation: join with `sameAs` patterns

Evaluating a rewritten `sameAs` query requires special attention in order to ensure that the result is complete. Our SAM approach minimizes the amount of materialized `sameAs` triples for better efficiency. Thus, the dataset does not contain any `sameAs` reflexive triple x `sameAs` x. Remind that our solution aims to bring `sameAs` reasoning capability to query engines that do not support `sameAs` reasoning natively. Such query engine do not infer x `sameAs` x for any individual. Therefore, a regular evaluation of a `sameAs` query may lead to incomplete result. For instance, let us remind the example dataset of § 8.5.3 including the materialized `sameAs` triples:

```
Id(pDoc1) type PostDoc
Id(pDoc2) name "Mary"
Id(pDoc3) email "mary@gmail.com"
Id(pDoc1) sameAs Id(pDoc2)
Id(pDoc1) sameAs Id(pDoc3)
```

Consider the query:

```
SELECT ?x, ?x1, ?x2
WHERE {
     ?x sameAs ?x1.  ?x1 type PostDoc.
     ?x sameAs ?x2.  ?x2 name ?n.
}
```

That query result is empty because the triple pattern $?x$ `sameAs` $?x_1$ does not bind to $?x = Id(pDoc1)$ and $?x_1 = Id(pDoc1)$ due to the absence of the reflexive `sameAs` triple in the dataset.

To overcome this limitation, while keeping the materialized data as small as possible, we devise an extended query evaluation process. The idea is to take into account the implicit reflexive `sameAs` triples while joining a non-`sameAs` triple pattern with a `sameAs` one in order to ensure that the result is complete. The key phases of the query evaluation are:

(i) Decomposition. A query containing n non-`sameAs` triple patterns is decomposed into n chains (or sub-queries). A chain contains one non-`sameAs` triple pattern and the `sameAs` patterns it is joined to. A chain has exactly one non-`sameAs` triple pattern and at most 2 `sameAs` triple patterns. For example, the decomposition for query Q_8 in Section 8.6 has 6 chains, among which a chain of length 2 is `?x sameAs`

`?x3. ?x3 memberOf ?o` and a chain of length 3 is `?s sameAs ?s2. ?s2 advisor ?x1. ?x sameAs ?x1`.

(ii) Planning. Based on the chains that somehow hide the `sameAs` patterns, the query planner assesses a join order and generates an execution plan as usual (ignoring the `sameAs` patterns).

(iii) Execution. During the query execution phase, if a chain contains a `sameAs` pattern then a dedicated operator ensures that all the bindings are produced. More precisely, to execute the chain `?x sameAs?y. ?y p ?z` consists in evaluating the triple pattern `?y p ?z` which results in a set of $(?y, ?z)$ bindings. Then, for each binding, produce a set of $(?x, ?y, ?z)$ bindings such that $?x$ binds to the $?y$ value and also to each individual equivalent to $?y$ value. This ensures a complete result.

8.6. Evaluation

Putting together the contributions presented in Sections 8.4 and 8.5, we are able to combine LiteMat with one of the two methods to reason over `sameAs` individuals, denoted RB (representative-based) and SAM (SameAs Materialization). It thus defines two forms of reasoners for RDFS with `sameAs`:

- the LiteMat + RB approach is, in most use cases, the best performing approach and is hence the default approach.
- the LiteMat + SAM provides additional features, *e.g.*, a need for origin-preserving scenario, and improves the processing performance of BGPs containing a single triple pattern with inference.

8.6.1. Computing Setup

We evaluate Strider^{R} on an Amazon EC2/EMR cluster of 11 machines (type m3.xlarge) and manage resources with Yarn. Each machine has 4 CPU virtual cores of 2.6 GHz Intel Xeon E5-2670, 15 GB RAM, 80 GB SSD, and 500 MB/s bandwidth. The cluster consists of 2 nodes for data flow management via the Kafka broker (version 0.8.x) and Zookeeper (version 3.5.x)[89], 9 nodes for Spark cluster (1 master, 8 workers, 16 executors). We use Apache Spark 2.0.2, Scala 2.11.7 and Java 8 in our experiment. The number of partitions for message topic is 16, generated stream rate is around 200,000 triples/second.

8.6.2. Datasets, Queries and Performance metrics

As explained in Section 8.3, we can not use any existing RSP benchmarks to evaluate the performances of Strider$^{\mathrm{R}}$. Hence, we are using our LUBM-based stream generator configured with 10 universities, *i.e.*, 1.4 million triples. For the purpose of our experimentation, we extended LUBM with triples containing the `sameAs` property. This extension requires to set two parameters: the number of cliques in a dataset and the number of distinct individuals per clique. To define these parameters realistically, we ran an evaluation over different LOD datasets. The results are presented in Table 8.3. It highlights that although the number of cliques can be very large (over a million in Yago), the number of individuals per clique is rather low, *i.e.*, a couple of individuals. Given the size of our dataset, we will run most of our experimentations with 1,000 cliques and an average of 10 individuals per clique, denoted 1k-10. Nevertheless, on queries requiring this form of reasoning, we will stress Strider$^{\mathrm{R}}$ with up to 5,000 cliques and an average of 100 individuals per clique (see Fig.**??** for more details). More precisely, we will experiment with the following configurations: 1k-10, 2k-10, 5k-10, 1k-25, 1k-50 and 1k-100. For the SAM approach, the number of materialized triples can be computed by $nc * ipc$ with nc the number of cliques and ipc the number of individuals per clique.

We have defined a set of 8 queries[2] to run our evaluation (see Appendix for details). Queries Q1 to Q5 are limited to concept or/and property subsumption reasoning tasks. Query Q6 implies sameAs only inferences while Q7 and Q8 mix subsumptions and sameAs inferences.

Finally, we need to define which dimensions we want to evaluate. According to *Benchmarking Streaming Computation Engines at Yahoo!*[3], a recent benchmark for modern distributed stream processing framework, we take system throughput and query latency as two performance metrics. In this chapter, throughput refers to how many triples can be processed in a unit of time (*e.g.*, triples per second). Latency indicates the time consumed by an RSP engine between the arrival of the input and the generation of its output. More precisely, for a windowing buffer w_i of the i-th query execution containing N triples and executed in t_i, then $throughput = \frac{N}{t_i}$ and the $latency = t_i$.

[2] https://github.com/renxiangnan/strider/wiki

[3] https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

8.6.3. Quantifying joins and unions over reasoning approaches

As stated before, we can not compare Strider^{R} to other available RSP systems. This is mainly due to the high stream rate generated of our experiment which can not be supported by state-of-the-art reasoning-enabled RSPs, *e.g.*, C-SPARQL and SparqlStream. This is probably due to the lack of data flow management scalability of in these RSPs. In fact, their design was not intended for large-scale streaming data processing. Moreover, RSP system that could handle such rate either do not support reasoning or are not open source, *e.g.*, CQELS-cloud.

To assess the performance benefit of our solution for processing complex queries, specially comprising many joins, we compare LiteMat + RB and Lite + SAM with a more classical query rewriting approach. This combines SAM with UNION clauses between combinations of BGP reformulation (this approach is henceforth denoted UNION + SAM). Notice that the UNION + SAM approach acts as a baseline for our experiments. Such a rewriting comes at the cost of increasing the number of joins. Table 8.4 sums up the join and union operations involved in the 8 queries of our experimentation. In particular, queries Q5, Q7 and Q8 present an important number of joins (resp. 90, 45 and 180) due to a large number of union clauses (resp. 17, 14, 29).

datasets	#triples	#sameAs cliques	max	avg
Yago*	3696623	3696622	2	2
Drugbank	4215954	7678	2	2
Biomodels	2650964	187764	2	1.95
SGD	14617696	15235	8	3
OMIM	9496062	22392	2	2

Table 8.3.: SamesAs statistics on LOD datasets (ipc = number of distinct individuals per sameAs clique, max and avg denotes resp. the maximum and average of ipc,*: subsets containing only `sameAs` triples with DBpedia, Biomodels contains triples of the form a `sameAs` a

8.6.4. Results evaluation & Discussion

The window size for involved continuous SPARQL queries with LiteMat reasoning support is set to 10 seconds, which is large enough to hold all the data generated from the dataset. However, since the impacts of extra data volume and more complex overheads are introduced in SAM query processing, we have to increase the window size (up to 60 seconds) to ensure that both LiteMat and SAM approaches return

Queries	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
# Joins								
LMRB	1	4	0	2	5	2	2	5
USAM	7	84	0	42	210	2	120	420
USAM*	3	24	0	18	90	2	60	210
# Union keywords								
USAM	6	20	3	20	41	0	29	59
USAM*	2	5	2	8	17	0	14	29
# Filter clauses								
LMRB	1	2	1	2	3	0	2	3

Table 8.4.: Number of joins, unions and filter per query for LiteMat + RB (LMRB) and UNION + SAM (USAM) approaches. Here, the number of UNIONs correspond to the number of UNION keywords. USAM* relies on a simplified LUBM ontology

the same result. In a nutshell, we approximately adjust the window size and the incoming stream rate by checking the materialized data volume.

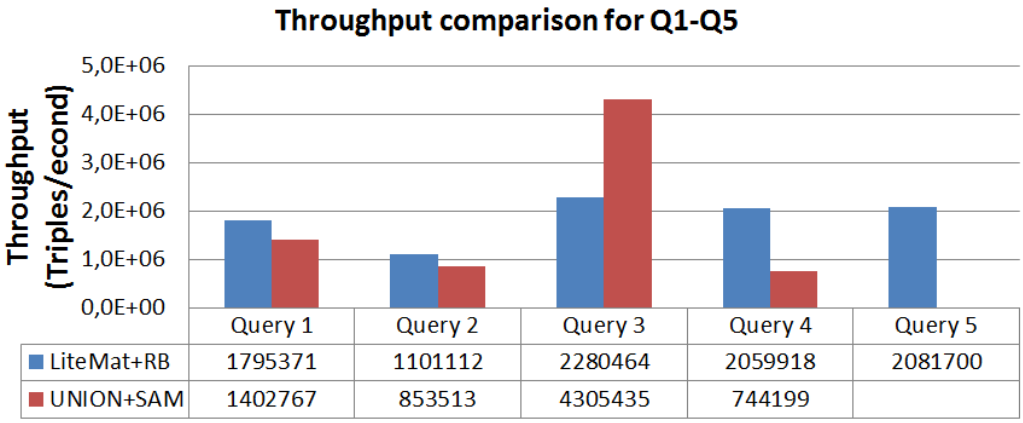
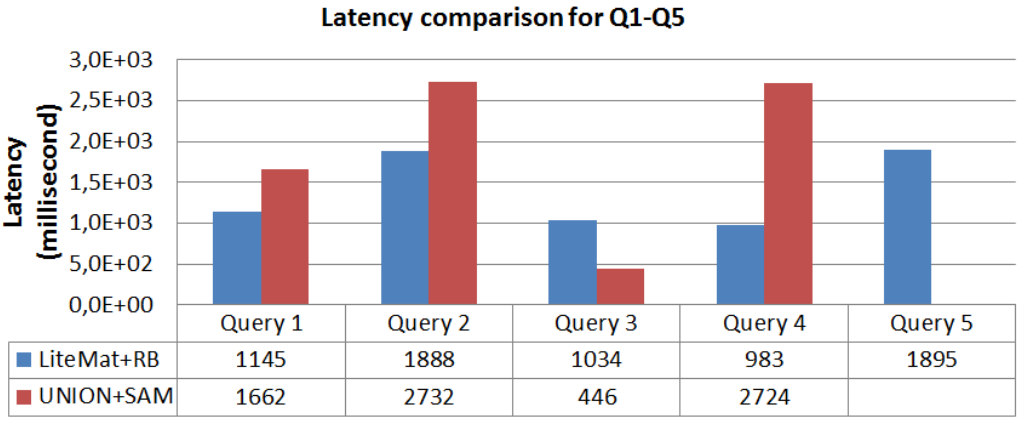


Figure 8.7.: Throughput Comparison between LiteMat+RB and UNION+SAM for **Q1** to **Q5**

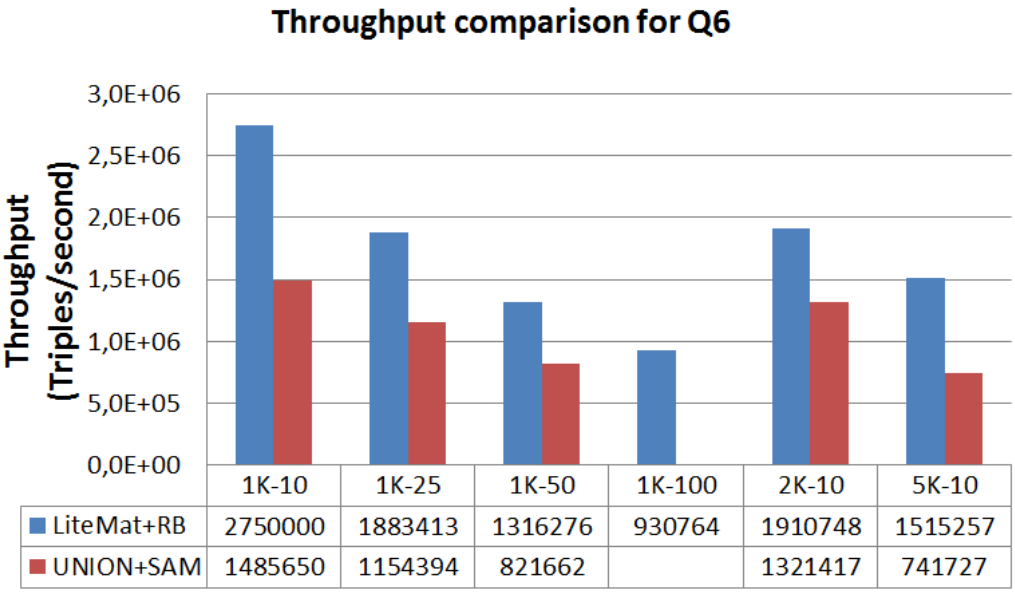


	Query 1	Query 2	Query 3	Query 4	Query 5
LiteMat+RB	1145	1888	1034	983	1895
UNION+SAM	1662	2732	446	2724	

Figure 8.8.: Latency Comparison between LiteMat+RB and UNION+SAM for **Q1** to **Q5**

