POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN COMPUTER SCIENCE AND ENGINEERING

ON UNIFIED STREAM REASONING

Doctoral Dissertation of:
Daniele Dell'Aglio

Supervisor:
Prof. Emanuele Della Valle

Tutor:
Prof. Stefano Ceri

The Chair of the Doctoral Program:
Prof. Andrea Bonarini

2016 – Cycle XXVIII

Acknowledgments

I would like to express gratitude to my advisor, Prof. Emanuele Della Valle. He has supported me in those years, since the decision to start my Ph.D. His patience, his suggestions and his knowledge helped me in the development of my thesis.

I would like to thank the reviewers of my thesis: Prof. Frank van Harmelen and Prof. Axel Polleres. Their comments, insights and recommendation offered different points of view and contributed to improve the quality of the work.

I am thankful to all the co-authors of my publications. I feel very lucky for the opportunity to meet and interact with so brilliant researchers. I did my best to learn as much as possible from each of them, and they have key roles in my growth as researcher.

My gratitude goes also to Fiorenzo, Lorenzo, Matteo, Fabrizio, Marina and Andrea. Their effort in reminding me that the development of the Ph.D. is not the only thing to do is exemplary, and they do not deserve all the times I ignored the advise.

A special thank goes to Soheila. She helped me in finding equilibrium, and she has always been there when I needed to talk and discuss about my problems and the huge amount of concerns that I had.

Finally, saying that I am grateful to my parents is not enough to express what I feel. They have been always there, and their continuous support allowed me to work on my Ph.D. in the best possible conditions.

Abstract

The real-time integration of huge volumes of dynamic data from heterogeneous sources is getting more and more attention, as the number of data-stream sources is keeping growing and changing at very high pace. Cities and the Internet of Things are perfect illustrations of such need. For instance, in the urban setting, semantic interpretation of road sensors and social networks can supply (directly and indirectly) continuous and up-to-date information about the traffic causes and their impacts, the progress of city-scale events or the trending activities around a user. While Data Stream and Event Processing deal with data streams and reactiveness, reasoning is a potential solution for the data heterogeneity: ontologies are key to access the data streams from the different sources and to make explicit hidden information. Stream Reasoning aims at bringing together those areas, with techniques to perform continuous reasoning tasks over data streams.

In this context, the problem I investigate is how to unify the current Stream Reasoning techniques, as they substantially differ from each others. This fact is evident when these techniques are designed to reach different goals, e.g. aggregating data in the stream vs. detecting events. However, it happens even when they perform the same task and final users may expect the same behaviour. Understanding peculiarities and common points is mandatory in order to compare, contrast and integrate them.

My research begins with the analysis of the state of the art in the area of Stream Reasoning, and in particular RDF Stream Processing

(RSP), i.e. systems that focus on the continuous query answering task. Next, I build a formal model to capture their behaviour and their evaluation semantics. I proceed iteratively starting with a core set of features from Data Stream Processing and Semantic Web and, next, extending that by integrating concepts from Complex Event Processing and reasoning.

The main outcome of my research is RSEP-QL, a formal reference model to describe the evaluation semantics of Stream Reasoning systems in the context of continuous query answering. RSEP-QL extends SPARQL by adding operators to manage streams such as sliding windows (also known as RSP-QL fragment of RSEP-QL) and event patterns. Similarly to SPARQL, RSEP-QL works under entailment regimes, which introduce deductive inference in the continuous query answering process.

I show the value of RSEP-QL through an application in the area of comparative testing. I formalise a notion of correctness of the query answering process with regards to RSP-QL. The definition is at the basis of CSRBench, an extension of the SRBench benchmark to assess the correctness of existing RDF Stream Processing operators. CSR-Bench is composed by input data streams, continuous queries and an oracle that automatically verify if an answer provided by a system is correct.

Sommario

L'integrazione in tempo reale di enormi flussi di dati da fonti etereogenee sta diventando un bisogno sempre più centrale nella realizzazione di servizi avanzati.

Gli scenari delle smart city e dell'Internet of Things esemplificano alla perfezione questo bisogno. Nell'ambiente urbano, combinare i dati che vengono prodotti dalla città ha un grande valore: dai sensori che rilevano i passaggi d'auto fino ai messaggi sui social network dei cittadini. L'integrazione e l'elaborazione di questi dati può portare allo sviluppo di nuovi sistemi per studiare il traffico, per monitorare l'evoluzione di eventi di larga scala o per scoprire quali sono le attività di tendenza in corso.

Se da un lato le tecniche di Data Stream Processing e Complex Event Processing offrono soluzioni per gestire questi flussi di dati in maniera reattiva, dall'altro le tecniche di reasoning sono una base per gestire l'eterogeneitá di questi dati. L'utilizzo di ontologie abilita l'accesso ai flussi di dati esposti dalle diverse sorgenti, esplicitando le informazioni nascoste in essi. L'area di ricerca dello Stream Reasoning studia come combinare le tecniche di queste aree, con soluzioni per eseguire reasoning in maniera continua sui flussi di dati.

Il problema che affronto in questa tesi è come unificare le attuali tecniche di Stream Reasoning. Capita infatti che queste tecniche possano essere molto diverse le une dalle altre. Ciò è evidente quando i compiti che svolgono sono differenti (ad esempio aggregare dati o identificare sequenze rilevanti di eventi), ma può accadere anche quando gli

obiettivi sono comuni e ci si potrebbe quindi attendere comportamenti simili. Capire le peculiarità e in punti in comune è importante per poter confrontare e integrare queste soluzioni.

La mia attività di ricerca inizia con un'analisi dello stato dell'arte nell'area dello Stream Reasoning e in particolare in quella dell'RDF Stream Processing (RSP), sistemi che valutano query in maniera continua all'arrivare di nuove informazioni sui flussi di dati. Successivamente, la tesi costruisce un modello formale per catturare la semantica operazionale e il comportamento di tali sistemi. Per fare ciò, segue un approccio iterativo, iniziando con un insieme di concetti base di Data Stream Processing e Semantic Web, per poi integrare i concetti di Complex Event Processing.

Il risultato principale della mia ricerca è RSEP-QL, un modello di riferimento per descrivere la semantica operazionale dei sistemi di Stream Reasoning nel contesto di compiti di interrogazione continua. RSEP-QL estende SPARQL aggiungendo operatori per gestire finestre e pattern di eventi. Come SPARQL, RSEP-QL opera considerando gli entailment regime, che introducono processi di inferenza deduttiva nel calcolo delle risposte.

Per mostrare il valore di RSEP-QL, la tesi presenta un'applicazione nel dominio del test comparativo. Dopo aver formalizzato la nozione di correttezza per un frammento di RSEP-QL, costruisce CSRBench, un'estensione del benchmark SRBench. L'obiettivo di CSRBench è quello di verificare il corretto funzionamento dei sistemi basati sulle finestre di tipo sliding. CSRBench è composto da un insieme di dati in ingresso, da una serie di query continue e da un sistema (chiamto oracolo) per verificare automaticamente se la risposta fornita dal sistema è corretta.

Contents

CHAPTER 1

Introduction

Logical reasoning in real time on multiple, heterogeneous, gigantic and inevitably noisy data streams in order to support the decision process of extremely large numbers of concurrent users.

With this sentence, Stuckenschmidt, Ceri, Della Valle and van Harmelen defined the notion of Stream Reasoning [98]. The goal of this research trend, started in 2008 [45], is to study if and how reasoning techniques are valid solutions to solve the problem of *integrating huge amounts of rapidly changing and heterogeneous data in real time.*

This problem is contemporary and present in a multitude of use cases, such as Smart Cities, Social Media analytics and Internet of Things. In the Smart City context, where different stakeholders supply data, as traffic sensor detections, public transport service statuses and user positions, data integration plays a key role in order to build novel and advanced services. The idea behind Social Media analytics is to listen and capture the thoughts of the crowd through social networks and blogs, inferring new and non-trivial knowledge to improve existing solutions such as recommender systems, surveys and big scale event monitors. The rise of the Internet of Things [10] stressed even more the problem, as it requires interconnection of embedded computing devices and sensors, and integration of large, high-speed data streams across the Web.

Stream Reasoning aims at tackling the integration of high dynamic and heterogeneous data by putting together Data Stream and Complex Event Processing [39] with Semantic Web [26]: while the former ones proved to be valid solutions to cope with data velocity, the latter shown to be a valid solution to integrate heterogeneous data on a Web scale.

Even if the goal can be easily stated, bridging those two worlds together is not straightforward: there are multiple ways on which these approaches can be combined. The combination can bring problems at both theoretical level – to create comprehensive data and processing models – and practical level – to guarantee the reactiveness, satisfying the real time computation requirements.

Since 2008, Stream Reasoning has moved forward [78]. Different research groups proposed (i) data models and vocabularies to capture data streams through RDF and ontologies [21, 65]; (ii) continuous query models, languages and prototypes [6, 22, 31, 73], inspired by SPARQL [63] and collected under the *RDF Stream Processing* (RSP) label; (iii) extensions of the reasoning tasks over streams, as consistency check and closure [74, 93**?**]; (iv) applications built on top of the aforementioned results [23, 68, 99].

The growth of Stream Reasoning brought several positive results, but also some drawbacks that can bring new challenge to be addressed in the future research.

Most of the solutions have been designed and proposed w.r.t. different use cases and input data. Rigorous comparisons are missing, and as a result, the connections among current approaches are missing as well. This fact drastically limits the possibility to determine when one approach is preferable to another one. Moreover, it is an obstacle to interoperability, making hard, if not impossible, to let these different solutions interoperate in complex architectures.

Even when solutions seems to be comparable, i.e. they operate on the same input, the query languages are similar and consequently the user may expect the the same outputs, we observe that actually it does not happen. In other words, two RSP engines with similar query models can produce different results given the same continuous query that processes the same data stream. And, surprisingly, the same system can produce different results while processing the same input in different runs.

The main goal of this research is to study how to *unify* current Stream Reasoning results. A comprehensive framework to describe continuous query answering and reasoning over stream of RDF data is missing. In addition to comparison and interoperability, mentioned above, such framework would be a basis to study RDF Stream Processing related problems and a candidate to build a standard RSP query language.

1.1 Research Questions

To investigate this problem, hard to tackle in its entirety, we asked ourselves two main research questions, related to different but still connected) parts of the problem. The first question states:

RQ.1 How can the behaviour of existing RDF Stream Processing systems be captured and compared when reasoning processes are not involved?

At the beginning of the research, we work under the assumption of absence of reasoning tasks. SPARQL, the language for querying RDF repositories, defines this setting as query answering under the *simple* entailment regime [59]. The goal is to verify if it is possible to capture in a common formalism the semantics of existing RDF Stream Processing engines: if we do not succeed in absence of inference processes, it follows that we cannot do it when we relax this constraint. We break down **RQ.1** in the two following sub questions:

RQ.1.1 Is it possible to create a formal continuous query model that can be used to describe existing RSP systems based on Data Stream Processing features?

RQ.1.2 Is it possible to create a formal continuous query model that can be used to describe existing RSP systems based on Complex Event Processing features?

The answer of those questions require to put together elements from Data Stream Processing, Complex Event Processing, SPARQL semantics and current state of the art on RDF Stream Processing. If both answers are affirmative, it is possible to design a *reference model* that captures the query models of the different systems.

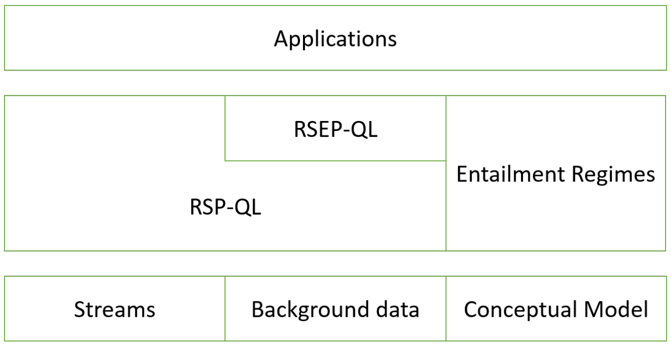

Figure 1.1: *The RSEP-QL model*

At this point, we can move a step forward and consider entailment regimes different from the simple one. We do it through the following question:

RQ.2 What is the correct behaviour of a continuous query engine while processing a semantic stream under entailment regimes?

To investigate the question, we investigate how SPARQL entailment regime can be used in this context of continuous query answering over semantics streams.

1.2 Summary of Contributions

This section summarizes the main outcomes of the thesis.

RSEP-QL. RSEP-QL is a reference model to continuously query RDF streams. It extends SPARQL by introducing operators to process and produce streaming data.

RSEP-QL is built in a modular way, as depicted in Figure 1.1. The left block, RSP-QL (Chapter 5), is a model that defines the continuous evaluation semantics; the sliding windows – a typical data Stream Processing operator to manage subsequences of the stream and limit the amount of data to be queried; a new dataset – that extends SPARQL dataset to include also streams; a set of streaming operators – used to build the output stream. The model extends the SPARQL semantics and is inspired by two Data Stream Processing models, CQL [9] and SECRET [29].

RSEP-QL (Chapter 6), extends RSP-QL by adding Complex Event Processing operators. That means, it defines (i) a new window operator, the landmark window, used to have a large view over the underlying stream; and (ii) event pattern operators to identify complex events on the stream. The evaluation semantics of the new operators is built on the results of RSP-QL.

Finally, the block on the right defines the entailment regimes in RSEP-QL (Chapter 7). It extends the SPARQL entailment regimes to take into account the presence of streams in the dataset and of event pattern operators. We adopt the ontology stream notion [66] to capture the portion of the stream to be considered and to describe the inference processes w.r.t. a conceptual model. At the same time, RSEP-QL entailment regimes maintain the compatibility with description logics without time extensions, e.g. $\mathcal{EL}^{++}$ and $DL - Lite^{\mathcal{H}}_{core}$. This fact is important with regards to computational complexity.

CSRBench. To supply evidence on the fact that RSEP-QL captures the semantics of existing systems, we use it in the context of correctness assessment (Chapter 9). We show that RSP-QL captures the semantics of a set of existent RDF Stream Processing solutions, and we propose CSRBench, a benchmark that extends SRBench [104] with correctness tests. We also designed and implemented an open source framework to automatically run the correctness tests.

Triple Wave. To fill an existent gap on RDF stream availability on the Web, the last contribution is Triple Wave (Chapter 4), a reusable and generic tool to spread and exchange RDF streams on the Web. The features of Triple Wave have been derived from requirements of real use-cases, and consider a diverse set of scenarios, independent of any specific RSP implementation. Triple Wave is fed with existing Web streams (e.g. Twitter and Wikipedia streams) It can also be invoked through both pull- and push-based mechanisms, thus enabling RSP engines to automatically register and receive data from Triple Wave.

1.3 Outline

This thesis is organised in three parts.

In the first part, *Problem Setting*, we set the basis for the research. In Chapter 2, we present the basic concepts on Semantic Web and reasoning that are used in the rest of the document: RDF, SPARQL, description logics and inference processes. Next, in Chapter 3, we review the state of the art, from Data Stream and Complex Event Processing to Stream Reasoning, where we describe different paradigms to process data streams, as querying and deductive reasoning.

In the second part, titled *A Reference Model for RDF Stream Processing*, we provide the core contribution of the thesis and we investigate the two Research Questions **RQ.1** and **RQ.2**. In Chapter 4, we add the time dimension in the RDF data model in two ways: *RDF streams* to model rapidly changing information; *time-varying RDF graphs* to model quasi-static RDF graph and their evolution over time. We also propose Triple Wave, a system to publish RDF streams on the Web. In Chapters 5 and 6, we describe respectively RSP-QL and RSEP-QL, introduced in the previous section. We then study the Research Question **RQ.2** in Chapter 7: we complete the RSEP-QL stack in Figure 1.1 by introducing the non-simple entailment regimes.

The third part, *Effectiveness of RSEP-QL: coverage and testing*, focuses on studying if RSEP-QL addresses the Research Question **RQ.1**. Chapter 8 presents a qualitative analysis about the coverage of RSEP-QL w.r.t. exiting RDF Stream Processing query languages and engines. Chapter 9 describes CSRBench, where we formalise the notion of correct answer with regards to RSP engines captured by RSP-QL. Moreover, we present the CSRBench data and queries, our test framework implementation, named *oracle*, and our main findings in applying CSRBench to existing RSP implementations.

Chapter 10 closes with a review of the research questions, a description of the future work and final remarks.

1.4 Publications

This thesis is based on the articles [17, 19, 42, 43, 47, 48, 49, 50, 51, 52, 79], listed above. I describe my contribution in each article at the beginning of the chapters.

- Daniele Dell’Aglio, Marco Balduini and Emanuele Della Valle: “Applying Semantic Interoperability Principles to Data Stream Management”. Data Management in Pervasive Systems, 2015: 135-166.

- Daniele Dell’Aglio, Jean-Paul Calbimonte, Emanuele Della Valle, Óscar Corcho: “Towards a Unified Language for RDF Stream Query Processing”. ESWC (Satellite Events) 2015: 353-363

- Andrea Mauri, Jean-Paul Calbimonte, Daniele Dell'Aglio, Marco Balduini, Emanuele Della Valle, Karl Aberer: "Where Are the RDF Streams? On Deploying RDF Streams on the Web of Data with TripleWave". International Semantic Web Conference (Posters & Demos) 2015
- Soheila Dehghanzadeh, Daniele Dell'Aglio, Shen Gao, Emanuele Della Valle, Alessandra Mileo, Abraham Bernstein: "Approximate Continuous Query Answering over Streams and Dynamic Linked Data Sets". ICWE 2015: 307-325
- Soheila Dehghanzadeh, Daniele Dell'Aglio, Shen Gao, Emanuele Della Valle, Alessandra Mileo, Abraham Bernstein: "Online View Maintenance for Continuous Query Evaluation". WWW (Posters & Demos) 2015: 25-26
- Daniele Dell'Aglio, Emanuele Della Valle, Jean-Paul Calbimonte, Óscar Corcho: "RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems". Int. J. Semantic Web Inf. Syst. 10(4): 17-44 (2014)
- Marco Balduini, Irene Celino, Daniele Dell'Aglio, Emanuele Della Valle, Yi Huang, Tony Kyung-il Lee, Seon-Ho Kim, Volker Tresp: "Reality mining on micropost streams - Deductive and inductive reasoning for personalized and location-based recommendations". Semantic Web 5(5): 341-356 (2014)
- Daniele Dell'Aglio, Emanuele Della Valle: "Incremental Reasoning on RDF Streams". Linked Data Management, 2014: 413-435
- Daniele Dell'Aglio, Jean-Paul Calbimonte, Marco Balduini, Óscar Corcho, Emanuele Della Valle: "On Correctness in RDF Stream Processor Benchmarking". International Semantic Web Conference (2) 2013: 326-342
- Marco Balduini, Emanuele Della Valle, Daniele Dell'Aglio, Mikalai Tsytsarau, Themis Palpanas, Cristian Confalonieri: "Social Listening of City Scale Events Using the Streaming Linked Data Framework". International Semantic Web Conference (2) 2013: 1-16
- Daniele Dell'Aglio: "Ontology-Based Top-k Query Answering over Massive, Heterogeneous, and Dynamic Data". ISWC-DC 2013: 17-24

Part I

Problem Setting

CHAPTER 2

Preliminary concepts: Managing Semantic Data on the Web

2.1 RDF and SPARQL

RDF is a W3C recommendation for data interchange on the Web [40]. RDF data is structured as directed labelled graphs, where the nodes are *resources*, and the edges represent relations among them. Each node of the graph can be a named resource (identified by an IRI), an anonymous resource (a blank node) or a literal. We identify with I, B and L respectively the sets of IRIs, blank nodes and literals. We define an *RDF term* as an element of the set $I \cup B \cup L$.

Definition 2.1 (RDF statement and RDF graph)**.** *An* RDF statement *d is a triple $(s, p, o) \in (I \cup B) \times (I) \times (I \cup B \cup L)$. A set of RDF statements is an* RDF graph*.*

Example 1. *The Sirius Cybernetics Corporation offers real-time geo-marketing services to shop owners to increase their sales by distributing instantaneous discount coupons to potential shoppers nearby. Alice and Bob, who respectively own shops A and B, decided to try that service. We can represent those facts in the following RDF graph G_{shops}:*

```
:a      rdf:type  :Shop .
:b      rdf:type  :Shop .
:alice  :owns     :a .
:bob    :owns     :b .
```

Given an RDF graph, it is possible to query it through the SPARQL query language [63], another W3C Recommendation. A SPARQL query typically contains one or more triple patterns called a basic graph pattern. Triple patterns are similar to RDF triples except that they may contain variables in place of resources. These patterns may match a subgraph of the RDF data, by substituting variables with RDF terms, resulting in an equivalent RDF subgraph.

Definition 2.2 (Triple pattern and Basic Graph Pattern)**.** *A* triple pattern *tp is a triple (sp, pp, op) such that*

$$(sp, pp, op) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V),$$

where V is the infinite set of variables. A basic graph pattern *is a set of triple patterns.*

Graph patterns in a SPARQL query can include basic graph patterns and other compound expressions defined recursively as[1]:

1. A set of triple patterns is a basic graph pattern;
2. If P_1 and P_2 are graph patterns, then (P_1 AND P_2), (P_1 OPT P_2) and (P_1 UNION P_2) are graph patterns;
3. If P is a graph pattern and u is a symbol in IUV, (GRAPH u P) and (SERVICE u P) are graph patterns;
4. If P is a graph pattern and R is a SPARQL built-in condition, then (P FILTER R) is a graph pattern.

A SPARQL built-in condition is composed of elements of the set $I \cup L \cup V$ and constants, logical connectives ($\neg$, $\wedge$, $\vee$), ordering symbols ($<, \leq, \geq, >$), the equality symbol ($=$), unary predicates like `bound`, `isBlank`, `isIRI` and other features.

To define the semantics of the evaluation of a SPARQL query, we summarise the notion of solution mappings, and evaluation of SPARQL graph patterns, as detailed in [63, 88].

[1]We present the subset of SPARQL needed to understand the rest of the thesis, and we refer to [63] for the complete language syntax and semantics.

Definition 2.3 (Solution mappings). *A* solution mapping μ *is a partial function* $\mu : V \rightarrow I \cup B \cup L$. *It maps a set of variables to a set of RDF terms. A mapping has a domain* $dom(\mu)$ *which is the subset of* V *over which it is defined. We denote as* $\mu(x)$ *the RDF term resulting by applying the solution mapping to variable* x. *We denote as* Ω *a* multiset of solution mappings*, and as* Ψ *a sequence of* solution mappings*. Typical relational algebraic operators can be applied to multiset of solution mappings:*

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1 \wedge \mu_2 \in \Omega_2 \wedge \mu_1 \sim \mu_2\} \tag{2.1}$$

$$\Omega_1 \cup \Omega_2 = \{\mu | \mu \in \Omega_1 \vee \mu \in \Omega_2\} \tag{2.2}$$

$$\Omega_1 \setminus \Omega_2 = \{\mu | \mu \in \Omega_1 \wedge \nexists \mu' \in \Omega_2 : \mu \sim \mu'\} \tag{2.3}$$

$$\Omega_1 ⟕ \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2) \tag{2.4}$$

In Formula 2.1, $\mu_1 \sim \mu_2$ expresses the mapping compatibility, i.e., mappings μ_1 and μ_2 are compatible if $\forall x \in dom(\mu_1) \cap dom(\mu_2)$, then $\mu_1(x) = \mu_2(x)$.

The input data is organized in RDF datasets, collections of one or more RDF graphs, namely RDF datasets.

Definition 2.4 (RDF dataset). *An* RDF dataset DS *is a set:*

$$DS = \{G_0, (u_1, G_1), (u_2, G_2), ...(u_n, G_n)\}$$

where G_0 *and* G_i *are RDF graphs, and each corresponding* u_i *is a distinct IRI.* G_0 *is called the* default graph*, while the others are called* named graphs*. During the evaluation of a query, the graph from the dataset used for matching the graph pattern is called* active graph. *Multiple graphs can become active during the evaluation, but only one at time.*

SPARQL defines four *query forms*: ASK, SELECT, CONSTRUCT and DESCRIBE. The most common query form, SELECT, produces a result of variable bindings matching the graph pattern; a CONSTRUCT produces a new RDF graph with the query solutions; ASK produces a boolean value that is true if at least a solution exists; and DESCRIBE produces an RDF description of resources in the solution. For instance, a select query is declared as follows:

$$query \rightarrow \text{ SELECT } v_1, ..., v_n \text{ WHERE } P$$

where $v_1, ...v_n$ is a list of variables, subset of the variables of the graph pattern P.

There are other constructs such as *solution modifiers* (e.g. LIMIT, DISTINCT, and ORDER BY) that are applied after pattern matching. These and other modifiers can be found in the SPARQL Query Language specification [63]. With all these concepts at hand, we can define a SPARQL query as follows.

Definition 2.5 (SPARQL Query)**.** *A* SPARQL query *is defined as a tuple* (E, D, QF)*, where* E *is a SPARQL algebraic expression,* D *is an RDF dataset, and* QF *is a query form.*

Example 2. *We are interested in querying the graph in Example 1 to find out who owns the shop :a. In this case the dataset is formed by the default graph containing the graph in Example 1, the query form is* SELECT *and the algebraic expression is composed of a single triple pattern.*

```
SELECT ?person WHERE {?person :owns :a }
```

Given a SPARQL query over an RDF graph, a query solution can be represented as a bag of solution mappings, each of which assigns terms of RDF triples in the graph, to variables of the query. The evaluation semantics of a SPARQL query algebraic expression w.r.t. an RDF dataset is defined for every operator of the algebra, and it is expressed through an evaluation function.

Definition 2.6 (SPARQL evaluation semantics)**.** *The* SPARQL evaluation semantics *of an algebraic expression* E *is denoted as* $[\![E]\!]_{D(G)}$*, where* $D(G)$ *is the dataset* D *with active graph* G*.*

We present now the evaluation semantics of a basic graph pattern. In order to do it, we need to introduce some blank nodes related concepts. Blank nodes can lead to infinite solutions, due to the fact that two solution mappings μ_1 and μ_2 can be equal but for the blank nodes they contain, leading to a situation of duplicate results. SPARQL treats the problem through RDF instance mappings and scoping graph:

Definition 2.7 (RDF instance mapping)**.** *An* RDF instance mapping σ *is a partial function* $\sigma : B \to I \cup B \cup L$*. It maps a set of blank nodes to a set of RDF terms. An RDF instance mapping has a domain* $dom(\sigma)$ *which is the subset of* B *over which it is defined. A* Pattern Instance Mapping M *is a combination of an RDF instance mapping* σ *and a solution mapping* μ*:* $M(x) = \mu(\sigma(x))$*.*

Pattern instance mappings are used in order to define the evaluation semantics of a basic graph pattern.

Definition 2.8 (Basic Graph Pattern evaluation)**.** *Let P be a basic graph pattern and G an RDF graph. The solution mapping μ is a solution of $[\![P]\!]_G$ if there exists a pattern instance mapping M such that $M(P) = \mu(\sigma(P)) \subseteq G$.*

In the context of the evaluation of a basic graph pattern P over the active graph $D(G)$ of a dataset D, P is matched against a special RDF graph SG named *scoping graph*. The scoping graph SG is graph-equivalent to the active graph $D(G)$ and it does not share blank nodes with the dataset D and the basic graph pattern P. For the sake of simplicity, when not needed, in the following we will not consider RDF and pattern instance mappings, and we will consider μ as a solution of $[\![P]\!]_{D(G)}$ when $\mu(P) \subseteq G$.

The evaluation of most of other SPARQL operators is defined through algebraic operation. For example, given two graph patterns P_1 and P_2 the evaluation of the join operator on the active graph $D(G)$ is given:

$$[\![Join(P_1, P_2)]\!]_{D(G)} = [\![P_1]\!]_{D(G)} \bowtie [\![P_2]\!]_{D(G)}$$

2.2 Description logics and reasoning

Description Logics are fragments of First Order Logic (FOL) [54]; they aim to be more expressive than Propositional Logic maintaining the decidability on the reasoning tasks (FOL is semi-decidable). Description Logics define a proper syntax, different by the one used in FOL: in general it imposes constraints limiting the expressiveness (i.e., it is not possible to have the same expressiveness of FOL).

Description Logics have three main elements: concepts, roles and individuals. *Concepts* (or classes) denote relevant sets of elements (individuals). Concepts can be partitioned in atomic and complex concepts. An *atomic concept* is a concept defined through a name, e.g., `Female` and `Person`; we denote with $\mathsf{N_C}$ the set of the atomic concepts of the ontology we are considering. A *complex concept* is a concept defined through a composition of other concepts (atomic and/or complex) through specific operators, e.g., `Woman` is the intersection of `Female` and `Person`. In the following, we indicate atomic concepts with $A, A_1, A_2, \ldots \in \mathsf{N_C}$, while complex concepts with $C, D, C_1, C_2, \ldots$. Each DL provides different rules on how complex concepts can be composed and which operators can be used. Two special (atomic) concepts are *top* ($\top$), that indicates the whole set of elements of the universe, and *bottom* ($\bot$), that refers to the empty set of elements of the universe.

Roles (or properties) are binary relations between elements. Similarly to concepts, roles can be classified in *atomic roles*, defined by a name (e.g., `parentOf`), and *complex roles*, defined through a combination of other roles (e.g., `motherOf` is a sub relation of `parentOf`). We denote with $\mathsf{N_R}$ the set of the atomic role names in the ontology we are considering. Additionally roles can be characterized by *domain* and *range*, expressed as sets of concepts. These two sets provide information about individuals that are involved in relations and their inclusion in domain (and range) concepts. We indicate atomic roles with $P, P_1, P_2, \ldots \in \mathsf{N_R}$, and generic roles (both atomic and complex) with $R, R_1, R_2, \ldots$. Additionally, roles can have properties, such as transitivity, symmetry, function. Each description logic indicates if and how complex roles are defined, and which properties can be used in the role definitions.

Individuals are instances of concepts and they can be related each other through roles. Individuals are constants and they are defined through a name. In DLs, and in particular in the context of studying their complexity, an important assumption is the *Unique Name Assumption* (UNA): it states that each individual cannot have more than one name. In the recent years, the research community has studied DLs without UNA. For example, OWL languages [60] do not assume UNA (so two names can refer to the same individual). We indicate individuals with $a, a_1, a_2, \ldots \in \mathsf{N_I}$.

Concepts, roles and individuals are related through *axioms*. There are two kind of axioms: terminological axioms and assertional axioms. *Terminological axioms* describe the concepts and the roles through the inclusion ($\sqsubseteq$) and the equivalence ($\equiv$) operators. Equivalence can be seen as a particular case of inclusion: given two concepts C and D, $C \equiv D$ is equivalent to $C \sqsubseteq D$ and $D \sqsubseteq C$. The set of terminological axioms composes the *TBox*, the set of intensional knowledge (the logical statements that describe the model in a formal way). An example of TBox is the one that contains the following statements:

```
1  Student ⊑ ⊤
2  Course ⊑ ⊤
3  dom(subscribe) ⊑ Student
4  ran(subscribe) ⊑ Course
```

The first two lines state that `Student` and `Course` are two concepts, while Lines 3 and 4 define the `subscribe` property by defining respectively its domain (`Student`) and range (`Course`). If the considered ontological language supports the $\exists$ and $\forall$ operators, the domain and range restrictions can be alternatively expressed as $\exists$`subscribe`.$\top \sqsubseteq$ `Student` and $\top \sqsubseteq \forall$`subscribe`.`Course`.

Assertional axioms describe the individuals. This is done through concept assertions and role assertions. A *concept assertion* states that an individual is in a concept, while a *role assertion* states that two individuals are related through a role. This knowledge is known as extensional knowledge, and it compose the ABox. An example of ABox is the following:

```
1  adam : Student
2  math : Course
3  (adam, math) : subscribe
```

Line 1 introduces the individual `adam` as member of the class `Student` (i.e., `adam` is a `Student`); similarly, Line 2 defines `math` as member of the class `Course` (i.e., `math` is a `Course`). An alternative notation to represent the class membership is $C(a)$ (e.g., `Student`(`adam`) and `Course`(`math`)). Finally, Line 3 expresses the fact that `adam` subscribes to the `math` course (alternatively written `subscribe`(`adam`, `math`), or in a OWL-like syntax: ⟨`adam subscribe math`⟩).

Given a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$ (w.r.t. $\mathcal{T}$), we define Knowledge Base $\mathcal{K}$ the repository composed by $\mathcal{T}$ and $\mathcal{A}$, i.e., $\mathcal{K} = (\mathcal{T}, \mathcal{A})$. Given a KB, it is possible to perform different inference tasks, known as *reasoning problems* (described in Section 2.2.1).

Finally, the semantics of a DL is defined through an *interpretation* $\mathcal{I}$. An interpretation is a model composed by a pair $\langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$, where $\Delta^{\mathcal{I}}$ is a non-empty set known as *domain* and $\cdot^{\mathcal{I}}$ is the *interpretation function.* This function maps:

- each individual name a to an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$;
- each concept C to a set $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$;
- each role R to a set $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$;

Each DL defines its own interpretation function, depending on the constructs and operators it admits.

2.2.1 Reasoning Problems

We present the most common reasoning problems in DLs are: ontology consistency, concept satisfiability, concept subsumption, instance checking and query answering. Given a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, we define the following problems.

Ontology consistency. Also known as knowledge base satisfiability, it verifies if $\mathcal{K}$ does not contain contradictions, e.g., given a concept C and an individual a, at the same $\mathcal{K} \models a : C$ and $\mathcal{K} \models (a{:}\neg C)$ (respectively a is an instance of C and a is not an instance of C w.r.t. $\mathcal{K}$). In other words, this task verifies if there exists a valid interpretation $\mathcal{I}$ of $\mathcal{K}$;

Concept satisfiability. It verifies if a concept C is satisfiable, i.e., it exists an instance a in C. Formally, a concept C is satisfiable w.r.t. $\mathcal{T}$ if there exists an interpretation $\mathcal{I}$ of $\mathcal{T}$ such that $C^{\mathcal{I}} \neq \emptyset$;

Concept subsumption. It verifies if a concept C is a subset of another concept D, i.e., each instance of C is also an instance of D. Formally, a concept C is subsumed by a concept D ($C \sqsubseteq D$) w.r.t. $\mathcal{T}$ if, in every interpretation $\mathcal{I}$ of $\mathcal{T}$, is true that $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$;

Cnstance checking. It verifies if it is always true that an object a is an instance of the concept C. Formally, an individual a is an instance of the concept C ($a : C$) w.r.t. $\mathcal{K}$ if, in every interpretation $\mathcal{I}$ of $\mathcal{K}$, is true that $a^{\mathcal{I}} \in C^{\mathcal{I}}$;

An interesting property of the aforementioned problems is that, if the considered DL is expressive enough, they can be reduced among them.

Conjunctive query answering. Let V denote the set of variable names, and let $x, y, x_1, x_2, \ldots$ indicate some of its members (i.e., variables). Let's also define the *query atoms*: they can be *concept query atoms* in the form $t : C_k$, and *role atoms* in the form $(t_1, t_2){:}R_k$, where t, t_1, t_2 are elements of the set $\mathsf{N_I} \cup \mathsf{V}$ (i.e., the union set of individual and variable names). A *conjunctive query* q is a formula composed by the conjunction of query atoms.

We denote with $var(q)$ the set of variable names in q ($var(q) \sqsubseteq \mathsf{V}$). We define the function π as a function associated to an interpretation $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$. π relates the elements of $var(q) \cup \mathsf{N_I}$ with the elements of the domain δ^I (i.e., $\pi : var(q) \cup \mathsf{N_I} \to \delta^I$). Given an individual $a \in \mathsf{N_I}$, $\pi(a) = (a)^{\mathcal{I}} = a^{\mathcal{I}}$ (in other words, π relates the individual names to the elements of the interpretation domain as defined by the interpretation function). Given a variable in $var(q)$, π maps it with an element of the domain. Given $\mathcal{I}$, π:

- $\mathcal{I}, \pi \models t : C_k$ if $\pi(t) \in C_k^{\mathcal{I}}$
- $\mathcal{I}, \pi \models (t_1, t_2){:}R_k$ if $(\pi(t_1), \pi(t_2)) \in R_k^{\mathcal{I}}$

If t is an individual a, then $\mathcal{I}, \pi \models C_k(a)$ is true if $a^{\mathcal{I}} \in C_k^{\mathcal{I}}$; if t is a variable x and $\pi(x) = a_x^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, then $\mathcal{I}, \pi \models C_k(a)$ is true if $a_x^{\mathcal{I}} \in C_k^{\mathcal{I}}$.

If there exists a function π such that for all the atoms C_k and R_k in the query q, the query is satisfied by the interpretation and π and we denote it in the following way: $\mathcal{I}, \pi \models C_k$ and $\mathcal{I}, \pi \models R_k$.

Finally, the conjunctive query answering is the problem to verify if, given a knowledge base $\mathcal{K}$ and a query q, there exists a function π such that $\mathcal{I}, \pi \models q$ for every interpretation $\mathcal{I}$ of $\mathcal{K}$ (if it is true, we write $\mathcal{K} \models q$).

2.2.2 The $\mathcal{EL}^{++}$ logic

As example of DL logic, we present $\mathcal{EL}^{++}$. The language $\mathcal{EL}^{++}$ is defined by the following rules:

$$\begin{aligned} &C ::= \top \mid \bot \mid A \mid \exists R.D \mid C_1 \sqcap C_2 \mid \{a_1, \ldots, a_n\} \\ &R ::= P \mid R_1 \circ \ldots \circ R_k \end{aligned}$$

The two definitions are recursive: the first states that C can be an atomic concept $(\top, \bot, A, \exists R.C_1)$, an union of two concepts C_1 and C_2 (C_1 and C_2 can be atomic or complex concepts), or an instance enumeration $\{a_1, \ldots, a_n\}$ (e.g., $\{male, female\}$ represents the gender concept). The second definition states that a role R can be an atomic predicate or a composition of predicates.

A $\mathcal{EL}^{++}$ TBox allows the following terminological axioms:

$$\begin{aligned} C &\sqsubseteq D \\ R_1 \circ R_2 \circ \ldots \circ R_k &\sqsubseteq R \\ dom(R) &\sqsubseteq C \\ ran(R) &\sqsubseteq C \end{aligned}$$

The first axiom is the General Concept Inclusion (GCI), the second is the Role Inclusion (RI), while the other two axioms are the domain and range restrictions (respectively DR and RR). In RI, the k parameter can be zero ($\varepsilon \sqsubseteq R$). It is worth to note that $\mathcal{EL}^{++}$ allows to define:

$$\begin{aligned}
\text{disjoint classes: } & C \sqcap D \sqsubseteq \bot \\
\text{role hierarchies: } & R \sqsubseteq S \\
\text{transitive roles: } & R \circ R \sqsubseteq R \\
\text{reflexive roles: } & \varepsilon \sqsubseteq R \\
\text{right-identity rules: } & R \circ S \sqsubseteq R \\
\text{left-identity rules: } & S \circ R \sqsubseteq R
\end{aligned}$$

As we will see below, to maintain the tractability of $\mathcal{EL}^{++}$ is necessary to impose a restriction on the TBox [11]. In particular, it is necessary to limit the ways on which role inclusions and range restrictions interact. This restriction imposes that if the TBox $\mathcal{T}$ contains $R_1 \circ R_2 \circ \ldots \circ R_n \sqsubseteq S$ ($n \geq 1$) and $\mathcal{T}$ implies a range restriction on S, then R_n will have the same range of S, i.e., if $R_1 \circ R_2 \circ \ldots \circ R_n \sqsubseteq S \in \mathcal{T}$ (with $n \geq 1$) and $\mathcal{T} \models ran(S) \sqsubseteq C$, then $\mathcal{T} \models ran(R_n) \sqsubseteq C$.

The interpretation $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$ for this individuals, concepts and roles of $\mathcal{EL}^{++}$ is defined in the following way:

$$\begin{aligned}
(a)^{\mathcal{I}} &= a^{\mathcal{I}} (\in \Delta^{\mathcal{I}}) \\
(\top)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\
(\bot)^{\mathcal{I}} &= \emptyset \\
(A)^{\mathcal{I}} &= A^{\mathcal{I}} (\subseteq \Delta^{\mathcal{I}}) \\
(\exists R.C)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} | \exists y \in \Delta^{\mathcal{I}} : (x, y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \\
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(\{a_1, \ldots, a_n\})^{\mathcal{I}} &= \{a_1^{\mathcal{I}}, \ldots, a_n^{\mathcal{I}}\} \\
(P)^{\mathcal{I}} &= P^{\mathcal{I}} (\subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}) \\
(R_1 \circ \ldots \circ R_k)^{\mathcal{I}} &= R_1^{\mathcal{I}} \circ \ldots \circ R_k^{\mathcal{I}}
\end{aligned}$$

Additionally, terminological axioms (CGIs, RIs, DRs and RRs) and assertion axioms (concept and role assertions) are interpreted in the following way:

$$
\begin{gathered}
(C \sqsubseteq D)^{\mathcal{I}} = C^{\mathcal{I}} \subseteq D^{\mathcal{I}} \\
(R_1 \circ \ldots \circ R_k \sqsubseteq R)^{\mathcal{I}} = R_1^{\mathcal{I}} \circ \ldots \circ R_k^{\mathcal{I}} \subseteq R^{\mathcal{I}} \\
(dom(R) \sqsubseteq C)^{\mathcal{I}} = R^{\mathcal{I}} \subseteq C^{\mathcal{I}} \times \Delta^{\mathcal{I}} \\
(ran(R) \sqsubseteq C)^{\mathcal{I}} = R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times C^{\mathcal{I}} \\
(a : C)^{\mathcal{I}} = a^{\mathcal{I}} \in C^{\mathcal{I}} \\
((a_1, a_2) : R)^{\mathcal{I}} = (a_1^{\mathcal{I}}, a_2^{\mathcal{I}}) \in R^{\mathcal{I}}
\end{gathered}
$$

Complexity of the $\mathcal{EL}^{++}$ reasoning tasks. The importance of $\mathcal{EL}^{++}$ is strictly related to its complexity properties. Baader et al. in [12, 14] set a milestone on this topic, proving that the complexity of the concept subsumption problem is P-complete. To do it, they defined a polynomial time algorithm that *classifies* the TBox, i.e., computes the subsumption relationships between all the concepts of the TBox. The algorithm is a milestone in the study of the $\mathcal{EL}^{++}$ logic, and it is at the basis of most of the current $\mathcal{EL}^{++}$ reasoners (able to outperform by orders of magnitude the classical Tableau-based reasoners), as CEL [13] and ELK [67].

An overview over the complexity of the reasoning tasks associated to the $\mathcal{EL}^{++}$ logic is available in Table 2.1, where the four complexity dimensions are :

- *Data complexity*: only the ABox is part of the input, the TBox and the query (if it is a conjunctive query answering problem) are fixed;
- *Taxonomic complexity*: the variable part is the TBox;
- *Query complexity*: the query is the input, while the knowledge base if fixed;
- *Combined complexity*: the whole knowledge base and the query are variable, and consequently, they are input.

The $\mathcal{EL}^{++}$ logic is expressive enough to reduce concept satisfiability, ontology consistency and instance checking problems to the concept subsumption one (Section 2.2.1), and vice-versa. That means, the four problem shares the same complexity results, P-complete.

Table 2.1: *Summary of the OWL 2 EL problem complexities (source: [83])*

Problem	*Data*	*Taxonomic*	*Query*	*Combined*
concept subsumption	P-complete	P-complete		P-complete
concept satisfiability				
ontology consistency				
instance checking				
conjunctive query answering		P-complete EXP	NP-complete	PSPACE-complete EXP

Regarding the $\mathcal{EL}^{++}$ conjunctive query answering problem, in general, th problem is undecidable, as proved in [95]. To demonstrate it, the author reduces the emptiness problem for intersection of context-free messages to a $\mathcal{EL}^{++}$ conjunctive query answering problem. Anyway, by the introduction of a set of constraints to the logic, it is possible to have the P-completeness for data and taxonomic complexities, NP-completeness for query complexity and PSpace-completeness for combined complexity. These complexities have been studied mainly by Krötzsch et al.. in [70, 71].

As reported in Table 2.1, the $\mathcal{EL}^{++}$ conjunctive query answering problem is PSpace-complete in the combined complexity, NP-complete in the query complexity and P-complete in the taxonomic and data dimensions.

2.2.3 Reasoning in SPARQL

SPARQL manages the presence of ontological languages through the entailment regimes extension [58, 59]. In addition to the elements defined in Definition 2.5, an entailment regime E is added: it indicates the ontological language semantics to be considered in the context of interpreting the RDF graphs in the data set. That means, certain terms in the graphs acquire special meanings and can be used in the context of inference processes, to make explicit knowledge implicitly present in the graphs.

The introduction of logical inference in the query answering process introduces the challenges of determining which are the ontologies associated to the RDF graphs, and of determining which is the answer the SPARQL engine should provide. The core of the entailment regime extension focuses on the basic graph pattern evaluation: intuitively, the idea is that the evaluation of a BGP P over the active graph G under the entailment regime E should produce answers that can be entailed from G and P under E.

The introduction of the entailment regime should satisfy a set of requirements to guarantee that, when the input graph is compliant to the entailment regime (*well-formed*), the answer is finite and correct (i.e. sound and complete). The problems arise mainly from the presence of blank nodes, that the solution has to take into account. Glimm and Krötzsch in [58] elaborate the entailment regime conditions defined in SPARQL 1.0 [90]. An entailment regime E should satisfy the following conditions:

1. for any active graph $D(G)$, the entailment regime E specifies a unique E-equivalent scoping graph to $D(G)$;
2. for any basic graph pattern P, scoping graph SG and solution mapping $\mu \in [\![P]\!]_{SG}$ under entailment regime E, $\mu(P)$ is well formed for E;
3. for any basic graph pattern P, scoping graph SG and for any solution mapping $\mu \in \Omega = [\![P]\!]_{SG}$ under entailment regime E, there exists a family of RDF instance mappings σ_μ such that:
$$SG \models SG \bigcup_{\mu \in \Omega} \mu(\sigma_\mu(P));$$
4. the entailment regime E must prevent trivial infinite solutions.

The SPARQL entailment regime recommendation [59] defines the entailment regimes for semantics, such as the one of RDFS, OWL 2 RL, EL and QL.

CHAPTER 3

Background: Processing Dynamic Data

3.1 Data Stream and Event Processing

The rise of data stream sources introduced new problems about how to manage, process and query infinite sequences of data with high frequency rate [39]. Two proposed approaches are Data Stream Processing [9] and Complex Event Processing [76].

3.1.1 Data Stream Processing

Data Stream Processing (or Data Stream Management Systems, shortened DSMS) solutions transform data streams in timestamped relations to be processed with well known techniques such as algebras [9]. In the following, we present CQL and SECRET. The first is a model that describes the algebra and a processing model of a Data Stream Processing engine, while the latter focuses on the behaviour of window operators.

CQL. The CQL stream processing model were proposed by the DB group of the Stanford University [9] and defines a generic Data Stream

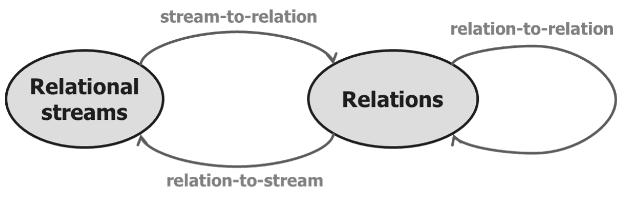

Figure 3.1: *The CQL model.*

Management System through three classes of operators, depicted in Figure 3.1.

The *stream-to-relation* operators manages the data stream: due to the fact that a stream is a potentially infinite bag of timestamped data items (also known as stream items and information items), those operators extract relations (as finite bags). One of the most studied operators of this class is the sliding window. Given a stream S, a sliding window dynamically selects subsets of the data items of S. The intuition behind this operator is that the more recent the data is, the more relevant. That means, it selects the most recent items in the stream and queries them, repeating this operations as the time goes ahead. The *windows* are the basic elements of the sliding windows. Given a stream S, a *time-based window* W defined through two parameters o and c selects the data items d of S with associated timestamp in $(o, c]$. A sliding window generates multiple windows (at different time instants) to create a time-varying view over the stream. Time-based sliding windows and tuple-based sliding windows are two of the most relevant sliding windows.

A *time-based sliding window* generates a sequence of windows at regular time intervals (e.g. a window every two seconds). Then, it selects the contents of the streams accordingly to each window $(o, c]$ (where o and c are the opening and the closing time instants). The o and c values change (*slide*) periodically, modifying the content of the sliding window. A time-based sliding window is described through two parameters, *width* ω (the dimension of the window, $c - o$) and *slide* β (the distance between two consecutive windows).

Tuple-based sliding windows select a fixed number of data items. Similarly to time-based sliding windows, they are described by the width and the slide parameters, but the width indicates the number of data items that are collected in the current view, while the slide indicates how many data items are removed/added at each window slide. Consequently, the difference between the closing time instant and the opening time instant of the generated windows is not constant.

When the slide parameter is equal to the width parameter, the sliding-window is also known as *tumbling window*: it is important because it partitions the stream: each data item would only in one window.

After that sliding window computes a finite relation from the stream, it may be transformed in another relation through a *relation-to-relation* operator. Relational algebraic expressions are a well-known cases of this class of operators.

Finally, the system outputs. If the output of the query processor should be a stream, it is necessary to include a *relation-to-stream* operator. When applied, it appends the set of relations to the output stream. At each time instant at which the continuous query is evaluated, the set of relations is processed by the relation-to-stream operator, which is in charge to determine which data items have to be streamed out. RStream, IStream and DStream are the three relation-to-stream operators defined in CQL.

RStream streams out the computed timestamped set of relations at each step. Rstream answers can be verbose as the same relation can be computed at different evaluation times, and consequently streamed out. IStream streams out the difference between the timestamped set of relations computed at the current step and the one computed at the previous step. Answers are usually short (they contain only the differences) and consequently this operator is used when data exchange is expensive.

DStream does the opposite of Istream: it streams out the difference between the computed timestamped set of relations at the previous step and at the current step. Dstream is normally considered less relevant than Rstream and Istream, but it can be useful, e.g. to retrieve attractions that are no more popular.

SECRET. SECRET [29] supports the task of integrating streaming data processors, by means of explaining the different behaviour of existing stream processing engines.

SECRET associates two time instant to each stream item, the application and system time. *Application time*, denotes the time instant associated to the event represented by the stream element. It can be shared by multiple elements (introducing *contemporaneity*) and consequently defines a partial order among the stream elements. *System* *time* is the "wall-clock" time, which must be unique, thus introducing a total order in the stream. Even if the application time is the relevant information from a conceptual point of view, it is important to take into account the system time to explain the correct behavior of stream processors.

The SECRET framework focuses on the window operator specification, by exploiting the notions of scope, content, report and tick.

Scope is a function that associates an evaluation time instant t to the time interval of the active window at t. Key to the computation of the scope is the t^0 parameter, which is the application timestamp when the first active window starts. It is an absolute time instant and depends on both the query (its registration time) and the Data Stream Processing engine (how it processes the query and instantiates the window).

The *Content* function identifies the set of elements of $\mathbb{S}$ in the active window. This function depends not only on the scope (and consequently the application time), but also on the system time: it means that asking for the content of the active window at the same application time at two different system time instants can produce two different results.

Report is a function that defines the required conditions to pass the window content to the relation-to-relation operators. SECRET identifies four reporting strategies (Data Stream Processing systems may use combined strategies as well):

- *Content change:* system reports if the content changes.
- *Window close:* system reports if the active window closes.
- *Non-empty content:* system reports if the active window is not empty.
- *Periodic:* system reports only at regular intervals.

The *Tick* function defines the conditions under which the input can be added to the window, becoming processable by the query engine. SECRET defines different strategies: tuple-driven and time-driven. Systems with *tuple-driven* tick strategy add input tuples in the window operator as soon as they arrive, while systems with *time-driven* tick strategy add sets of tuples to the window at each (application) time instant.

3.1.2 Complex Event Processing

Beyond data streams in Data Stream Processing, Complex Event Processing (CEP) focuses on events, which Luckham defines as *an object that represents, or records an activity that happens, or is thought of as happening* [76].

While Data Stream Processing operators are well suited in tasks that require aggregations or assessing occurrence in intervals, CEP operators provide the possibility to compose complex event by means of temporal relations among the event occurring in the stream. CEP operators allow to express several types of relationships between events [80] which may include simultaneousness, precedence [77], absence (as form of negation) [24].

Examples of temporal relations are depicted in Figure 3.2, that informally shows the application of operators between two temporal intervals as defined in Allen's Algebra [3, 96]. Assume that compatible relations exist for three event patterns P_1, P_2, and P_3 in the time intervals shown in Figure 3.2, the horizontal bars represent the result of evaluating the operators on the relations at the different time units depicted with vertical dashed lines.

Temporal operators in Complex Event Processing can be viewed as special types of joins that add to the boolean semantics of the join operator, the temporal semantics of the specific temporal relation of the operator. For instance, the relations of P_1 SEQ P_2 from a membership perspective are those of (P_1 JOIN P_2), but the SEQ operator keeps only the relations of the JOIN operator for which a relation of P_2 starts after the end of a relation of P_1.

In the following, we present two Complex Event Processing solutions, Tesla and Esper.

Tesla. Tesla [38] proposed a complete set of operators including selection of primitive events, definition of sequences, negation, aggregation, and recursive rules. Notably, it also allows customizable selection and consumption policies for event matching. The *selection policy* determines if a pattern matches once or multiple times for each selected stream item, while the *consumption policy* determines if a pattern consumes the statements it matches (i.e., once it is matched it is not available for the next evaluation).

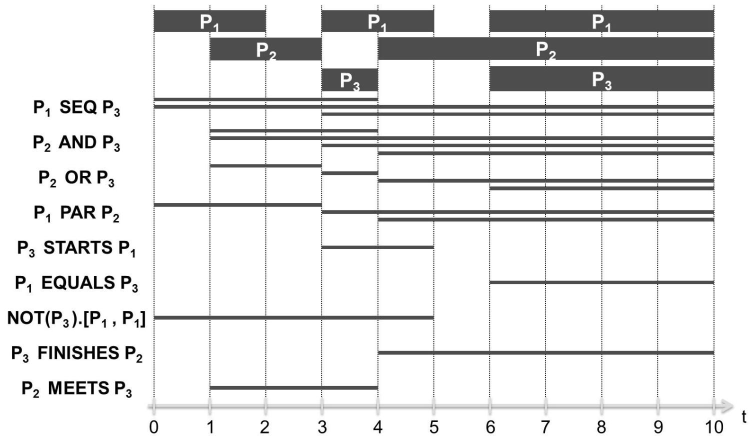

Figure 3.2: *Temporal relations between two temporal intervals as defined in Allen's Algebra.*

Esper. The open-source system Esper[1] also includes CEP features in its query language, EPL. In particular, it supports a series of pattern operators such as followed-by ($A \rightarrow B$), which looks for an event A and if encountered, looks for B. Esper allows more flexibility through the EVERY operator, which indicates that the pattern should restart after it is matched on the stream. For instance EVERY ($A \rightarrow B$) detects an event A followed by a B. When this pattern matches, the matching restarts and looking for the next A. On the other hand EVERY $A \rightarrow$ EVERY B matches for every event A followed by a B.

3.2 RDF Streams and RDF Stream Processing Engines

When moving from relational data streams to RDF Stream Processing, one of the most notable differences relies on the data model: instead of traditional relational data, it uses the RDF data model. The rest of the section first introduces the RSP data and query models proposed in the literature.

3.2.1 RDF streams

Different definitions of RDF streams were proposed. It is useful to introduce two axes to classify them: the data item and the time annotation.

[1]Esper: `http://www.espertech.com/esper/index.php`

Data item dimension. The data item is the minimal informative unit in the stream. Existing work in RSP considers two alternatives for this role: RDF statements and RDF graphs, as in Definition 2.1. The simplest case is the one where the stream is composed of *RDF statements* [44]. Even if the RDF statement stream model is easy to be managed, the amount of information that a single RDF statement carries may be not enough when modelling real use cases. For this reason, recent work [21] proposes to use as informative unit *RDF graphs*.

Example 3. *For example, let us consider this RDF stream on which each triple states the presence of a person in a room of The Louvre:*

```
  :alice  :detectedAt :monaLisaRoom  [1] .
  :bob    :detectedAt :parthenonRoom [2] .
  :conan  :detectedAt :monaLisaRoom  [5] .
  :bob    :detectedAt :monaLisaRoom  [5] .
```

Turtle[2]-like syntax is adopted: in each row there is an RDF statement enriched with the time annotation (the fourth element between square brackets). When adopting this data model, each statement may contain enough information to be processed as informative unit.

Example 4. *Let us consider the following RDF stream, expressing check-in operations in a social network:*

```
:g1 {
  :alice :posts :c1 .
  :c1    :where :monaLisaRoom .
  :c1    :with  :conan .
} [1]
:g2 {
  :bob   :posts :c2 .
  :c2    :where :parthenonRoom .
  :c2    :with  :diana .
} [3]
:g3 {
  :conan :posts :c3 .
  :c3    :where :monaLisaRoom .
} [3]
```

This example adopts a Trig-like syntax[3]: RDF resources `:g1`, `:g2` *and* `:g3` *identify three RDF graphs with the relative time annotations (the number between squared brackets). The blocks of RDF statements (enclosed in {}) are the contents of the graphs. As it is possible to observe,*

[2]Cf. http://www.w3.org/TR/turtle/.
[3]Cf. http://www.w3.org/TR/trig/.

in this example single RDF statements are not enough to represent a whole informative unit (e.g. a check-in post).

Time annotation dimension. The time annotation is a set of time instants associated with each data item. The choice to consider the time as an annotation on data items, and not as part of the schema, is inherited by the DSMS research [15], and it is motivated by both modelling and technical reasons: the time information could be part of the schema, but it should not be mandatory (i.e. there are scenarios on which it is not). Moreover, DSMSs and CEPs usually do not allow explicit accesses to the time annotations through the query languages: conditions are usually expressed with time relative constraints, e.g. select events that *happen before a given one*, or identify the events that *hold in the last five minutes*.

The term *application time* refers to the time annotation of a data item [29]. Usually, application time is represented through sets of timestamps, i.e. identifier of relevant time instants. The classification along the time annotation axis depends on the number of timestamps that compose the application time.

In the simplest case, the application time consists of *zero timestamps*: in other words, there is not explicit time information associated with the data items. It follows that the RDF stream is an ordered sequence of elements that arrive to the processing engine over time, like in the stream S represented in the Figure 3.3.

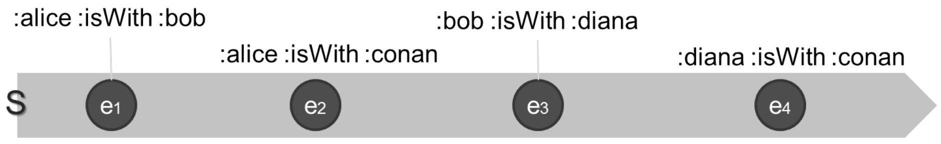

Figure 3.3: *Example of stream with zero timestamps per data item.*

Rounds labelled with e_i (i in $[1, 4]$) represent the data items of the stream; the time flows from left to right, e.g. item e_1 happens before item e_2. Even if the data items do not have explicit timestamps, it is possible to process those streams by defining queries that exploit the order of the elements, such as:

q$_1$. Does Alice meet Bob before Conan?

q$_2$. Who does Conan meet first?

Let us consider now the case on which the application time is modelled introducing a metric [96] and each data item has *one timestamp* like in Figure 3.4.

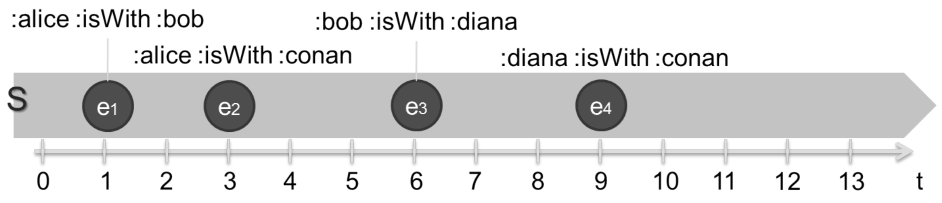

Figure 3.4: *Example of stream with one timestamp per data item.*

In most of the existing works, the timestamp used in the application time represents the time instant at which the associated event occurs, but other semantics are possible, e.g. time since the event holds. Due to the fact that data items are still ordered by recency (as in the previous case), it is possible to issue queries of the previous case, as q_1 and q_2. Additionally, it is possible to write queries that take into account the time, such as:

q_3. Does Diana meet Bob and then Conan within 5m?

q_4. How many people has Alice met in the last 5m?