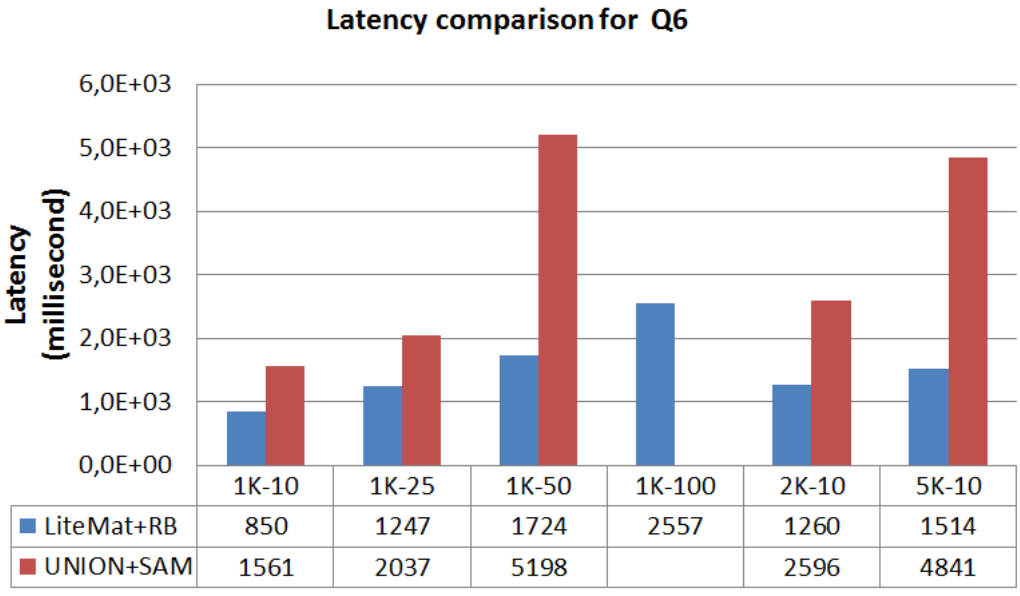
All the evaluation results include the cost of LiteMat encoding and, for the LiteMat + SAM solution, the cost of `sameAs` triple materialization. Figure 8.7, 8.8 reports the throughput and query latency of Q1 to Q5. Reasoning LiteMat + RB achieves the highest throughput (up to 2 millions triples/seconds) and the query latency remains at the second-level. To the best of our knowledge, such performances have not been achieved by any existing RSP engines. When both original and rewritten query patterns are relatively simple, *e.g.*, Q1 and Q2, LiteMat + RB has 30% gain over UNION+SAM on throughput and latency. The improvement gain of LiteMat + RB over UNION + SAM is increasing for queries involving multiple inferences, *i.e.*, Q4 is 75% faster. For Q5, UNION + SAM does not even terminate. This is mainly due to the insufficient computing resources (*e.g.*, number of CPU cores, memories) on Spark driver nodes, which is not capable of handling such intensive overheads for jobs/tasks scheduling and memory management.

Nevertheless, UNION + SAM is more efficient than LiteMat + RB on Q3 which contains a single triple pattern needing to reason over a property hierarchy of length two. This is due to the automatic parallelism provided by Spark on the execution of the `UNION` queries, thus benefiting from a good usage of cluster resources. With its `FILTER` clause, LiteMat + RB on Q5 does not benefit from such a parallel execution. Under this circumstance, a filter operator with numeric range determination seems to be more costly than the union of three selections. In fact, a filter operator to evaluate a range predicate (LiteMat + RB) is longer to process than evaluating three equality predicates (UNION + SAM).



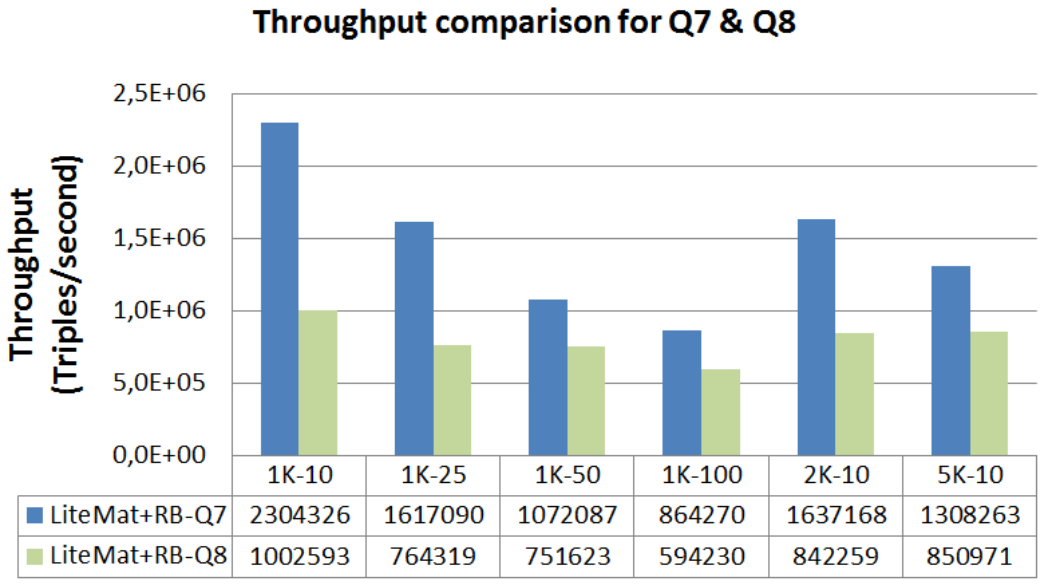
	1K-10	1K-25	1K-50	1K-100	2K-10	5K-10
LiteMat+RB	2750000	1883413	1316276	930764	1910748	1515257
UNION+SAM	1485650	1154394	821662		1321417	741727

Figure 8.9.: Throughput Comparison between LiteMat+RB and UNION+SAM for **Q6** by varying the size of clique.



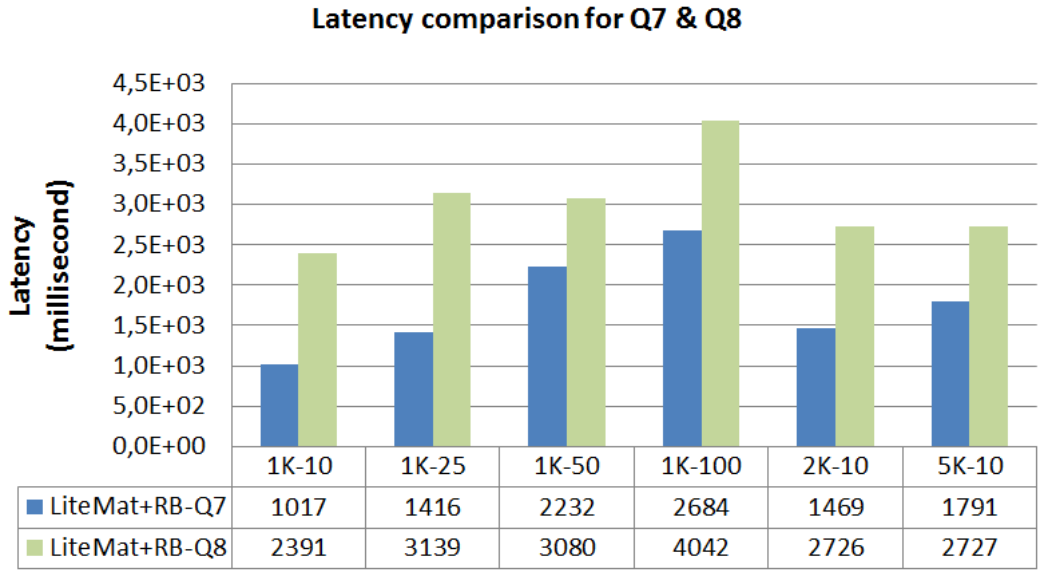
	1K-10	1K-25	1K-50	1K-100	2K-10	5K-10
LiteMat+RB	850	1247	1724	2557	1260	1514
UNION+SAM	1561	2037	5198		2596	4841

Figure 8.10.: Latency Comparison between LiteMat+RB and UNION+SAM for **Q6** by varying the size of clique.



	1K-10	1K-25	1K-50	1K-100	2K-10	5K-10
LiteMat+RB-Q7	2304326	1617090	1072087	864270	1637168	1308263
LiteMat+RB-Q8	1002593	764319	751623	594230	842259	850971

Figure 8.11.: Throughput Comparison between LiteMat+RB and UNION+SAM for **Q7, Q8** by varying the size of clique.



	1K-10	1K-25	1K-50	1K-100	2K-10	5K-10
LiteMat+RB-Q7	1017	1416	2232	2684	1469	1791
LiteMat+RB-Q8	2391	3139	3080	4042	2726	2727

Figure 8.12.: Latency Comparison between LiteMat+RB and UNION+SAM for **Q7, Q8** by varying the size of clique.

Figures 8.9, 8.10, 8.11 and 8.12 illustrate the impact on engine throughput and latency of Q6 to Q8 with varying `sameAs` clique sizes. As noted previously, "1K-10" means 1,000 cliques, and 10 individuals per clique. The number of materialized triples for `sameAs` reasoning support follows the $nc * ipc$ formula presented in Section 8.6.2. The number of materialized triples obviously increases with greater number of cliques and/or number of individuals per clique. The data throughput and latency can only be compared on Q6 since on Q7 and Q8, LiteMat + SAM does not terminate. The same non termination issue than on Q5 is observed (Q7 and Q8 respectively have 60 joins and 210 joins). Although stream rate is controlled at a low level, the system quickly fails after the query execution is triggered.

Data throughput and latency is always better for LiteMat + RB than LiteMat + SAM by up to respectively two and three times. For the same computing setting, when the number of individuals per clique increases for a given number of cliques or when the number of cliques increases for the same number of individuals per clique, the performances of the LiteMat approaches decrease.

The same evolution for LiteMat + RB is witnessed on the more complex Q7 and Q8 queries. Nevertheless, for these queries, a throughput of over 800,000 triples per second can be achieved. Given our computing setting of 11 machines, this is still a major breakthrough compared to existing RSP engines. Moreover these systems are currently not able to support important constructors such as `sameAs`.

8.6.5. Cost analysis of the SAM approach

The SAM approach allows for retrieving the specific members of a clique that match the original dataset. As explained in Section 8.5.3, we propose an evaluation strategy that efficiently generates the result of a join operation between a non-`sameAs` pattern and its adjacent `sameAs` patterns, without actually processing the join. For example, SAM only executes 5 out of 12 joins for query Q_8, the result of each of the remaining 7 joins is directly obtained from the clique metadata, (see $Cl(x)$ in Table 8.2). The SAM approach implies to customize the query engine and add the specific logic for joining `sameAs` triple patterns. Therefore, SAM only suits to extensible query processors and prevents a 'black box' SPARQL processors from being used.

Due to the extensible Spark APIs, it is possible to implement such an approach. It would more difficult to obtain such a behavior with an out-of-the box SPARQL query processor.

8.7. Conclusion

In this chapter, we have presented the integration of several reasoning approaches for RDFS plus `sameAs` within our Strider RSP engine. For most queries, LiteMat together with the representative-based (RB) approach for `sameAs` cliques is the most efficient. Nevertheless, LiteMat + SAM proposes an unprecedented provenance-awareness feature that can not be obtained in other approaches. Lite + SAM can also be useful for very simple queries, *e.g.*, a single triple patter in the WHERE clause.

To the best of our knowledge, this is the first scalable, production-ready RSP system to support such an ontology expressiveness. Via a thorough evaluation, we have demonstrated the pertinence of our system to reason with low latency over high throughput data streams. One of the limitations of our system corresponds to the potential large memory footprint of the generated dictionaries. Comparing to conventional stream reasoning approach, in the case of small workload or simplistic query, Strider^{R} does not have a definitive advantage.

As future work, we consider adding the support for the ontology which contains cycle and multiple hierarchies. We will also investigate novel semantic partitioning solutions. This could be applied to elements such as dictionaries, streaming data and continuous queries. We are aiming to support data streams that would update the ontology and thus our dictionaries. An improvement of the FILTER operator in Strider^{R} is also in the scope of consideration. Finally, we are also working on increasing the expressiveness of supported ontologies, *e.g.*, including transitive properties.

9. BigSR: An Empirical Study of Real-time Expressive RDF Stream Reasoning on Modern Big Data Platforms

9.1. Introduction

In this chapter, we study the feasibility of applying modern Big Data technique to support real-time expressive RDF stream reasoning. We have conducted a series of experiments based on benchmarking different streaming model and parallelism levels. To do so, we implement a reusable distributed RDF stream reasoning prototype, namely BigSR to support our evaluations.

Thus our system addresses important problems that are being met frequently in modern applications. For instance, projects like Waves, SEAS[1] (European ITEA2), Optique[2] (European FP7) and many others require processing data streams with rich semantics in close to real time. At the same time, industrial systems based on Datalog (Logiblox[78], Yedalog [95], datomic[3]) are emerging. It hence makes sense to mix these two features (*i.e.*, close to real-time stream processing and rule-based reasoning) in a single framework to fulfill an emerging kind of systems.

Some available stream reasoning systems like StreamRule [72], Ticker [76, 96] and Laser [75] have opted for a centralized design to benefit from existing ASP solver such as Clingo [74]. Their scalability is hence limited by single machine/process and thus can not scale. Actually, high expressive queries often involve recursion or complex temporal logic operators, which are considered as the main performance bottleneck for stream reasoning. Additionally, the optimization tailored for static query evaluation *e.g.*, data indexing, data preprocessing, neither meet the real-time nor the defined

[1] https://www.the-smart-energy.com
[2] http://optique-project.eu/
[3] www.datomic.com

temporal logic requirements. Finally, distributed environments often adopt a shared-nothing architecture where the memory of different data partitions are isolated, thus preventing some advanced optimization for Datalog program materialization [97] to be applied.

This chapter first introduces our BigSR prototype which addresses distributed computing techniques on expressive stream reasoning. This system possesses two broad classes of streaming models: Bulk Synchronous Parallel (BSP) and Record-at-A-Time(RAT) which are respectively implemented using Apache Spark Streaming and Apache Flink. The adoption of these two models is motivated by different needs from real-world use cases, *e.g.*, ASP programs with or without recursion, the capacity to support multiple streaming windows and constraints on acceptable processing latency.

9.2. Stream Reasoning with LARS in BSP and RAT models

Recall the definitions given in 3.5.2 (in the Background knowledge chapter), we use LARS as the theoretical foundation. LARS is a rule-based logical framework defined as an extension of Answer Set Programming (ASP) which we are using as a theoretical foundation. Ticker and Laser[75] are recent systems also based on LARS. In this section, we recall some basic definition of LARS, and we describe the general methodologies to parallelize the evaluation of Datalog programs and their relations to LARS. Finally, we reformulate the streaming models on Spark and Flink (*i.e.*, BSP and RAT) with a simple example.

9.2.1. Parallel Datalog evaluation

Before illustrating the distributed stream reasoning in BigSR, we summarize the three Parallelism Levels (PL) mentioned in [98] for parallel instantiation of Datalog programs.

(PL1) Components level. Consider a stratified Datalog program $\mathcal{P}$ and its dependency graph $G_{\mathcal{P}} = \langle V, E \rangle$. $\mathcal{P}$ can be split into n subprograms $\{p_i\}_{i \in 1,\ldots,n}$ where each subprogram p_i is associated to a strongly connected component (SCC) C_i of $G_{\mathcal{P}}$. In accordance with the topological order of C_i, we can identify the subprograms that can be executed in parallel.