It is worth to note that q_3 and q_4 do not refer to absolute time instants, but on relative ones w.r.t the time instant of another event (as in q_3) or the current time instant (as in q_4).

As a final case, let us introduce the application time composed of *two timestamps*. The semantics that is usually associated to the two timestamps is the time range $(from, to]$ on which the data item is valid, as shown in Figure 3.5.

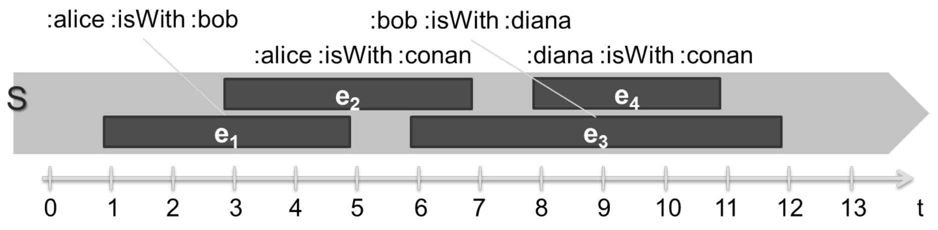

Figure 3.5: *Example of stream with two timestamps per data item.*

Each square represents a data item, and the application time is represented by the initial and the final timestamps, e.g. e_1 has application time $(1, 5)$, so it is valid from the time instants 1 to the time instants 5. Similarly to the previous case, it is still possible to process to the queries presented in the first two cases (e.g. q_1, …, q_4), and additionally more complex constraints can now be written, such as:

q_5. Which are the meetings that last less than 5m?

q_6. Which are the meetings with conflicts?

Other cases exist, where the application time is composed of three or more timestamps, or where the application time semantics has other meanings. Even if they can be useful in some use cases, they are still under investigation in RSP research and no relevant results have been reached, yet.

3.2.2 RSP query languages and systems

After the presentation of the RDF stream definitions, this section discusses the problem of processing these kinds of data. As for the data model, the RSP query model is inspired by research in Data Stream and Complex Event Processing: RSP solutions rely on the key idea that existing Data Stream and Complex Event Processing operators can be combined with Semantic Web ones to enable advanced tasks as inference, and heterogeneous data integration over data streams.

Table 3.1 lists the RSP available in the state of the art, and classify them according to the data model they manage and on the operators they support. Analysing the table, it is possible to observe that most systems focus on RDF streams with RDF statement as data item and the application time is composed of one timestamp. The C-SPARQL engine [20], CQELS [73] and Morph$_{stream}$ [31], adopting respectively the C-SPARQL, CQELS-QL and SPARQL$_{stream}$ languages, manage data streams where data items are RDF statements. Their query models are similar, but they are designed in different ways and they target different use cases.

The data model with one timestamped RDF statements is the one on which the initial RSP research focused, and novel trends started to consider data model variants. The SLD platform [18] has features similar to C-SPARQL, CQELS and SPARQL$_{stream}$, but it is able to process RDF streams with RDF graphs as data items. INSTANS [94] follows a completely different approach: it takes sequences of RDF

	C-SPARQL	CQELS	SPARQL$_{stream}$	INSTANS	ETALIS	SLD
Data item	statement	statement	statement	statement	statement	graph
Applitaction time	1	1	1	0	2	1
Time-based sliding windows	✓	✓	✓			✓
Triple-based sliding windows	✓	✓				
Graph-based sliding windows						✓
stream-to-relation	RStream	IStream	RStream IStream DStream	RStream	RStream	RStream
CEP operators	∼				✓	∼

Table 3.1: *Comparison of RSP systems. ∼ indicates that C-SPARQL and SLD have a limited support to Complex Event Processing operators, through a function named* timestamp*.*

statements without timestamps as input and processes them through the RETE algorithm. Finally, ETALIS and EP-SPARQL [6] work on RDF streams with application time composed of two timestamps: they use CEP concepts and are able to perform Stream Reasoning tasks. We review them in the rest of the section.

C-SPARQL. Continuous SPARQL (C-SPARQL) [20] is a language for continuous queries over streams of RDF data that extends SPARQL 1.1 by adding Data Stream Processing inspired operators, as sliding windows and streaming operators.

The language is implemented in the C-SPARQL engine, an open-source software that allows to register queries that are continuously evaluated over time. The C-SPARQL engine is built on top of two sub-components, ESPER[4] and Jena[5]. The former is responsible of executing continuous queries over RDF streams, producing a sequence of RDF graphs over time, while the latter runs a standard SPARQL query against each RDF graph in the sequence, producing a continuous result. This result is finally formatted as specified in the Query Form.

C-SPARQL has a limited support to Complex Event Processing operators through the timestamp function, that allows to retrieve the most recent time instant associated to a RDF statement. Timestamps can be compared in the context of the FILTER clause, enabling the expression of simple sequence patterns.

CQELS. The Continuous Query Evaluation over Linked Streams – shortened CQELS – accepts queries expressed in CQELS-QL [73], a declarative query language that, similarly to C-SPARQL, extends the SPARQL 1.1 grammar with operators to query RDF streams. The main difference between C-SPARQL and CQELS-QL is the relation-to-stream operator they support: CQELS-QL supports only Istream, whereas C-SPARQL supports only Rstream.

Differently from the C-SPARQL engine that uses a "black box" approach which delegates the processing to other engines, CQELS proposes a "white box" approach and implements the required query operators natively to avoid the overhead and limitations of closed system regimes. CQELS provides a flexible query execution framework with the query processor dynamically adapting to the changes in the input data. During query execution, it continuously reorders operators ac-

[4]Cf. `http://espertech.com/`.
[5]Cf. `http://jena.apache.org/`.

cording to heuristics that improve query execution in terms of delay and complexity.

SPARQL$_{stream}$. SPARQL$_{stream}$ [31] is another extension of SPARQL that supports operators over RDF streams such as time windows. Unlike CQELS and the C-SPARQL engine, SPARQL$_{stream}$ supports all the streaming operators.

The language is implemented in morph-streams: it is an RDF stream query processor that uses Ontology-Based Data Access techniques [32] for the continuous execution of SPARQL$_{stream}$ queries against virtual RDF streams that logically represent relational data streams. Morph-streams uses R2RML[6] to define mappings between ontologies and data streams. SPARQL$_{stream}$ queries are rewritten first in a relational algebra expression extended with time window constructs, that is optimized, and then translated in the Data Stream Processing target language, e.g. SNEE[7], Esper and GSN[8].

INSTANS. The Incremental eNgine for STANding Sparql [94], INSTANS, takes a different perspective on RDF Stream Processing. Users model their task as multiple interconnected SPARQL 1.1 queries and rules. Next, INSTANS performs continuous evaluation of incoming RDF data against the compiled set of queries, storing intermediate results into a Rete-like structure. When all the conditions are matched, the result is instantly supplied. In this sense, INSTANS does not require continuous extensions to RDF or SPARQL.

ETALIS and EP-SPARQL. ETALIS (Event TrAnsaction Logic Inference System) [7] is a Complex Event Processing inspired RSP engine. At the best of our knowledge, this is the only RSP engine that processes RDF streams with application time composed by two timestamps. Users can specify event processing tasks in ETALIS using two declarative rule-based languages, ETALIS Language for Events (ELE) and Event Processing SPARQL (EP-SPARQL) [7]. The former language is more expressive than the latter, even if it is less usable. A common point is that complex events are derived from simpler events by means of deductive prolog rules.

[6]Cf `http://www.w3.org/TR/r2rml/`.
[7]Cf. `http://code.google.com/p/snee/`.
[8]Cf. `http://lsir.epfl.ch/research/current/gsn/`.

ETALIS is a pluggable system that can use multiple prolog engines such as YAP[9] and SWI[10]. ETALIS supports three different policies [4]:

- *unrestricted*: all input elements are selected for matching the event patterns.
- *chronological*: only the earliest input that can be matched are selected for matching the event patterns; then, they are ignored in the next evaluations.
- *recent*: only the latest input that can be matched are selected for matching the event patterns; then, they are ignored in the next evaluations.

Notably, the EP-SPARQL query does not change in the three cases, as the setting is a configuration parameter set at the startup of the engine. Moreover, independently on the setting, all the system outputs happen as soon as they are available.

SLD. The Streaming Linked Data (SLD) framework [18] is not a proper RSP engine, but it wraps the C-SPARQL engine and it adds new features. SLD offers: a set of adapters that transcode relational data streams in streams of RDF graphs (e.g. a stream of micro-posts as an RDF stream using the SIOC vocabulary [30], or a stream of weather sensor observation using the Semantic Sensor Network vocabulary [37]), a publish-subscribe bus to internally and externally exchange RDF streams (following the Streaming Linked Data Format [21]), facilities to record and replay RDF streams, and extendible layer to plug components that decorate RDF streams (e.g. adding sentiment annotations to micro-posts).

3.3 Stream Reasoning

In this section we introduce the most relevant work in the area of Stream Reasoning. We grouped them in three categories. In *Continuous Query Answering under Entailment Regimes*, we collect the work focusing on the query answering problem in presence of streams and logical based inference processes. *Materialization and Incremental Maintenance* presents contributions in computation of the closure of streams of axioms and the techniques to make them work incrementally

[9]Cf. `http://www.dcc.fc.up.pt/~Evsc/Yap/`.
[10]Cf. `http://swi-prolog.org/`.

(i.e. without restarting from scratch every time the ontology changes). Finally, in *Formal Semantics* we describe two recent work on providing foundations on Stream Reasoning.

3.3.1 Continuous Query Answering under Entailment Regimes

ETalis, EP-SPARQL. Most of the system we presented in Section 3.2 evaluates query under *simple* regime, where basic graph patterns are evaluated in terms of subgraph matching against the data in the active graph of the dataset. One of the exceptions is ETalis/EP-SPARQL, that supports back-ward temporal reasoning over RDFS.

For example, as CEPs, it allows to check for sequences such as A SEQ B; however, it also allows stating that C is a subclass of A. Therefore the condition A SEQ B is matched also if an event of type C is followed by an event of type B, because all events of type C are also of type A.

Stream Reasoning with ASP. Answer Set Programming (ASP) [53] is a declarative problem solving paradigm. Its roots can be found in deductive databases, logic programming and knowledge representation. The problem is modelled through a set of logic rules (a logic program) and the solution is composed by the set of answers. ASP is characterized by a rich modelling language, e.g. it captures integrity constraints, weak constraints, negations. At the same time, ASP solvers proved to have very high performance in solving the tasks, by exploiting techniques from constraint solving.

One limit of ASP is the fact that its programs are written to work with static knowledge, and results has to be recomputed every time the underlying data changes. Incremental ASP [56] overcomes this limit, and extends ASP in order to incrementally compute the solutions. Stream Reasoning and Incremental ASP are connected through the Time-Decaying Logic Programs [57], that introduce the notions of emerging and expiring data.

Results are used in StreamRule [81], an approach to compute continuous queries under ASP entailment regime. StreamReuls is a two-layer approach: the first layer is composed by an RSP engine acting as filter, to reduce the amount of data to be considered in the inference process. The second layer is the logic program, based on Incremental ASP, that computes the answer set.

3.3.2 Materialization and Incremental Maintenance

The origin of incremental maintenance approach can be found in maintenance of materialized views in active databases [34, 97]. In these works, authors researched on how to generate a materialized view in active databases and how to maintain it incrementally through set of updates. They proved that when the number of modification in the database is under a threshold, the incremental maintenance techniques perform orders of magnitude faster than the whole re-computation of the view.

Ontology Maintenance with DRed. Volz et al. [101] proposes an algorithm to incrementally maintain an ontological entailment. It does not focus explicitly on streaming data, its presence in this list is motivated by the relevance with the following algorithms. This technique is a declarative variant of the *Delete and Re-derive* (DRed) algorithm of [97]. The general idea of DRed is a three-steps algoritm:

1. Overestimate the deletions: starting from the facts that should be deleted, compute the facts that are deducted by them;
2. Prune the over-estimated deletions: determine which facts can be rederived by other facts;
3. Insert the new deducted facts: derive facts that are consequences of added facts and insert them in the materialization.

The version of DRed proposed in [101] is written in Datalog$^{\neg}$. The materialization is in a Datalog predicate T and the goal of the algorithm is the computation of two predicates, T^{+} and T^{-}, when updates occur. T^{+} and T^{-} should be respectively added and removed to T to obtain a new correct materialization.

IMaRS. The Incremental Materialization for RDF Streams algorithm (IMaRS) [48**?**] is a variation of DRed for the incremental maintenance of the window content materializations. In general, the main problems of the incremental maintenance are the deletions: it is a complex task to determine which consequences are not valid anymore when statements are removed from the knowledge base. IMaRS exploits the window mechanism in order to cope with this problem: it allows to determine when a statement is going to expire and should be deleted from the materialization. IMaRS, when axioms are inserted in the window, computes their *expiration* time instants. This allows IMaRS to work out a new complete and correct materialization whenever a new window should be computed by dropping explicit statements and entailments that are no longer valid.

The ontological language targeted by IMaRS is RDFS+ [2], an extension of RDFS with transitive and inverse properties and without concrete domains. Experiments showed that in the streaming scenario it outperforms DRed.

Sparkwave. Sparkwave [69] is a solution to perform continuous pattern matching over RDF data streams under RDFS[11] entailment regime. It allows to express temporal constraints in the form of time windows while taking into account RDF schema entailments. It is based on the Rete algorithm [55] which was proposed as a solution for production rule systems, but it offers a general solution for matching multiple patterns against multiple objects. The Rete algorithm trades memory for performance by building two memory structures that check intra- and inter-pattern conditions over a set of objects, respectively. Sparkwave adds another memory structure, which computes RDFS entailments, in front of the original two. Under the assumption that the ontology does not change, RDFS can be encoded as rules that are activated by a single triple from the stream. Therefore, each triple from the stream can be treated independently and in a stateless way. This guarantees for high throughput. Moreover, Sparkwave adds time windows support to Rete in an innovative way. While the state of the art [103] uses a separate thread to prune expired matchings, Sparkwave prunes them after each execution of the algorithm without risking deadlocks and keeping the throughput stable.

Sparkwave is very similar to IMaRS on a conceptual level. However, the approach proposed by Sparkwave can hardly be extended to more expressive languages. As stated above, RDFS can be encoded as rules that are activated by a single triple from the stream, whereas others, like owl:transitiveProperty, are activated by multiple statements from the stream. This means that the stateless approach of Sparkwave is no longer sufficient.

Streaming Knowledge Bases. Streaming Knowledge Bases [102] is one of the earliest stream reasoners. It uses TelegraphCQ [35] to efficiently handle data stream, and Jena rule engine to incrementally

[11]In particular, its description logic fragment

materialise the knowledge base. The architecture of Streaming Knowledge Bases is similar to the one of the C-SPARQL Engine. It supports RDFS and owl:inverseOf construct (i.e., rules that are activated by a single triple from the stream), therefore the discussion reported above for Sparkwaves also applies to it. Unfortunately, the prototype has never been made available and no comparative evaluation results are available.

Truth Maintenance System. Ren and Pan [92] take a different perspective; they investigate the possibility to optimise Truth Maintenance Systems so to perform expressive incremental reasoning when the knowledge base is subject to a large amount of random changes (both updates and deletes). They optimise their approach to reason with $\mathcal{EL}^{++}$, building a directed graph to track the inferences and consequently the justifications. Removals are performed by traversing the graph and removing the nodes. On the contrary, addition operations generate new nodes and edges in the graph. Authors provide experimental evidence that their approach outperform re-materialisation up to 10% of changes.

DynamiTE. Urbani et al. [100] propose DynamiTE, a framework to efficiently compute a materialization and to keep it up to date. The approach differentiates by the previous ones for the introduction of parallelization, that drastically improve the performance.

The use cases targeted by DynamiTE is similar to the one of DRed and Truth Maintenance System: an ontology modified by a huge number of frequent update operations, similar to transactional settings in data base systems. On additions, DynamiTE re-computes the materialization to add the new entailments through a parallel Datalog evaluation. On removals, it deletes the explicit and entailed axioms no longer valid. Several algorithms can perform this action: authors considered DRed and a "counting" algorithm they defined, that exploits the idea of counting the number of justifications that entailed it.

Despite DRed and Truth Maintenance System, that targets expressive ontological languages, DynamiTE focuses on ρDF [84], a fragment of RDFS. In this setting, Experiments show that the "counting" algorithm outperforms DRed by orders of magnitude.

3.3.3 Formal semantics

LARS. The *Logic-based framework for Analyzing Reasoning over Streams* (LARS) [25] defines a logic for capturing typical operations of the Stream Reasoning context. Regarding the data, they model the notion of stream as sequence of time-annotated formulas. In addition to the usual boolean operators (and, or, implies, not), authors define the two temporal logic operators $\diamond$ and $\Box$, to express the fact that a formula holds respectively at some time in the past and every time in the past. Additionally, it introduces the @ and $\boxplus$: the former is used to state that a formula holds at a specific time instant, while the latter is the *window* operator, that restricts the scope on which the enclosed formula applies. Authors proved that LARS captures the CQL language (including aggregates) and ETalis semantics.

STARQL. In parallel with the development of LARS, the *Spatial and Temporal ontology Access with a Reasoning-based Query Language* [85, 86] (STARQL) proposes an alternative view on Stream Reasoning. The framework is designed to access and query heterogeneous sensor data through ontologies, in a OBDA approach similar to the SPARQL$_{stream}$ one.

STARQL is structured as a two-layer framework, expressed through the $STARQL(OL, ECL)$ notation, where OL denotes an Ontology Language to model the data and its schema, and ECL is an Embedded Constraint Language to compose the queries. For example, $STARQL(DL-Lite^{\mathcal{H}}_{core}, UCQ)$ indicates that the ontology is modelled in the $DL-Lite^{\mathcal{H}}_{core}$ description logic and queries are expressed as union of continuous queries. STARQL offers window operators, clauses to express event matching and a layer to integrate static and streaming data.

Part II

A Reference Model for RDF Stream Processing

CHAPTER 4

Extending the RDF model for RDF Stream Processing

Several works studied how to add the temporal dimension to RDF [61, 62, 82, 91], for instance by adding a fourth element to the triple to express the validity time. In the context we are studying, we have to take into account the time by two different points of view. On the one hand, there are streams, as flows of timestamped data items that are served on sequences potentially infinite, as for example the Twitter streaming API, or the Wikipedia update change. On the other hand, there is the background data (e.g. data stored in repositories), that evolves and changes, as it is possible for example observe in DBPedia live.

The two cases require different time-related extensions, and we present them in this chapter: in Section 4.1, we formalize the notion of RDF stream that we are going to consider in the rest of this work; Section 4.2 formalises the background data through time-varying and instantaneous RDF graphs. Section 4.3 describes Triple Wave, a prototype to provide RDF streams, to give an example of RDF stream serialization and publishing over the Web.

Section 4.3 is based on "Where Are RDF Streams? On Deploying RDF Streams on the Web of Data With TripleWave", a poster published at the 14th International Web Conference. A. Mauri implemented and deployed the prototype, while J.P. Calbimonte and I drove the writing: J.P. Calbimonte defined with the part of R2RML mappings, while I wrote the rest of the paper. Prof. Della Valle and M. Balduini shared with us their experience in creating and consuming RDF streams inside the SLD framework, and helped us in deciding how to publish RDF streams as Linked Data. Prof. K. Aberer supervised the work.

4.1 Streaming extension of the RDF model

In this section, we formalise the notion of RDF stream associating a time instant to each RDF statement, as in [20, 32, 69, 73, 100]We start by defining the notion of time as in [9].

Definition 4.1 (Time). *A* time line T *is an infinite, discrete, ordered sequence of time instants* $(t_1, t_2, \ldots)$*, where* $t_i \in \mathbb{N}$*. A* time unit *is the difference between two consecutive time instants* $(t_{i+1} - t_i)$ *and it is constant.*

It is now possible to associate temporal annotations to RDF concepts (as statements and graphs) and, consequently, define RDF streams as sequences of them.

Definition 4.2 (RDF data stream). *An* RDF data stream S *is a data stream where information items are RDF graphs:*

$$S = (d_1, t_1), (d_2, t_2), (d_3, t_3), (d_4, t_4), \ldots$$

The pair (d_i, t_i) *represents the i-th RDF graph* d_i *and* $t_i \in T$ *is the relative time annotation.*

As learnt in Section 3.2.1, existent models assume RDF streams where the information item is either RDF graph or RDF statement. We adopt the former, due to the fact it can capture the latter, e.g. by grouping all statements with the same time annotation, or by defining a graph for every single statement.

Example 5. *The Sirius Cybernetics Corporation offers for free-download a mobile App that delivers instantaneous discount coupons to shoppers while they are near by shops like A and B. The shoppers Carl, Diana* *and Eve have such an App on their mobiles. When they are within 200 meters from shops a or b, the App records it on the RDF stream S_{nearby} with the timestamped RDF graphs (we use a TriG [33] like notation, where graphs are enriched with the time relative instant) in Listing 4.1.*

```
:d_n1 {:diana :isNearby :a} [2] .
:d_n2 {:eve :isNearby :b} [2] .
:d_n3 {:carl :isNearby :a} [5] .
:d_n4 {:eve :isNearby :a} [7] .
:d_n5 {:diana :isNearby :b} [12] .
:d_n6 {:carl :isNearby :b} [19] .
:d_n7 {:carl :isNearby :b} [21] .
```

Listing 4.1: *The RDF stream S_{nearby}*

The statements assert that Diana, Carl and Eve are nearby the shop A respectively at the time instants 2, 5 and 7; Eve, Diana are nearby the shop B at time instants 2 and 12; Carl is near shop B at time instants 19 and 21.

4.2 Time-varying and Instantaneous RDF Graphs

Introduction of RDF streams does not exclude the presence of static and quasi-static data (background data). An additional extension is required, due to the fact that RDF does not capture the evolution over time, as explained in the RDF 1.1 primer:

> The RDF data model is atemporal: RDF graphs are static snapshots of information.

We introduce now the concepts of the time-varying RDF graph and instantaneous RDF graph. Intuitively, time-varying graphs capture the dynamic evolution of a graph over time, while instantaneous graphs represent the content of the graph at a fixed time instant.

Definition 4.3 (Evolution of RDF graphs over time)**.** *A* time-varying graph $\overline{G}$ *is a function that relates time instants $t \in T$ to RDF graphs:*

$$\overline{G} : T \rightarrow \{G \mid G \textit{ is an RDF graph}\}$$

An instantaneous RDF graph $\overline{G}(t)$ *is the RDF graph identified by the time-varying graph $\overline{G}$ at the given time instant t.*

We indicate with the capital letter G $(G, G_1, G_2, \ldots)$ the RDF graphs and with d $(d, d_1, d_2, \ldots)$ the RDF graphs that act as information items in a stream. Moreover, we adopt the bar capital letter $\overline{G}$ to indicate the

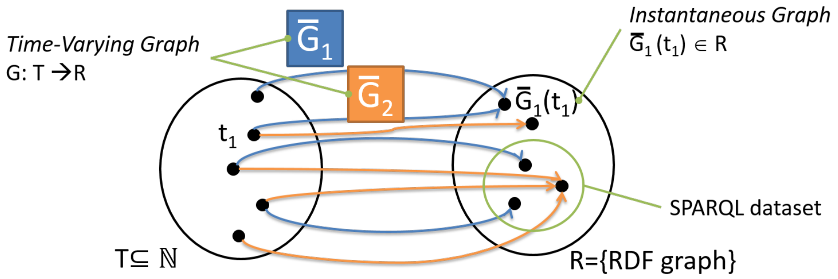

Figure 4.1: *Time-varying and instantaneous graph*

time-varying graphs $(\overline{G}, \overline{G}_1, \overline{G}2, \ldots)$. Figure 4.1 helps in understanding the difference between the two concepts: it shows two time-varying graphs, $\overline{G}_1$ and $\overline{G}_2$. Each time-varying graph associates time instants to RDF graphs; for each time instant t at which $\overline{G}_1$ ($\overline{G}_2$) is defined, $\overline{G}_1(t)$ ($\overline{G}_2(t)$) refers to an instantaneous RDF graph; being $\overline{G}_1$ ($\overline{G}_2$) a function, each time instant is associated to one and only one RDF graph. It is worth to note that the instantaneous graph contains RDF statements (without any timestamp). It follows that instantaneous graphs can be queried through the SPARQL query language without any continuous extension.

Example 6. *The Sirius Cybernetics Corporation manages to convince both Alice and Bob to use its instantaneous discount coupon service from the time instant 2 to the time instant 13. After that, Alice leaves the service, and only Bob keeps using it. The time-varying graph $\overline{G}_{shops}$, which captures the shops using the instantaneous coupon service, is built as follows:*

- *at time $t < 2$, $\overline{G}_{shops}(t)$ is the empty graph;*
- *at time $t \in [2, 13]$, $\overline{G}_{shops}(t)$ is the graph G_{shops} presented in Example 1 including both shops;*
- *at time $t > 13$, $\overline{G}_{shops}(t)$ is the RDF graph G'_{shops} in Listing 4.2, that includes only shop b.*

```
:b rdf:type :Shop .
:bob :own :b .
```

Listing 4.2: *RDF graph G'_{shops}*

4.3 On Publishing RDF Streams on the Web

While time-varying graphs capture existing RDF graphs, we asked ourselves if RDF streams are available on the Web. Surprisingly, public RDF Streams are generally missing in the landscape of RDF stream processing. Existing RSP engines have circumvented this issue in different ways, e.g. the SLD framework has internal components to lift data streams in RDF streams, while SPARQL$_{stream}$ uses the notion of virtual RDF streams. Other engines, like C-SPARQL and CQELS, expose programmatic APIs and delegate to the users the task to manage the streams and to feed the system through the relative API.

Inspired by the experience of the community in publishing static data sets as RDF data sets, we built Triple Wave[1], a framework to transform existing streams in RDF streams and publish them on the Web. The high velocity on which the data is generated makes it difficult to to permanently store the whole data stream and to publish it as Linked Data. To overcome this limit, we extend the initial proposal about publishing RDF streams in [21], defining two types of triple graphs: stream graphs and instantaneous graphs, as detailed in Section 4.3.2: the former represents a time-varying graph with the list of the recent content of the stream; the latter represents a data item of the stream. Triple Wave produces a JSON stream in the JSON-LD format, compliant with the model described in Definition 4.2. An advantage of using JSON-LD is on the possibility to process RDF streams not only with RSP engines, but also with existing frameworks and techniques for RDF processing (e.g. SPARQL engines).

As a case study, we consider the change stream of Wikipedia[2]. This stream features all the changes that occur on the Wikipedia website. This stream is characterized by heterogeneity: it comprehends not only elements related to the creation or modification of pages (e.g., articles and books), but also events related to users (new registrations and blocked users), and discussions among them.

```
{
  "page": "Naruto: Ultimate Ninja",
  "pageUrl": "http://en.wikipedia.org/wiki/Naruto:_Ultimate_Ninja",
  "url": "https://en.wikipedia.org/w/index.php?diff=669355471&oldid
      =669215360",
  "delta": -7, "comment": "/ Characters /",
  "wikipediaUrl": "http://en.wikipedia.org",
```

[1]Cf. https://streamreasoning.github.io/TripleWave/.
[2]Cf. https://en.wikipedia.org/wiki/Special:RecentChanges

```
  "channel": "#en.wikipedia", "wikipediaShort": "en",
  "user": "Jmorrison230582", "userUrl": "http://en.wikipedia.org/wiki/User/
      Jmorrison230582",
  "unpatrolled": false, "newPage": false, "robot": false,
  "namespace": "article"
}
```

Listing 4.3: *A fragment of the change stream of Wikipedia*

Listing 4.3[3] shows a fragment of the stream of changes of Wikipedia. In particular, it shows that the user `Jmorrison230582` modified an article of the `English` Wikipedia about `Naruto: Ultimate Ninja`. Furthermore, the delta attribute tell us that the user deleted some words, and the `url` attribute refers the to the Wikipedia page that describes the event.

4.3.1 R2RML to create RDF streams

Streams on the Web are available in a myriad of formats, so to adapt and transform them to RDF streams we use a generic transformation process that is specified as R2RML[4] mappings. Although these mappings were originally conceived for relational database inputs, we can use light extensions that support other formats such as CSV or JSON (e.g. as in RML[5]). The example below specifies how a Wikipedia stream update can be mapped to a graph of an RDF stream[6]. This mapping defines first a triple that indicates that the generated subject is of type `schema:UpdateAction`. The `predicateObjectMap` clauses add two more triples, one specifying the object of the update (e.g. the modified wiki page) and the author of the update. The graph is specified using the `graphMap` property.

```
:wikiUpdateMap a rr:TriplesMap; rr:logicalTable :wikistream;
rr:subjectMap [ rr:template "http://131.175.141.249/TripleWave/{time}";
rr:class schema:UpdateAction; rr:graphMap :streamGraph ];
rr:predicateObjectMap [rr:predicate schema:object; rr:objectMap [ rr:column
      "pageUrl" ]];
rr:predicateObjectMap [rr:predicate schema:agent;  rr:objectMap [ rr:column
      "userUrl"] ];.
```

Additional mappings can be specified, as in the example below, for providing more information about the user (e.g. user name):

[3]Data collected with the API provided by `https://github.com/edsu/wikistream`
[4]R2RML W3C Recommendation: `http://www.w3.org/TR/r2rml/`
[5]RML extensions: `http://rml.io`
[6]We use schema.org as the vocabulary in the example.

```
:wikiUserMap a rr:TriplesMap; rr:logicalTable :wikistream;
nlyyrr:subjectMap   [ rr:column "userUrl";
rr:class schema:Person; rr:graphMap :streamGraph ];
rr:predicateObjectMap [ rr:predicate schema:name; rr:objectMap  [ rr:column
     "user" ]];.
```

A snippet of the resulting RDF Stream graph, serialized in JSON-LD, is shown in in Listing 4.4.

```
{
  "http://www.w3.org/ns/prov#generatedAtTime": "2015-06-30T16:44:59.587Z",
  "@id": "http://131.175.141.249/TripleWave/1435682699587",
  "@graph": [
    { "@id": "http://en.wikipedia.org/wiki/User:Jmorrison230582",
      "@type": "https://schema.org/Person",
      "name": "Jmorrison230582"
    },{
      "@id": "http://131.175.141.249/TripleWave/1435682699587",
      "@type": "https://schema.org/UpdateAction",
      "object": {"@id": "http://en.wikipedia.org/wiki/Naruto_Ultimate_Ninja
          "},
      "agent":  {"@id": "http://en.wikipedia.org/wiki/User:Jmorrison230582
          "}
    }
  ],
  "@context": "https://schema.org/"
}
```

Listing 4.4: *Portion of the timestamped element in the RDF stream.*

4.3.2 Publishing stream elements as Linked Data

Triple Wave is implemented in Node.js and streams out the RDF stream using HTTP with chunked transfer encoding. Consumers can register at the endpoint `http://131.175.141.249/TripleWave/wiki.json` and receive the data following a push paradigm. In cases where consumers may want to pull the data, Triple Wave allows publishing the data accordingly to the Linked Data principles [27]. Given that the stream supplies data that changes very frequently, data is only temporarily available for consumption, assuming that recent stream elements are more relevant. In order to allow the consumer to discover which are the currently available stream elements, we use and extend the framework proposed in [21]. According to this scheme, Triple Wave distinguishes between two kinds of Named Graphs: the Stream Graph (sGraph) and the Instantaneous Graphs (iGraphs). Intuitively, an iGraph represents one stream data item, while the sGraph is an index to the recent content of the stream.

```
tr:sGraph  sld:contains (tr:1435682699954 tr:1435682699587) ;
sld:lastUpdate "2015-06-29T15:46:05"^^xsd:dateTime .
tr:1435682699587 sld:receivedAt "2015-06-30T16:44:59.587Z"^^xsd:dateTime .
tr:1435682699954 sld:receivedAt "2015-06-30T16:44:59.954Z"^^xsd:dateTime .
```

Listing 4.5: *The sGraph pointing to the iGraph described in Listing 4.4.*

As an example, for the Wikipedia RDF stream, the sGraph is published at the address `http://131.175.141.249/TripleWave/sGraph`. By accessing the sGraph, consumers discover which are the stream elements (identified by iGraphs) available at the current time instants. The sGraph in Listing 4.5 describes the sGraph and the current content that can be retrieved. The ordered list of iGraphs is modeled as an `rdf:list` with the most recent iGraph as the first element, and with each iGraph having its relative timestamp annotation. Next, the consumer can access the iGraphs dereferencing the iGraph URL address. As example, when the consumer accesses the graph[7] at `http://131.175.141.249/TripleWave/1435682699587`, it retrieves the content of the graph reported in Listing 4.4.

4.4 Remarks

In this section, we formalized the main concepts related to data in the RDF Stream Processing context, RDF Streams and Time-varying graphs. The former ones model infinite sequences of timestamped data items, that are the main input of RSP engines; the latter ones model the evolution of RDF graphs over time.

We then presented Triple Wave, a system that allows deploying RDF Streams on the Web, taking existing streams of non-RDF data and converting them to stream graphs using declarative mappings. Triple Wave allows publishing these streams as Linked Data, as well as a live stream of RDF graphs. We have implemented and deployed a use-case that feeds from a live stream of Wikipedia updates. This constitutes a first step towards ubiquitous deployment of RDF streams, based on the ever-increasing amount of data streams available on the Web and the upcoming Internet of Things. Moreover, the provision of RDF stream data will prompt new challenges to existing Linked Data solutions, and will contribute to the maturity of RSP technologies.

[7]This iGraph is expired at the time of submission. It is possible to consult the sGraph to get the list of the non-expired graphs.

CHAPTER 5

RSP-QL: A Continuous Extension of SPARQL to Unify Existing RSP Languages

In the previous chapter, we defined the RSP data model by adding the temporal dimension in the RDF model in three different ways: *timestamped RDF statements (graphs)* are RDF statements (graphs) with a time annotation; *RDF streams* are ordered sequences of timestamped RDF statements (graphs) and *time-varying and instantaneous RDF* graphs capture the evolution of RDF graphs over time. In this section, we present RSP-QL, an extension of the SPARQL language to unify existing RSP languages. The two main requirements that drive the design of RSP-QL are: 1) the evaluation of a query over an input data should produce a unique solution; 2) RSP-QL should capture the operational semantics of the most relevant extensions of SPARQL for RDF streams, i.e. C-SPARQL, CQELS and SPARQL$_{stream}$.

This chapter is based on "RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems", published on the International Journal of Semantic Web Information Systems. I drove this work. I wrote the first draft of the paper under the supervision of Prof. Della Valle, that supported me in finding a clear and readable way to expose the concepts in the paper, and provided the running example scenario. The article was later on edited by J.P. Calbimonte, that reviewed the draft and improved the running example across the paper. Finally, Prof. Óscar Corcho further improved the clearness and the style of the work.

5.1 From SPARQL to RSP-QL

One of the main differences between SPARQL and RSP-QL is the way in which queries are evaluated. Adopting the DSMS nomenclature [16, 36], SPARQL allows to issue *one-time queries*, i.e. queries that are evaluated once by the SPARQL engine. In contrast, RSP-QL allows to register *continuous queries*, queries issued once and continuously evaluated [16], i.e. they are evaluated multiple times and the answer is composed by emitting over time the results of each evaluation iteration.

Example 7. *(cont'd). The instantaneous discount coupon service offered by the Sirius Cybernetics Corporation allows shop owners to propose discounts to shoppers nearby, who use the App associated to the service, e.g., when their shops are empty. Those instantaneous coupons are published in a social network and the RDF stream S_{social} in Listing 5.1 shows two of them.*

```
:ds1 {:post1  :author  :alice ; :contains  :c1  .
       :c1  :in  :a ; :on  :armadillo ; :reduce 30 } [8]  .
:ds2 {:post2  :author  :bob ; :contains  :c2  .
       :c2  :in  :b ; :on  :panda ; :reduce 25 } [15]  .
```

Listing 5.1: *The RDF stream S_{social}*

The first item, d_{s1}, states that Alice publishes at time instant a message with a coupon for a 30% discount on Armadillo shoes, while the second item reports on a offer from Bob at the time instant 15 for a 25% discount on Panda glasses.

Sirius Cybernetics Corporation monitors: (i) the streams S_{nearby} (the shops with shoppers nearby) and S_{social}, and (ii) the time-varying graph $\overline{G}_{shops}$ (the shops that use the service, which are changing over time). It sends instantaneous coupons proposed by the shop owners to shoppers nearby their shops. This query requires a continuous evaluation, because it has to notify coupons to shoppers, who are nearby a shop, every time that a shop proposes a new coupon and it has to notify shoppers, who get nearby a shop, with the most recent coupons of that shop.

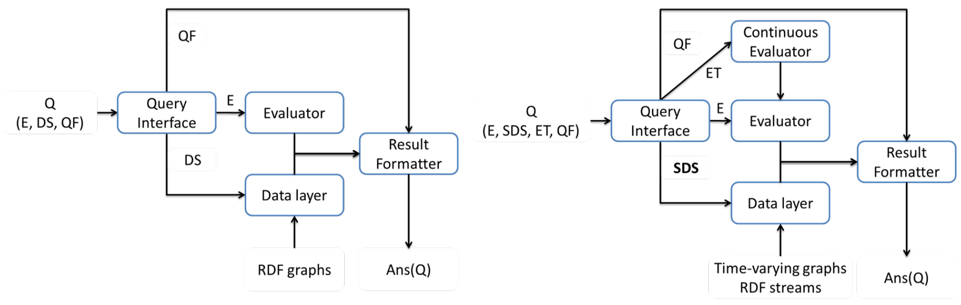

Figure 5.1: *From SPARQL (left) to RSP-QL engine (right)*

We present the definition of RSP-QL query, which extends the notion of SPARQL query presented in the Section 2.1.

Definition 5.1 (RSP-QL query)**.** *An* RSP-QL query *Q is defined as (SE, SDS, ET, QF) where*

- *SE is an RSP-QL algebraic expression*
- *SDS is an RSP-QL dataset*
- *ET is the sequence of time instants on which the evaluation occurs*
- *QF is the Query Form*

The continuous-evaluation paradigm influences the definition of RSP-QL query. We go in depth in the remaining of the section, but we provide now some intuitions about this model. We refer to Figure 5.1 to highlight the difference of the RSP-QL query model w.r.t the SPARQL one.

First, the dataset has to take into account time, both to manage the RDF streams and to cope with time-varying RDF graphs. Next, we need to define the continuous query evaluation semantics. It requires two main operations: we need to extend the one-time SPARQL evaluation semantics, and we need to let the SPARQL operators process time-changing data. Regarding the first point, we exploit the ET time instant sequence and push it in the evaluation process of SPARQL. To maintain backward compatibility with the SPARQL query model, we do not modify the SPARQL operators, but we work on their inputs and outputs. As we see above, most of the RSP-QL operators are compliant with the relative SPARQL ones. The intuition behind this choice is that the continuous evaluation can be viewed as a sequence of instantaneous evaluations, so, fixed a time instant, the operators can work in a time-agnostic way.

Last, we need to work on the output of the query, in order to generate streams as output. We do it by introducing the new class of *streaming operators that takes as input sequences of solution mappings and produces sequences of timestamped solution mappings; finally, we extend the SPARQL query forms to be able to convert a sequence of time-annotated solution mappings into a RDF stream.

5.2 Assumptions

In the rest of this section, we make the two assumptions. The first assumption is related to the time required for query evaluation: we assume that the time required to evaluate the query over the current input and to produce the portion of answer is lower than the time unit. It is a common assumption made in this kind of work [39], so as to guarantee that the stream processor works without accumulating delays in the continuous-evaluation process.

We also assume no duplicates in the same window. This assumption is made for the sake of simplicity and to describe the RSP-QL model exploiting the notion of RDF graph, that is a set of RDF statements (and not a bag, like relations in DSMS). This constraint influences the result of query processing with some queries (such as the ones with aggregations). Anyway, as we explain in Section 5.10, this constraint can be relaxed by introducing simple bookkeeping information, i.e. statement counters.

5.3 RSP-QL dataset

The addition of the time dimension and the presence of RDF streams requires a new notion of RDF dataset that determines the input data of the RSP-QL query. We introduce the concept of window over a stream, that creates RDF graphs by extracting relevant portions of the stream.

Definition 5.2 (Fixed window). *A* fixed window *(or window) $W(S)$ is a set of timestamped RDF graphs extracted from a RDF stream S. A* time-based window *$W^{(o,c]}$ is a fixed window defined through two time instants o, c (respectively named opening and closing time instants) such that:*

$$W^{(o,c]}(S) = \{(d,t) | (d,t) \in S \wedge t \in (o,c]\}$$

A time-based window selects a portion of the data contained in the stream.

Example 8. *(cont'd) Listing 5.2 shows content of the time-based window $W^{(4,8]}$ over S_{nearby} with opening time $o = 4$ and $c = 8$.*

```
:dn3 {:carl  :isNearby  :a} [5] .
:dn4 {:eve   :isNearby  :a} [7] .
```

Listing 5.2: *Content of the window $W^{(4,8]}(S_{nearby})$*

In order to be able to process the content of the stream at different time instants, we need an operator that creates multiple (fixed) windows over the stream. This operator is called *time-based sliding window*, and it operates by creating a sequence of time-based windows (with different opening/closing time instants) over the stream.

Definition 5.3 (Time-based sliding window)**.** *A* time-based sliding window $\mathbb{W}$ *takes as input a stream S and, given a time instant t produces a time-based window $W^{(o,c]}(S)$ (Figure 5.2). $\mathbb{W}$ is defined through a set of parameters (α, β, t^0), where:*

- *α is the width parameter*
- *β is the slide parameter*
- *t^0 is the time instant on which $\mathbb{W}$ starts to operate*

We denote with $\mathbb{W}(S)$ the application of sliding window $\mathbb{W}$ on the stream S, and with $\mathbb{W}(S, t)$ the time-based window produced at time instant t

The above definition explains which are the inputs and outputs of the sliding window. In the following, we refer to time-based window (Definition 5.2) with the term fixed window, to make clearer the distinction between them and the time-based sliding windows (Definition 5.3). To distinguish the fixed windows from the time-based sliding windows we use the capital letter, e.g. $W^{(o,c]}$, $W_1^{(o,c]}$, and $W_2^{(o,c]}$ for the former and the blackboard bold letter, e.g., $\mathbb{W}$, $\mathbb{W}_1$ and $\mathbb{W}_2$ for the latter. For the sake of readability, we omit the superscript interval $(o, c]$ in the time-based windows when not strictly required. Next, we provide the description of how time-based sliding windows work.

Given a time-based sliding window $\mathbb{W}$ defined by α, β and t^0, the RSP engine generates a sequence of fixed windows $(W_1, W_2, \ldots)$, defined through the following constraints:

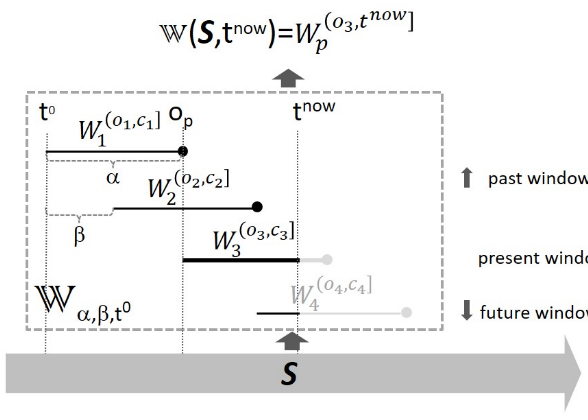

Figure 5.2: *Fixed window and sliding window.*

- the opening time of the first window $W_1^{(o_1,c_1]}$ is $t^0 = o_1$
- each window has width α, i.e. for each window W_i of $\mathbb{W}$ defined through $(o_i, c_i]$, $i \in \mathbb{N}^+$, it holds:

$$c_i - o_i = \alpha$$

- the difference between the opening time instants of two consecutive windows is β, i.e. given two windows $W_i^{(o_i,c_i]}$ and W_{i+1} of $\mathbb{W}$, defined respectively in the time intervals $(o_i, c_i]$ and $(o_{i+1}, c_{i+1}]$, holds:

$$o_{i+1} - o_i = \beta$$

Given a time instant t^{now}, we name *present window*[1] the window W_p of $\mathbb{W}$ defined with $(o_p, c_p]$ such that o_p is the most recent opening time, i.e.

$$\nexists W_j \text{ of } \mathbb{W} \text{ defined in } (o_j, c_j] : o_p < o_j < t^{now}$$

It is now possible to describe the interval of the fixed window produced by a time-based sliding window $\mathbb{W}$ at time t^{now}. The window $W(S, t^{now})$ is a sub-window of the present window $W_p^{(o_p,c_p]}$:

$$\mathbb{W}(S, t^{now}) = W^{(o_p,t^{now}]} \subseteq W_p^{(o_p,c_p]}$$

[1]In DSMS literature this window is known as *active window*; we changed its name in order to clearly distinguish it from the *active graph* notion as in Definition 2.4

Example 9. *(cont'd) The time-based sliding window $\mathbb{W}_1$ over S_{nearby} defined through ($\alpha = 5, \beta = 2, t^0 = 1$) generates the following fixed windows: W_1 between $(1, 6]$, W_2 between $(3, 8]$, W_3 between $(5, 10]$ and so on. If $\mathbb{W}_1$ is evaluated at the time instant 9, the present window is W_3 and $\mathbb{W}_1(S_{nearby}, 9)$ contains:*

```
:dn4 {:eve  :isNearby   :a} [7] .
```

To summarise, given a time instant t and a time-based sliding window $\mathbb{W}$ defined through the parameters (α, β, t^0), $\mathbb{W}$ takes as input a stream S, produces a fixed window $\mathbb{W}(S, t) = W_p^{(o,c]}$ with a sequence of timestamped items (d_i, t_i) of S such that $o < t_i \leq c$. Finally, we can define the concept of RSP-QL dataset.

Definition 5.4 (RSP-QL dataset)**.** *An* RSP-QL dataset *SDS is a set composed by an (optional) default element A_0, n ($n \geq 0$) named time-varying graphs and m ($m \geq 0$) named sliding windows over $o \leq m$ streams:*

$$\begin{aligned} SDS =&\{(def, A_0),\\ &(u_1, \overline{G}_1), \ldots, (u_n, \overline{G}_n),\\ &(w_1, \mathbb{W}_1(S_1)), \ldots, (w_j, \mathbb{W}_j(S_1)),\\ &(w_{j+1}, \mathbb{W}_{j+1}(S_2)), \ldots, (w_k, \mathbb{W}_k(S_2)),\\ &\ldots\\ &(w_l, \mathbb{W}_l(S_o)), \ldots, (w_m, \mathbb{W}_m(S_o))\} \end{aligned} \tag{5.1}$$

where

- *$def \notin I$ is a special symbol used to denote the default element A_0*
- *u_p, w_q are IRIs ($u_p, w_q \in I$) for each $p \in [1, n]$ and $q \in [1, m]$*
- *$(u_p, \overline{G}_p)$ identifies a time-varying named graph, for each $p \in [1, n]$*
- *$(w_q, \mathbb{W}_q(S_r))$ identifies a named time-based sliding window over an RDF stream, for each $q \in [1, m]$ and $r \in [1, o]$*

When composing a SPARQL query, it is possible to declare different graphs that will be merged to compose the default graph. In this case, the default graph notion is replaced by the default element one, denoted A_0: it is a collection of time-varying graphs and sliding windows. In the evaluation process, the A_0 is processed to generate a RDF graph to be queried. Intuitively, it merges the content of the unnamed time-varying graphs at time t and the RDF graphs contained in the present windows of the unnamed sliding windows.

Example 10. *(cont'd) One of the RSP-QL dataset that can be built to answer the query presented in Example 7 is:*

$$\begin{aligned} SDS = &\{(def, \overline{G}_{shops}),\\ &(w_1, \mathbb{W}_1(S_{nearby})), (w_2, \mathbb{W}_2(S_{social}))\} \end{aligned}$$

where the default graph is the time-varying graph describing the shops that use the instantaneous discount coupon service, w_1 and w_2 identify respectively the sliding window $\mathbb{W}_1$ over the stream S_{nearby}, and the sliding window $\mathbb{W}_2$ over the stream S_{social}. $\mathbb{W}_2$ is defined as $(\alpha = 2, \beta = 2, t^0 = 0)$.

5.4 Time-varying collections of solution mappings

After defining the notion of RSP-QL dataset, we move to the query evaluation process. In this section, we treat the problem of processing data that change over time; in the next section, we extend the SPARQL evaluation semantics in order to support the continuous evaluation.

As explained in Section 5.1, the continuous query evaluation consists in evaluating the query multiple times at different instants. At each iteration, fixed a time instant, the RSP-QL engine can determine on which data the algebraic expression should be evaluated and from now on, the evaluation process is atemporal. In other words, we need to push the time dimension in the data types exchanged by the operators and we do not need to redefine the existing SPARQL 1.1 operators to work with timestamped data. This approach is different from the one in [28], where the authors redefine the SPARQL algebraic operators in order to cope with streaming data.

As explained in the background section, SPARQL algebra operators work on RDF graphs or on collections of solution mappings [63]. For example, BGPs receive as input the active RDF graphs and produce as output multisets (bags) of solution mappings; JOIN, UNION and DIFF operators consume and produce multisets of solution mappings; and ORDER BY and DISTINCT operators work on sequences of solution mappings.

In the previous section, we introduced the notions of time-varying and instantaneous RDF graphs, to take into account the time dimension: the time-varying RDF graph $\overline{G}$ is a mapping between the time and the RDF graph set, and given a time instant t the instantaneous graph $\overline{G}(t)$ identifies an RDF graph. A BGP operator, as defined in the SPARQL specification, can operate over an instantaneous RDF graph (since it is an RDF graph). Generalising, in our model each operator processes instantaneous inputs and produces instantaneous outputs; the sequence of instantaneous inputs (outputs) at different time instants are time-varying inputs (outputs). It follows that we need to define the time-varying and instantaneous extensions for multisets (identified respectively by $\overline{\Omega}$ and $\overline{\Omega}(t)$) and on sequences (identified by $\overline{\Psi}$ and $\overline{\Psi}(t)$) of solution mappings. It is worth noting that we are modifying the concept of collections of solution mappings and not the definition of solution mapping itself: it is a key-point to guarantee that the existing SPARQL 1.1 operators continue to work as by their original definition.

Definition 5.5 (Time-varying collections of solution mappings)**.** *A* time-varying sequence of solution mappings $\overline{\Psi}$ *maps time instants $t \in T$ to the set of solution mapping sequences:*

$$\overline{\Psi} : T \to \{\Psi \mid \Psi \textit{ is a sequence of solution mappings}\}$$

Given a time-varying sequence of solution mapping $\overline{\Psi}$, we use the term instantaneous sequence of solution mappings $\overline{\Psi}(t)$ *to refer to the sequence of solution mappings at time t.*

A time-varying multiset of solution mappings $\overline{\Omega}$ *maps the time T to the set of solution mapping multisets*

$$\overline{\Omega} : T \to \{\Omega \mid \Omega \textit{ is a multiset of solution mappings}\}$$

Given a time-varying multiset of solution mappings $\overline{\Omega}$, we use the term instantaneous multiset of solution mappings $\overline{\Omega}(t)$ *to refer to the multiset of solution mappings at time t.*

The RSP-QL model ensures that existing SPARQL 1.1 operators continue to work on RDF statements and solution mappings. As example, in the following, we show the Join definition adjusted to take into account time-aware collections of inputs (outputs). The differences w.r.t. SPARQL 1.1 Join definition presented in Section 2.1 are underlined.

Example 11. *For a given time instant t, let $\underline{\overline{\Omega}_1(t)}$ and $\underline{\overline{\Omega}_2(t)}$ be instantaneous multisets of solution mappings. We define* RSP-QL Join *as:*

$$\begin{aligned}\text{Join}(\underline{\overline{\Omega}_1(t)}, \underline{\overline{\Omega}_2(t)}) = \{\mu_1 \cup \mu_2)| \\ \mu_1 \in \underline{\overline{\Omega}_1(t)} \wedge \mu_2 \in \underline{\overline{\Omega}_2(t)} \wedge \mu_1 \sim \mu_2\}\end{aligned}$$

Notably, fixed a time instant t, $\overline{\Omega}_1(t)$ and $\overline{\Omega}_2(t)$ are two bags of solution mappings: the Join operator works on their content (solution mappings) as in the original definition.

5.5 Continuous evaluation semantics

At this point, RSP-QL operators can process instantaneous inputs and produce instantaneous outputs. What we need to do now is to model the continuous evaluation process. To do it, we include the evaluation time in the SPARQL evaluation semantics; then, we explain that the continuous query answering is done by executing the query at each time instant of the sequence ET (the evaluation time instants defined in the RSP-QL query presented at the beginning of the section).

We now extend the definition of SPARQL evaluation semantics (Definition 2.6) to take into account the time dimension: we add a third parameter, evaluation time t, in the *eval* function signature.

Definition 5.6 (RSP-QL evaluation semantics)**.** *Given an RSP-QL dataset SDS, an algebraic expression SE and an evaluation time instant t, we define*

$$[\![SE]\!]^t_{SDS(A)}$$

as the evaluation of SE at time t with respect to the RSP-QL dataset SDS having active element A (that can be either a time-varying graph or a window operator).

This new concept requires a revision of the definitions of the existing SPARQL evaluation of algebraic operators. For the sake of brevity, we show the continuous evaluation semantics of BGP and Join operators.

Definition 5.7 (Evaluation of BGP)**.** *The* evaluation of a Basic Graph Pattern *operator is defined in the following way:*

$$[\![\text{BGP}]\!]^t_{SDS(A)} = [\![\text{BGP}]\!]_{SDS(A,t)}$$

The solution of the BGP *is computed with regard to the RSP-QL dataset SDS having $\overline{G}$ as active element A at time t, i.e. $SDS(A,t)$*

denotes a RDF graph. The output of the evaluation is an instantaneous multiset of solution mappings $\overline{\Omega}(t)$.

If A is a time-varying graph $\overline{G}$, $SDS(A,t)$ refers to the instantaneous graph $\overline{G}(t)$ at time t; otherwise, if A is a sliding window $\mathbb{W}$ over a stream S, $SDS(A,t)$ is an RDF graph obtained by merging all the graphs of the time-based window $\mathbb{W}(S,t)$. To sum up:

$$SDS(A,t) = \begin{cases} \overline{G}(t), & \textit{if } A \textit{ is a time-varying graph } \overline{G} \\ \bigcup_{(G_i,t_i)\in\mathbb{W}(S,t)} G_i, & \textit{if } A \textit{ is a sliding window } \mathbb{W}(S) \end{cases}$$

The above evaluation of the BGP is important because it shows that the BGP is evaluated over an RDF graph. The evaluation definitions of the other existing SPARQL 1.1 algebraic operators propagates the evaluation time to the evaluation of the algebraic expressions.

It is worth noting what happens when A is a sliding window. At time t, the sliding window produces a present window. All the stream items (i.e. RDF graphs) in the present window are merged together according to the RDF merge operation[2] [64]. The result is a RDF graph. When the active element is the default element A_0, for each time-varying graph and sliding window a RDF graph is computed; finally, the graphs are merged together in a new RDF graph.

The next definition presents the evaluation of the join operator.

Definition 5.8 (Evaluation of Join)**.** *The* evaluation of Join *is defined as follows:*

$$[\![\mathrm{Join}(P_1,P_2)]\!]^t_{SDS(A)} = \mathrm{Join}([\![P_1]\!]^t_{SDS(A)}, [\![P_2]\!]^t_{SDS(A)})$$

where $SDS(A)$ indicates the active element A in the RSP-QL dataset SDS and P_1, P_2 are graph patterns.

In the algebraic tree the Join operators have two children, represented by the two graph patterns P_1 and P_2. The evaluation of the Join operator consists in applying the Join (Definition 11) to the two multisets of solution mappings computed by evaluating P_1 and P_2 at time t with regards to the RSP-QL dataset with active element A.

5.6 Streaming operators

In previous sections, we extended the query model of SPARQL to consume dynamic data (data that changes over time) and to process it

[2]It follows that that blank nodes cannot be shared across stream items. We plan to investigate in future work if and how unnamed resources can be shared among stream items.

in a continuous fashion. Now, we need to define what is the output of the query: our model should produce not only time-varying RDF graphs, but also RDF streams. To enable this feature, we add a set of *streaming operators, that are similar to the relation-to-stream operator described in Section 3.1.1. The *streaming operators take as input sequences of solution mappings and produce sets of timestamped solution mappings. As we see above, in the RSP-QL model the timestamped solution mappings can be processed only by the Query Form operators. That means, the *streaming operators are thought to be the outer elements of the algebraic trees.

Despite other RSP-QL operators that are inherited by SPARQL, the *streaming operators require a time instant as input parameter because they are time-aware: they need to know the current evaluation time in order to produce their outputs. They reintroduce the temporal dimension in the data, appending a time instant on the solution mappings; *streaming operators can be considered the dual operators of sliding windows, that process timestamped RDF graphs removing the time annotation. We maintain the relation-to-stream names and we redefine them to work in the RSP-QL setting. We start by defining the RStream operator.

Definition 5.9 (RStream). *Let $\overline{\Psi}$ be a time-varying sequence of solution mappings and $t \in T$ the evaluation time instant. We define* RStream *in the following way:*

$$\text{RStream}(\overline{\Psi}(t), t) = \{(\mu, t) | \mu \in \overline{\Psi}(t)\}$$

We define the RStream evaluation semantics *as follows:*

$$[\![\text{RStream}(L)]\!]^t_{SDS(A)} = \text{RStream}([\![L]\!]^t_{SDS(A)}, t)$$

where L is a solution sequence.

The RStream operator is the simplest one among the three that we present in this section. It takes as input a sequence of solution mappings $\Psi(t)$ and annotates each of them with the evaluation time t. This operator allows streaming out the whole answer produced at each evaluation iteration.

Definition 5.10 (IStream). *Given a time-varying sequence of solution mappings $\overline{\Psi}$ and two consecutive time instants t_{j-1} and t_j in the ET sequence (i.e. there is no time instant $t \in ET$ such that $t \in (t_{j-1} <$*

$t_j)^3$), we define the IStream *operator as follows:*

$$\begin{aligned}&\text{IStream}(\overline{\Psi}(t_{j-1}), \overline{\Psi}(t_j), t_j) = \\ &\qquad \{(\mu, t_j) | \mu \in \overline{\Psi}(t_j) \wedge \mu \notin \overline{\Psi}(t_{j-1})\}\end{aligned}$$

and we define the IStream evaluation semantics *as follows:*

$$[\![\text{IStream}(L)]\!]^{t_j}_{SDS(A)} = \text{IStream}([\![L]\!]^{t_{j-1}}_{SDS(A)}, [\![L]\!]^{t_j}_{SDS(A)}, t_j)$$

IStream streams out the difference between the answer of the current evaluation and the one of the previous iteration. IStream generally produces shorter answers and it is used in cases where it is important to put the focus on what is new.

Definition 5.11 (DStream)**.** *Given a time-varying sequence of solution mappings $\overline{\Psi}$ and two consecutive time instants t_{j-1} and t_j in the ET sequence, we define the* DStream *operator as follows:*

$$\begin{aligned}&\text{DStream}(\overline{\Psi}(t_{j-1}), \overline{\Psi}(t_j), t_j) = \\ &\qquad \{(\mu, t_j) | \mu \notin \overline{\Psi}(t_j) \wedge \mu \in \overline{\Psi}(t_{j-1})\}\end{aligned}$$

We define the DStream evaluation semantics *as following:*

$$[\![\text{DStream}(L)]\!]^{t_j}_{SDS(A)} = \text{DStream}([\![L]\!]^{t_{j-1}}_{SDS(A)}, [\![L]\!]^{t_j}_{SDS(A)}, t_j)$$

The output produced by DStream is the part of the answer at the previous iteration that is not in the current one (for example, a continuous query over $\overline{G}_{shops}$ to stream out which discount coupons end).