(PL2) Rules level. When recursion occurs in C_i, p_i is concurrently evaluated through bottom-up semi-naive algorithm [99].

(PL3) Single Rule level. Consider a program $\mathcal{P} = \mathrm{T}(X) \leftarrow \mathrm{R}(Y, X)$ where $\mathcal{P}$ contains a limited number of rules. As a result, $\mathcal{P}$ is neither benefiting from PL1 nor PL2. In this situation, the idea is to divide a single rule instantiation into a number of subtasks. All the subtasks are able to run independently.

Computing the answer stream of a positive LARS program can be regarded as evaluating a Datalog program at each time point. PL1, PL2, and PL3 are thus enabled to be applied in LARS program.

9.2.2. Streaming Models on Spark and Flink

Streaming Models. In general, two broad classes of execution models exist in distributed stream processing frameworks: BSP and RAT. Representative streaming systems, *e.g.*, Spark Streaming, Google Dataflow [47], based on the BSP model buffer and process data by batch. Intuitively, BSP organizes the communication between processes and synchronizes the data processing across records by setting barriers at

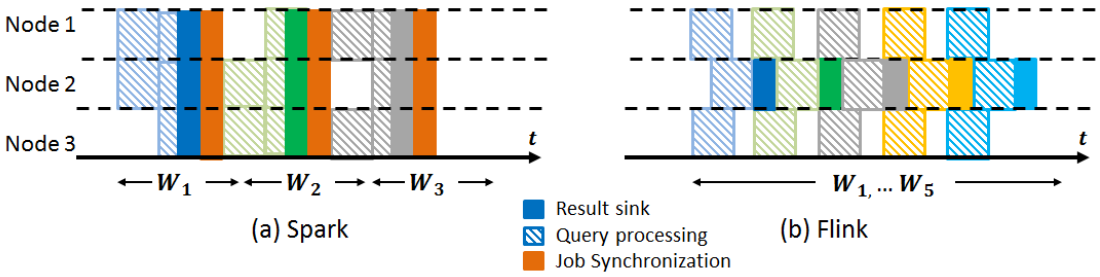


Figure 9.1.: Blocking and non-blocking query processing.

the end of each batch. On the contrary, RAT systems like Flink and Storm handle data processing record by record, where operators are regarded as long running tasks, which rely on mutable local states. The computation is done through the data flowing from one operator to another. We choose Spark and Flink as the underlying systems of BigSR. Both systems ensure fault tolerance, automatic work distribution, and load balancing.

Before we present the implementation details of BigSR, we use an example to demonstrate a fundamental difference between BSP and RAT. Considering a program $\mathcal{P} = \mathrm{T}(X) \leftarrow \boxplus_{\tau}^{w(l,d)} \Diamond(\mathrm{R1}(Y) \wedge \mathrm{R2}(Y, X))$, Figure 9.1 roughly gives a runtime example of $\mathcal{P}_0$ on Spark (Figure 9.1(a)) and Flink (Figure 9.1(b)). Spark processes the data stream synchronously, the next query execution will be launched after the previous one is finished. Conversely, Flink serializes, caches, and pushes forward each record to the next operator eagerly right after the current computation is done. Such behavior minimizes the data processing delay, and operators are able to perform asynchronously. In Figure 9.1, although Spark uses 3 nodes to compute the second part of every query with higher parallelism than Flink, it still needs some time for synchronization between two jobs. Thus, within 9-time units, Flink is able to finish 5 continuous queries while Spark is only able to process 3 queries during this same period of time. This clearly showcases a better use of computing resources in favor of Flink.

LARS does not include any notion of *state*. Thus, to build the connection between LARS and Spark/Flink's execution model, we define stateful and stateless operators as follows: A stateless operator over a stream transforms a stream into another stream. In contrast, a stateful operator is a function which takes a pair of a stream and a state, and returns another pair of stream and state. In other words, the data processing in stateless operator only looks up the current record. The evaluation of the stateful operator requires the system to hold an internal state, *e.g.*, use local

memory or an external database for window operator.

9.3. Distributed Stream Reasoning

This section is organized as follow: Section 9.3.1 presents the system architecture of BigSR; Section 9.3.2 gives some description of the data structure on Spark, Flink, and the RDF stream representation in our system. Section 9.3.3 explains the translation of LARS window operator to the BSP and RAT models. Then, Section 9.3.5 and 9.3.6 explore some details about distributed RDF stream reasoning on Spark and Flink, respectively. Finally, we introduce some discussions and partial conclusions in Section 9.3.7.

9.3.1. Architecture of BigSR

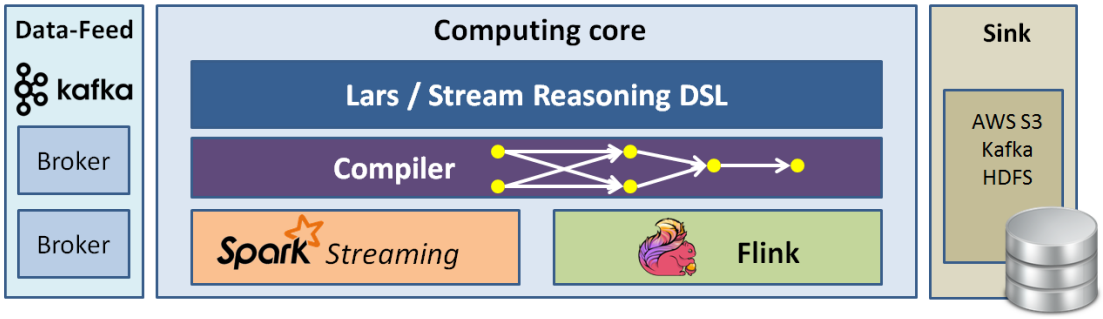


Figure 9.2.: BigSR system architecture

Figure 9.2 gives a high-level view of the BigSR architecture. It consists of three principal modules: (i) *Data-feed* is built on top of Apache Kafka (a distributed message queue) and ensures high throughput and fault-tolerant data stream injection/management; (ii) *Sink* persists query outputs into a storage component such as Amazon S3, HDFS or even Kafka; (iii) *Computing core* first registers and compiles a given LARS program into BigSR's logical execution plan. Then, the system binds the obtained logical plan to the physical operators of Spark (Streaming) or Flink for real-time distributed RDF stream reasoning.

9.3.2. Data Structure

BigSR comes with a LARS stream reasoning Domain Specific Language (DSL). Listing D.2 showcases a query example from the SRBench dataset. The query

grammar follows a general Datalog program writing style. For instance, on line 2 of Listing D.2, atom_tp2 denotes an atom of extensional predicate *type* with variable *Obs* and constant *rainObs*. Rule r is constructed with a head atom atom_res, two body atoms atom_1, atom_2 and a time-based window timeWindow. timeWindow accepts two parameters l and d to define a sliding window over the conjunction of atom_tp1 and atom_tp2. Finally, we construct program p, where p can be expressed by the following LARS rule:

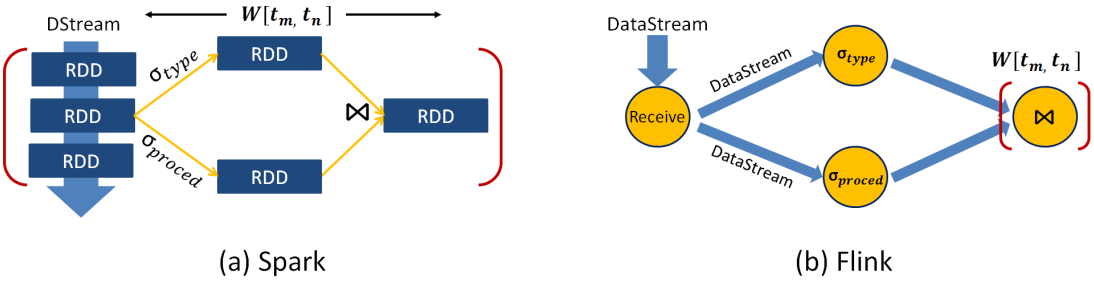
$$\text{resIRI}(Obs, Sen) \leftarrow \boxplus_{\tau}^{w(l,d)} \Diamond (\text{procedure}(Obs, Sen), \text{type}(Obs, rainObs).$$

In order to capture the previously-stated parallelism paradigms of Section 9.2.1 with BSP and RAT streaming models, we detail the query evaluation on Spark and Flink. BigSR adopts set semantics to handle all stateful operators, *i.e.*, each IDB inferred by stateful Datalog formulas will be deduplicated.

```
val atom_tp1 = Atom(procedure, Term("Obs"), Term("Sen"))
val atom_tp2 = Atom(type, Term("Obs"), Term("rainObs"))
val atom_res = Atom(resIRI, Term("Obs"), Term("Sen"))
val r = Rule(atom_res, Set(atom_tp1, atom_tp2), timeWindow(l, d))
val p = Program(Set(r))
```

Listing 9.1: BigSR DSL code snippet

Both Spark and Flink keep their own data structures to support BSP and RAT, respectively: (1) Spark abstracts a sequence of RDD as *DStream*[50] to enable near real-time data processing. The system buffers incoming data streams periodically as a micro-batch RDD. Each RDD encapsulates the data in a certain time interval, *i.e.*, corresponding to wall-clock times. Intuitively, a micro-batch RDD refers to the minimum allowable data operation granularity. In addition, the timestamp assigned by the system does not bring any impact to the query's semantics. (2) Flink takes *DataStream* as a basic data structure. DataStream represents a parallel data flow running on multiple partitions where all data transformations are processed at a record-level. Such a fine-grained data transformation makes event-timestamp-based operation feasible. In BigSR, we use the so-called *ingestion time* to handle time-based windows. Practically, each record gets the stream source's current time as a timestamp. Moreover, internally, Flink handles ingestion and event time in the same



(a) Spark (b) Flink

Figure 9.3.: Logical plan of query D.2 on Spark and Flink

manner. Instead of using event timestamp, we choose ingestion time for data stream processing in Flink for the purpose of simplifying our experiment.

9.3.3. Window Operation in BigSR

We introduce the translation of LARS window semantics to the physical operators in Spark and Flink. As described in Section **??**, only time-based window is involved in BigSR for the current implementation. We thus cover the most common use-cases of stream reasoning.

Here, we clarify the translation of LARS time-based window operator to Spark (BSP) and Flink (RAT). Specifically, we provide, under BSP and RAT, the translations of atoms of the form $\boxplus_\tau^{w(l,d)}\Diamond$, for an input stream S, with range size l, sliding size d and a sliding window operator $w(l,d)$.

Spark. A Spark Streaming application only allows a single global window operator. The system buffers input data stream in real-time, and launches the computation *w.r.t.* its predefined logical plan (more details are given in Section 9.3.4).

For an input stream S, the translation of $\boxplus_\tau^{w(l,d)}\Diamond S$ in Spark is illustrated in Figure 9.4. A window operator with range size l in LARS is firstly translated into four micro-batches of size $l_b = l/4$ in Spark, and then a batch of these four micro-batches with sliding size d, $l_b = d/2$. Each micro-batch $batch_i$ is assigned to a time interval $T_i = [t_i, t_{i+l_b}]$ containing data within T_i. A micro-batch is the basic unit of data processing in Spark's BSP model.

Recall that LARS defines the window operator at ASP's *formula* level. It is not obvious how to extend such translation to the program level in LARS when there are multiple different window definitions, because the micro-batch approach can only applied to one window over the whole program.

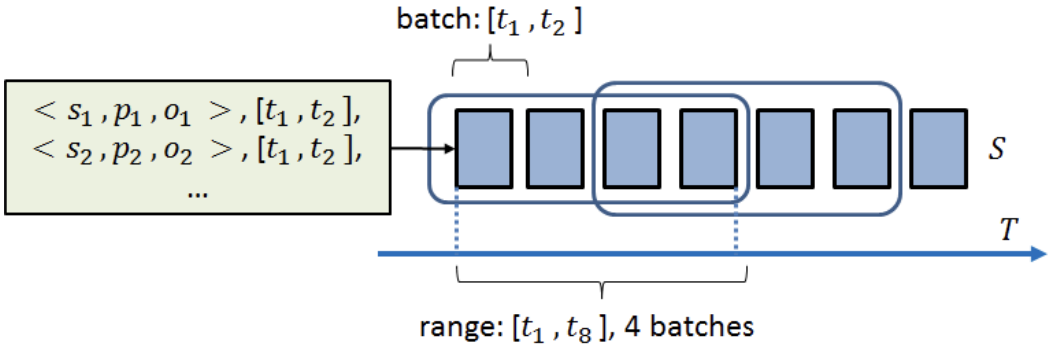


Figure 9.4.: The translation of window operator on Spark.

Flink. Comparing to the coarse-grained, micro-batch-level timestamp operation in Spark Streaming, Flink manipulates timestamps at the record-level. Each RDF triple is annotated by a timestamp.

In Figure 9.5, for a given start time point t_i of a window function $w(l, d)$, $\boxplus_{\tau}^{w(l,d)} \Diamond S$ holds all the data from $[t_i, t_i + l]$, *e.g.*, in Figure 9.5, $[t_1, t_4] \in [t_1, t_1 + l]$, and 4 triples are buffered by $w(l, d)$. $w(l, d)$ slides periodically over S by $d = 3$ time points.

In our implementation, we identify stream S by a certain predicate of RDF triple. E.g, the answer stream of $\boxplus_{\tau}^{w(l,d)} \Diamond(\mathrm{p}(Y))$ only contains the atoms with predicate p.

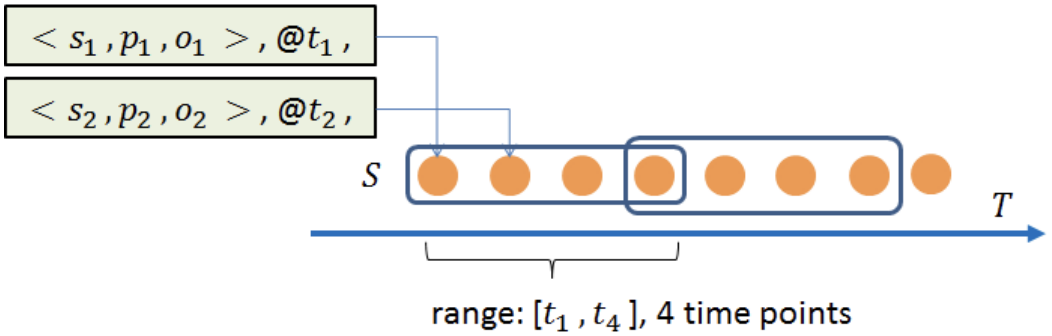


Figure 9.5.: The translation of window operator on Flink.