5.7 Query Form

Depending on the presence of the *streaming operator, the output of each evaluation of the algebraic expression E of the query can be a either a sequence of solution mappings or a sequence of timestamped solution mappings.

If the algebraic expression SE does not contain the *streaming operator, a case allowed by C-SPARQL and SPARQL$_{stream}$, at each iteration the query produces a compliant SPARQL answer, i.e. a variable binding for SELECT, a boolean value for ASK and a set of RDF

[3]Consequently, $\overline{\Psi}(t_i)$ and $\overline{\Psi}(t_j)$ exists and $\overline{\Psi}(t_k)$ is undefined $\forall t_k \in T$ such that $t_{j-1} < t_k < t_j$.

statements for CONSTRUCT and DESCRIBE. This decision preserves interoperability between RSP systems and SPARQL engines.

When the *streaming operator is in the algebraic expression the output of the RSP-QL engine is a stream: at each instantaneous evaluation, the engine appends a new set of elements at the output stream. Similarly to the first case, the output format depends on the query form. In the SELECT case, the output is a relational data stream, in the case of ASK is a stream of boolean values, and finally, in the case of CONSTRUCT/DESCRIBE, the output is an RDF stream. It is worth noting that only in the last case the output can be consumed by another RSP-QL engine; in the case of SELECT/ASK query forms, the output stream can feed a relational stream processor.

5.8 Evaluation time instants

We defined ET as the sequence of time instants at which the evaluation occurs. It is an abstract concept which is key to the RSP-QL query model and its continuous-evaluation semantics, but it is hard to use it in practice when designing the RSP-QL syntax. In fact, the ET sequence is potentially infinite, so the syntax needs a compact representation of this set. Moreover, the ET set could be unknown when the query is composed: the time instants on which the query has to be evaluated could depend on the data that is streaming through the RSP engine, e.g. the query should be evaluated every time the window content changes. In other words, query designers can be interested in associating the query evaluation to some relevant events, that can be known a priori (e.g. periodical evaluation) or not (e.g. status of the window content).

To address this issues, we introduce in RSP-QL the notion of *policy* to express the time instant set ET. The concept was initially proposed by Botan et al. in SECRET [29].

Definition 5.12 (Evaluation Time Instant Set). *A* policy *P is a combination of one or more boolean conditions (shortly* strategy*, according to the SECRET model) that allow identifying the potentially infinite set of time instants ET. Each strategy is associated to a window and could set constraints to the window content or its parameters. Given a policy P, the* evaluation time instant set *ET_P is the set of time instants on which the policy in P is satisfied, i.e.*

$$t \in ET_P \textit{ iff } P \textit{ is satisfied at time } t$$

We can now indicate with $Q = (SE, SDS, ET_P, QF)$ the RSP-QL query where ET_P is represented through the policy P. The four strategies presented in SECRET are:

CC Content Change: the sliding window reports if the content changes.

WC Window Close: the sliding window reports if the present window closes.

NC Non-empty Content: the sliding window reports if the present window is not empty.

P Periodic: the sliding window reports only at regular intervals.

Each window can have zero or more policies associated to it, and the policy P is a combination of them, as we show in the next example.

Example 12. *(cont'd) Let's define the policy P for the query described in Example 7: it should detect the shoppers recently spotted in nearby shops that offer instantaneous discount coupons. The query involves two streams (S_{nearby} and S_{social}) and for each of them there is an associated sliding window $\mathbb{W}_1(S_{nearby})$ and $\mathbb{W}_2(S_{social})$ with parameters ($\alpha_1 = 5$, $\beta_1 = 2$, $t_1^0 = 1$) and ($\alpha_2 = 2$, $\beta_2 = 2$, $t_2^0 = 0$). As policy P, we set the Window Close and the Non-empty Content to $\mathbb{W}_2(S_{social})$:*

$$P = WC[\mathbb{W}_2(S_{social})] \wedge NC[\mathbb{W}_2(S_{social})]$$

As a result, the ET_P set contains pair time instants such that the window content of $\mathbb{W}_2(S_{social})$ is not empty. There fact that there are no conditions over S_{nearby} implies that it does not contribute to the report strategy. When the policy triggers, the instantaneous evaluation starts, and the content of the present window of S_{nearby} is retrieved.

5.9 RSP-QL query evaluation

We can now put all the pieces together and explain how an RSP-QL query is evaluated by an engine.

Definition 5.13 (Continuous Evaluation). *Let Q a continuous query $Q = (SE, SDS, ET, QF)$, where SE is an RSP-QL algebraic expression, SDS is an RSP-QL dataset (Definition 5.4), ET is the sequence of evaluation time instants (Definition 5.12) and QF is the query form. The* continuous evaluation *of Q produces an output $Ans(Q)$ and it is computed in the following way: for each $t \in ET$,*

1. *evaluate the algebraic expression E over the RSP-QL dataset, as explained in the continuous evaluation semantics:*

$$[\![SE]\!]^t_{SDS(A_0)}$$

2. *each operator works on instantaneous collections of inputs (e.g. RDF statements, solution mappings) and produce instantaneous collections of outputs accordingly to the definition of time-varying solution mappings.*

3. *format the output of evaluation according to the Query Form QF: if the algebraic expression has as outer element a *streaming operator the output is a stream; otherwise it is a SPARQL compliant answer.*

Example 13. *(cont'd) We can now sum up and formalise the query described in Example 7. The query Q^{ex} is defined through the four parameters (SE, SDS, QF, ET). The RSP-QL dataset of Q^{ex} is the one described in Example 26:*

$$\begin{aligned} SDS =&\{\overline{G}_0 = \overline{G}_{shops}, \\ &(w_1, \mathbb{W}_1(S_{nearby})), (w_2, \mathbb{W}_2(S_{social}))\}, \end{aligned}$$

where $\mathbb{W}_1$ and $\mathbb{W}_2$ are defined respectively through ($\alpha = 5$, $\beta = 2$, $t^0 = 1$) and ($\alpha = 2$, $\beta = 2$, $t^0 = 0$).

Instead of the RSP-QL algebraic expression, we write the WHERE *clause in a SPARQL-like syntax shown in Listing 5.3*

```
WHERE{
WINDOW :w1 { ?shopper :isNearby ?shop  }
WINDOW :w2 { ?post :author ?shop_owner ; :offers ?coupon }
?shop_owner :owns ?shop
}
```

Listing 5.3: WHERE *clause capturing the conditions of the running example.*

For simplicity, we set RStream *as *streaming operator,* SELECT *as Query Form, all the variables are projected. The clause matches shoppers nearby shops whose owner is offering an instantaneous discount coupon. The sequence of solution mappings are streamed out using the* RStream *operator.*

The evaluation time instant set is defined through the policy:

$$P = WC[\mathbb{W}_2(S_{social})] \wedge NC[\mathbb{W}_2(S_{social})]$$

?shopper	?shop	?post	?shop_owner	?coupon	timestamp
:*carl*	:*a*	:$post_1$	:*alice*	:c_1	8
:*eve*	:*a*	:$post_1$	:*alice*	:c_1	8
:*diana*	:*b*	:$post_2$	:*alice*	:c_2	16

Table 5.1: *Timestamped solution mappings computed in the continuous evaluation process.*

Now, we can apply Definition 5.13 and determine which is the unique correct answer of Q^{ex}. First, the ET set is determined; given the policy P, the input data and the sliding window definitions, it follows that the evaluation occurs at time 8 and 16. At these time instants, the algebraic expression is evaluated over the RSP-QL dataset. The computed timestamped solution mappings are shown in Table 5.1. Finally, the timestamped solution mappings are appended at the $Ans(Q^{ex})$ stream.

5.10 Relaxing the no duplicate data assumption

To close this section, we explain how to relax the assumption presented at the beginning of this section: the input data does not contain duplicates. We made this assumption to explain the RSP-QL model using concepts familiar to the reader, in particular the one of RDF graph. We use RDF graphs to represent the content of the sliding windows over the streams and this choice allowed us to use well-known operations such as RDF graph merging and basic graph pattern evaluation. Anyway, RDF defines the concept of RDF graph as a set of RDF statements, and consequently no duplicates are admitted.

Relaxing the constraint, we can cope with the presence of duplicates by introducinga simple bookkeeping mechanism and by annotating RDF statements with the number of repetitions in the windows. To put this new annotation, it is necessary to extend several components of the RSP-QL model. For example, sliding windows should initialise the counter; the evaluation semantics of aggregates and joins has to take the counters into account; in RDF graph merging, if both RDF graphs to be merged contain the same statement, then the relative counters have to be summed up.

CHAPTER 6

RSEP-QL: Complex Event Processing Operators in RSP-QL

Luckham [76] proposes one of the first definitions of event in the context of CEP, as:

> an object that represents, or records an activity that happens, or is thought of as happening to.

After him, several definitions were proposed, with several level of details (distinguishing simple/primitive events and complex events). In this section, we keep a general notion of event and we consider as event as a set of statements captured in one information item of the stream. That means, an event is a set of one or more statements in one of the items (i.e., an RDF Graph) of an RDF stream.

In this chapter, we study how to integrate CEP opreators in RSP-QL. In particular, we stud the SEQ operator, being the most basic building block in CEP. Even if it may seem straightforward to formalize this operator, its execution in different engines produces different and hardly comparable results. We therefore register the presence of this operator is two systems, EP-SPARQL and C-SPARQL. Interestingly, the two SEQ operators behave in different ways, and we aim at capturing all of them.

As explained in Section 3.2.2, EP-SPARQL is the RSP language with largest support for CEP features, with a wide range of operators to define complex events, e.g., SEQ, OPTIONALSEQ, EQUALS and EQUALSOPTIONAL. EP-SPARQL supports three different policies – unrestricted, recent and chronological – that determine its behaviour.

On the other hand, C-SPARQL is based on DSMS techniques, but it has a naive support to some CEP features. C-SPARQL implements a function, named timestamp, which takes as input a triple pattern and returns the time instant associated to the *most recent* matched triple. This function can be used inside a FILTER clause to express time constraints among events. The evaluation in C-SPARQL strictly relies on the notion of time-based sliding window, which selects a portion of the stream to be used as input and the time instants on which evaluations occur.

While EP-SPARQL is an engine built and modeled to perform CEP operations, C-SPARQL is a DSMS-inspired RSP engine that offers a naive support to event pattern matching. As shown above, even when the event pattern to be matched is simple, the two systems behave in completely different ways, and none of them is able to capture the other. It is out of the scope of this paper to determine which system is the most suitable to be used given a use case and the relative set of requirements. Our goal is to build a model able to capture the behaviour of both engines.

Example 14. *The Sirius Cybernetics Corporation aims at improving their mobile application with a social recommender. Shop owner discounts are published as micro-post in a social network, and the user App finds out whether the coupons are socially relevant by checking whether someone whom the user follows repost about the coupons. If yes, it delivers the coupons to the user.*

The $G_{followers}$ graph captures the following relation between the users in the social network. Listing 6.1 shows $G_{followers}$: Carl follows both Diana and Eve, while $:evefollows\ :carl$. It holds that $\overline{G}_{followers}(t) = G_{followers}$ for every t in $[0, 20]$.

```
:carl  :follows  :diana  .
:carl  :follows  :eve  .
:eve  :follows  :carl  .
```

Listing 6.1: *The $G_{followers}$ graph.*

As shown in Example 7 the stream S_{social} describes the publications of two coupons, respectively at time instant 8 *by Alice and at time* 10 *by Bob. Listing 6.2 shows the portion of S_{social} after* 15*. There are two new items at time* 16 *and* 18*, stating that Diana and Eve mentions the Bob's coupon in two new micro-posts.*

```
:ds3 {:post3 :author :diana ; :mentions :c2 .
      :c2 :in :b ; :on :panda ; :reduce 25 } [16] .
:ds4 {:post4 :author :eve ; :mentions :c2 .
      :c2 :in :b ; :on :panda ; :reduce 25 } [18] .
```

Listing 6.2: *Portion of the S_{social} stream.*

Carl has the App installed on his mobile phone and he is walking near shop b at time 18 *(as described in S_{nearby} in Example 5). The application of Carl monitors the social network to discover if some of the users he follows retweet any coupon (event E_1) of a shop nearby him (event E_2). The two events should be in a temporal sequence (i.e. E_1 before E_2): in this case, there are matchings (both the content of d_{s3} and d_{s4} can verify the pattern). To reduce the number of messages, the application should produce only one notification per social event: given the situation above, one notification has to be produced at the first time Carl goes near the shop aafter 16 and one at the first time Carl goes near the shop bafter 18. Carl is nearby shop B at time 19 and 21, as described in S_{nearby}, so one matching has to be produced at 19 and no notification should be produced at 21.*

In the following, we first introduce the landmark windows in Section 6.1, a class of window operators typical of the Event Processing domain. Then, Section 6.2 then move to the syntax of the operators needed to express event patterns in RSEP-QL queries in , and we then define their semantics in Section 6.3. We close with final remarks in Section 6.6

This chapter is based on a joint research with M. Dao-Trao, J.P. Calbimonte, E. Della Valle and D. Le Phouc. M. Dao-Trao and I drove the work. We defined the syntax and the semantics of the operators. M. Dao-Trao and J.P Calbimonte wrote the running examples along the article; D. Le Phouc and E. Della Valle edited and improved the draft.

6.1 Landmark window operators

Before moving on the definition of the semantics of the event patterns, we introduce two new window operators, the landmark windows and the within windows. These operators are typical of CEP world: the former captures the whole stream from a starting point, while the latter defines a time-based fixed window in the context of an event pattern.

Definition 6.1 (Landmark window). *A landmark window $\mathbb{L}$ takes as input a stream S, and given a time instant t produces a time-based fixed window $W^{(t^0,t]}$. $\mathbb{L}$ is defined through a parameter t^0, that represents the time instant on which $\mathbb{L}$ starts to operate. We denote with $\mathbb{L}(S)$ the application of the landmark window $\mathbb{L}$ on the stream S, and with $\mathbb{L}(S,t)$ the time-based window produced at time t.*

The idea behind landmark window is to select a large portion of the stream, over which event patterns are usually evaluated. An interesting fact of this operator is that the size of the generated windows always increase. That means, while the sliding windows has an intrinsic mechanism to consume the data (data exists the sliding window,), landmark windows should usually be associated to external consuming mechanism. In this sense, their application in Event Processing is natural, due to the fact that it is possible to implement them using mechanisms as load shedding and removing data after that event pattern matches.

6.2 Syntax

To capture and process events, we introduce the notion of *event patterns*, recursively defined as follows.

1. If P is a Basic Graph Pattern, $w \in I$, then the expressions (EVENT w P) is an event pattern, named *Basic Event Pattern* (BEP);[1]

2. If E_1 and E_2 are event patterns, then the expressions (FIRST E_1) and (LAST E_1) are event patterns;

3. If E_1 and E_2 are event patterns, then the expression (E_1 SEQ E_2) is an event patterns.

Finally, we define the MATCH operator to integrate event patterns into SPARQL: given an event pattern E, the expression (MATCH E) is a graph pattern. Being MATCH E a graph pattern, it can be used in combination with SPARQL syntactical rules [63] to build queries.

[1] We do not tackle here the case where $w \in I \cup V$, which is one of our future works.

6.3 Semantics

We now define the semantics of the event patterns introduced in Section 6.2. Event patterns are defined in the context of event graph patterns (i.e., by using MATCH). The main difference w.r.t. the evaluation semantics given in Chapter 5 is that this decomposition process should take into account the temporal aspects related to event matching, i.e., the evaluation should (i) produce time-annotated solution mappings, and (ii) control the time range in which a sub-pattern is processed. We address (i) by introducing the notion of *event mapping*, defined as follows.

Definition 6.2 (Event mapping). *An* event mapping *is a triple* (μ, t_1, t_2)*, composed by a solution mapping* μ *and two time instants* t_1 *and* t_2*, representing the initial and the final time instants that justify the matching, respectively.*

We assume that we are given a partial order $\prec$ to compare the pair of timestamps in event mappings. Depending on particular applications, specific ordering can be chosen. Regarding (ii), we propose a new evaluation function. Before proceeding with its presentation, we introduce the notion of window composition. We explained in Definition 5.2 that a fixed window defines a sequence of timestamped RDF graphs, i.e. a substream. Being it a substream, it is possible to apply a new fixed window on it. In other words, fixed windows can nest. Intuitively, the result is the portion of the stream in the intersection of the two windows. We formalise it in the next definition.

Definition 6.3 (Fixed Window Composition). *Let* $W^{(o_1,c_1]}$ *and* $W^{(o_2,c_2]}$ *be two fixed windows. The composition of* $W^{(o_1,c_1]}$ *and* $W^{(o_2,c_2]}$*, denoted* $W^{(o_1,c_1]} \bullet W^{(o_2,c_2]}$*, produces a fixed window* $W^{(o_3,c_3]}$ *in the interval:*

$$(o_3 = \max\{o_1, o_2\}, c_3 = \min\{c_1, c_2\}]$$

Window composition is key for the definition of the event pattern evaluations, as it restricts the area on which the event pattern can match. If $W^{(o_2,c_2]}(S)$ indicates the application of the fixed window $W^{(o_2,c_2]}$ to a stream S, $W^{(o_1,c_1]} \bullet W^{(o_2,c_2]}(S)$ denotes the application of $W^{(o_1,c_1]}$ to the substream identified by $W^{(o_2,c_2]}(S)$, alternatively written as $W^{(o_1,c_1]}(W^{(o_2,c_2]}(S))$. If two windows do not overlap, it follows that their composition is empty and the application to a stream produces an empty sequence. Finally, it is worth to note that composition commutes, i.e.

$$W^{(o_1,c_1]} \bullet W^{(o_2,c_2]} = W^{(o_2,c_2]} \bullet W^{(o_1,c_1]}$$

The following definition defines the identity window, as the identity value of the window composition.

Definition 6.4. *An* identity fixed window *(or identity window), denoted W^{id}, is the identity element of the window composition operation, i.e.*

$$W^{id} \bullet W^{(o_1,c_1]} = W^{(o_1,c_1]}$$

The application of the identity fixed window to a stream identifies the stream itself:

$$W^{id}(S) = S$$

We now have the elements to proceed in the definition of event pattern evaluation semantics. The following definition introduces a new evaluation function. It is worth noting that while RSP-QL function $[\![\cdot]\!]^t_G$ (Definition 5.6) is still used to evaluate graph patterns, this new one is intended to be used in the context of event pattern evaluation.

Definition 6.5 (Event Pattern Evaluation Semantics)**.** *Given an event pattern E, a window function W (*active window*), and an evaluation time instant $t \in ET$, we define*

$$(\!|E|\!)^t_W$$

as the evaluation *of E in the scope defined by W at t .*

The function associates the evaluation with an active fixed window that sets the boundaries of the valid ranges for evaluating event patterns. Important in the evaluation of event patterns is the role of the contextual window. The contextual window denotes a fixed window W to be composed with the ones produced by the window operators defined in the dataset *SDS*. We use the contextual window to restrict intervals of time in matching sub-event-patterns, to guarantee correctly computing complex event patterns, as we explain below. By default, the contextual window is the identity window, that does not impose any time restriction over the window on which it is applied.

The next definition presents the evaluation of Basic Event Patterns. Similar to BGPs, (BEP) are the simplest building block. The idea behind their semantics is to produce a set of SPARQL BGP evaluations over the stream items from a window of *SDS*, restricted by the active window.

Definition 6.6 (Basic Event Pattern evaluation). *Let P be a Basic Graph Pattern and w_i a window identifier. The evaluation of* FIRST EVENT w_i P *is defined as:*

$$\begin{aligned}(\!|\text{EVENT } w_i\ P|\!)^t_W = \{(\mu, t_k, t_k) \mid \mu \in [\![P]\!]_{G_k} \wedge \\ (G_k, t_k) \in W \bullet W_i(S_j, t) \wedge \\ (w_i, W_i(S_j)) \in SDS\} \qquad (6.1)\end{aligned}$$

It is worth comparing the evaluation semantics of a BEP with the one of a BGP as defined in Section **??**. They both exploit the SPARQL BGP evaluation, but while the former defines an evaluation for each stream item (i.e., an RDF graph), the latter is a unique evaluation over the merge of the stream items in one RDF graph.

Example 15. *Let's start to build the query of Example 14. The RSP-QL dataset SDS contains the time-varying graph $\overline{G}_{followers}$ to retrieve the follower relations between users and two time-based sliding windows $\mathbb{W}_3$ and $\mathbb{W}_4$ over the streams S_{nearby} and S_{social}, defined through the parameters $(\alpha = 30, \beta = 1, t^0 = 0)$. To summarize:*

$$\begin{aligned}SDS =&\{(g_f, \overline{G}_{followers}), \\ &(w_3, \mathbb{W}_3(S_{nearby})), (w_4, \mathbb{W}_4(S_{social}))\}\end{aligned}$$

Let E_1 be the event that identifies when Carl is detected close to a shop. That means,

$$E_1 = \text{EVENT } w_3\ \{carl\ \ :isNearby\ ?shop\}$$

We show how to evaluate $(\!|E_1|\!)^{20}_{W^{id}}$. First of all, the content of the window $\mathbb{W}_3$ at time 22 ($\mathbb{W}_3(S_{nearby}, 22)$) is the whole stream depicted in Listing 4.1. The content of the composed window $W^{id} \bullet \mathbb{W}_3(S_{nearby}, 22)$ is the same.

Now we evaluate $[\![:carl\ \ :isNearby\ ?shop]\!]_{d_{n,k}}$ for $1 \leq k \leq 22$. The graphs d_{n5}, d_{n19} and d_{n21} have matches, which are $\mu_a = \{?shop \mapsto :a\}$ and $\mu_b = \{?shop \mapsto :b\}$. Combining with the timestamps 5, 19 and 21 when d_{n5}, d_{n19} and d_{n21} respectively appear in S_{nearby}, we have:

$$(\!|E_1|\!)^{22}_{W^{id}} = \{(\mu_a, 5, 5), (\mu_b, 19, 19), (\mu_b, 21, 21)\}.$$

Next is the semantics of other event patterns, starting with those that identify the *first* and *last* event matching a pattern, based on the ordering $\prec$.

Definition 6.7 (FIRST and LAST event patterns)**.** *Let E be an event pattern. The evaluation semantics of* FIRST E *and* FIRST E *is defined as:*

$$\begin{aligned}(\!|\text{FIRST } E|\!)^t_W = \{(\mu, t_1, t_2) \in (\!|E|\!)^t_W \mid \\ \nexists(\mu', t_3, t_4) {\in} (\!|E|\!)^t_W \colon (t_3, t_4) {\prec} (t_1, t_2)\} \quad (6.2)\\ (\!|\text{LAST } E|\!)^t_W = \{(\mu, t_1, t_2) \in (\!|E|\!)^t_W \mid \\ \nexists(\mu', t_3, t_4) {\in} (\!|E|\!)^t_W \colon (t_1, t_2) {\prec} (t_3, t_4)\} \quad (6.3)\end{aligned}$$

The next definition presents the definition of the SEQ operator.

Definition 6.8 (SEQ event pattern)**.** *Let E_1 and E_2 be two event patterns. The evaluation of E_1 SEQ E_2 is defined as:*

$$\begin{aligned}(\!|E_1 \text{ SEQ } E_2|\!)^t_W = \{(\mu_1 {\cup} \mu_2, t_1, t_4) \mid \\ (\mu_2, t_3, t_4) {\in} (\!|E_2|\!)^t_W {\wedge} \\ (\mu_1, t_1, t_2) {\in} (\!|\mu_2(E_1)|\!)^t_{W^{\sqcup}[0,t_3-1] \bullet W}\} \quad (6.4)\end{aligned}$$

Intuitively, for each event mapping (μ_2, t_3, t_4) that matches E_2, Equation (6.4) seeks for (a) *compatible* and (b) *preceding* event mappings matching E_1. The two demands are guaranteed by introducing constraints on the evaluation of E_1:

- (a) is imposed by, in E_1, substituting the shared variables with E_2 for their values from μ_2, denoted by $\mu_2(E_1)$.
- (b) is ensured by restricting the time range on which input graphs are used to match $\mu_2(E_1)$: we only consider graphs appearing before t_3, thus $W^{\sqcup}[0, t_3 - 1] \bullet W$.

Example 16. *The event E_2 that looks for the posts related to a shop is defined as follows:*

$$\begin{aligned}E_2 = EVENT\ w_4\ \{&?post\ \ :author\ ?following\ .\\ &?post\ \ :mentions\ ?c\ .\\ &?c\ \ :in\ ?shop\}\end{aligned}$$

We show how $(\!|E_2$ SEQ $E_1|\!)^{22}_{W^{id}}$ is evaluated. In Example 15 we computed the result of $(\!|E_1|\!)^{22}_{W^{id}}$. Applying the function in Definition 6.8, it follows that the result of the evaluation is:

$$\begin{aligned}&(\!| E_2 \text{ SEQ } E_1 |\!)^t_W = \\ &\quad \{(\mu \cup \mu_a, t_1, 5) \mid (\mu, t_1, t_2) \in (\!| \mu_a(E_2) |\!)^t_{W^{\sqcup}[0,4] \bullet \mathbb{W}_4}\} && (6.5)\\ &\cup \{(\mu \cup \mu_b, t_3, 19) \mid (\mu, t_3, t_4) \in (\!| \mu_b(E_2) |\!)^t_{W^{\sqcup}[0,18] \bullet \mathbb{W}_4}\} && (6.6)\\ &\cup \{(\mu \cup \mu_b, t_5, 21) \mid (\mu, t_5, t_6) \in (\!| \mu_b(E_2) |\!)^t_{W^{\sqcup}[0,20] \bullet \mathbb{W}_4}\} && (6.7)\end{aligned}$$

The set in (6.5) *is empty, due to the fact that* $W^{\sqcup}[0,4] \bullet \mathbb{W}_4(t)$ *is empty. The sets in* (6.6) *and* (6.7) *are not empty, as the evaluation of* $(\!| \mu_b(E_2) |\!)^t_{W^{\sqcup}[0,18] \bullet \mathbb{W}_4}$ *and* $(\!| \mu_b(E_2) |\!)^t_{W^{\sqcup}[0,20] \bullet \mathbb{W}_4}$ *produce three event mappings* $(\mu_c = \{?post \mapsto :post_2, ?author \mapsto :bob, ?c \mapsto :c_2\}, 15, 15)$, $(\mu_d = \{?post \mapsto :post_3, ?author \mapsto :diana, ?c \mapsto :c_2\}, 16, 16)$ *and* $(\mu_e = \{?post \mapsto :post_4, ?author \mapsto :eve, ?c \mapsto :c_2\}, 18, 18)$.

It follows that the result of the evaluation is

$$\begin{aligned}(\!| E_2 \text{ SEQ } E_1 |\!)^t_W = \{&(\mu_c \cup \mu_b, 15, 19), (\mu_c \cup \mu_b, 15, 21),\\ &(\mu_d \cup \mu_b, 16, 19), (\mu_d \cup \mu_b, 16, 21)\\ &(\mu_e \cup \mu_b, 18, 19), (\mu_e \cup \mu_b, 18, 21)\}\end{aligned}$$

Finally, we define the semantics of the MATCH operator. It removes the time annotations from event mappings and produces a bag of solution mappings. Thus, the result of this operator can be combined with results of other SPARQL graph pattern evaluation (i.e., other bags of solution mappings).

Definition 6.9 (MATCH evaluation)**.** *The evaluation of* MATCH E *is defined as:*

$$[\![\text{MATCH } E]\!]^t_{SDS(A)} = \{\mu \mid (\mu, t_1, t_2) \in (\!| E |\!)^t_{W^{id}}\} \tag{6.8}$$

The MATCH clause encloses the search of event patterns. Its precedence w.r.t. other operators is the lowest: this operators is the last one that has to be evaluated, to ensure the correct identification of the event patterns.

Example 17. *The evaluation of* MATCH E_2 SEQ E_1 *is the follow*[2]*:*

$$\begin{aligned}&[\![\text{MATCH } E_2 \text{ SEQ } E_1]\!]^{22}_{G_0} = \\ &\{\mu \mid (\mu, t_1, t_2) \in (\!| E_2 \text{ SEQ } E_1 |\!)^{22}_{W^{id}}\} = \\ &\qquad \{(\mu_c \cup \mu_b), (\mu_d \cup \mu_b), (\mu_e \cup \mu_b)\}\end{aligned}$$

[2]As in Chapter 5, we assume set semantics.

*The resulting set of solution mappings can be combined with other graph pattern results. Following Example **??**, the query contains the graph pattern* $P_1 = \text{GRAPH } g_f$ *{:carl :follows* $?follower$*}. It is possible to join it with the* MATCH*graph pattern above on the* $?following$ *variable.*

The evaluation of the graph pattern is the set $\{(?follower \mapsto : diana), (?follower \mapsto : eve)\}$*. It follows that the evaluation of the graph pattern* P_1 JOIN MATCH E_2 SEQ E_1 *produces the set of mappings* $\{(\mu_d \cup \mu_b), (\mu_e \cup \mu_b)\}$.

6.4 Event Selection Policies

Evaluating the SEQ operator as in Equation (6.4) takes into account all possible matches from the two sub-patterns. This kind of evaluation captures only the "unrestricted" behaviour of EP-SPARQL and C-SPARQL. With the purpose of formally capturing the CEP semantics of C-SPARQL and EP-SPARQL, we introduce in this section different versions of the sequencing operator that allows different ways of selecting stream items to perform matching, known as *selection policies*.

Firstly, for C-SPARQL's *naive* CEP behaviour, Equation (6.9) simply picks the two latest event mappings that match the two sub-patterns and compare their associated timestamps.

$$\begin{aligned}(\!|E_1 \text{ SEQ}^n\ E_2|\!)^t_W = \ & \{(\mu_1 \cup \mu_2, t_1, t_4) \mid (t_1, t_2) \prec (t_3, t_4) \wedge \\ & (\mu_1, t_1, t_2) \in (\!|\text{LAST } E_1|\!)^t_W \wedge (\mu_2, t_3, t_4) \in (\!|\text{LAST } E_2|\!)^t_W\}\end{aligned} \tag{6.9}$$

For the *chronological* and *recent* settings from EP-SPARQL, we need more involved operators SEQ^c and SEQ^r. In the sequel, let $W^\star = W^\sqcup[0, t_3 - 1] \bullet W$.

$$\begin{aligned}(\!|E_1 \text{ SEQ}^c\ E_2|\!)^t_W = \ & \{(\mu_1 \cup \mu_2, t_1, t_4) \mid (\mu_2, t_3, t_4) \in (\!|E_2|\!)^t_W \wedge \\ & (\!|\mu_2(E_1)|\!)^t_{W^\star} \neq \emptyset \wedge (\mu_1, t_1, t_2) \in (\!|\text{FIRST } \mu_2(E_1)|\!)^t_{W^\star} \wedge \\ & (\nexists (\mu_2', t_3', t_4') \in (\!|E_2|\!)^t_W \colon (\!|\mu_2'(E_1)|\!)^t_{W^\star} \neq \emptyset \wedge (t_3', t_4') \prec (t_3, t_4))\}.\end{aligned} \tag{6.10}$$

Compared to (6.4), Equation (6.10) selects an event mapping (μ_2, t_3, t_4) of E_2 that:

- has a compatible event mappings in E_1 which appeared before μ_2. This is guaranteed by the condition $(\!|\mu_1(E_2)|\!)^t_{W^\star} \neq \emptyset$ and the window function $W^\star = W^\sqcup[0, t_3 - 1] \bullet W$;

- is the first of such event mappings. This is ensured by stating that no such (μ'_2, t'_3, t'_4) exists, where $(t'_3, t'_4) \prec (t_3, t_4)$.

Once (μ_2, t_3, t_4) is found, (μ_1, t_1, t_2) is taken from $(\!|\text{FIRST } \mu_2(E_1)|\!)^t_{W^\star}$, which makes sure that it is the first compatible event that appeared before (μ_2, t_3, t_4). Finally, the output event matching E_1 SEQ^c E_2 is $(\mu_1 \cup \mu_2, t_1, t_4)$.

Equation (6.11) follows the same principle as Equation (6.10), except that it selects the last instead of the first event mappings.

$$\begin{aligned}(\!|E_1 \text{ SEQ}^r\ E_2|\!)^t_W = \ &\{(\mu_1 \cup \mu_2, t_1, t_4) \mid (\mu_2, t_3, t_4) \in (\!|E_2|\!)^t_W \wedge \\ &(\!|\mu_2(E_1)|\!)^t_{W^\star} \neq \emptyset \wedge (\mu_1, t_1, t_2) \in (\!|\text{LAST } \mu_2(E_1)|\!)^t_{W^\star} \wedge \\ &(\nexists (\mu'_2, t'_3, t'_4) {\in} (\!|E_2|\!)^t_W \colon (\!|\mu'_2(E_1)|\!)^t_{W^\star} {\neq} \emptyset \wedge (t_3, t_4) {\prec} (t'_3, t'_4))\}.\end{aligned} \tag{6.11}$$

6.5 Event Consumption Policies

Selection policies are not sufficient to capture the behaviour of EP-SPARQL in the chronological and recent settings. As described in Section **??**, under these settings, stream items that contribute to an answer are not considered in the following evaluation iterations. We complete the model by formalizing this feature, known as *consumption policies*.

Let $ET{=}t_1, t_2, \ldots, t_n, \ldots$ be the set of evaluation instants. Abusing notation, we say that a window function w_j *appears* in an event pattern E, denoted by $w_j \hat{\in} E$, if E contains a basic event pattern of the form (EVENT w_j P).

Consumption policies determine input for the evaluation. Our next two definitions cover this aspect. Definition 6.10 is about a possible input for the evaluation while Definition 6.11 talks about the new incoming input. We first define such notions for a window in an RDF streaming dataset, and then lift them to the level of structures that refer to all windows appearing in an event pattern.

Definition 6.10 (Potential Input & Input Structure). *Given an RDF streaming dataset SDS, we denote by* $I_i(w_j) \subseteq SDS_{w_j}(t_i)$ *a* potential input *at time* t_i *of the window identified by* w_j*. For initialization purposes, we let* $I_0(w_j) = \emptyset$*.*

Given an event pattern E*, an* input structure I_i *of* E *at time* t_i *is a set of potential inputs at* t_i *of all windows appearing in* E*, i.e.,* $I_i = \{I_i(w_j) \mid w_j \hat{\in} E\}$*.*

Definition 6.11 (Delta Input Structure). *Given an RDF streaming dataset SDS and two consecutive evaluation times t_{i-1} and t_i, where $i > 1$, the new triples arriving at a window w_j are called a* delta input*, denoted by $\Delta_i(w_j) = SDS_{w_j}(t_i) \setminus SDS_{w_j}(t_{i-1})$. For initialization purposes, let $\Delta_1(w_j) = SDS_{w_j}(t_1)$.*

Given an event pattern E, a delta input structure *at time t_i is a set of delta inputs at t_i of all windows appearing in E, i.e., $\Delta_i = \{\Delta_i(w_j) \mid w_j \hat{\in} E\}$.*

We can now define consumption policies in a generic sense.

Definition 6.12 (Consumption Policy & Valid Input Structure). *A* consumption policy function $\mathcal{P}$ *takes an event pattern E, a time instance $t_i \in ET$, and a vector of additional parameters $\vec{p}$ depending on the specific policy, and produces an input structure for E.*

The resulted input structure is called valid *if it is returned by applying $\mathcal{P}$ on a set valid parameters $\vec{p}$, where the validity of $\vec{p}$ is defined based on each specific policy.*

This generic notion can be instantiated to realize specific policies in practice. For example, the policy $\mathcal{P}^u$ that captures the EP-SPARQL’s unrestricted setting requires no further parameters, thus $\vec{p} = \emptyset$ and returns full input at evaluation time. To be more concrete:

$$\mathcal{P}^u(E, t_i) = \{I_i(w_j) = SDS_{w_j}(t_i) \mid w_j \hat{\in} E\}$$

For the chronological and recent settings, we describe here only informally the two respective functions $\mathcal{P}^c$ and $\mathcal{P}^r$. Their additional parameters include I_{i-1} (the input structure at t_{i-1}) and Δ_i (the delta input structure at t_i), and they return an input structure I_i such that its elements $I_i(w_j)$ contain $\Delta_i(w_j)$ and the triples in $I_{i-1}(w_j)$ that are not used to match E at t_{i-1}. The validity of input can be guaranteed by starting the evaluation with $I_1(w_j) = SDS_{w_j}(t_1)$ which is valid by definition.

We now proceed to incorporate consumption policies into event patterns evaluation. The idea is to execute the evaluation function $(\!|.|\!)$ also with a policy function $\mathcal{P}$, that is, to evaluate an event pattern E with $(\!|E|\!)^t_{W,\mathcal{P}}$. Then, when the evaluation process reaches a basic event pattern at the leaf of the operator tree, $\mathcal{P}$ is applied to filter out already consumed input. Formally:

$$(\!|\text{EVENT } w_j\ P|\!)^{t_i}_{W,\mathcal{P}} = [\![P]\!]_{\mathcal{I}},$$

where $\mathcal{I}$ denotes $I_i(w_j) \cap (\bigcup_{(G_k,t_k) \in W \bullet W_j(S_\ell, t_i)} G_k)$ and $I_i(w_j) \in I_i = \mathcal{P}(E, t_i, I_{i-1}, \Delta_i)$.

6.6 Remarks

In this chapter, we presented RSEP-QL, that extends RSP-QL to query RDF streams with CEP features. We give a formal semantics for RSEP-QL and compare its CEP features with those that currently be possible with EP-SPARQL and C-SPARQL. The formal semantics of CEP operators provides a vital foundation for enabling CEP features in RSP engines.

Our approach can be extended to represent other selection and consumption policies in CEP such as *strict contiguity*, *partition contiguity*, *skip till next match*, and *skip till any match* [1].

Key for the implementation and the realization of an evaluation module is the study and the optimization of RSEP-QL queries: study of algebraic equivalences and cost models are needed to improve performance and reduce responsiveness.

CHAPTER 7

Reasoning in RSEP-QL

In the last five years, several techniques were proposed under the Stream Reasoning label, as described in Section 3.3, but they are heterogeneous in terms of input, output and use cases. In this chapter, we move a step forward by introducing an extension of RSEP-QL to describe Stream Reasoning techniques in terms of continuous query answering process in the context of RDF Stream Processing.

The research question we investigate in this chapter is:

RQ.2 What is the correct behaviour of a continuous query engine while processing a semantic stream under entailment regimes?

To investigate the question, we study if and how the SPARQL entailment regimes can be applied. The remainder of the chapter is structured as follows: after a presentation of the assumptions in Section 7.1, Section 7.2 introduces the entailment regime extension for RSEP-QL at a glance, while Sections 7.3–7.5 go in depth with the descriptions of the entailment levels. We close with a comparison of the levels in Section 7.6, and with final remarks in Section 7.7.

7.1 Assumptions

Let us first detail the assumptions we make on the RDF stream content, the conceptual model and its evolution over time. Most of those assumptions are the same of existing work in stream reasoning (Section 3.3). The first is that we focus on entailment regimes with a DL-related semantics. The choice let us adopt the notions of ontology, TBox and ABox.

Regarding the assumptions on the RDF streams, we assume that they carry only assertional axioms, and the content of each stream is compliant to a TBox $\mathcal{T}$ containing the terminological axioms. This assumptions captures a large portion of reality, where the conceptual model is static (or quasi-static), and the facts rapidly changes over time. It happens in several scenarios, such as the ones presented in various work of the stream reasoning area.

Another assumption on RDF stream comes from the following consideration on the semantics of the time instants: up to know, we defined time instants as time annotations (*application time*) associated to the stream items. This definition can be interpreted in different ways, depending on the nature of the data stream item. When the stream brings events like "Alice opens the door d" with time instant t, the information is related to the fact that the *action* happens at time t. On the other side, a stream item "the door d is open" at time t refers to a *state* of the door, and t can refer to the time instant on which this state starts to be true, ends to be true, or simply is true at that time. We leave the investigation of this problem as future work; in this chapter we assume that the time instant indicates that the stream item starts to be true at time t. This assumption contributes in guaranteeing the correctness of the inference process: if we consider a query evaluation at time t over the content of one fixed window, the inference process should take into account axioms that are still true at that time. That means, the axioms in the fixed window starts to be true at a time t_i, and are still true at t.

We also defines a set of assumptions related to the TBox. The terminological axioms are stored in a time-varying graph $\overline{G}_{\mathcal{T}}$, that is optionally part of the dataset. Being a time-varying graph, $\overline{G}_{\mathcal{T}}$ allows to track the evolution of the TBox $\mathcal{T}$ over time. The second assumption is that, entailment regime E associated to the ontological language $\mathcal{L}$, for each time instant t where $\overline{G}_{\mathcal{T}}$ is defined, the RDF graph is $\overline{G}_{\mathcal{T}}(t)$ is well formed for E. The assumption is made in the sake of clearness, and we refer to SPARQL entailment regime results [59] for error management when not differently stated.

The last assumption is on the fact that the TBox does not change, i.e. for each time instant t where $\overline{G}_{\mathcal{T}}$ is defined, $\overline{G}_{\mathcal{T}}(t) = G_{\mathcal{T}}$. From the above assumptions, given an entailment regime E, $G_{\mathcal{T}}$ can be represented in the TBox $\mathcal{T}$.

7.2 The Stream Reasoning stack

SPARQL entailment regimes [58] extends the SPARQL model in order to take into account ontological languages and to make available to the query processor implicit information in the data. The main idea behind the extension is to redefine the notion of evaluation of basic graph pattern through the concept of semantic entailment relation. That means, given an entailment regime E, the basic graph pattern should be evaluated over a scoping graph E-equivalent to the active graph, and not over the active graph itself (as explained in Section 2.2.3).

When we move from SPARQL to RSEP-QL, we should take into account two main differences. The first is that the active element in the dataset *SDS* can be a time-varying graph, a (sliding or landing) window operator, or a combination of both (in the default element A_0). The second is that, in addition to basic graph patterns, there are also basic event patterns that access the content of the active element of *SDS*.

While basic event patterns are evaluated over a window operator[1] content, basic graph patterns can be evaluated over either windows or time-varying graphs. In the latter case, when the content of the active element A is a time-varying graph $\overline{G}$, Definition 5.7 explains that the basic graph pattern at time t is matched over a RDF graph $SDS(A, t) = \overline{G}(t)$. In this case, the extension of RSEP-QL with the SPARQL entailment regimes is straightforward: fixed an entailment regime E, conditions and theory of [58] can be applied in the context of the RDF graph $\overline{G}(t)$.

Different is the case when the active element is a window operator applied to a stream S. The output of a window operator can be queried in the context of basic graph pattern and basic event pattern evaluations. The process is shown in Figure 7.1, considering the blue blocks: in the dataset *SDS* the stream S is associated to

[1]When not specified, we refer to *window operator* to indicate a generic sliding or landing windows.

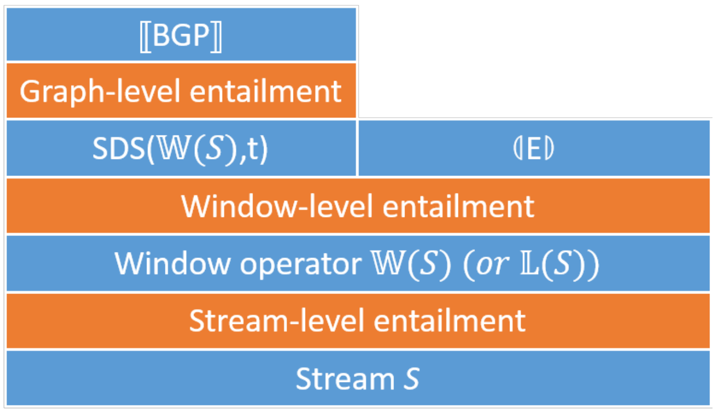

Figure 7.1: *The stream reasoning stack: the blue box represents RSEP-QL elements, while the orange ones represent the different levels on which reasoning can be applied.*

a set of window operators. Let $\mathbb{W}(S)$ be one of those window operators: as explained in Definitions 5.3 and 6.1, fixed a time instant t, $\mathbb{W}(S,t)$ produces a fixed window $W^{(o,c]}$ containing a subset of the stream with timestamped RDF graphs (d_i, t_i). In the context of a basic graph pattern evaluation having $\mathbb{W}(S)$ as active element, Definition 5.7 explains that the window content is merged in a RDF graph (i.e., $SDS(\mathbb{W}(S),t) = \{(s,p,o)|(s,p,o) \in d_i \wedge (d_i,t_i) \in \mathbb{W}(S,t)\}$). In the context of an basic event pattern evaluation, Definition 6.6 shows how each stream item (d_i, t_i) in the fixed window is considered separately, and the event pattern matches on one of those elements.

Given this process, the question that may arise is: when is the right moment to consider the entailment regime? We envision three different points (*levels*) where it can happen, denoted by the orange boxes in Figure 7.1: *graph-level entailment*, where the entailment regime is applied in the context the RDF graph $SDS(\mathbb{W}(S),t)$; *window-level entailment*, applied in the context of to the window operator $\mathbb{W}(S)$; and *stream-level entailment*, applied to the portion of the stream data S observed up to the current evaluation time instant. The application of the entailment regime to the three levels is not equivalent: each level considers different input and output data models, and brings the continuous query to compute different results.

7.3 Graph-level entailment

When the entailment regime is applied at *graph level*, the event pattern extension of RSEP-QL does not gain any advantage. In other words, graph-level entailment just affects the result of basic graph pattern evaluations.

Definition 5.7 explains that the evaluation at time t of a basic graph pattern P having $SDS(A)$ as active elements is equivalent to the SPARQL evaluation $[\![P]\!]_{SDS(A,t)}$. When the active element is a sliding (or a landing) window $\mathbb{W}(S)$, the active element $SDS(\mathbb{W}(S), t)$ identifies the RDF graph with the whole content of the fixed window $\mathbb{W}(S, t)$.

In this sense, the graph-level entailment is a direct application of the SPARQL entailment regime extension: considering $SDS(\mathbb{W}(S), t)$ as the active RDF graph, most of the results presented in Section 2.2.3 hold, and when $SDS(\mathbb{W}(S), t)$ is well formed for E, it is possible to convert it in an ontology, as defined in the OWL 2 Web Ontology Language Mapping to RDF Graphs document [87]. Anyway, this ontology would be composed by ABox axioms only. This is the main difference w.r.t. SPARQL: while SPARQL entailment regimes assume that both terminological and assertional axioms are stored in the active graph, when the active elements is a window, the resulting graph contains only triples describing the ABox axioms. To solve the problem, we introduce the notion of Stream TBox Mapper.

Definition 7.1 (Stream TBox Mapper)**.** *A* Stream TBox Mapper M *is a partial function that maps RDF streams to time-varying graphs. Let S be a stream, $M(S)$ identifies a time-varying graph $\overline{G}_{S,\mathcal{T}}$ containing the concepts and relations used in S information items.*

The Stream TBox Mapper identifies the conceptual models related to the RDF streams. Given a RSEP-QL query, for each stream S available in the dataset SDS, the Stream TBox Mapper returns either a time-varying graph $\overline{G}_{S,\mathcal{T}}$ is S is part of the domain, or the empty RDF graph otherwise.

The Stream TBox Mapper supplies the terminological axioms, introducing the missing elements to enable the inference process. For each basic graph pattern evaluation over a window content under graph-level entailment regime E, the active graph considered in the evaluation is

$$SDS(\mathbb{W}(S), t)) \cup M(S).$$

Let $\mathcal{O}$ be the ontology associated to $SDS(\mathbb{W}(S), t)) \cup M(S)$, we indicate with $Cl(\mathcal{O})$ its closure w.r.t. the ontological language $\mathcal{L}$ associated to E.

We envision different possible ways on which the Stream TBox Mapper can be provided to the Stream Reasoner. A solution is to define it in the query, declaring up to one Stream TBox Mapper for each stream considered. Another possibility, is to let the stream provider publish the data schema. We discuss this idea in the future work (Section 10.2.2).

Example 18. *Let S_{social} be the stream defined in Examples 7 and 14, and let $M(S_{social}) = \overline{G}_{\mathcal{T}_{social}}$. For the assumption that the TBox does not change, $\overline{G}_{\mathcal{T}_{social}}$ is $G_{\mathcal{T}_{social}}$ for any time value t where is defined, and $\mathcal{T}_{social}$ in Listing 7.1 is the relative $\mathcal{EL}^{++}$ TBox.*

$Post \sqsubseteq \exists hasAuthor.Person$
$CouponPubPost \sqsubseteq Post \sqcap \exists contains.Coupon$
$Coupon \sqsubseteq \exists in.Shop \sqcap \exists on.Produt \sqcap \exists reduce$
$MentionedCoupon \sqsubseteq Coupon \sqcap \exists mentionedIn.Post$
$contains \circ mentionedIn \sqsubseteq propagatedBy$

Listing 7.1: *The TBox representing $G_{\mathcal{T}_{social}}$, the Stream TBox Mapper value of S_{social}*

Let's now consider a query that looks for the posts that are propagated by other posts (as in Line 5). The query contains a time-based sliding window $\mathbb{W}$ over S_{social} defined through parameters $\omega = 20, \beta = 10, t^0 = 0$. At evaluation time 20, the ontology considered in the entailment regime is composed by $\mathcal{T}_{social}$ and the ABox $\mathcal{A}_{social}$ in Listing 7.2, obtained by $SDS(\mathbb{W}(S_{social}), 20)$, i.e. the merge of the data in $W^{(0,20]}(S_{social})$.

$(\exists hasAuthor.\{Alice\} \sqcap \exists countains.\{c_1\})(:post_1)$,
$(\exists in.\{a\} \sqcap \exists on.\{armadillo\} \sqcap \exists reduce.\{30\})(c_1)$,
$(\exists hasAuthor.\{Bob\} \sqcap \exists countains.\{c_2\})(:post_2)$,
$(\exists hasAuthor.\{Diana\})(:post_3)$,
$(\exists hasAuthor.\{Eve\})(:post_4)$,
$(\exists in.\{b\} \sqcap \exists on.\{panda\} \sqcap \exists reduce.\{25\})(c_2)$
$(\exists mentionedIn.\{:post_3\} \sqcap \exists mentionedIn.\{:post_4\})(c_2)$

Listing 7.2: *ABox $\mathcal{A}_{social}$ obtained by $SDS(\mathbb{W}(S_{social}), 20)$*

The answer of the query is $:post_2$, as $propagatedBy(:post_2, :post_3)$ and $propagatedBy(:post_2, :post_4)$ can be inferred through the axioms in Line 5 of $\mathcal{T}_{social}$ and Lines 3 and 7 of $\mathcal{A}_{social}$.

On the contrary, under simple entailment, the query does not return *any result, as no posts are explicitly propagated in the stream, and consequently there are not matching on the graph* $SDS(\mathbb{W}(S_{social}), 20)$.

7.4 Window-level entailment

The application of entailment regime at graph level is a direct extension of the SPARQL entailment regimes, but it has some drawbacks: it affects the answer only in the context of basic graph pattern evaluation, while it does not have any effect on the basic event pattern evaluation. Moreover, the process uses only a subset of the available information bought in the stream – it considers the data item contents but not the relative temporal annotations. *Window-level entailment regime* overcomes these limits, by applying the entailment regime E in the context of the fixed windows produced by the window operators.

Differently from the graph-level entailment, which works on RDF graphs (and ontologies), the window-level entailment receives as input a fixed window, i.e. an RDF substream. We need a formalism that (i) enables inference operations and (ii) captures the dynamic and evolution of the knowledge. We adopt the notion of *ontology streams*: initial definition is given by Huang and Stuckenschmidt in [65], and has been used in other work of the ares as [75, 92]. Starting from that definition, we define *ontology substreams* as follows.

Definition 7.2 (DL $\mathcal{L}$ ontology substream)**.** *A DL* $\mathcal{L}$ ontology substream *is a pair* $(\mathcal{T}, \mathcal{S}_o^c)$*, where* $\mathcal{T}$ *is static TBox expressed in* $\mathcal{L}$*, and* $\mathcal{S}_o^c$ *(*$o, c \in \mathbb{N}$ *and* $o < c$*) is an* ABox substream*, i.e., a sequence of ABox snapshots. An ABox snapshot is denoted with* $\mathcal{S}_o^c(i, j)$ *and represents the* j*-th snapshot*[2] *at time instant* i*.*

Let $n^i \geq 0$ *be the number of snapshots with time instant* i*,* $\mathcal{S}_o^c$ *is defined as:*

$$\begin{aligned}\mathcal{S}_o^c = (&\mathcal{S}_o^c(o+1,1), \mathcal{S}_o^c(o+1,2), \ldots, \mathcal{S}_o^c(o+1, n^{o+1}),\\ &\mathcal{S}_o^c(o+2,1), \mathcal{S}_o^c(o+2,2), \ldots, \mathcal{S}_o^c(o+2, n^{o+2}),\\ &\ldots\\ &\mathcal{S}_o^c(c,1), \mathcal{S}_o^c(c,2), \ldots, \mathcal{S}_o^c(c, n^{c})),\end{aligned}$$

As in the original definition, data (ABox) and its inferred statements (entailments) are evolving over time while its schema (TBox) remains unchanged. The main difference of this new definition is the order:

[2]For the sake of clearness, we assume that the snapshots are enumerated.

ontology stream definition in [65, 75, 92] describes a total order over the ABox snapshots, while the one in Definition 7.2 uses a partial order: given a time instant t, the ontology substream carries a set of simultaneous snapshots.

Example 19 (DL $\mathcal{EL}^{++}$ ontology substream)**.** *Listing 7.3 illustrates an $\mathcal{EL}^{++}$ ontology substream $\mathcal{OC}_0^{20} = (\mathcal{T}_{social}, \mathcal{SC}_0^{20})$ related to microposts in a social network. Listings 7.1 and 7.3 respectively shows the TBox $\mathcal{T}_{social}$ and the ABox substream $\mathcal{SC}_0^{20}$.*

$$\begin{aligned}
&\mathcal{SC}_0^{20}(8,1) : (\exists hasAuthor.\{Alice\} \sqcap \exists countains.\{c_1\})(:post_1),\\
&\qquad\qquad (\exists in.\{a\} \sqcap \exists on.\{armadillo\} \sqcap \exists reduce.\{30\})(c_1)\\
&\mathcal{SC}_0^{20}(15,1) : (\exists hasAuthor.\{Bob\} \sqcap \exists countains.\{c_2\})(:post_2),\\
&\qquad\qquad (\exists in.\{b\} \sqcap \exists on.\{panda\} \sqcap \exists reduce.\{25\})(c_2)\\
&\mathcal{SC}_0^{20}(16,1) : (\exists hasAuthor.\{Diana\})(:post_3),\\
&\qquad\qquad (\exists in.\{b\} \sqcap \exists on.\{panda\} \sqcap \exists reduce.\{25\} \sqcap \exists mentionedIn.\{:post_3\})(c_2)\\
&\mathcal{SC}_0^{20}(18,1) : (\exists hasAuthor.\{Eve\})(:post_4),\\
&\qquad\qquad (\exists in.\{b\} \sqcap \exists on.\{panda\} \sqcap \exists reduce.\{25\} \sqcap \exists mentionedIn.\{:post_4\})(c_2)
\end{aligned}$$

Listing 7.3: *ABox substream $\mathcal{SC}_0^{20}$ of $\mathcal{OC}_0^{20}$*

The ABox substream $\mathcal{SC}_0^{20}$ captures posts published on a social network, where some posts carries coupons, that can be mentioned and discussed by other posts. The snapshots without time points are empty.

7.4.1 Mapping Fixed Windows and Ontology Substreams

Comparing the ontology substream definition with the one of RDF stream in Definition 4.2, it is possible to observe that both models assume a partial order. While the RDF stream is an infinite sequence of data items, an ontology substream is a finite sequence (similarly to a fixed window). Intuitively, the $\mathcal{O}_o^c$can capture the data in a fixed window, and can enable the inference processes in the context of RSEP-QL continuous query answering. We formalise the notion in the following definition.

Definition 7.3 (Mapping Fixed Windows in $\mathcal{O}_o^c$)**.** *Let S be an RDF stream and $W^{(o,c]}(S)$ a fixed window over S. Given an entailment regime* E*, the ABox substream $\mathcal{S}_o^c$ maps $W^{(o,c]}(S)$ w.r.t* E *if the following condition holds:*

$$(d,t) \in W^{(o,c]}(S) \iff \exists! j : \mathcal{S}_o^c(t,j) = \mathcal{A}_d,$$

where $\mathcal{A}_d$ is the ontology that maps the RDF graph d w.r.t. E.

The definition uses the mappings between RDF graphs and ontologies to map the RDF stream items into ABox snapshots (and vice versa).

Example 20. *The ontology substream in Example 19 is the mapping of the RDF stream S_{social} defined in Examples 7 and 14.*

7.4.2 Reasoning in Ontology Substreams

To define reasoning over ontology substreams, we distinguish two cases: intra-snapshot and so-far inference. The intra-snapshot inference process considers as ontology the TBox and one snapshot of the $\mathcal{O}_o^c$.

Definition 7.4 (Intra-snapshot inference)**.** *Given an ontological language $\mathcal{L}$ and an ontology substream $(\mathcal{T}, \mathcal{S}_o^c)$, the* intra-snapshot inference *is an inference process that considers the TBox $\mathcal{T}$ and one ontology substream snapshot $\mathcal{S}_o^c(i,j)$ w.r.t. $\mathcal{L}$. The* intra-snapshot closure *of a snapshot $\mathcal{S}_o^c(i,j)$ is defined as follows:*

$$Cl^{intra}(\mathcal{S}_o^c(i,j)) \models \alpha \iff \mathcal{T}, \mathcal{S}_o^c(i,j) \models \alpha$$

As the name suggests, the so-far inference involves different snapshots, and combine their axioms to infer new knowledge.

Definition 7.5 (So-far inference)**.** *Given an ontological language $\mathcal{L}$ and an ontology substream $(\mathcal{T}, \mathcal{S}_o^c)$, the* so-far inference *is an inference process that considers the TBox $\mathcal{T}$, one ontology substream snapshot $\mathcal{S}_o^c(i,j)$ and all the snapshots that precedes it or are contemporary w.r.t. $\mathcal{L}$. The* so-far closure *of a snapshot $\mathcal{S}_o^c(i,j)$ is defined as the closure of $\mathcal{S}_o^c(i,j)$ w.r.t the TBox $\mathcal{T}$, and the snapshots $\mathcal{S}_o^c(p,q)$ such that $p \leq i$.*

Let α be an entailment of the so-far closure of $\mathcal{S}_o^c(i,j)$, and let $\alpha_1, \ldots, \alpha_k, \ldots, \alpha_n$ be a group of axioms that justifies α. It holds:

$$\begin{gathered} Cl^{sofar}(\mathcal{S}_o^c(i,j)) \models \alpha \\ \iff \\ \mathcal{T}, \mathcal{S}_o^c(o+1,1), \ldots, \mathcal{S}_o^c(i,1), \ldots, \mathcal{S}_o^c(i,j), \ldots, \mathcal{S}_o^c(i,n^i) \models^{\alpha_1,\ldots,\alpha_k,\ldots,\alpha_n} \alpha \wedge \\ \mathcal{T}, \mathcal{S}_o^c(i,j) \models \alpha_k, \end{gathered} \tag{7.1}$$

The so-far closure of a snapshot $\mathcal{S}_o^c(i,j)$ contains the explicit axioms, the ones of the intra-snapshot closure, and the ones that can be inferred taking into account also the axioms in the ontology substream up to know. The intuition behind the so-far inference is that a formula α can be inferred only when all the inferring axioms are true, and it happens

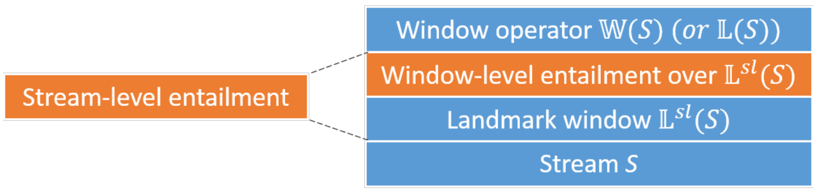
Stream-level entailment
Window operator $\mathbb{W}(S)$ (or $\mathbb{L}(S)$)
Window-level entailment over $\mathbb{L}^{sl}(S)$
Landmark window $\mathbb{L}^{sl}(S)$
Stream S

Figure 7.2: *Stream-level entailment: the entailment-regime is applied at a window level over a landmark window $\mathbb{L}^{sl}(S)$, and the window operator defined in the query $\mathbb{W}$or$\mathbb{L}$ is applied on the substream identified by $\mathbb{L}^{sl}$.*

when the most recent of them appears in the ontology substream. For example, if a formula α can be inferred by α_1 in a snapshot at time t_1 and α_2 in a snapshot $t_2 > t_1$, the time instant when α starts to hold is t_2, due to the fact that before t_2 α_2 trueness is not known.