9.3.4. Program plans generation.

Figure 9.3 compares the logical plan of our Listing D.2 query example. Both Spark and Flink can naturally embed the three parallelism strategies of Section 9.2.1 into their own native physical plan. Due to reliable cluster resource allocation, continuous Spark jobs are launched synchronously, *i.e.*, a link connects two consecutive query

executions (Figure 9.3, (a)). Input data from stream sources are collected through a window of duration T which consists of n micro-batches of duration t. The buffered data are processed by the entire DAG of operators. On the other hand, the execution bound only exists on stateful operators, *e.g.*, join and window, in Flink (Figure 9.3 (b)). Except for the conjunction of σ_{type} and $\sigma_{procedure}$ which run synchronously, data reception and selections σ_{type}, $\sigma_{procedure}$ run asynchronously.

9.3.5. Distributed Stream Reasoning on Spark

In the following section, in order to provide a fair comparison between BSP and RAT, we keep identical query semantics for these two models. Hence, consider the following program $\mathcal{P}_0$ consisting of rules R_1, R_2, and R_3:

$$
\begin{aligned}
R1: &\quad \mathrm{p}_2(X,Y) \leftarrow \boxplus_{\tau}^{w(l,d)} \Diamond(\mathrm{p}_0(X,Y)) \\
R2: &\quad \mathrm{p}_1(X,Y) \leftarrow \boxplus_{\tau}^{w(l,d)} \Diamond(\mathrm{p}_2(X,Y) \wedge \mathrm{p}_0(Y,Z)) \\
R3: &\quad \mathrm{p}_2(X,Y) \leftarrow \boxplus_{\tau}^{w(l,d)} \Diamond(\mathrm{p}_1(X,Y) \wedge \mathrm{p}_0(X,Y))
\end{aligned}
$$

We assume that R_1, R_2, and R_3 hold the same window operator in their bodies. Be aware that this is not a prerequisite for Flink in the general case. Predicate p_0 is an EDB predicate while p_1 and p_2 are two IDB predicates.

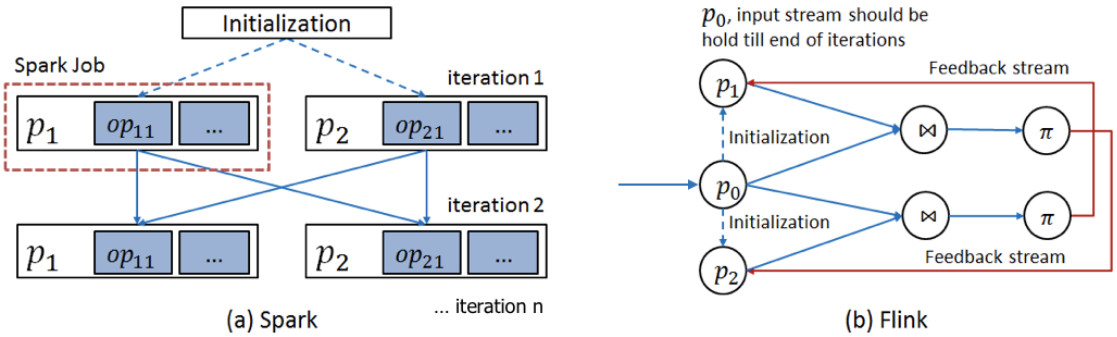


Figure 9.6.: Recursive program ($\mathcal{P}_0$) evaluation on Spark and Flink.

Algorithm 3 adopts the semi-naive evaluation of program $\mathcal{P}_0$ on Spark Streaming. p_0 is the set of input facts over DStream. Spark captures p_0 by window operator $\boxplus_{\tau}^{w(l,d)}\Diamond$ applied over input streams, and initializes IDBs p_1, $\delta\mathrm{p}_1$, p_2, and $\delta\mathrm{p}_2$. The evaluation terminates when a fixed-point is reached on IDB relations. During the whole semi-naive evaluation, the system omits the notion of *time*, and the evaluation

is performed as usual in a statical data processing.

We now present how this evaluation of $\mathcal{P}_0$ is parallelized on Spark:

- **PL1.**: The system starts the evaluation by initializing p_1 and p_2 using input EDB p_0. p_1 and p_2 forms an SCC C_i, for any other SCC C_j, where C_j does not depend on C_i, the evaluations of C_i and C_j can be computed in parallel.
- **PL2.** In each iteration step of the semi-naive evaluation, the set of operations (*e.g.*, selection, join, union) which compute each IDB predicate are chained together as a Spark job, *i.e.*, , two Spark jobs for the evaluation of p_1 and p_2 are involved. The iterations are completed until a fix-point is reached. The system outputs p_1 and p_2 for further calculation.
- **PL3.** Inside a single Spark Job, each operator performs a transformation of RDD. As an RDD is a distributed data collection, multiple tasks may execute concurrently across different data partitions.

The program $\mathcal{P}_0$ corresponds to a series of BSP Spark *jobs* which execute in a Spark Streaming context. One restriction is that $\mathcal{P}_0$ is only allowed to possess a single global window ($\boxplus_{\tau}^{w(l,d)}$).

For a non-recursive program, the logical plan is first mapped into a single Spark job's DAG logical plan. Next, Spark compiles the logical plan to its physical plan and is then evaluated. We mainly discuss the recursive program evaluation on Spark here. Considering the previously defined program $\mathcal{P}_0$, Figure 9.6 (a) gives a running example by using the semi-naive evaluation of [99].

Limitation & Envisioned Optimization. The implementation of recursion support in BigSR is straightforward. Some discussions about [81] and our envisioned optimization for recursive query handling are worth mentioning. Of the four proposed solutions in [81], two of them play key roles: (i) extending immutable RDD to mutable *SetRDD*; (ii) adding *recursive stage* support in Spark job scheduler. Both (i) and (ii) sacrifice fault-tolerance to gain system performance. For a streaming service running 24×7, such a fault-tolerance trade-off has a high potential impact on system's robustness and reliability.

In addition to the optimizations of [81], we propose two other envisioned solutions: (a) Integration with distributed index. For stratified LARS program, iterations and join operations become the performance bottleneck since an iteration potentially involves intensive job scheduling and network shuffling in a distributed environment.

Algorithm 3 Semi-Naive evaluation of $\mathcal{P}_0$ in Spark

Input : program $\mathcal{P}_0$ with global window function $w(l, d)$,
DStream S
Output : $\mathrm{R_{p_1}}$,$\mathrm{R_{p_2}}$ (RDDs)
foreach *Substream* S' *of* S, *computed by* $\boxplus_{\tau}^{w(l,d)}\Diamond$ **do**
// Initialize RDDs for $\delta'\mathrm{p}_1$, $\delta'\mathrm{p}_2$
Let $\delta\mathrm{p}_1 = \delta\mathrm{p}_2 = \varnothing$
Let $R_{\mathrm{p}_1} = R_{\mathrm{p}_2} = \varnothing$
$R_{p_0} = \delta\mathrm{p}_1 = \delta\mathrm{p}_2 := \{\mathrm{p}_0(s, o) \mid \mathrm{p}_0(s, o) \in S'\}$;
$R_{\mathrm{p}_1} := \delta\mathrm{p}_1, R_{\mathrm{p}_2} := \delta\mathrm{p}_2$;

do
// Compute δ'p1, δ'p2 in parallel
$\delta'\mathrm{p1} := \delta\mathrm{p}_2 \bowtie \pi_{Y,Z}(R_{\mathrm{p}_0}) - R_{\mathrm{p}_1}$;
$\delta'\mathrm{p2} := \delta\mathrm{p}_1 \bowtie \pi_{X,Y}(R_{\mathrm{p}_0}) - R_{\mathrm{p}_2}$;

// Compute R_{p1}, R_{p2} in parallel
$R_{\mathrm{p}_1} := R_{\mathrm{p}_1} \bigcup \delta'\mathrm{p}_1$;
$R_{\mathrm{p}_2} := R_{\mathrm{p}_2} \bigcup \delta'\mathrm{p}_2$;
$\delta\mathrm{p1} = \delta'\mathrm{p1}$, $\delta\mathrm{p2} = \delta'\mathrm{p2}$
while $\delta\mathrm{p}_1 \neq \varnothing$ *or* $\delta\mathrm{p}_2 \neq \varnothing$;
end

Instead of scanning each partition entirely in RDD, the work[4] integrates adaptive radix tree [100] to enable efficient data access and update. Our preliminary experimentation emphasizes that a performance gain for each iteration is up to 2.5-3 with indexed RDD compared to standard RDD. However, the index needs to be reconstructed after each iteration, which comes with a non-negligible overhead. This approach thus is in the experimental and envisaged stage. (b) Parallel job scheduling. Spark scheduler submits jobs in a centralized way via a master node. Frequent job submissions cause unavoidable latency. Rather than merging multiple jobs together into a single job [81], a distributed job scheduler [101] would provide millisecond latency for job scheduling without sacrificing any fault-tolerance.

9.3.6. Distributed Stream Reasoning on Flink

For a non-recursive program, BigSR compiles a LARS program into Flink's streaming topology (*i.e.*, DAG). Then, it is evaluated through data flowing between operators in the DAG.

[4] `https://github.com/amplab/spark-indexedrdd`

In the case of a recursive program (*e.g.*, $\mathcal{P}_0$), *i.e.*, different from the BSP model of Spark, Flink needs to handle the timestamp during the whole life cycle of the LARS program evaluation. For input stream S, Flink computes S' and continuously append data to p_0. The program evaluation on Flink follows the main spirit of semi-naive algorithm. However, each IDB/EDB relation is mapped into the independent data stream, and all stateful relational operators (*e.g.*, distinct, stream join) in $\mathcal{P}_0$ is restricted by $w(l, d)$. In particular, the RAT model requires Flink to feed back the IDB stream after each iteration before the fixed-point is reached. Algorithm 4 gives the general steps to evaluate program $\mathcal{P}_0$ on Flink. Figure 9.6(b) gives a high-level vision of the workflow of Flink of $\mathcal{P}_0$:

- **PL1.** Similar to Spark, any other SCC which is independent from C_i can be computed in parallel.
- **PL2.** Different from Spark, which splits the recursion into a series of independent jobs. Flink achieves recursion with streaming feedback. In iteration i, the system needs to feed back $S_f^i(\mathrm{p}_1)$ and $S_f^i(\mathrm{p}_2)$ (downstream for computing p_1 and p_2, respectively) as the input for iteration $i+1$. And the processes on $S_f^i(\mathrm{p}_1)$ and $S_f^i(\mathrm{p}_2)$ occur in parallel in a single iteration step.
- **PL3.** Similar to Spark, the DataStream flows through each operator across in parallel, each operator consists of multiple tasks (over some partitions) which can be performed concurrently (*w.r.t.* PL3).

Limitation & Envisioned Optimization. There are two main limitations we found by using the RAT model:

(1) We require that the conjunction of two atoms a_1, a_2 should share the same window operator, *e.g.*, , a formula $\alpha = \boxplus^{w_1}\Diamond a_1 \wedge \boxplus^{w_2}\Diamond a_2$ currently imposes that $w = w_1 = w_2$ (*i.e.*, $\alpha = \boxplus^{w}\Diamond(a_1 \wedge a_2)$). Given our experience, this limitation shows up when input stream is of type S and the underlying process relies on multi-cores/distributed environment. The main difficulties come from synchronization of clocks, task progress, and window trigger mechanisms. However, a single-core/centralized system with input stream $S^\star$ does not suffer from such a synchronization problem. Since all the computations are done sequentially, the program evaluation performs in a quasi-static way without considering the fast update of $S^\star$. To the best of our knowledge, CQELSCloud, which is the only implementation with distributed setting, has a similar semantic. Nevertheless, to avoid above-mentioned

Algorithm 4 Semi-Naive evaluation of $\mathcal{P}_0$ on Flink

Input : program $\mathcal{P}_0$ with global window function $w(l, d)$,
DataStream S
Output : $\mathrm{S_{p_1}}$,$\mathrm{S_{p_2}}$ (DataStream)
Initialize DataStreams for $\mathrm{p_0}$, $\mathrm{p_1}$, $\mathrm{p_2}$ as $\mathrm{S_{p_0}}$, $\mathrm{S_{p_1}}$, $\mathrm{S_{p_2}}$, respectively

foreach *DataStream* S **do**
  Buffer the current state $state(\mathrm{S_{p_0}})$ of $\mathrm{S_{p_0}}$
  $\delta\mathrm{S_{p_1}} := \pi_{X,Y}(state(\mathrm{S_{p_0}})), \delta\mathrm{S_{p_2}} := \pi_{X,Y}(state(\mathrm{S_{p_0}}))$;

  while $\delta\mathrm{S_{p1}} \neq \varnothing$ *or* $\delta\mathrm{S_{p2}} \neq \varnothing$ **do**
    $\delta'\mathrm{S_{p_1}} := \boxplus_{\tau}^{w(l,d)} \Diamond(\delta\mathrm{S_{p_2}} \bowtie \pi_{Y,Z}\mathrm{S_{p_0}} - \mathrm{S_{p_2}})$
    $\delta'\mathrm{S_{p_2}} := \boxplus_{\tau}^{w(l,d)} \Diamond(\delta\mathrm{S_{p_1}} \bowtie \pi_{X,Y}\mathrm{S_{p_0}} - \mathrm{S_{p_1}})$

    // Append derived facts to $\mathrm{p_1}, \mathrm{p_2}$
    $\mathrm{S_{p_1}} := \boxplus_{\tau}^{w(l,d)} \Diamond(\mathrm{S_{p_1}} \bigcup \delta'\mathrm{S_{p_1}})$;
    $\mathrm{S_{p_2}} := \boxplus_{\tau}^{w(l,d)} \Diamond(\mathrm{S_{p_2}} \bigcup \delta'\mathrm{S_{p_2}})$;

    $\delta\mathrm{S_{p_2}} = \delta'\mathrm{S_{p_2}}, \delta\mathrm{S_{p_1}} = \delta'\mathrm{S_{p_1}}$
    // Feed back streams for next iteration
    Feed back $\mathrm{S_{p_1}}, \mathrm{S_{p_2}}$;
  end
end

problems, CQELSCloud forbids event timestamp and "sliding" mechanisms in their window operators. The record is emitted eagerly right when the computation is done.

(2) We do not support recursive queries on Flink yet. Within the LARS framework, the implementation of recursion with RAT could be quite challenging. The example given earlier skips window operator. Once the body atoms of $\mathrm{p_1}$ and $\mathrm{p_2}$ are restricted by the temporal logic operator (*e.g.*, window operator), the synchronization problem mentioned in (1) will reappear. A possible solution is to merge multiple input streams together and perform the recursion with a recursive operator. The system would then have to cut off the query processing pipeline to handle the recursion. In such a situation, the recursive operator behaves similarly to the BSP model which is against the original intention of using the RAT model.