Example 21 (Reasoning in ontology substreams)**.** *Listing 7.4 shows three examples of intra-snapshot entailments: TBox $\mathcal{T}_{social}$ and snapshot $\mathcal{SC}_0^{20}(8,1)$ entail that Alice is a Person (Line 1), :post$_1$ is a Post and CouponPubPost (respectively Lines 2 and 3).*

```
1  $\mathcal{T}, \mathcal{S}_o^c(8,1) \models Person(Alice)$
2  $\mathcal{T}, \mathcal{S}_o^c(8,1) \models Post(:post_1)$
3  $\mathcal{T}, \mathcal{S}_o^c(8,1) \models CouponPubPost(:post_1)$
4  $\mathcal{T}, \mathcal{S}_o^c(15,1), \mathcal{S}_o^c(16,1) \models \exists propagatedBy.\{:post_3\}(:post_2)$
```

Listing 7.4: *Entailment examples in the ontology substream $\mathcal{O}_o^c$*

An example of so-far inference is shown on Line 4, where the axiom $\exists propagatedBy.\{:post_3\}(:post_2)$ is entailed on $Cl^{sofar}(\mathcal{SC}_0^{20}(16,1))$ due to the fact that it is inferred through axioms in $\mathcal{SC}_0^{20}(16,1)$, $\mathcal{SC}_0^{20}(15,1) \in$ prev(16) *and $\mathcal{T}_{social}$.*

Abusing the notation, we indicate with $Cl^{intra}(\mathcal{O}_o^c)$ and $Cl^{sofar}(\mathcal{O}_o^c)$ the intra-snapshot and so-far closure of all the snapshots of $\mathcal{O}_o^c$.

7.5 Stream-level Entailment

On of the main limits of the window-level entailment is that it does consider only a recent portion of the ontology substream: every time a new window is computed, what happened in the past is forgotten, and the entailment regime is applied from scratch restarting by the data contained in the new fixed window.

The entailment regime at stream level overcomes this limit, considering a larger portion of the ontology substream as data to be considered in the entailment regime. Even if the name suggests that the entailment regime considers the whole RDF stream S, in this case the reasoning is made on the top of a landmark window $\mathbb{L}^{sl}$ over S, as depicted in Figure 7.2. This fact is explained by the following considerations. First, the RDF stream has a start point, e.g. the time on which the source starts to supply the data, or the time on which the engine registers to the source and starts to receive the RDF stream. Moreover, in the continuous query answering context, the RDF stream has an end: at each step of instantaneous evaluations, the process considers the RDF stream available up to that moment, i.e. it cannot access data still not available in the future. This behaviour is captured by a landmark window, as in Definition 6.1. Given a query with a dataset *SDS* including a set of sliding and landmark windows over an RDF stream S, S can be represented without loss of generality with $\mathbb{L}^{sl}(S)$, where $\mathbb{L}^{sl}$ starts to operate at the minimum t^0 time value of the window operators defined over S in *SDS*.

Moving from a RDF stream to a landmark window over a RDF stream and fixed an evaluation time t, a fixed window in $(t_0, t]$ is produced, and it can be mapped to an ontology substreams as in Definition 7.3. Anyway, in this setting, we should take into account also the window operators defined by the query (the top block in Figure 7.2): given the evaluation time instant t, it generates a fixed window $W^{(a,b]}$ that should include a portion of the ontology substream obtained by the fixed window $\mathbb{L}^{sl}(S, t)$.To describe this step, in the following we provide the definition of fixed window in the context of an ontology substream.

Definition 7.6 (Fixed windows in ontology substreams)**.** *Given an ontology substream $\mathcal{O}_o^c = (\mathcal{T}, \mathcal{S}_o^c)$ and two time instants a, b such that $o \leq a < b \leq c$, we define a fixed window over an ontology substream $\mathcal{O}_o^c$, denoted $\mathcal{O}_o^c(a, b]$ as:*

$$\{Cl^{sofar}(\mathcal{S}_o^c(i, j))|a < i \leq b\}$$

Similarly to the fixed window in RDF streams (Definition 5.2), the fixed window of the above definition selects a subsequence of the ontology substream using a time condition, where snapshots are closed w.r.t. so-far inference. When the fixed window is applied on an ABox substream (without a TBox), it works similarly to fixed windows on an RDF stream: it selects the set of axioms in the interval defined by the window time interval.

Example 22. *Let $\mathbb{W}$ be a sliding window with parameters $\omega = 2, \beta = 2, t^0 = 0$ over the RDF stream S_{social}. Under stream-level entailment regime, at evaluation time $t = 18$, the basic (or event) graph pattern is evaluated over the RDF substream corresponding to $\mathcal{OC}_0^{18}(16, 18]$ as in Listing 7.5.*

It is worth noting that under simple entailment regime, the scoping element contains only the statements relative to axioms in Lines 1 and 2, i.e. $W^{(16,18]}(S_{social})$. Under window-level entailment regime, the scoping element is composed by the statements relative to $\mathcal{OC}_{16}^{18}$, i.e. the axioms between Lines 1 and 5. The axioms in is not considered because it needs the snapshot $\mathcal{OC}_0^{20}(15, 1)$ to be entailed.

```
1  $(\exists hasAuthor.\{Eve\})(:post_4)$
2  $(\exists in.\{b\} \sqcap \exists on.\{panda\} \sqcap \exists reduce.\{25\} \sqcap \exists mentionedIn.\{:post_4\})(c_2)$
3  $Person(Eve)$
4  $Coupon(c_2)$
5  $MentionedCoupon(c_2)$
6  $\exists propagatedBy.\{:post_3\}(:post_2)$
```

Listing 7.5: $\mathcal{OC}_0^{18}(16, 18]$ *contains the snapshot* $\mathcal{OC}_0^{18}(18, 1)$ *and its so-far closure.*

7.6 Comparing the entailment regime levels

To close this section, we compare the entailment levels.

In the context of a BGP evaluation, the graph-level entailment and the window-level entailment produces the same result. We can show it by exploiting the following property.

Theorem 1. Let $W^{(o,c]}$ be a fixed window over a stream S and $\mathcal{L}$ an ontological language. Let merge be merge of data items in a fixed window, i.e.

$$\mathsf{merge}(W^{(o,c]}(S)) = \bigcup_{(d_i,t_i)\in W^{(o,c]}(S)} d_i,$$

and let closure be either the closure of the content of $W^{(o,c]}(S)$ w.r.t. $M(S)$ under $\mathcal{L}$ or the closure of an ontology $\mathcal{O}$, i.e.

$$\mathsf{closure}(A) = \begin{cases} Cl^{sofar}(A) & \text{if } A = \mathcal{O}_o^c \\ Cl(A) & \text{if } A = \mathcal{O} \end{cases}$$

Under the assumption that no inconsistency is entailed, merge and closure commutes with each other, i.e.

$$\mathsf{merge}(\mathsf{closure}(W^{(o,c]}(S))) = \mathsf{closure}(\mathsf{merge}(W^{(o,c]}(S))) \qquad (7.2)$$

To show that the property holds, the so-far closure of the ontology substream associated to $W^{(o,c]}(S)$ should infer the same axioms of the closure of the ontology associated to the window graph content. It is easy to show that any formula inferred in the former process is inferred also in the latter. Let α be an entailment in the so-far closure $\mathsf{closure}(W^{(o,c]}(S))$, and let $J = \{\alpha_1, \ldots, \alpha_n\}$ be a justification of α, i.e. a minimal set of axioms in the ontology responsible for the entailment. The axioms in J that are not in $\mathcal{T}$, are in $W^{(o,c]}(S)$: it means that they are also in the merge of the snapshots $\mathsf{merge}(W^{(o,c]}(S))$, and the same justification J holds in its closure $\mathsf{closure}(\mathsf{merge}(W^{(o,c]}(S)))$.

To show that the vice versa holds, each formula inferred by the graph closures should be inferred in (at least) one snapshot of the ontology substream. Given an entailment α in $\mathsf{closure}(\mathsf{merge}(W^{(o,c]}(S)))$, and a justification $J = \{\alpha_1, \ldots, \alpha_n\}$, α is entailed in the so-far closure of the snapshot containing the most recent axiom $\alpha_i \in J$.

The equivalence of the two processes in case of basic graph pattern evaluation can be exploited when implementing the system: if the query does not contains basic event patterns, the RSP engine can assume graph-level entailment even if the query should be evaluated under window-level entailment, with a potential gain in performance. In fact, the enrichment introduced by the window-level entailment is useful in the context of event matching, where it can increase the number of answers.

Different is the case of stream-level entailment, where the number of answers in both basic graph and event pattern evaluation is equal or greater the number of answers w.r.t. graph and window level entailment regimes. As explained in Section 7.5, the stream-level entailment regime considers as input a superset of the data considered by graph and window level ones.

Example 23. *Let $\mathbb{W}$ be a sliding window over S_{social} defined through parameters $\omega = 4, \beta = 2, t^0 = 13$. The task to solve is to find the posts that propagates coupon posts. At evaluation time 17, $\mathbb{W}$ produces a fixed window defined in $(13, 17]$. Under all the three entailment regime levels, the answer matches :post$_2$ propagated by :post$_3$.*

At evaluation time 19, $\mathbb{W}$ produces a fixed window in $(15, 19]$. In this case, the stream-level entailment regime produces two results, while the *other two entailments do not produce any result. In fact, the former considers the whole stream, and it infers* $:post_2$ *propagated by* $:post_3$ *at time instant 16, and* $:post_2$ *propagated by* $:post_4$ *at time 18, that are selected in the window and consequently matched as results. The latter ones do not produce any result, because they consider the closure of* $W^{(15,19]}(S)$*, that does not contain the description of* $:post_2$*, i.e., it is not possible to infer any propagatedBy relation.*

7.7 Remarks

In this chapter, we presented how to extend the SPARQL entailment regime extension in the context of RSEP-QL. The main difference is given when the BGP (or, alternatively, the basic event pattern) should be evaluated over a window: in this case, we identified three levels of application of the regime, that may lead to different results.

Part III

Effectiveness of RSEP-QL: coverage and testing

CHAPTER 8

Capturing the semantics of the existing RSP engines

Existing RSP query languages have different underlying semantics, and even if their syntaxes is similar, these differences have fundamental consequences at query evaluation time. In this chapter, we ask: *how can RSP-QL and RSEP-QL be used to capture the heterogeneity of the operational semantics of existing RSP engines?*

We go in depth and we study the query models of C-SPARQL, SPARQL$_{stream}$, CQELS and EP-SPARQL. Those languages and engines have implicit assumptions on how the results of a continuous query are streamed out and when the system should react to changes on the sliding windows. We provide a qualitative analysis of the coverage of RSEP-QL of the aforementioned RSP systems. Next, in Chapter 9, we will formalise how RSP-QL captures the models of C-SPARQL, SPARQL$_{stream}$ and CQELS.

Table 8.1 summarises the comparison of the systems and highlights their differences. As we depicted in the previous chapters, a main difference in RSP engines is given by the support of data stream processing features and the event processing ones. It follows that the engines

Feature	**C-SPARQL**	**CQELS**	**SPARQL$_{stream}$**	**EP-SPARQL**
Window operators	Time-based sliding windows	Time-based sliding windows	Time-based sliding windows	Landmark windows, within operators
Sliding window parameters	α and β	α and β	α and β	
Event pattern operators	timestamp function			SEQ, OPTSEQ, etc.
Stream in the dataset	Sliding windows associated to the default element A_0	Each sliding window has a named element $(w, \mathbb{W})$	Sliding windows associated to the default element A_0	One stream as default element
Evaluation time instants	Window close and Non-empty content	Content-change	Window close and Non-empty content	Content-change
Streaming operator	RStream	IStream	RStream, IStream and DStream	RStream

Table 8.1: *Comparison of RSP engines. On the one hand, C-SPARQL, CQELS and SPARQL$_{stream}$ offers Data Stream Processing features through sliding window operators. On the other hand, EP-SPARQL provides several operators to define Event Patterns, while C-SPARQL supports event matching through the* timestamp *function.*

support different kind of windows: while C-SPARQL, CQELS and SPARQL$_{stream}$ implement time-based sliding windows, EP-SPARQL has landmark windows and within operators. We start from here the analysis, in Section 8.1 we discuss the event processing operators, while in Section 8.2 we analyse the differences in sliding windows in the different solutions. Next, in Section 8.3 we discuss the different types of datasets that are built by the RSP engine; in Section 8.4 we discuss the continuous evaluation process; while in Section 8.5 we report on the *streaming operators. We close with some considerations about the query language under development at W3C in Section 8.6, and with some final remarks in Section 8.7.

This chapter is based on "Towards Unified Language for RDF Stream Query Processing", published on the Satellite Events of Extended Semantic Web Conference. I drove the development of this paper: I wrote the first draft of the paper, that was later on edited by J.P. Calbimonte, which added examples in the sections. The work has been finally read and refined by Prof. E. Della Valle and Prof. Ó. Corcho.

8.1 Complex Event Processing operators

In general, in every RSP engine that supports SPARQL, it is possible to verify the existence of an event (i.e. EVENT w_i P), by inserting P in the WINDOW pattern of w_i. As discussed in Chapter 6, EP-SPARQL is the RSP language with highest support to CEP features. We compare EP-SPARQL and RSEP-QL on two features: the data model and the event pattern matching.

The two query models adopt two different data models: EP-SPARQL assumes as input a stream of statements annotated with intervals, while RSEP-QL adopts streams of graphs annotated with time instants. This fact brings to a slightly difference in the way time instants associated to event mappings are interpreted. In EP-SPARQL, the *simple* events are already characterized by two different time instants (validity interval), while in RSEP-QL simple event mappings have one time instant (the time on which the event happens). Consequently, the time annotations of the complex events have a different meaning: in EP-SPARQL, they indicate the validity interval of the events; in RSEP-QL, the temporal annotation associated to the event mappings are the time instants that justify it.

Moving to the operator and the operational semantics, we have evidence of the fact that RSEP-QL can capture the behaviour of EP-

RSEP-QL	EP-SPARQL/C-SPARQL
$\mathbb{L} + \text{SEQ}$	EP-SPARQL unrestricted
$\mathbb{L} + \text{SEQ}^c + \mathcal{P}^c$	EP-SPARQL chronological
$\mathbb{L} + \text{SEQ}^r + \mathcal{P}^r$	EP-SPARQL recent
$\mathbb{W} + \text{SEQ}^n$	C-SPARQL SEQ (timestamp)
$\mathbb{W}$	C-SPARQL time-window

Table 8.2: *Coverage of DSMS and CEP features of RSEP-QL compared to EP-SPARQL and C-SPARQL.*

SPARQL and C-SPARQL. RSEP-QL models both the event patterns and their evaluation semantics takes into account the presence of selection and consumption policies. While models for event patterns were already available in [5, 41], a formalization for policies was missing. These policies are key to determine the answer that a query should produce for a given input stream. Therefore, it is not possible to consider those aspects as only technical or implementation related.

RSEP-QL captures the behavior of the sequential event pattern matching features of EP-SPARQL and C-SPARQL [R3], including the different selection and consumption policies that they provide. To illustrate the coverage of these features, Table 8.2 shows the equivalence of the main features in RSEP-QL with their counterparts in both EP-SPARQL and C-SPARQL. For instance, one can observe that an EP-SPARQL sequence pattern (with recent policy) can be captured by the SEQ^r operator and the $\mathcal{P}^r$ function on a landmark window in RSEP-QL.

8.2 Sliding window operators

RSEP-QL defines the sliding window operator through three parameters: width (α), slide (β) and t^0 (Section 5.3). The query models of C-SPARQL, SPARQL$_{stream}$ and CQELS only allow controlling the width and slide values of the sliding windows. The t^0 parameter is left to be managed internally by systems and none of the query languages provide syntactic constructs to constrain it. The query designer cannot determine when the first window of the sliding window opens: each sliding window can start at different time instants, and consequently, the system can produce different outputs.

```
REGISTER STREAM :longStream AS
CONSTRUCT {?shop :totalLong ?totalLong}
FROM STREAM :in [RANGE 60m STEP 20m]
FROM :shops
WHERE{
  SELECT ?shop ((COUNT(?shop) AS ?totalLong)
  WHERE{ ?user :isNearby ?shop}
  GROUP BY ?shop
}
```

Listing 8.1: *C-SPARQL Query Q_1^{CS}: it counts every 20 minutes the number of shops to which a user was near by in the last 60 minutes*

Example 24. *Let's consider the C-SPARQL query Q_1^{CS} in Listing 8.1: it processes the input stream S_{nearby} in order to count the number of times a user goes near a shop. The query defines a time-based sliding window $\mathbb{W}$ with the parameters ($\alpha = 60, \beta = 20$). The t^0 parameter cannot be explicitly defined and the sliding window can open at different time instants, as shown in Figure 8.1. This fact influences the portion of the stream that is captured by the sliding window operators: if we focus on the first window of each sliding window operator, we can notice that in the first case ($t^0 = 0$), it contains the elements s_1, s_2 and s_3; in the second case ($t^0 = 1$), it contains the elements s_1, s_2, s_3 and s_4; while in the third case ($t^0 = 2$), the first window captures s_3 and s_4 only.*

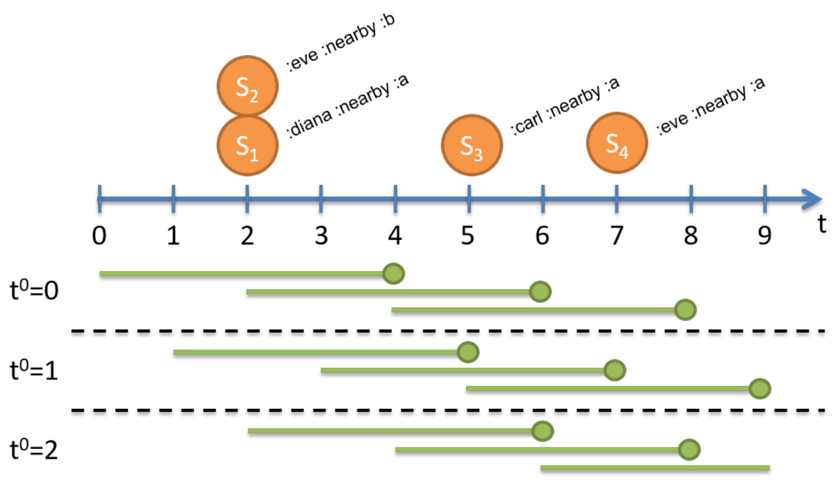

Figure 8.1: *Sliding windows with different t^0 values*

Similarly, CQELS and SPARQL$_{stream}$ have the same problem. On the contrary, EP-SPARQL does not suffer of this issue related to

t^0because it does not provide time-based sliding window operators and the within operators determines fixed windows in the context of events. Thus, the time on which the within operators start to operate does not matter.

Example 25. *In the query in Listing* **??**, *the two computed fixed window are in the scopes* $(2, 12]$ *and* $(7, 17]$, *independently by the time on which the query is registered when the within operator starts to operate (assuming that the query is registered before time instant 2). The equivalent C-SPARQL query, listed in Listing* **??** *suffers of the* t^0 *issue depicted above: if the window starts to operate at time instant* 0, *the answer containing shop b is not detected.*

8.3 RSP-QL datasets

The RSP-QL dataset (Section 5.3) is a generic definition, which is constrained by the syntaxes of query languages of C-SPARQL, CQELS, SPARQL$_{stream}$ and EP-SPARQL.

In CQELS-QL, a named time-varying graph is associated to each window; window content can be accessed using the STREAM operator. This operator is analogous to the RSP-QL's WINDOW: it sets as active graph the fixed window generated by a sliding window. In CQELS is not possible to put the content from the sliding window in the default element of the dataset. The syntax of CQELS-QL brings to assign an implicit name to each sliding window; in other words, it is not possible to assign explicit identifiers to the sliding windows. In this way, the language gains in usability, but it forbids to add sliding windows contents to the default element.

```
CONSTRUCT {?shop a :PopularShop}
FROM :shops
WHERE{
  STREAM :in [RANGE 60m STEP 20m] {
    SELECT ?shop (COUNT(*) AS ?totalLong)
    WHERE{ ?user1 :isNearby ?shop }
    GROUP BY ?shop
  }
  STREAM :in [RANGE 20m STEP 20m] {
    SELECT ?shop (COUNT(*) AS ?totalShort)
    WHERE { ?user2 :isNearby ?shop }
    GROUP BY ?shop
  }
  ?shop a :Shop
  FILTER (?totalShort-?totalLong > $threshold$)
16 }
```

Listing 8.2: *CQELS query to find the emerging shops*

Example 26. *We want to search of the trending shops. One way of characterizing trending shops is by finding out those which are frequently appearing in the last time period, and less frequently in the past. This apparently simple query contains some interesting elements that reveal differences among existing RSP solutions and challenge some of their capabilities. The query that solves this task has to look at the same stream from two different perspectives: in the one hand it needs to keep track of very recent shops, while on the other hand needs to be aware of a longer time span, so that it can make sure that the new shops were not present before.*

Listing 8.2 reports the CQELS-QL query that models the task. The query declares two sliding windows over the same input stream `:in`*: the first,* $\mathbb{W}_l^{CQ}$ *(Line 4), has width* $\alpha_l = 60$ *minutes and slide* $\beta_l = 20$ *minutes; the second,* $\mathbb{W}_s^{CQ}$ *(Line 9), has width and slide* $\alpha_s = \beta_s = 20$ *minutes (it is a tumbling window). Each sliding window contains a subquery to compute the shop and the total number of their mentions (respectively* `?totalLong` *and* `?totalShort`*). The trending value is computed at Line 15: if this value is greater than the* ***$threshold$*** *value, then the shop is selected as trending and it is streamed out according to the* CONSTRUCT *clause at Line 1. The RSEP-QL dataset of this query is the following:*

$$SDS^{CQ} = \{\texttt{:shops}, (w_l, \mathbb{W}_l^{CQ}(\texttt{:in})), (w_s, \mathbb{W}_s^{CQ}(\texttt{:in}))\}$$

The two sliding windows, $\mathbb{W}_l^{CQ}$ *and* $\mathbb{W}_s^{CQ}$*, do not have an explicit name in the query.*

C-SPARQL does the opposite: its query language does not allow to name the time-varying graphs computed by the sliding windows. As a result, all the graphs computed by the sliding windows are merged and set as default graph.

Example 27. *The task in Example 26 cannot be written in one C-SPARQL query, as the syntax of C-SPARQL does not allow to distinguish among multiple windows defined over the same stream. Let us consider the query in Listing 8.3, the RSP-QL dataset built by the query is the following:*

$$SDS^{CS} = \{A_0 = \{\mathbb{W}_l^{CS}(\texttt{:in}), \mathbb{W}_s^{CQ}(\texttt{:in})\}\}$$

The dataset SDS^{CS} has the two sliding windows in the default graph position, i.e. the graphs produced by the sliding windows are merged in the default graph. In fact, C-SPARQL does not allow to name the sliding windows and, consequently, the generated windows.

```
REGISTER STREAM :out AS
CONSTRUCT {?shop a :PopularShop}
FROM STREAM :in [RANGE 60m STEP 20m]
FROM STREAM :in [RANGE 20m STEP 20m]
WHERE{
?m sioc:topic ?shop.
}
```

Listing 8.3: *C-SPARQL Query: the triple pattern is evaluated against the union of the two sliding windows*

It is actually possible to solve the task through a network of three C-SPARQL queries. First, two queries Q_1^{CS} and Q_2^{CS} process the input stream `:in` *in order to compute the shop mentions in the long and in the short windows. Q_1^{CS}, reported in Listing 8.1, builds a stream* `:longStream` *that brings the shops and the number of appearance of the shop in the last 60 minutes (according to the sliding window definition at Line 3). Query Q_2^{CS} builds a stream* `:shortStream` *with the shops and their number of mentions in the previous 20 minutes. Q_2^{CS} is similar to Q_1^{CS} but for the window size, the name of the output stream and the property name in the* CONSTRUCT *clause (we omit it for brevity). Those streams are the input of query Q_3^{CS}, reported in Listing 8.4, which computes the trending value of the shops and adds the shop in the output stream* `:out` *if the trending value is greater than the threshold (Line 7).*

```
1 REGISTER STREAM :out AS
2 CONSTRUCT {?shop a :PopularShop}
3 FROM STREAM :longStream [RANGE 20m STEP 20m]
4 FROM STREAM :shortStream [RANGE 20m STEP 20m]
5 WHERE{
6   ?shop :countLong ?totalLong; :countShort ?totalShort.
7   FILTER (?totalShort-?totalLong > $threshold$)
8 }
```

Listing 8.4: *C-SPARQL Query Q_3^{CS}: computation of the trending shops*

In SPARQL$_{stream}$, similarly, named stream graphs can be declared but not used inside the query body. Therefore, graphs derived by time-based sliding windows are logically merged in the default graph of the query dataset.

On the one hand, it may be considered easier to write queries in C-SPARQL and SPARQL$_{stream}$ than in CQELS: all the sliding windows are declared before the WHERE clause and the data from the streams is available in the default graph. On the other hand, CQELS allows to write more complex queries, such as queries with multiple sliding window over the same stream. However, in the examples above we shown that the same task can be solved in all the languages we analysed: CQELS allows to express the task in one query, while C-SPARQL and SPARQL$_{stream}$ require network of queries.

Finally, EP-SPARQL assumes the existence of "only" one stream S and the dataset default element is always a landmark window over S. It is worth noting that, despite what happens in other languages, EP-SPARQL can write query outputs on S.

8.4 Evaluation time instants

In Section 5.8, the concepts of policy and strategy were presented. They allow determining the set of time instants ET on which evaluations occur. Looking at the available RSP systems, it is possible to observe that policy and strategy are features of the implementation, i.e. neither the query model nor the query language syntax allow expliciting control policies and strategies. This is a major source of heterogeneity among RSP systems.

C-SPARQL and SPARQL$_{stream}$ adopt a Window Close and Non-empty Content policy to the windows of the query, while CQELS and EP-SPARQL implement the Content-Change policy (it evaluates the query as every time new statements enter the window). It follows that the systems build different evaluation time instant sets and consequently, they stream out new results at different time instants.

Example 28. *Let's consider the sliding window $\mathbb{W}$ in Figure 8.2: it is over S_{nearby} and it is defined through $(\alpha = 5, \beta = 2, t^0 = 1)$. The lower part of the figure shows the effect of different policies on the set time instants the evaluation occurs: each diamond represent an evaluation time in the relative RSP system. While CQELS and EP-SPARQL evaluate the query as soon as it receives new timestamped RDF statements, C-SPARQL and SPARQL$_{stream}$ follow a regular pattern: they report every time the windows close (except for the cases where windows are empty).*

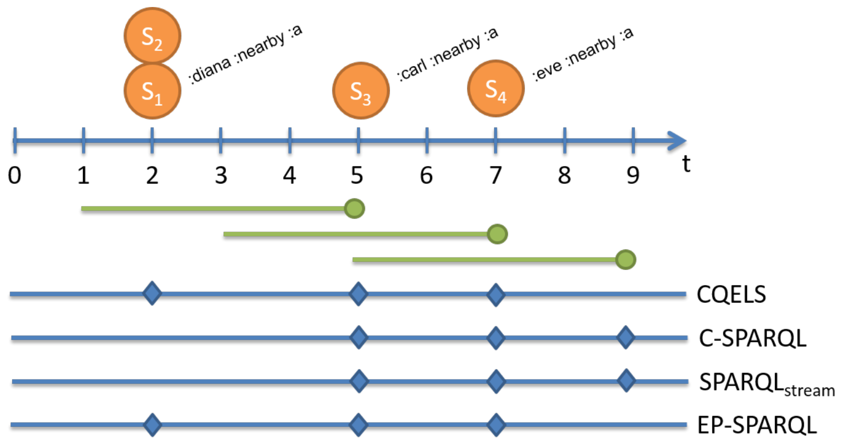

Figure 8.2: *Policy implementations in RSP systems*

8.5 Streaming operators

The last feature on which we focus is the streaming operator support. In Section 5.6, we defined three streaming operators: RStream, IStream and DStream, but only SPARQL$_{stream}$ supports all of them. C-SPARQL and EP-SPARQL permit the RStream operator only (it streams out the whole output at each evaluation), while CQELS admits only the IStream one (it streams out only the new statements).

C-SPARQL and EP-SPARQL answers can be verbose as the same solution mapping could be in different portions of the output stream computed at different evaluation time instants. It is suitable when it is important to have the query answer at each step.

CQELS streams out the difference between the timestamped set of mappings computed at the last step and the one computed at the previous step. It means that CQELS uses an IStream operator to produce the output. That is, it is impossible to produce an RStream with the whole result of each operator. In other words, the algebraic expressions of CQELS-QL always assume Istream as outer element of the algebraic expression. Consequently, answers are usually short (they contain only the difference) and it is a good solution when data exchange is expensive.