9.3.7. Discussions

Table 9.1 briefly compares BSP and RAT for implementing a stream reasoning framework like LARS. With BSP on Spark, the manipulation of temporal logical operators

Model	Expressiveness	Recursion
BSP (Spark)	Low, coarse-grained	Implementation easy
RAT (Flink)	High, fine-grained	Implementation difficult

Table 9.1.: Intuitive comparison between BSP and RAT

is rather coarse-grained with low expressiveness. The query should be evaluated in batches by a global window. Furthermore, the semantic of data timestamps does not influence the semantic of the query. The combination of window operators is not flexible. However, the query evaluation of the BSP model in a window is practically similar to a static data processing. Therefore, BSP greatly simplifies the implementation of recursion. On the other hand, the RAT model handles data processing record by record. The evaluation of operators can be performed asynchronously. RAT enables to manipulate the timestamp and window operators in a fine-grained fashion. Different types of time (*e.g.*, event time, ingestion time, system processing time) can be integrated on RAT. The combination of multiple window operators can be chained together and ran independently. For recursive query evaluation, RAT suffers from synchronization of processes over multiple streams and multiple windows. This makes the implementation of recursion within the LARS framework a challenging problem.

9.4. Evaluation

The code base (written using the Scala programming language), data sources and test queries are available on GitHub[5]. We conduct our experiments on a Amazon EMR cluster with a Yarn resource manager. The EMR cluster consists of a total of 9 nodes of type m4.xlarge. One node is setup for the Kafka broker and message producer, one node for Apache Zookeeper, seven nodes for Spark/Flink application (one master node and six worker nodes). Each node has 4 CPU virtual cores of 2.4 GHz Intel Xeon E5-2676 v3 processors, 16 GB RAM and 750 MB/s bandwidth. We use Spark 2.2.1, Flink 1.4.0 (broadcast join is disabled), Scala 2.11.7 and Java 8 as evaluation baselines.

[5] `https://github.com/renxiangnan/bigsr`

9.4.1. Benchmark Design

Dataset & Queries. Our evaluation is based on synthetic and real-world datasets which involve 4 different datasets and 15 queries (Table 9.2). The 4 datasets correspond to Waves, SRBench, CityBench and LUBM. All the data captured by the Waves, SRBench, and CityBench datasets come from real-world IoT sensors. The Waves dataset describes measures of a potable water network, *e.g.*, values of flow, pressure and chlorine levels, etc. SRBench, one of the first RSP benchmark, contains USA weather observations ranging from 2001 to 2009. CityBench simulates a smart city context for RSP applications and concerns sensor measures on vehicle traffic, parking lot utilization and user location use cases. All the aforementioned datasets come from RSP contexts. It is hard to design a recursive query, because the generated RDF data streams are usually directly converted from flat data (CSV) with few references between entities. Therefore, we use LUBM for recursive query evaluations.

Query Q_1 to Q_{11} include stateful operators for windowing and recursion, where Q_{12} to Q_{15} only contain stateless operators (*e.g.*, selection, filter, projection). We evaluate Q_{12} to Q_{15} to highlight the engine performance with BSP and RAT model for low-latency use cases, such as reactive applications.

	Q_1 - Q_3	Q_4 - Q_6	Q_7 - Q_8	Q_9 - Q_{11}	Q_{12} - Q_{15}
Dataset	Waves	SRBench	CityBench	LUBM	Synthetic
Recursive	No	No	No	Yes	No

Table 9.2.: Test queries and datasets.

To guarantee query semantics consistency between BSP and RAT, the result sets of a given query should be the same on the two distributed models. Input data streams are generated by Kafka message producer and injected into BigSR in parallel. The average stream rate is around 250,000 to 300,000 triples/second.

Performance metrics. Considering *Benchmarking Streaming Computation Engines at Yahoo!*, the well-known benchmark for distributed streaming systems[6], we take system throughput and query latency as the principal performance criteria. In particular, we categorize the evaluations into two groups:

- **Group 1:** Q_1 to Q_{11} (queries with stateful operators). We denote throughput

[6]https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

as the number of triples processed per second by the engine (*i.e.*, triples/second). Latency corresponds to the duration taken by BigSR between the arrival of an input and the generation of its output.

- **Group 2:** Q_{12} to Q_{15} (queries with only stateless operators). We focus on the minimum latency that the engine is able to attain. On Spark, we first reduce the micro-batch size as much as possible, then we record the query latency for completing the process of current micro-batch. On Flink, the latency of a record r indicates the time difference between the moment r enters the system and the moment r' outputs from the system.

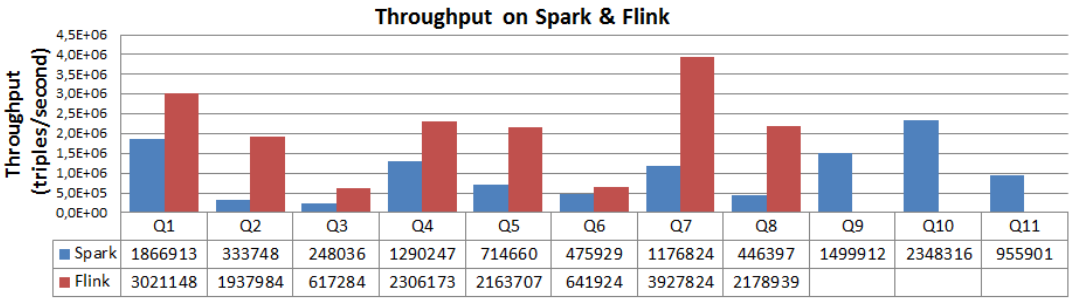
Performance tuning is one of the most important steps for the deployment of Spark and Flink applications. Based on our previous experience, we list three important factors which bring significant impact on engine performance, *i.e.*, parallelism level, memory management, and data serialization. Unfortunately, there is no fixed rule to configure these parameters in an optimal way. The tuning has to be done empirically. Besides, recursion on Spark may generate long RDD lineage in the driver memory which can lead to stack overflow. We thus periodically trigger the local checkpoint of RDD to truncate the RDD lineage.

9.4.2. Evaluation Results & Discussion

In this section, we present and discuss the evaluation result over the queries presented in Section 9.4.1. We do not compare BigSR to the state of the art RSP/(Streaming) nor to ASP systems due to the following reasons: (1) Compared to our previous work, on Strider [102] , the Spark implementation in BigSR is approximately 30% less efficient. We partially attribute this to the *distinct* operation which satisfies the set semantic. Nevertheless, performance evaluation in [102] emphasizes 1 to 2 orders of magnitude performance gains over available RSPs (*i.e.*, C-SPARQL and CQELS). We keep exactly the same cluster settings as our previous work [102] , we can consider that the distributed design of BigSR takes a substantial performance advantage over existing centralized RSP engines.

(2) The system likes Laser currently does not allow to continuously inject data stream in real-time. The system considers the stream is finite and loads the entire stream into memory for program evaluation. Moreover, the support of recursion is also missing (*i.e.*, occurrence of a runtime-error) in the current version of Laser.

Throughput. Figure 9.7 reports the engine throughput for Q_1 to Q_{11}. Both implementation with BSP (Spark) and RAT (Flink) achieves high throughput at the



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Spark	1866913	333748	248036	1290247	714660	475929	1176824	446397	1499912	2348316	955901
Flink	3021148	1937984	617284	2306173	2163707	641924	3927824	2178939			

Figure 9.7.: System throughput (triples/second) on Spark and Flink for Q_1 to $Q11$.

level of million triples per second. We observe that the throughput of Flink is 1.x - 3.x times superior to Spark. This difference is more substantial when the query has more intensive joins/conjunctions. This can be explained by the job scheduling of Spark which imposes join operations to be performed on different compute nodes, thus causing network shuffles. Moreover, Spark's job scheduling is difficult to control in a fine-grained manner. On the contrary, Flink is able to avoid shuffles with an appropriate system configuration, *i.e.*, the join operation can be managed by a *task manager* and performed locally on a single compute node.

For recursive queries Q_9 to Q_{11}, Spark achieves a throughput of up to 2.3M triples/second. Although we design three recursive queries for LUBM, the length of transitivity in LUBM is rather small which limits the number of iterations in the semi-naive evaluation. Additionally, the intermediate results generated in Q_9 to Q_{11} are of moderate size, which reduces the performance penalty implied by shuffle operations.

Latency. We summarize the query latency of Group 1 (Q_1 to Q_{11}) in Figure 9.8. Spark and Flink hold second/sub-second delay in general (only Q_3 exceeds one second on Flink). Flink has a lower latency than Spark. The obtained latency on Spark and Flink are already acceptable for most streaming applications.

Here, we highlight the experiment over queries in group 2 (Table 9.3). Intuitively, Q_{12} to Q_{15} have been designed to stress BSP and RAT on the latency dimension. In fact, the micro-batch interval size of Spark is set to 500 ms. Even though the average latency on Spark is around 100 ms, but 500 ms is approximately the minimum "safe" batch size we can configure on Spark. The reason is that the garbage collection (GC) triggers periodically in a long-running Spark Streaming application (on driver and workers), GC pause occurs from time to time. The query latency thus can grow up to

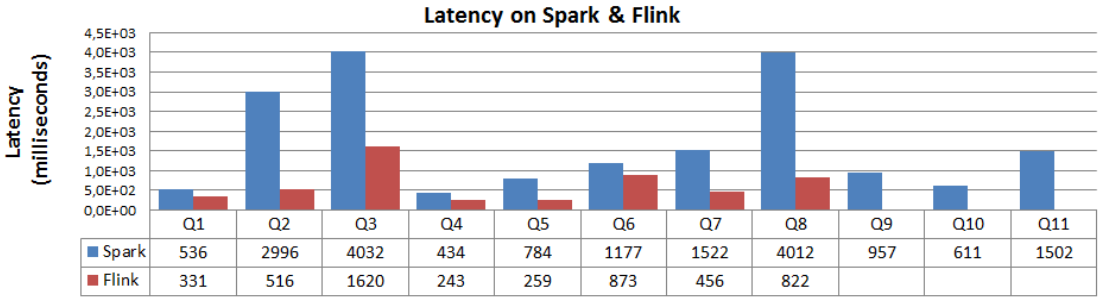


Figure 9.8.: Query latency (milliseconds) on Spark and Flink for Q_1 to $Q11$.

400 ms. We conclude that Spark satisfies the near-real-time use case with sub-second delay requirement.

On Flink, we calculate the record latency by subtracting the output timestamp from input system-assigned timestamp. The minimum observable time unit is millisecond (limited by Flink), and the vast majority obtained latency is 0 ms. Apparently, sub-millisecond delay meets most real-time, latency-sensitive use cases.

	Q_{12}	Q_{13}	Q_{14}	Q_{15}
Spark	110	96	115	99
Flink	<1	<1	<1	<1

Table 9.3.: Stateless query latency (millisecond); Spark micro-batch size = 500 ms.

9.5. Conclusion

This work bridges the gap between theoretical work in progress on RDF stream reasoning and modern cutting-edge Big Data technologies. In fact, our BigSR system is able to reach the millions of triples per second processing mark on complex queries and second/subsecond latency in general. In order to tackle scalability, BigSR considers the standard BSP and RAT approaches through implementations with state of the art open source frameworks, respectively Apache Spark and Apache Flink. Both these systems offer rich APIs (*e.g.*, obviously for stream processing but also for machine learning, graph analytics), fault-tolerance, load balancing, and automatic work distribution. In terms of reasoning, we address logic programming through the ASP-based LARS framework.

Our experimentation presents some interesting results on the current state of these systems. With its large programmer community, Spark is easier than Flink to get into and implement applications with. Nevertheless, it may be difficult to configure, and tune this parallel computing framework. The support for recursive rules was not a difficult problem. The overall performance of Flink on both data throughput and latency is superior to Spark and is quite impressive without requiring a lot of tuning. Nonetheless, the design and implementation of an evaluation approach for recursive programs are not straightforward. This is in fact in our future work list together with a more efficient incremental model maintenance.

10. Conclusion and Future Work

In this thesis, we have addressed some major problems of distributed RDF stream processing and reasoning by handling continuously SPARQL query, optimizing reasoning query and executing expressive temporal Datalog/ASP program.

The thesis starts with a survey of existing RSP benchmarks. We propose some new performance metrics and design a specific evaluation plan. In particular, we take into account the specific implementation of each RSP engine. We perform many experimentations to evaluate the impact of Stream Rate, Number of Triples, Window Size, Number of Streams and Static Data Size on Execution Time and Memory Consumption. Several queries with different complexities have been considered. The main results of this complete study are that each RSP engine has its own advantage and are adapted to a particular context and use case, e.g., C-SPARQL excels on complex and multi-stream queries while CQELS stands out on queries requiring static data.

Given this evaluation of existing RSP engines, we are able to design Strider, a distributed RDF stream processing engine for large scale data stream. It is built on top of Apache Spark Streaming and Apache Kafka to support continuous SPARQL query evaluation and thus possesses the characteristics of a production-ready RSP. Strider comes with a set of hybrid AQP strategies: i.e., static heuristic rule-based optimization, forward and backward adaptive query processing. We insert the trigger into the optimizer to attain the automatic strategy switching at query runtime. Moreover, with its micro-batch approach, Strider fills a gap in the current state of RSP ecosystem which solely focuses on record-at-a-time. Through our micro-benchmark based on real-word datasets, Strider provides a million/sub-million- level throughput and second/sub-second latency, a major breakthrough in distributed RSPs. And we also demonstrate the system reliability which is capable of handling the structurally instable RDF streams.

Then, we extend Strider to enable distributed RDF stream reasoning by integrating several novel reasoning approaches, *i.e.*, LiteMat for RDFS + `sameAs` reasoning over streaming data. For most queries, LiteMat together with the representative-based

(RB) approach for `sameAs` cliques is the most efficient. Nevertheless, LiteMat + SAM proposes an unprecedented provenance-awareness feature that can not be obtained in other approaches. Lite + SAM can also be useful for very simple queries, e.g., a single triple pattern in the WHERE clause. To the best of our knowledge, this is the first scalable, production-ready RSP system to support such ontology expressiveness. Via a thorough evaluation, we have demonstrated the relevance of our system to reason with low latency over high throughput data streams.

Finally, we bridge the gap between theoretical work in progress on RDF stream reasoning and modern cutting-edge Big Data technologies. We emphasize that a trade-off between expressiveness of reasoning and scalability is possible in RDF stream reasoning. In fact our BigSR system is able to reach millions triples per second processing mark on complex queries and second and subsecond latency in general. In order to tackle scalability, BigSR considers the standard BSP and RAT approaches through implementations with state-of-the-art open source frameworks, respectively Apache Spark and Apache Flink. Both these systems offer rich APIs (e.g., obviously for stream processing but also for machine learning, graph analytics), fault-tolerance, load balancing and automatic work distribution. In terms of reasoning, we address logic programming through the ASP-based LARS framework. Our experimentation presents some interesting results on the current state of these systems. With its large programmer community, Spark is easier than Flink to get into and implement applications. Nevertheless, it may be difficult to configure, tune this parallel computing framework. The support for recursive rules was not a difficult problem. The overall performance of Flink on both data throughput and latency is superior to Spark and is quite impressive without requiring a lot of tuning. Nonetheless, the design and implementation of an evaluation approach for recursive programs is not straightforward.

In a next step, we plan to explore more features of reasoning RDF stream in distributed environment. This implies four general aspects: (1) Querying over compressed data stream. Since original RDF data format is redundant, efficient RDF serializing would be considerably smaller than the original; (2) we found that LiteMat could be integrated with conventional Datalog/ASP materialization; (3) the further support of recursion for RAT model is also in the scope of consideration; (4) neither Strider nor BigSR adopts any incremental evaluation strategy. Such behavior may involve a lot of recomputations over input data stream which have been already materialized. Incremental query evaluation could potentially leverage the system performance by avoiding redundant computation.

Prefix

- ex: <http://myexample.org/>
- f: <http://larkc.eu/csparql/sparql/jena/ext#>
- a: <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
- ssn: <http://purl.oclc.org/NET/ssnx/ssn/>
- qudt: <http://data.nasa.gov/qudt/owl/qudt/>
- waterML: <http://www.cuahsi.org/waterML/>

A. Queries for the evaluation in Chapter 5

Q_1

```
SELECT DISTINCT ?observation
FROM STREAM <http://myexample.org/stream> [RANGE 1s STEP 1s]
WHERE {
?message ex:observeChlorine ?observation .
?observation ex:hasTag ?tag . }
```

Q_2

```
SELECT DISTINCT ?observation (COUNT(?tag) AS ?numberOfTags)
FROM STREAM <http://myexample.org/stream> [RANGE 1s STEP 1s]
WHERE {
?message   ex:observeChlorine   ?observation .
?observation   ex:hasTag   ?tag . }
GROUP BY ?observation
ORDER BY ASC(?observation)
```

Q_3

```
SELECT ?observation
FROM STREAM <http://myexample.org/stream> [RANGE 1s STEP 1s]
WHERE {
?message ex:observeChlorine ?observation .
```

```
?observation ex:hasTag ?tag .
FILTER ( regex(str(?observation), '00$', 'i')
       || ( regex(str(?observation), '50$', 'i'))) }
GROUP BY ?observation
```

Q_4

```
SELECT ?observation (COUNT(?tag) AS ?numberOfTags)
FROM STREAM <http://myexample.org/stream> [RANGE 1s STEP 1s]
WHERE {
?message ex:observeChlorine ?observation .
?observation ex:hasTag ?tag .
FILTER ( regex(str(?tag), '1$', \"i\")
 || regex(str(?tag), '2$', \"i\")
 || regex(str(?tag), '3$', \"i\"))
FILTER (f:timestamp(?observation, ex:hasTag, ?tag)
        >= f:timestamp(?message, ex:observeChlorine, ?observation)) }
GROUP BY ?observation ?numberOfTags
```

Q_5

```
SELECT ?observation (COUNT(?tag) AS ?numberOfTags)
FROM STREAM <http://myexample.org/stream> [RANGE 1s STEP 1s]
WHERE { {
  ?message    ex:observeChlorine   ?observation .
  ?observation    ex:hasTag   ?tag .
FILTER ( regex(str(?observation), '00$', 'i') )
FILTER (f:timestamp(?observation, ex:hasTag, ?tag)
        >= f:timestamp(?message, ex:observeChlorine, ?observation))
 } UNION {
 ?message1   ex:observeFlow   ?observation.
 ?observation   ex:hasTag   ?tag .
FILTER ( regex(str(?observation), '10$', 'i') ) } }
GROUP BY ?observation
```

```
HAVING (COUNT(?tag) = 3)
ORDER BY ASC(?observation)
```

Q_6

```
SELECT ?sector ?timestamp ?label
FROM NAMED STREAM <http://myexample.org> [RANGE 1s STEP 1s]
FROM <http:...>
WHERE {
?message ex:observeChlorine ?observation .
?observation ex:isProducedBy ?sensorId .
?sensorId ex:belongsTo ?sector .
?sensorId ex:isCreatedBy ?manufacture_ID .
rdfs:label ?label . }
```

B. Queries for the evaluation in Chapter 7

Q_1

Q_2

Q_3

Q_4

```
SELECT  ?s  ?o1
WHERE {
?s    a   ?o ;
?s    ssn:isProducedBy  ?o1 .}
```

Q_5

```
SELECT  ?s   ?o1   ?o2   ?o3   ?o4   ?o5
WHERE  { {
 ?s    a    ?o .
 ?s    ssn:isProducedB   ?o1 .
 ?s    ssn:hasValue    ?o2 . }
UNION {
 ?o2     a    ?o3 .
 ?o2    ssn:startTime ?o4 .
 ?o2    qudt:numericValue ?o5 . }}
```

Q_6

```
SELECT  ?s   ?o1   ?o2   ?o3   ?o4   ?o5   ?o6
WHERE {
?s      a    ?o .
?s      ssn:isProducedBy    ?o1 .
?s      ssn:hasValue   ?o2 .
?o2    a    ?o3 .
?o2    ssn:startTime   ?o4 .
?o2    qudt:unit   ?o5 .
?o2    qudt:numericValue  ?o6 . }
```

Q_7

```
SELECT ?s ?o1 ?o2 ?o3
WHERE {
?s     ssn:hasValue    ?o1 .
?s     ssn:hasValue    ?o2 .
?s     ssn:hasValue    ?o3 .
?o1    a      waterML:flow .
?o2    a      waterML:temperature .
?o3    a      waterML:chlorine. }
```

Q_8

```
SELECT ?s ?o1 ?o2 ?o3
WHERE {
?s     ssn:hasValue    ?o1 .
?s     ssn:hasValue    ?o2 .
?s     ssn:hasValue    ?o3 .
?o1    a      waterML:flow .
?o2    a      waterML:temperature .
?o3    a      waterML:chlorine. }
```

Q_9

```
SELECT ?o11 ?o21 ?o31
WHERE {
?s    ssn:hasValue    ?o1 .
?s     ssn:hasValue   ?o2 .
?s     ssn:hasValue   ?o3 .
?o1    a   waterML:flow.
?o1    qudt:numericValue   ?o11 .
?o2    a   waterML:temperature.
?o2     qudt:numericValue   ?o21 .
?o3     a   waterML:chlorine .
?o3     qudt:numericValue   ?o31 .}
```

C. Queries for the evaluation in Chapter 8

C.1. Queries

C.1.1. Queries with inferences over concept hierarchies

Q_1: Inferences are required on the Professor concept which has no direct instances in LUBM datasets.

```
SELECT ?n
WHERE {
    ?x  rdf:type  lubm:Professor;
    ?x  lubm:name  ?n.}
```

Q_2: Inferences are required on both the Professor and Student concepts.

```
SELECT ?ns ?nx
WHERE {
    ?x  rdf:type lubm:Professor;
    ?x  lubm:name ?nx.
    ?s  lubm:advisor ?x;
    ?s  rdf:type lubm:Student.
    ?s  lubm:name ?ns. }
```

C.1.2. Query with inferences over property hierarchies

Q_3: Inferences are required for the memberOf property which has on direct sub property and one indirect sub property.

```
SELECT ?x ?o  WHERE { ?x  lubm:memberOf  ?o.}
```

C.1.3. Queries with inferences over both concept and property hierarchies

Q_4: This query mixes the Q1 and Q3 and thus necessitates to reason over the Professor and memberOf hierarchies

```
SELECT ?o ?n
WHERE {
    ?x  rdf:type  lubm:Professor;
    ?x  memberOf  ?o;
    ?x  lubm:name ?n.
}
```

Q_5: This query goes further than Q4 by mixing Q2 and Q3, i.e., it requires reasoning over the Professor and Student concept hierarchies and the memberOf property hierarchy.

```
SELECT ?ns ?nx ?o
WHERE {
     ?x  rdf:type  lubm:Professor;
     ?x  lubm:name  ?nx;
     ?x  lubm:memberOf  ?o.
     ?s  lubm:advisor  ?x;
     ?s  rdf:type  lubm:Student;
     ?s  lubm:name ?ns.
 }
```

C.1.4. Query with inferences over the owl:sameAs property

Q_6: Inferences are required over a clique of similar individuals of the type PostDoc.

```
SELECT ?n ?e
WHERE {
     ?x  rdf:type  lubm:PostDoc;
     ?x  lubm:name  ?n;
     ?x  lubm:emailAddress ?e.
   }
```

C.1.5. Queries with inferences over concept, property hierarchies and owl:sameAs

Q_7: Inferences over the Faculty concept hierarchy, which includes PostDoc sameAs individuals and the memberOf property.

```
SELECT ?o ?n
WHERE {
     ?x  rdf:type  lubm:Faculty;
     ?x  memberOf  ?o;
     ?x  lubm:name  ?n.}
```

Q_8: The most complex query of our evaluation with two inferences over concept hierarchies (Faculty and Student), with the former containing sameAs individual cliques, and inferences over the memberOf property hierarchy.

```
SELECT ?ns ?nx ?o
WHERE {
     ?x  rdf:type  lubm:Faculty;
     ?x  lubm:name  ?nx;
     ?x  lubm:memberOf  ?o.
     ?s  lubm:advisor  ?x;
     ?s  rdf:type  lubm:Student;
     ?s  lubm:name  ?ns.}
```

C.2. Details on our continuous query extension

The `STREAMING` clause is used to initialize a Spark Streaming context. As in other RSP query languages, the `WINDOW` and `SLIDE` keywords respectively specify the range and size of a windowing operator. Since Spark Streaming is based on a micro-batch processing model, we defined a `BATCH` clause to assign the time interval of each micro-batch. Basically, a single micro-batch represents a RDD, Spark's main abstraction. For each triggered query execution, Spark Streaming receives a segment of Dstreams which essentially consists of an RDD sequence.

The `STREAMING` clause is used to initialize a Spark Streaming context. As in other RSP query languages, the `WINDOW` and `SLIDE` keywords respectively specify the range and size of a windowing operator. Since Spark Streaming is based on a

micro-batch processing model, we defined a `BATCH` clause to assign the time interval of each micro-batch. Basically, a single micro-batch represents a RDD, Spark's main abstraction. For each triggered query execution, Spark Streaming receives a segment of Dstreams which essentially consists of an RDD sequence.

The `REGISTER` clause concerns the SPARQL queries to be processed. Strider^{R} allows to register multiple queries, and uses a thread pool to launch all registered queries asynchronously. However, the optimization of multiple SPARQL queries is beyond the scope of this paper. Inside `REGISTER`, each continuous SPARQL query possesses a query ID. The `REASONING` clause enables the end-user to select a combination of concept/property hierarchy and `sameAs` inferences. Once `REASONING` service is triggered, Strider^{R} automatically rewrites the given SPARQL query to its LiteMat mapping. Moreover, incoming data stream will also be encoded within the rules of LiteMat KBs.

D. Queries for the evaluation in Chapter 9

D.1. Waves dataset, non-recursive

Q_1:

```
val atom_tp1 = Atom(addPref(ssnPref, "hasValue"), Term("S"), Term("O"))
val atom_res = Atom(resIRI, Term("S"))
val rule1 = Rule(atom_res, Set(atom_tp1), f_win(range, slide))
Program(Set(rule1))
```

Q_2:

```
val atom_tp1 = Atom(addPref(rdfSyntaxPref, "type"), Term("S"), Term("O1"))
val atom_tp2 = Atom(addPref(ssnPref, "startTime"), Term("S"), Term("O2"))
val atom_tp3 = Atom(addPref(qudtPref, "unit"), Term("S"), Term("O3"))
val atom_tp12 = Atom(addTempPref("tp12"), Term("S"), Term("O2"))
val atom_result = Atom(resIRI, Term("S"), Term("O3"))

val rule1 = Rule(atom_tp12, Set(atom_tp1, atom_tp2), f_win(range, slide))
val rule2 = Rule(atom_result, Set(atom_tp12, atom_tp3), f_win(range, slide))
Program(Set(rule1, rule2))
```

Q_3:

```
val atom_tp1 = Atom(addPref(rdfSyntaxPref, "type"), Term("S"), Term("O1"))
val atom_tp2 = Atom(addPref(ssnPref, "startTime"), Term("S"), Term("O2"))
val atom_tp3 = Atom(addPref(qudtPref, "unit"), Term("S"), Term("O3"))
val atom_tp4 = Atom(addPref(qudtPref, "numericValue"), Term("S"), Term("O4"))
val atom_tp5 = Atom(addPref(ssnPref, "isProducedBy"), Term("O"), Term("O5"))
val atom_tp6 = Atom(addPref(rdfSyntaxPref, "type"), Term("O"), Term("O6"))
val atom_tp7 = Atom(addPref(ssnPref, "hasValue"), Term("O"), Term("S"))
val atom_star1 = Atom(addTempPref("star1"), Term("S"), Term("O1"), Term("O2"), Term("O3"),
    Term("O4"))
val atom_star2 = Atom(addTempPref("star2"), Term("O"), Term("S"), Term("O5"), Term("O6"))
```

```
val atom_result = Atom(resIRI, Term("O"), Term("O1"), Term("O2"), Term("O3"), Term("O4"),
    Term("O5"), Term("O6"))

val rule1 = Rule(atom_star1, Set(atom_tp1, atom_tp2, atom_tp3, atom_tp4), f_win(range,
    slide))
val rule2 = Rule(atom_star2, Set(atom_tp5, atom_tp6, atom_tp7), f_win(range, slide))
val rule3 = Rule(atom_result, Set(atom_star1, atom_star2), f_win(range, slide))
Program(Set(rule1, rule2, rule3))
```

D.2. SRBench dataset, non-recursive

Q_4:

```
val atom_tp1 = Atom(addPref(obsPref, "procedure"), Term("Obs"), Term("Sen"))
val atom_tp2 = Atom(addPref(rdfSyntaxPref, "type"), Term("Obs"), Term(addPref(whtPref,"
    RainfallObservation")))
val atom_result = Atom(resIRI, Term("Obs"), Term("Sen"))
val rule1 = Rule(atom_result, Set(atom_tp1, atom_tp2), f_win(range, slide))
Program(Set(rule1))
```

Q_5:

```
val atom_tp1 = Atom(addPref(obsPref, "procedure"), Term("Obs"), Term("Sen"))
val atom_tp2 = Atom(addPref(rdfSyntaxPref, "type"), Term("Obs"), Term(addPref(whtPref, "
    RainfallObservation")))
val atom_tp3 = Atom(addPref(obsPref, "result"), Term("Obs"), Term("Res"))
val atom_star1 = Atom(addTempPref("star1"), Term("Obs"), Term("Sen"))
val atom_result = Atom(resIRI, Term("Obs"), Term("Res"))

val rule1 = Rule(atom_star1, Set(atom_tp1, atom_tp2), f_win(range, slide))
val rule2 = Rule(atom_result, Set(atom_star1, atom_tp3), f_win(range, slide))
Program(Set(rule1, rule2))
```

Q_6:

```
val atom_tp1 = Atom(addPref(obsPref, "procedure"), Term("Obs"), Term("Sen"))
val atom_tp2 = Atom(addPref(rdfSyntaxPref, "type"), Term("Obs"), Term(addPref(whtPref, "
    RainfallObservation")))
val atom_tp3 = Atom(addPref(obsPref, "result"), Term("Obs"), Term("Res"))
val atom_tp4 = Atom(addPref(obsPref, "floatValue"), Term("Res"), Term("Value"))
val atom_tp5 = Atom(addPref(obsPref, "uom"), Term("Res"), Term("Uom"))
val atom_star1 = Atom(addTempPref("star1"), Term("Obs"), Term("Res"))
val atom_star2 = Atom(addTempPref("star2"), Term("Res"), Term("Uom"))
val atom_result = Atom(resIRI, Term("Obs"), Term("Uom"))

val rule1 = Rule(atom_star1, Set(atom_tp1, atom_tp2, atom_tp3), f_win(range, slide))
val rule2 = Rule(atom_star2, Set(atom_tp4, atom_tp5), f_win(range, slide))
val rule3 = Rule(atom_result, Set(atom_star1, atom_star2), f_win(range, slide))
Program(Set(rule1, rule2, rule3))
```

Q_7:

```
val atom_tp1 = Atom(addPref(ssnPref1, "observedBy"), Term("ObId"), Term(addPref(
    servicePref, "AarhusTrafficData182955")))
val atom_tp2 = Atom(addPref(saoPref, "hasAvgSpeed"), Term("ObId"), Term("AvgSpeed"))
val atom_tp3 = Atom(addPref(saoPref, "hasAvgMeasuredTime"), Term("ObId"), Term("
    AvgMeasuredTime"))
val atom_tp4 = Atom(addPref(saoPref, "hasVehicleCount"), Term("ObId"), Term("VehicleCount"
    ))
val atom_res = Atom(resIRI, Term("ObId"), Term("AvgSpeed"))

val rule = Rule(atom_res, Set(atom_tp1, atom_tp2, atom_tp3, atom_tp4), f_win(range, slide)
    )
```

Q_8:

```
 val atom_tp1 = Atom(addPref(ssnPref1, "observedBy"), Term("ObId"), Term(addPref(
     servicePref, "AarhusTrafficData182955")))
val atom_tp2 = Atom(addPref(saoPref, "status"), Term("ObId"), Term("Status"))
val atom_tp3 = Atom(addPref(saoPref, "hasAvgMeasuredTime"), Term("ObId"), Term("
    AvgMeasuredTime"))
val atom_tp4 = Atom(addPref(saoPref, "hasAvgSpeed"), Term("ObId"), Term("AvgSpeed"))
val atom_tp5 = Atom(addPref(saoPref, "hasExtID"), Term("ObId"), Term("ExtID"))
val atom_tp6 = Atom(addPref(saoPref, "hasMedianMeasuredTime"), Term("ObId"), Term("
    MedianMeasuredTime"))
val atom_tp7 = Atom(addPref(ssnPref1, "startTime"), Term("ObId"), Term("StartTime"))
val atom_tp8 = Atom(addPref(saoPref, "hasVehicleCount"), Term("ObId"), Term("VehicleCount"
    ))
val atom_star1 = Atom(addTempPref("star1"), Term("ObId"),Term("Status"), Term("
    AvgMeasuredTime"), Term("AvgSpeed"))
val atom_star2 = Atom(addTempPref("star2"), Term("ObId"), Term("ExtID"),Term("
    MedianMeasuredTime"), Term("StartTime"), Term("VehicleCount"))
val atom_res = Atom(resIRI, Term("ObId"),Term("Status"), Term("AvgMeasuredTime"), Term("
    AvgSpeed"), Term("ExtID"),Term("MedianMeasuredTime"), Term("StartTime"), Term("
    VehicleCount"))

val rule1 = Rule(atom_star1, Set(atom_tp1, atom_tp2, atom_tp3, atom_tp4), f_win(range,
    slide))
val rule2 = Rule(atom_star2, Set(atom_tp1, atom_tp5, atom_tp6, atom_tp7, atom_tp8), f_win(
    range, slide))
val rule3 = Rule(atom_res, Set(atom_star1, atom_star2), f_win(range, slide))
Program(Set(rule1, rule2, rule3))
```

Q_9:

```
val atom1 = Atom(resIRI, Term("Pub"), Term("Author"))
val atom2 = Atom(addPref(lubmPref, "publicationAuthor"), Term("Pub"), Term("Author"))
val atom3 = Atom(addPref(lubmPref, "publicationAuthor"), Term("Pub1"), Term("Author2"))
val atom4 = Atom(addPref(lubmPref, "publicationAuthor"), Term("Pub2"), Term("Author2"))
val atom5 = Atom(resIRI, Term("Pub2"), Term("Author1"))
val atom6 = Atom(resIRI, Term("Pub1"), Term("Author1"))
```

```
val rule1 = Rule(atom1, Set(atom2), f_win(range, slide))
val rule2 = Rule(atom5, Set(atom6, atom3, atom4), f_win(range, slide))
Program(Set(rule1, rule2))
```

Q_10:

```
val atomSubOrganizationOf_XY = Atom(addPref(lubmPref, "subOrganizationOf"), Term("X"),
    Term("Y"))
val atomBaseOrg = Atom(addTempPref("baseOrg"), Term("X"), Term("Y"))
val atomUpdateSubOrg_XY = Atom(addTempPref("updateSubOrg"), Term("X"), Term("Y"))
val atomUpdateSubOrg_XZ = Atom(addTempPref("updateSubOrg"), Term("X"), Term("Z"))
val atomUpdateSubOrg_YZ = Atom(addTempPref("updateSubOrg"), Term("Y"), Term("Z"))
val atomUpdateSubOrg_AB = Atom(addTempPref("updateSubOrg"), Term("A"), Term("B"))
val atomResult = Atom(resIRI, Term("A"), Term("B"))

val rule6 = Rule(atomBaseOrg, Set(atomSubOrganizationOf_XY), f_win(range, slide))
val rule7 = Rule(atomUpdateSubOrg_XY, Set(atomBaseOrg), f_win(range, slide))
val rule8 = Rule(atomUpdateSubOrg_XZ, Set(atomUpdateSubOrg_XY,atomUpdateSubOrg_YZ), f_win(
    range, slide))
val rule9 = Rule(atomResult, Set(atomUpdateSubOrg_AB), f_win(range, slide))

Program(Set(rule6, rule7, rule8, rule9))
```

Q_{11}:

```
val atom1_X = Atom(addPref(rdfSyntaxPref, "type"), Term("X"), Term(addPref(lubmPref, "
    GraduateStudent")))
val atom2_XY = Atom(addPref(lubmPref, "memberOf"), Term("X"), Term("Y"))
val atom3_SY = Atom(addPref(lubmPref, "undergraduateDegreeFrom"), Term("S"), Term("Y"))
val atom4_X = Atom(addPref(rdfSyntaxPref, "type"), Term("X"), Term(addPref(lubmPref, "
    University")))
val atom5_X = Atom(addPref(rdfSyntaxPref, "type"), Term("X"), Term("Department"))
val atom6_XY = Atom(addPref(lubmPref, "subOrganizationOf"), Term("X"), Term("Y"))
val atom6_YZ = Atom(addPref(lubmPref, "subOrganizationOf"), Term("Y"), Term("Z"))

val atomGraduateStudent = Atom(addTempPref("graduateStudent"), Term("X"))
val atomMemberOf_YX = Atom(addTempPref("memberOf"), Term("Y"), Term("X"))
val atomMemberOf_XZ = Atom(addTempPref("memberOf"), Term("X"), Term("Z"))
val atomUgDegreeFrom_YX = Atom(addTempPref("ugDegreeFrom"), Term("Y"), Term("X"))
val atomUgDegreeFrom_XY = Atom(addTempPref("ugDegreeFrom"), Term("X"), Term("Y"))
val atomUniv = Atom(addTempPref("univ"), Term("X"))
val atomUniv_Y = Atom(addTempPref("univ"), Term("Y"))
val atomDept = Atom(addTempPref("dept"), Term("X"))
val atomDept_Z = Atom(addTempPref("dept"), Term("Z"))
val atomBaseOrg = Atom(addTempPref("baseOrg"), Term("X"), Term("Y"))
val atomUpdateSubOrg_XY = Atom(addTempPref("updateSubOrg"), Term("X"), Term("Y"))
val atomUpdateSubOrg_XZ = Atom(addTempPref("updateSubOrg"), Term("X"), Term("Z"))
val atomUpdateSubOrg_YZ = Atom(addTempPref("updateSubOrg"), Term("Y"), Term("Z"))
val atomTemp1 = Atom(addTempPref("temp1"), Term("X"), Term("Y"))
val atomTemp2 = Atom(addTempPref("temp2"), Term("Y"))
val atomResult = Atom(resIRI, Term("X"), Term("Y"))

val rule1 = Rule(atomGraduateStudent, Set(atom1_X), f_win(range, slide))
val rule2 = Rule(atomMemberOf_YX, Set(atom2_XY), f_win(range, slide))
val rule3 = Rule(atomUgDegreeFrom_YX, Set(atom3_SY), f_win(range, slide))
val rule4 = Rule(atomUniv, Set(atom4_X), f_win(range, slide))
```

```
val rule5 = Rule(atomDept, Set(atom5_X), f_win(range, slide))
    //---------------------- Recursive Part ----------------------
val rule6 = Rule(atomBaseOrg, Set(atom6_XY))
val rule7 = Rule(atomUpdateSubOrg_XY, Set(atomBaseOrg), f_win(range, slide))
val rule8 = Rule(atomUpdateSubOrg_XZ, Set(atomUpdateSubOrg_XY,atomUpdateSubOrg_YZ), f_win(
    range, slide))
    //------------------------------------------------------------
val rule9 = Rule(atomTemp1, Set(atomGraduateStudent, atomMemberOf_XZ, atomUgDegreeFrom_XY)
    , f_win(range, slide))
val rule10 = Rule(atomTemp2, Set(atom6_YZ, atomUniv_Y, atomDept_Z), f_win(range, slide))
val rule11 = Rule(atomResult, Set(atomTemp1, atomTemp2), f_win(range, slide))
Program(Set(rule1, rule2, rule3, rule4, rule5, rule6, rule7, rule8, rule9, rule10, rule11)
    )
```

Q_{12}:

```
val atom_tp1 = Atom(addPref(qudtPref, "numericValue"), Term("S"), Term("O"))
val atom_res = Atom(resIRI, Term("S"), Term("O"))
val rule = Rule(atom_res, Set(atom_tp1))
Program(Set(rule))
```

Q_{13}:

```
val atom = Atom(addPref(obsPref, "procedure"), Term("Obs"), Term("Sen"))
val atom_result = Atom(resIRI, Term("Obs"), Term("Sen"))
val rule = Rule(atom_result, Set(atom))
Program(Set(rule))
```

Q_{14}:

```
val atom = Atom(addPref(ssnPref1, "observedBy"), Term("ObId"),Term(addPref(servicePref, "
    AarhusTrafficData182955")))
val atom_res = Atom(resIRI, Term("ObId"))
val rule = Rule(atom_res, Set(atom))
Program(Set(rule))
```

Q_{15}:

```
val atom1 = Atom(addPref(lubmPref, "publicationAuthor"), Term("Pub"), Term("Author"))
val atom_res = Atom(resIRI, Term("Pub"), Term("Author"))
val rule = Rule(atom_res, Set(atom1))
Program(Set(rule))
```

Bibliography

[1] K. Whitehouse, F. Zhao, and J. Liu, "Semantic streams: A framework for composable semantic interpretation of sensor data," in *EWSN*, vol. 3868 of *Lecture Notes in Computer Science*, pp. 5–20, Springer, 2006.

[2] "Linked data." https://www.w3.org/wiki/LinkedData, 2018.

[3] A. Gyrard, M. Serrano, J. B. Jares, S. K. Datta, and M. I. Ali, "Sensor-based linked open rules (S-LOR): an automated rule discovery approach for iot applications and its use in smart cities," in *Proceedings of the 26th International Conference on World Wide Web Companion, Perth, Australia, April 3-7, 2017*, pp. 1153–1159, 2017.

[4] T. Eiter, J. X. Parreira, and P. Schneider, "Detecting mobility patterns using spatial query answering over streams," in *WSP/WOMoCoE@ISWC*, vol. 1936 of *CEUR Workshop Proceedings*, pp. 17–32, CEUR-WS.org, 2017.

[5] M. B. Alaya and T. Monteil, "FRAMESELF: an ontology-based framework for the self-management of machine-to-machine systems," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 6, pp. 1412–1426, 2015.

[6] Z. Ush-Shamszaman and M. I. Ali, "Toward a smart society through semantic virtual-object enabled real-time management framework in the social internet of things," *IEEE Internet of Things Journal*, vol. 5, no. 4, pp. 2572–2579, 2018.

[7] A. Kamilaris, A. Pitsillides, F. X. Prenafeta-Boldu, and M. I. Ali, "A web of things based eco-system for urban computing - towards smarter cities," in *24th International Conference on Telecommunications, ICT 2017, Limassol, Cyprus, May 3-5, 2017*, pp. 1–7, 2017.

[8] Z. Ush-Shamszaman and M. I. Ali, "On the need for applications aware adaptive middleware in real-time RDF data analysis (short paper)," in *On the Move*

to Meaningful Internet Systems. OTM 2017 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2017, Rhodes, Greece, October 23-27, 2017, Proceedings, Part II, pp. 189–197, 2017.

[9] P. Patel, M. I. Ali, and A. P. Sheth, "On using the intelligent edge for iot analytics," *IEEE Intelligent Systems*, vol. 32, no. 5, pp. 64–69, 2017.

[10] "Waves fui 17." www.waves-rsp.org, 2015.

[11] M. Arenas, C. Gutiérrez, and J. Pérez, "On the semantics of SPARQL," in *Semantic Web Information Management*, pp. 281–307, Springer, 2009.

[12] "Esper." http://www.espertech.com/, 2013.

[13] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds, "Sparql basic graph pattern optimization using selectivity estimation," in *WWW*, 2008.

[14] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, "Jena: Implementing the semantic web recommendations," in *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers &Amp; Posters*, WWW Alt. '04, (New York, NY, USA), pp. 74–83, ACM, 2004.

[15] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: A generic architecture for storing and querying RDF and RDF schema," in *International Semantic Web Conference*, vol. 2342 of *Lecture Notes in Computer Science*, pp. 54–68, Springer, 2002.

[16] O. Erling and I. Mikhailov, "Virtuoso: RDF support in a native RDBMS," in *Semantic Web Information Management*, pp. 501–519, Springer, 2009.

[17] T. Neumann and G. Weikum, "x-rdf-3x: Fast querying, high update rates, and consistency for RDF databases," *PVLDB*, vol. 3, no. 1, pp. 256–263, 2010.

[18] B. Motik, Y. Nenov, R. Piro, I. Horrocks, and D. Olteanu, "Parallel materialisation of datalog programs in centralised, main-memory RDF systems," in *AAAI*, pp. 129–137, AAAI Press, 2014.

[19] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen, "S2rdf: Rdf querying with sparql on spark," *PVLDB.*, 2016.

[20] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, "A distributed graph engine for web scale rdf data," in *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pp. 265–276, VLDB Endowment, 2013.

[21] B. Shao, H. Wang, and Y. Li, "Trinity: a distributed graph engine on a memory cloud," in *SIGMOD Conference*, pp. 505–516, ACM, 2013.

[22] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, "Triad: A distributed shared-nothing rdf engine based on asynchronous message passing," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), pp. 289–300, ACM, 2014.

[23] L. Galárraga, K. Hose, and R. Schenkel, "Partout: A distributed engine for efficient rdf processing," in *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, (New York, NY, USA), pp. 267–268, ACM, 2014.

[24] M. Hammoud, D. A. Rabbou, R. Nouri, S.-M.-R. Beheshti, and S. Sakr, "Dream: Distributed rdf engine with adaptive query planner and minimal communication," *Proc. VLDB Endow.*, vol. 8, pp. 654–665, Feb. 2015.

[25] H. Kim, P. Ravindra, and K. Anyanwu, "From SPARQL to mapreduce: The journey using a nested triplegroup algebra," *PVLDB*, vol. 4, no. 12, pp. 1426–1429, 2011.

[26] F. Maali, P. Ravindra, K. Anyanwu, and S. Decker, "Syrql: A dataflow language for large scale processing of RDF data," in *International Semantic Web Conference (1)*, vol. 8796 of *Lecture Notes in Computer Science*, pp. 147–163, Springer, 2014.

[27] A. Schätzle, M. Przyjaciel-Zablocki, T. Hornung, and G. Lausen, "Pigsparql: A SPARQL query processing baseline for big data," in *International Semantic Web Conference (Posters & Demos)*, vol. 1035 of *CEUR Workshop Proceedings*, pp. 241–244, CEUR-WS.org, 2013.

[28] P. Ravindra, H. Kim, and K. Anyanwu, "Optimization of complex SPARQL analytical queries," in *EDBT*, pp. 257–268, OpenProceedings.org, 2016.

[29] A. Schätzle, M. Przyjaciel-Zablocki, T. Berberich, and G. Lausen, "S2X: graph-parallel querying of RDF with graphx," in *Biomedical Data Management and Graph Online Querying - VLDB 2015 Workshops, Big-O(Q) and DMAH,*

Waikoloa, HI, USA, August 31 - September 4, 2015, Revised Selected Papers, pp. 155–168, 2015.

[30] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *MSST*, pp. 1–10, IEEE Computer Society, 2010.

[31] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.

[32] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *OSDI*, pp. 599–613, USENIX Association, 2014.

[33] O. W. Group, "RDF Schema 1.1." `https://www.w3.org/TR/rdf-schema/`, 2014.

[34] O. W. Group, "Web Ontology Language." `https://en.wikipedia.org/wiki/Web_Ontology_Language`, 2009. [Online; accessed 2012-12-11].

[35] S. Muñoz, J. Pérez, and C. Gutiérrez, "Minimal deductive systems for RDF," in *ESWC*, vol. 4519 of *Lecture Notes in Computer Science*, pp. 53–67, Springer, 2007.

[36] Y. Guo, Z. Pan, and J. Heflin, "Lubm: A benchmark for owl knowledge base systems," *Journal of Web Semantics*, 2005.

[37] C. Gutiérrez, C. A. Hurtado, and A. A. Vaisman, "Introducing time into RDF," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 2, pp. 207–218, 2007.

[38] N. Lopes, A. Polleres, U. Straccia, and A. Zimmermann, "Anql: Sparqling up annotated RDFS," in *International Semantic Web Conference (1)*, vol. 6496 of *Lecture Notes in Computer Science*, pp. 518–533, Springer, 2010.

[39] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic, "Stream reasoning and complex event processing in ETALIS," *Semantic Web*, vol. 3, no. 4, pp. 397–407, 2012.

[40] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic, "EP-SPARQL: a unified language for event processing and stream reasoning," in *WWW*, pp. 635–644, ACM, 2011.

[41] A. Arasu, S. Babu, and J. Widom, "CQL: A language for continuous queries over streams and relations," in *Database Programming Languages, 9th International Workshop, DBPL 2003*, pp. 1–19, 2003.

[42] I. Botan, R. Derakhshan, N. Dindar, L. M. Haas, R. J. Miller, and N. Tatbul, "SECRET: A model for analysis of the execution semantics of stream processing systems," *PVLDB*, vol. 3, no. 1, pp. 232–243, 2010.

[43] D. L. Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth, "A native and adaptive approach for unified processing of linked streams and linked data," in *International Semantic Web Conference (1)*, vol. 7031 of *Lecture Notes in Computer Science*, pp. 370–388, Springer, 2011.

[44] T. Eiter, G. Ianni, and T. Krennwallner, "Answer set programming: A primer," in *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures*, pp. 40–110, 2009.

[45] H. Beck, M. Dao-Tran, T. Eiter, and M. Fink, "LARS: A logic-based framework for analyzing reasoning over streams," in *AAAI*, 2015.

[46] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica, "Drizzle: Fast and adaptable stream processing at scale," in *SOSP*, pp. 374–389, ACM, 2017.

[47] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *PVLDB*, 2015.

[48] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy, "Storm@twitter," in *SIGMOD Conference*, pp. 147–156, ACM, 2014.

[49] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink^{TM}: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, 2015.

[50] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the*

Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, (New York, NY, USA), pp. 423–438, ACM, 2013.

[51] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus, "Continuous queries and real-time analysis of social semantic data with c-sparql," in *SDoW2009*, 2009.

[52] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: A stream data management benchmark," in *VLDB*, pp. 480–491, Morgan Kaufmann, 2004.

[53] Y. Zhang, M. Pham, Ó. Corcho, and J. Calbimonte, "Srbench: A streaming RDF/SPARQL benchmark," in *International Semantic Web Conference (1)*, vol. 7649 of *Lecture Notes in Computer Science*, pp. 641–657, Springer, 2012.

[54] D. L. Phuoc, M. Dao-Tran, M. Pham, P. A. Boncz, T. Eiter, and M. Fink, "Linked stream data processing engines: Facts and figures," in *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference*, ISWC'11, 2012.

[55] D. Dell'Aglio, J. Calbimonte, M. Balduini, Ó. Corcho, and E. D. Valle, "On correctness in RDF stream processor benchmarking," in *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II*, pp. 326–342, 2013.

[56] M. I. Ali, F. Gao, and A. Mileo, "Citybench: A configurable benchmark to evaluate RSP engines using smart city datasets," in *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II*, pp. 374–389, 2015.

[57] M. Kolchin, P. Wetz, E. Kiesling, and A. M. Tjoa, "Yabench: A comprehensive framework for RDF stream processor correctness and performance assessment," in *ICWE*, vol. 9671 of *Lecture Notes in Computer Science*, pp. 280–298, Springer, 2016.

[58] D. L. Phuoc, J. X. Parreira, and M. Hauswirth, "Linked stream data processing," in *Reasoning Web. Semantic Technologies for Advanced Query Answering - 8th International Summer School 2012, Vienna, Austria, September 3-8, 2012. Proceedings*, pp. 245–289, 2012.

[59] "Esper complex event processing and streaming analytics." http://www.w3.org/TR/sparql11-query/, 2018.

[60] D. Anicic, P. Fodor, S. Rudolph, R. Stühmer, N. Stojanovic, and R. Studer, "A rule-based language for complex event processing and reasoning," in *RR*, vol. 6333 of *Lecture Notes in Computer Science*, pp. 42–57, Springer, 2010.

[61] J. Calbimonte, Ó. Corcho, and A. J. G. Gray, "Enabling ontology-based access to streaming data sources," in *The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I*, pp. 96–111, 2010.

[62] I. Galpin, C. Y. A. Brenninkmeijer, F. Jabeen, A. A. A. Fernandes, and N. W. Paton, "An architecture for query optimization in sensor networks," in *ICDE*, pp. 1439–1441, IEEE Computer Society, 2008.

[63] D. L. Phuoc, H. N. M. Quoc, C. L. Van, and M. Hauswirth, "Elastic and scalable processing of linked stream data in the cloud," in *International Semantic Web Conference (1)*, vol. 8218 of *Lecture Notes in Computer Science*, pp. 280–297, Springer, 2013.

[64] L. Fischer, T. Scharrenbach, and A. Bernstein, "Scalable linked data stream processing via network-aware workload scheduling," in *SSWS@ISWC*, vol. 1046 of *CEUR Workshop Proceedings*, pp. 81–96, CEUR-WS.org, 2013.

[65] K. Lab, "Serial Graph Partitioning and Fill-reducing Matrix Ordering." `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`, 2016. [Online; accessed 2013-3-30].

[66] E. D. Valle, S. Ceri, D. F. Barbieri, D. Braga, and A. Campi, "A first step towards stream reasoning," in *FIS*, vol. 5468 of *Lecture Notes in Computer Science*, pp. 72–81, Springer, 2008.

[67] X. Ren, O. Curé, H. Naacke, J. Lhez, and L. Ke, "Strider$^{\mathrm{r}}$: Massive and distributed RDF graph stream reasoning," in *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*, pp. 3358–3367, 2017.

[68] E. Thomas, J. Z. Pan, and Y. Ren, "TrOWL: Tractable OWL 2 Reasoning Infrastructure," in *the Proc. of the Extended Semantic Web Conference (ESWC2010)*, 2010.

[69] J. Z. Pan, Y. Ren, and Y. Zhao, "Tractable approximate deduction for OWL," *Artif. Intell.*, vol. 235, pp. 95–155, 2016.

[70] Y. Kazakov, M. Krötzsch, and F. Simančík, "The incredible elk," 2014.

[71] B. Glimm, I. Horrocks, B. Motik, and G. Stoilos, "Optimising ontology classification," in *ISWC*, pp. 225–240, 2010.

[72] A. Mileo, A. Abdelrahman, S. Policarpio, and M. Hauswirth, "Streamrule: A nonmonotonic stream reasoning system for the semantic web," in *RR*, 2013.

[73] T. Pham, A. Mileo, and M. I. Ali, "Towards scalable non-monotonic stream reasoning via input dependency analysis," in *33rd ICDE 2017*, 2017.

[74] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Clingo = ASP + control: Preliminary report," *CoRR*, vol. abs/1405.3694, 2014.

[75] H. R. Bazoobandi, H. Beck, and J. Urbani, "Expressive stream reasoning with laser," in *ISWC*, 2017.

[76] H. Beck, T. Eiter, and C. Folie, "Ticker: A system for incremental asp-based stream reasoning," *TPLP*, 2017.

[77] S. Gao, D. Dell'Aglio, J. Z. Pan, and A. Bernstein, "Distributed stream consistency checking," in *Web Engineering - 18th International Conference, ICWE 2018, Cáceres, Spain, June 5-8, 2018, Proceedings*, pp. 387–403, 2018.

[78] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, "Design and implementation of the logicblox system," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 1371–1382, ACM, 2015.

[79] T. J. Green, "Logiql: A declarative language for enterprise applications," in *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '15, (New York, NY, USA), pp. 59–64, ACM, 2015.

[80] J. Wang, M. Balazinska, and D. Halperin, "Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines," *PVLDB*, 2015.

[81] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo, "Big data analytics with datalog queries on spark," SIGMOD, 2016.

[82] M. Stonebraker and L. A. Rowe, "The design of postgres," in *SIGMOD Conference*, pp. 340–355, ACM Press, 1986.

[83] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus, "Querying RDF streams with C-SPARQL," *SIGMOD Record*, vol. 39, no. 1, pp. 20–26, 2010.

[84] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *VLDB*, pp. 411–422, ACM, 2007.

[85] T. Neumann and G. Moerkotte, "Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins," in *ICDE*, 2011.

[86] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. Boncz, "Heuristics-based query optimisation for sparql," in *EDBT*, 2012.

[87] A. Gubichev and T. Neumann, "Exploiting the query structure for efficient join ordering in sparql queries," in *EDBT*, 2014.

[88] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[89] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *USENIX*, 2010.

[90] X. Ren, H. Khrouf, Z. Kazi-Aoul, Y. Chabchoub, and O. Curé, "On measuring performances of C-SPARQL and CQELS," in *SWIT@ISWC*, 2016.

[91] H. Halpin, P. J. Hayes, J. P. McCusker, D. L. McGuinness, and H. S. Thompson, "When owl: sameas isn't the same: An analysis of identity in linked data," in *ISWC*, pp. 305–320, 2010.

[92] O. Curé and G. Blin, *RDF Database Systems: Triples Storage and SPARQL Query Processing*. Morgan Kaufmann Publishers Inc., 1st ed., 2014.

[93] O. Curé, H. Naacke, T. Randriamalala, and B. Amann, "Litemat: A scalable, cost-efficient inference encoding scheme for large RDF graphs," in *IEEE Big Data*, pp. 1823–1830, 2015.

[94] J. Greiner, "A comparison of parallel algorithms for connected components," in *ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pp. 16–25, ACM, 1994.

[95] B. Chin, D. von Dincklage, V. Ercegovac, P. Hawkins, M. S. Miller, F. Och, C. Olston, and F. Pereira, "Yedalog: Exploring knowledge at scale," in *1st Summit on Advances in Programming Languages (SNAPL 2015)*, (Dagstuhl, Germany), pp. 63–78, 2015.

[96] H. Beck, "Expressive rule-based stream reasoning," in *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pp. 4341–4342, 2015.

[97] B. Motik, Y. Nenov, R. Piro, I. Horrocks, and D. Olteanu, "Parallel materialisation of datalog programs in centralised, main-memory RDF systems," in *AAAI*, pp. 129–137, AAAI Press, 2014.

[98] S. Perri, F. Ricca, and M. Sirianni, "Parallel instantiation of ASP programs: techniques and experiments," *TPLP*, 2013.

[99] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.

[100] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," ICDE '13, 2013.

[101] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," SOSP, 2013.

[102] X. Ren and O. Curé, "Strider: A hybrid adaptive distributed RDF stream processing engine," in *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I*, pp. 559–576, 2017.