8.6 Analysis of the W3C RSP Query Language proposal

The problem of heterogeneity is currently studied by different groups. Initiatives have started with the goal of proposing a common model and query language for processing RDF Streams, converging in the RSP Community Group of the W3C[1]. The emergence of such a model is expected to take the most representative, significant and important features of previous efforts, but it will also require a careful design and definition of its semantics. In this context, it is essential to lay down the foundations of formal semantics for the standardized RSP query model, such that we consider beforehand the notions of correctness, continuous evaluation, evaluation time and operational semantics, to name a few. In this section, we briefly analyse the language under development by the W3C RDF Stream Processing community group[2]. We describe how limitations in current RSP languages are partially solved by the current proposed language of the W3C RSP Community Group, we show that the new language proposed in the RSP Group is covered by the RSEP-QL model, therefore providing a well-founded semantics for it. We also show that this new language allows covering cases that previous RSP languages are unable or only partially able to address.

```
REGISTER STREAM :out
AS CONSTRUCT RSTREAM{ ?shop a :PopularShop}
FROM NAMED WINDOW :lwin ON :in [RANGE PT60M STEP PT20M]
FROM NAMED WINDOW :swin ON :in [RANGE PT20M STEP PT20M]
FROM GRAPH :shops
WHERE{
  ?shop a :Shop
  WINDOW :lwin{
    SELECT ?shop (COUNT(*) AS totalLong)
    WHERE { ?m1 sioc:topic ?shop. }
    GROUP BY ?shop
    }
  WINDOW :swin{
    SELECT ?shop (COUNT(*) AS totalShort)
    WHERE { ?m2 sioc:topic ?shop. }
    GROUP BY ?shop
    }
  FILTER(?totalShort-?totalLong) > $threshold$)
}
```

Listing 8.5: *The task in Example 26 modelled through the W3C RSP Query Language*

[1]W3C RSP Group: http://www.w3.org/community/rsp

[2]We refer at the version of the language available at July 2015

Listing 8.5 shows the query that captures the task in Example 26. Observing the query, it is possible to note that the new language puts together the features of C-SPARQL, CQELS and SPARQL$_{stream}$ in order to overcome some of the limits highlighted in the previous sections.

First, the new language allows to declare the *streaming operator (RStream, at Line 2). Moreover, the new language allows to build both CQELS and C-SPARQL data sets: it is possible due to the the sliding windows declarations in the FROM clause, combined with the use of the NAMED keyword (Lines 3 and 4). Next, in the WHERE clause, the WINDOW keyword is used to refer to the content of the named sliding windows (similarly to the GRAPH keyword in SPARQL). The RSEP-QL dataset built by the query is:

$$SDS^{RSP} = \{(\texttt{:lwin}, \mathbb{W}_l^{RSP}(\texttt{:in})), (\texttt{:swin}, \mathbb{W}_s^{RSP}(\texttt{:in}))\}$$

Nevertheless, this syntax is not enough to determine a unique query following th RSEP-QL model. As we explained above, there is no explicit information to determine which is the report policy and when the sliding windows start to work (i.e. the t^0 value). A possible solution for the latter problem can be the introduction of a `STARTING AT` clause to express the t^0 value. Alternatively, the language could allow to define a pattern to express the t^0 value.

8.7 Remarks

In this chapter, we exploited RSEP-QL to explain and compare the model the query languages and models of C-SPARQL, SPARQL$_{stream}$, CQELS and EP-SPARQL. We shown that RSP-QL captures the semantics of those different engines and languages. Having well-defined RSP engine models would enable interoperability through common query interfaces even if the implementations architectural approaches. In this sense, RSEP-QL aims at constituting a contribution to ongoing efforts in the Semantic Web community to provide standardized and agreed definition of extensions to RDF and SPARQL for managing data streams.

Furthermore, it is worth noting the expressivity of RSEP-QL allows defining complex queries that combine both windows and event patterns, defining queries that none of the existing engines may express. For instance, consider that in a social network we want to find the post made by a user that is then followed by a popular user, defined as someone that gets a lot of mentions in the last hour and has a lot of followers. In this case a time window is needed to keep track of the number of mentions in the last hour. Then the sequence pattern is required to capture the fact that someone is followed after he made a post. The contextual information is used to look for the number of followers of a person, to determine if he is popular.

CHAPTER 9

Correctness assessment in RSP engines

Several initiatives started to test and compare RSP engines. Some early efforts have been done as well in the context of RDF Streaming benchmarking: SRBench [104] and LSBench [89]. SRBench is mainly focused on understanding coverage for SPARQL constructs. LSBench is mainly focused on understanding the throughput of existing RDF Stream processors and checking correctness by comparing the results of different processors and quantifying the mismatch among them. However, to make a step further towards a more precise assessment on correctness, it is necessary to consider the different operational semantics of the benchmarked systems. In Chapter 8, we discussed how RSP-QL and RSEP-QL can capture the query models of existing RSP systems. It follows that they are candidates to be at the basis of a formal definition of correctness to be used in the context of benchmarking. In this chapter, focus on RSP-QL and we study the following problem: *can RSP-QL be used to test whether an RSP system is correct or not?*

We use the RSP-QL model to address the problem of describing the execution of a query, given an input stream and a system previously characterized. This allows us to check the correctness of RSP engines, by comparing the modelled output and the system actual output.

Taking into account the properties described in the first part of the chapter, we propose CSRBench, a benchmark for correctness checking. CSRBench extends SRBench, with a set of queries and an oracle to assess the correctness of RSP engines. We apply CSRBench to a set of existent engines and report on our findings in Section 9.3, and we close with some remarks in Section 9.4.

This chapter is based on "On the Need to Include Functional Testing in RDF Stream Engine Benchmarks", published at the First International Workshop on Benchmarking RDF Systems and its extended version "On Correctness in RDF Stream Processor Benchmarking", published at the 12^{th} International Semantic Web Conference. The initial idea of this work started by Prof. E. Della Valle, that observed that RSP engines were actually not behaving as expected. I drove the investigation of the problem under his supervision: we designed the tests and we designed the oracle system, that I developed. J.P. Calbimonte integrated and adapted the tests in SRBench, defining the data and the queries of CSRBench; he also executed the tests over SPARQL$_{stream}$. Marco Balduini ran the experiments in C-SPARQL and CQELS. Prof. E. Della Valle and Prof. Ó. Corcho supervised the work and refined the final drafts of the articles.

9.1 Correctness assessment in RSP engines

In Section 8, we described that RSP engines have some parameters that are not described in their model. As consequence, they have different behaviours when they process the same query over the same data, and they produce multiple correct answers (e.g. the t^0 parameter described in Section 8.2). These parameters are *hidden*, in the sense that they are not defined in the models and there is no way to control them through the query languages. They are usually managed by the engines that adopts the models, or by constraints on the query languages.

Even if those parameters cannot be controlled, it is necessary to take them into account to determine if the answer is correct. In fact, while Intuitively, a query Q^R for an RSP system R is a *partially defined* RSP-QL query, i.e. some of its parameters are undefined or implicitly defined. Every RSP system can be analysed, so that it is possible to determine the values that those parameters assume and, consequentially, derive a set, denoted of RSP-QL queries that produces all the correct answers that Q^R may compute.

In fact, each RSP-QL query generates a different (but correct) answer: if the result of the query q^R matches one of them, we state that it is correct (Section 9.1.1). Anyway, to assess whether an RDF Stream Processing system behaves correctly or not, some assumptions and approximations are required, due to the infinite nature of input streams (Section 9.1.2).

9.1.1 Correctness based on RSP-QL

The RSP-QL model is built to capture the query model and the operational semantics of existing RSP systems. This model is able to reproduce each result of the targeted systems. Anyway, as explained in Section 8, each system R has a different behaviour, and the relative queries are not fully-defined RSP-QL queries. In the next definition, we use the analysis in Section 8 and we formalise the notion of correctness w.r.t. RSP-QL. Given an engine R and a query Q^R, we define set of well-defined RSP-QL queries that produces all the possible correct answers of Q^R. The process of finding the queries is captured by a function, named $rspqlQueries(\cdot)$, that is defined in the remaining of this section.

Definition 9.1 (Correctness of an RSP engine answer)**.** *Let R be an RSP system (C-SPARQL, CQELS or $SPARQL_{stream}$), let Q^R denote a continuous compliant to R and let $Ans_R(Q^R)$ be the answer produced by the continuous evaluation of Q^R in R. It exists a set $\mathbb{Q} = rspqlQueries(Q^R, R)$ of RSP-QL queries that can be derived by Q^R, such that the answer $Ans_R(Q^R)$ is* correct *with regards to the RSP-QL model iff it exists a query $Q \in \mathbb{Q}$ such that $Ans_R(Q^R) = Ans(Q)$.*

$Ans(Q)$ is the output of Q as defined in Section 5.9. To complete the definition, we need to explain how to build the *rspqlQueries* function. The function is strictly related to the peculiarities highlighted in Chapter 8: given a continuous query Q^R for R, it composes RSP-QL queries by adding to Q^R information that can be elicited by the operational semantics of R.

Given an RSP engine R and a query Q^R, the function *rspqlQueries* generates a set of RSP-QL queries $\mathbb{Q}$. For each query $Q \in \mathbb{Q}$, the algebraic expression and the query form are the same of Q^R. The set of evaluation time instants is set accordingly to the report policy of R (see Sections 5.8 and 8.4). Regarding the dataset, we analysed two main differences between the RSP-QL query model and the RDF Stream Processing systems: Section 8.3 shows that each system has syntactical constraints that limit the shape of the dataset and Section 8.2 discusses the fact that t^0 instants are system-related parameters and they are out of control of the query designer. While the former does not influence the number of queries in $\mathbb{Q}$, the latter generates multiple alternative sliding window operators and, consequently, different RSP-QL queries. The sliding windows operators of the queries in $\mathbb{Q}$ are built in the following way:

- for each time-based sliding window $\mathbb{W}_i$ in Q^R, determine the set $T^0_{\mathbb{W}_i}$ of possible t^0 instants – the time instants on which the first window of $\mathbb{W}_i$ opens:

  $$T^0_{\mathbb{W}_i} = \{a, a+1, \ldots\},$$

  where a is the query registration time.

- compute the set Z with the combinations of the starting time instants of the sliding windows in Q^R:

  $$Z = \prod_{i=1}^{n} T^0_{\mathbb{W}_i}$$

Each element of the Z set is a vector z of dimension n (where n is the number of sliding window operators in Q^R). The number of vectors in Z is the number of queries that $rspqlQueries$ generates. In particular, given the j-th vector $z_j \in Z$, the query Q_j has a dataset with n sliding windows. For each value z_{ji} of the vector z_j ($i \in [1, n]$), a sliding window $\mathbb{W}_i$ is defined as:

$$\mathbb{W}_i(\alpha_i, \beta_i, z_{ji}),$$

where (α_i, β_i) are the width and slide parameters that define the i-th sliding window of Q^R.

9.1.2 Correctness in practice

Definition 9.1 gives a notion of correctness assessment for RSP systems. The idea is to compare the output of a system with the ones generated through the RSP-QL query model, and check if the answers match. Anyway, the notion is theoretical and it is not feasible in reality:

1. the continuous and infinite nature of data streams do not allow determining whether two answers match. The infinite input lengths imply undecidability in the matching problem – it is possible to determine if inputs are different, but it requires infinite time to determine if they are equal;

2. the *rspqlQueries* function generates a RSP-QL query for each possible combination of t^0 values, but those combinations are infinite (sliding windows can start to work at any time instant).

We start to analyse the second problem. To cope with the infinite t^0 combinations, we exploit the following property of the sliding window operator.

Theorem 2. Let $\mathbb{W}'$ and $\mathbb{W}''$ be two sliding window operators, defined respectively through (α, β, a) and $(\alpha, \beta, a+n\beta)$, where $a \in T$ is a time instant and n a natural number. The sliding windows are applied to a generic stream S; for each time instant $t \in T$ on which $\mathbb{W}''$ operates, i.e. $t \geq a + n\beta$, it holds:

$$\mathbb{W}'(S, t) = \mathbb{W}''(S, t)$$

The proof is straightforward. Intuitively, the slide parameter introduces a periodicity in the window: windows generated by sliding windows with the same width and slide parameters, and t^0 values of a and $a + n\beta$ overlap, capturing the same portions of the streams. Consequently the results will be the same from the starting time of the most recent sliding window.

We exploit this property to limit the number of the t^0 combinations and consequently the number of queries that *rspqlQueries* generates. Exploiting this property, we modify the *rspqlQueries* function in the following way: given a query Q^R with n sliding windows, we define the set $T^0_{\mathbb{W}_i}$ of the i-th sliding window as:

$$T^0_{\mathbb{W}_i} = \{a, a+1, \ldots\}$$

Exploiting the property, we bound the set:

$$T^0_{\mathbb{W}_i} = \{a, a+1, \ldots, a+\beta_i - 1\}$$

Now the set Z (the Cartesian product of the $T^0_{\mathbb{W}_i}$ sets) is bound and the *rspqlQueries* function generates a finite number of RSP-QL queries.

We need also to introduce a constraint in Definition 9.1, to set a time instant on which the correctness assessment starts.

Definition 9.2 (Correctness in practice)**.** *The answer $Ans_C(q)$, produced by continuously executing Q^R on R, is* correct *with regards to the RSP-QL model iff from a time instant t_s, there exists a query $Q \in \mathbb{Q}$ such that $Ans_C(q) = Ans(Q)$. t_s is:*

$$t_s = \max\{t_j | t_j \in T^0_{\mathbb{W}_1} \cup \ldots \cup T^0_{\mathbb{W}_n}\}$$

This new definition sets a time instant on which the comparison starts. In particular, it cuts off the transient state of the query answering (the registration and the set up of the query), and it focus on the stable phase of the process. It is a suitable assumption, due to the fact that in stream processing it is common to focus on behaviour of the system when system is in a stable state [8].

Regarding the problem of the infinite input length, it is necessary to bind the length of the stream. This bound has to guarantee that the inputs are long enough to discover long-term running problems (burn-in tests [72]) of the RSP engine, e.g. windows misalignment over the data streams.

9.2 CSRBench: Correctness Extensions of SRBench

In the main problem dimensions are defined, then we explain the design of the benchmark queries in Section 9.2.2. Next, Section 9.2.3 presents the architecture and the implementation of the oracle, an open-source software that performs the assessment of the correctness.

9.2.1 Problem dimensions

Even if the result of a query answering should depend on the input (query and data), we explained above how in current RSP solutions, the engine has an active role and affects the output of the process. It means that we should take into account three dimensions while designing CSRBench: the system, query and input stream data. For the system dimension, we consider:

- *Reporting* policies, as described in Section 5.8.
- Sliding window initial time t^0, which is used by the system to determine the scope of the produced window (and consequently the content). While this parameter is usually not configurable, it can be inferred in post-execution analyses.
- Input stream timestamp policy.

For the query dimension, we focus on the time-based sliding window and *streaming operators, largely neglected in previous benchmarking efforts. Checking correctness with different window configurations is one of the key elements of the proposed extensions. In particular we consider:

- *Window size.* Varying the window sizes (e.g. 10s, 1s, 100ms) results in different scopes, and consequently different window contents.
- *Window slide.* Variations on the window slide (e.g. slide every 1s, every 10ms) also produce differences on the scope. A slide equal to the window size indicates a *tumbling* window, while a slide smaller than the size produces a *sliding* window. We propose testing these different combinations.
- **streaming operator.* While some RSP engines provide only one default operator for the output, others allow explicitly indicating the type of R2S that is expected in the results.

For the input stream dimension, we can briefly mention the input data rate (e.g. triples per second), the window content size (e.g. number of triples) and the data stream distribution (constant, normal, bursts in the input data, etc.).

9.2.2 CSRBench queries

CSRBench uses as input data the one of SRBench. The main difference is related to queries: the benchmark queries are modified in order to stress the time-based sliding window operators, adding the following three types of queries to the existing of SRBench: *queries with parametrized sliding window parameters*, *queries with parametrized aggregations* and *queries with joins involving data from different stream items*. Because the queries are parametrized, the different combinations are useful to produce a set of concrete queries that cover a wide range of cases. A full query descriptions in the three languages available at: `http://www.w3.org/wiki/CSRBench`.

Parametrized sliding window size and slide. By varying the window width and slide, the query in Listing 9.1 allows testing different cases: different window sizes and window slides (e.g. 10, 100, 1000 ms., etc.). Therefore, the value assigned to these two parameters will allow obtaining sliding or tumbling windows.

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-
    observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl
    #>
SELECT ?sensor ?tempvalue ?obs
```

```
FROM NAMED STREAM <http://cwi.nl/SRBench/observations> [NOW - %
    WSIZE% MS SLIDE %WSLIDE% MS]
WHERE { ?obs om-owl:observedProperty weather:_AirTemperature ;
             om-owl:procedure ?sensor ;
             om-owl:result [om-owl:floatValue ?tempvalue] .
        FILTER(?tempvalue > %TEMP%) }
```

Listing 9.1: *Parametrized window slide and size, example in SPARQL$_{stream}$. Parameters are indicated as strings through percentage symbols.*

Parametrized aggregate query. Aggregate queries are commonly used in data stream processing, as a common task is the computation of data trends and summarization rather than on individual data points. Aggregates pose challenges to the computation of the window content and depending on the streaming processor report and tick policies, the results of a sum, average or other function may greatly vary. This type of issues are often overlooked when querying single stream triples.

Joins of triples in different timestamps. The queries types described above include graph pattern matching of triples that are typically received at the same timestamp (or nearly), e.g. an observation and its value, its type, etc. However, there are cases where queries including joins at different timestamps may be relevant. This is more challenging for query engines and correctness checking. For instance, the query in Listing 9.2 asks for sensor stations that record a high atmospheric temperature variation, in a time window.

It is worth noting that this query produces answers when the temperature increases or decreases (there is no control about the order of the observations, so `?value1` could be before or after `?value2`). If we would like to write the query that looks for increasing temperature values, we should write a multi-window query, or we need a mechanism to put constraints on the application timestamps in the query, such as C-SPARQL's timestamp function.

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-
    observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl
    #>
REGISTER QUERY q AS SELECT ?sensor ?ob1 ?value1 ?obs
FROM NAMED STREAM <http://cwi.nl/SRBench/observations>[RANGE %
    WSIZE% S STEP %WSLIDE% S]
WHERE { ?ob1 om-owl:procedure ?sensor ;
             om-owl:observedProperty weather:_AirTemperature ;
             om-owl:result [om-owl:floatValue ?value1].
```

```
        ?ob2 om-owl:procedure ?sensor ;
             om-owl:observedProperty weather:_AirTemperature ;
             om-owl:result [om-owl:floatValue ?value2].
        FILTER(?value1-?value2 > %VARIATION_THRESHOLD%)  }
```

Listing 9.2: *Query joining triples of different timestamps, example in C-SPARQL. Parameters are indicated as strings through percentage symbols.*

9.2.3 An Oracle for Automatic Correctness Checking

Once the benchmark is defined, we need a way to check if the results, provided by a system to the benchmark queries and input, correspond to the expected ones according to the system operational semantics. For this we propose an *oracle* that generates and compares results of RDF stream processors and checks their correctness. As explained above, given an input stream S, a target system R and a query Q^R, the continuous evaluation Q^R in R produces an answer $Ans_R(Q^R)$. The oracle implements the process described above: given the query Q^R and a RSP-QL model M^R that captures operational semantics of R, it derives the set of equivalent queries $rspqlQueries(Q^R, R)$, it generates all the possible answers and checks if one of them matches $Ans_R(Q^R)$.

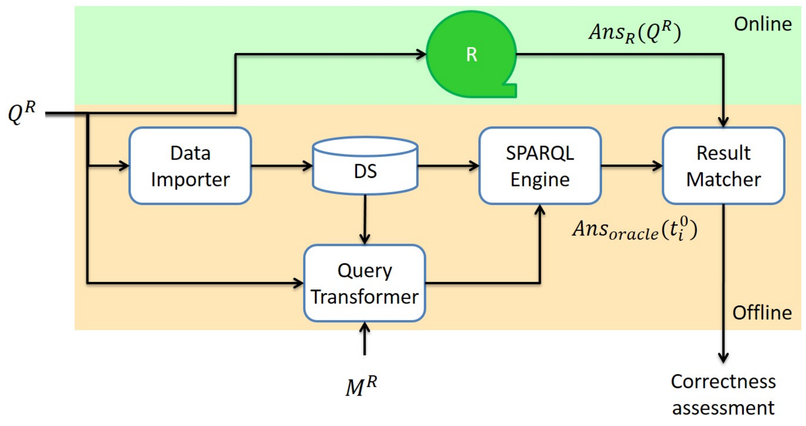

Figure 9.1: *Oracle for RSP query results correctness checking.*

The architecture of the oracle is shown in Figure 9.1. The oracle is based on the off-line execution of a continuous query Q^R, that is translated in a set of SPARQL queries evaluated over an RDF dataset that simulates an RDF stream. The oracle operates in two main stages: (i) the setup of the dataset, and (ii) the execution and comparison of the results.

The main goal of the Dataset Importer is to produce an RDF dataset that allows to compute the data of the time-varying graphs and sliding windows contained in the RSP-QL dataset for each evaluation time instant. This dataset is composed of a metadata RDF graph G_m and a set of RDF graphs $\{G_t\}$ that captures the input stream S. Given that all the engines we are considering use streams of timestamped statements, the original RDF stream S is composed of a sequence of elements $(d = \{(s, p, o)\}, t)$, where d is a graph embedding one RDF statement (s, p, o), and t is the application timestamp, following the model described in Section 4.1. For each timestamp t in S, a corresponding data graph G_t is created, and also the following metadata triple is added in G_m:

$$G_t \ \ \textit{:hasTimestamp} \ t.$$

Finally, for each t, the data items of S with timestamp t are imported into G_t, i.e.

$$G_t = \{(s, p, o) \mid (d = \{(s, p, o)\}, t) \in S\}.$$

Once the RDF dataset is set up, the oracle computes the correct answers of Q^R w.r.t. R. To do it, the oracle translates Q^R into a set a set of sequences of plain SPARQL queries to be executed over the RDF dataset. The execution of a sequence of SPARQL queries simulates the execution of a RSP-QL continuous query in $rspqlQueries(Q^R, R)$ over a limited amount of time. The translation performed by the oracle in this work mainly considers the window operators of the continuous query, and can be summarized as follows.

1 the oracle computes the set Z of values to be assigned to t^0, according to Section 9.1.2. $Z = \{t^0_1, t^0_2, \ldots, t^0_k\}$;

2 the oracle iterates over Z. At each iteration, the oracle assigns a value t^0_i of Z to t^0, and:

2.1 oracle computes the answer $Ans_{oracle}(t^0_i)$. To do it, oracle computes the set $ET^{t^0_i}$ of evaluation time instants, according to the RSP-QL policy of M^R (as in Section 5.8). For each evaluation time in $ET^{t^0_i}$:

2.1.1 oracle determines the next present window: it computes the time interval $(o_p, c_p]$ of the present window;

2.1.2 oracle computes the present window content, by selecting the G_t graphs in $(o_p, c_p]$, i.e.

$$\{G_t \mid (G_t, \textit{:hasTimestamp}, t) \in G_m \wedge o_p < t \leq c_p\};$$

2.1.3 oracle evaluates the query. It executes a plain SPARQL query over the merge of the graphs determined by the previous step. The SPARQL query preserves the graph patterns and output modifiers of Q^R;

2.1.4 the answer of the query is a timestamped set of mappings, and it is added to the oracle output stream $Ans_oracle(t_i^0)$. The conversion depends on the *streaming operator definition contained in M^R;

2.2 when $Ans_{oracle}(t_i^0)$ computation ends, oracle compares it with $Ans_R(Q^R)$. If they match, the process ends and the oracle notifies the positive matching, i.e. $Ans_R(Q^R)$ is correct w.r.t. its operational semantics M^R, and for input S and query Q^R. Otherwise it restarts from Step 2;

3 if none of the generated answers

$$Ans_{oracle}(t_1^0), Ans_{oracle}(t_2^0), \ldots, Ans_{oracle}(t_k^0)$$

matches the output stream $Ans_R(Q^R)$, the oracle notifies that the answer of R is not correct.

The oracle we implemented manages queries with one time-based sliding window over a stream and static background data, it supports the whole SPARQL 1.1 query language and it implements the three R2S operators RStream, IStream and DStream. The oracle is configurable and it is possible to change both the input stream and the benchmark queries. In this way it can also be used by RSP developers to set up testing environments while implementing their systems.

We have implemented the oracle on the top of the Sesame framework. We made it available as an open source project[1]. The project repository supplies also all the resources required to repeat the experiments: the input stream (with different streamer implementations for the analysed systems), the queries, and the code to execute them in C-SPARQL, CQELS and SPARQL$_{stream}$.

[1]Cf. `https://github.com/dellaglio/csrbench-oracle`

9.3 Experiments with RSP engines

We have used CSRBench to test[2] the C-SPARQL engine (version 0.9), CQELS (Aug 2011) and SPARQL$_{stream}$ (version 1.0.5). The dataset used for experiments consists of a subset of the LSD dataset of SR-Bench that comprises weather observations from hurricanes in the US. Only the data from hurricane Charley has been used, for a total of 3 hours of records. Data is replayed with parametrized input rates. We defined 7 queries, by instantiating the parameters of the three types of queries defined in Section 9.2.2:

- *Q1.* Query latest temperature observations and its originating sensor, filtered by a threshold. $\omega = 10s$, $\beta = 10s$, tumbling window.
- *Q2.* Query latest temperature observations and its originating sensor, filtered by a threshold. $\omega = 1s$, $\beta = 1s$, tumbling window.
- *Q3.* Query latest relative humidity observations and its originating sensor, filtered by a threshold. $\omega = 4s$, $\beta = 4s$, tumbling window.
- *Q4.* Query latest average temperature value, filtered by a threshold. $\omega = 4s$, $\beta = 4s$, tumbling window.
- *Q5.* Query latest temperature observations and its originating sensor, filtered by a threshold. $\omega = 5s$, $\beta = 1s$, sliding window.
- *Q6.* Query latest sensors having observations with a variation of temperature values higher than a threshold. $\omega = 5s$, $\beta = 5s$, tumbling window.
- *Q7.* Query latest sensors having observations with higher temperature values than a fixed sensor station. $\omega = 5s$, $\beta = 5s$, tumbling window.

As shown in Table 9.1, none of the RDF stream engines successfully passes all the tests. This provides an idea of the difficulty of assessing correctness in this type of systems. We now describe the cases where there are failures.

Queries *Q1*, *Q2* and *Q3* focus on variations of the window size and slide, for the case of tumbling windows. All the systems behave in

[2] C-SPARQL: `http://streamreasoning.org/download`, CQELS: `http://code.google.com/p/cqels/`, SPARQL$_{stream}$: `https://github.com/jpcik/morph-streams`

Query	C-SPARQL	CQELS	SPARQL$_{stream}$
Q1	✓	✓	✓
Q2	✓	✓	✓
Q3	✓	✓	✓
Q4	✓	×	×
Q5	×	✓	✓
Q6	✓	×	✓
Q7	✓	×	✓

Table 9.1: *Correctness checking: experimental results for C-SPARQL, CQELS and SPARQL$_{stream}$.*

the correct way and provide the correct answers. These results are for the most part very similar in terms of content, as the graph pattern of the queries operates over (almost) contemporaneous triples. The main difference is on the timing of the output. The report policy of CQELS enables an almost immediate answer after a match is produced. This behaviour interestingly hides any difference on the use of a slide, and consequently the results for *Q1* and *Q2* are virtually identical in CQELS. Also, smaller windows such as the one in *Q2* forced to configure the time resolution of the processing engine, in the case of SPARQL$_{stream}$, which otherwise would be unable to slide at the given rate. In the case of SPARQL$_{stream}$, the results in these three queries with C-SPARQL is noticeable on the absence of any output when no matches are produced. This situation must not be confused with the absence of data in the input stream.

The other four queries exploited unexpected behaviours of the analyzed systems. C-SPARQL does not pass the experiment with *Q5*: this query highlights the use of an explicitly controlled slide, smaller than the window size. The problem is related to the fact that when a query is registered in C-SPARQL, there is a transitory phase on which some open windows are erroneously reported. When the system becomes stable and the first window closes, C-SPARQL starts to behave correctly and works as expected. This wrong behaviour is related to the sliding windows, in case of tumbling windows C-SPARQL works correctly.

SPARQL$_{stream}$ does not behave in the correct way with *Q4*. In general, this query poses challenges in several aspects. The first and most obvious is related to the t^0 parameter. Because the first window opens at different points in each system, the resulting average values are completely different. The oracle adequately handles these variations, by computing results for different possible values of t^0. Nevertheless, other issues arise on the way aggregates are implemented in the absence of matches in the graph patterns. In this particular case, SPARQL$_{stream}$ outputs a *null* value instead of a 0 average value. This unexpected behaviour is the cause of the failure in *Q4*, although in other cases the resulting values are correct. It is also worth mentioning that because SPARQL$_{stream}$ uses a underlying Data Stream Processing system through query rewriting, by changing it with another implementation, its modelled parameters (e.g. report, tick, etc.) could also change. Therefore its operational semantics depend on the underlying system it uses in a particular deployment.

Even CQELS does not provide the correct answer on *Q4* and, additionally, it shows wrong behaviours on *Q6* and *Q7*. Both *Q6* and *Q7* focus on the evaluation of joins in triples with different timestamps. In the first case the equality join is on the observation sensor, which is a URI in the dataset, while in the second it is basically a cross-product of a single fixed observation against all observations in the window. In this case, the problem is given by the fact that CQELS does not correctly remove the RDF statements from the active window. As a result, there are aggregations and joins on elements that should not be in the window anymore and the system produces additional wrong mappings. This behaviour does not emerge with other queries due to to the IStream operator: when queries filters the input stream, the answers are computed looking for triple patterns over data with the same application timestamp. Consequently, only answers obtained from new data entering the window are output, due to the fact that the data already present in the window produced answers that were output in previous steps.

9.4 Remarks

In this chapter, we made an effort to cover an existing gap in current benchmarks for RSP engines. Checking the correctness of streaming query results is complementary to other tests such as functional coverage, performance, scalability, but it is also key to assess how a system complies to its operational semantics. A comparison among this type of systems is not possible if we are unable to judge whether their output is correct or not. To do so, we exploited RSP-QL to formalise a notion of correctness, and we have shown experimentally that RSP engines do not always comply to their semantics. We have shown the cases where this happens through the CSRBench extensions and the oracle.

We aim to improve this work in several directions. We plan to improve the oracle output mechanism: at the moment the match between an oracle result and the system result $Ans_R(Q^R)$ provides a boolean answer: true if $Ans_R(Q^R)$ is contained (or it is equal) to one of the oracle and false otherwise. We plan to provide a more expressive matching mechanism, through the introduction of a matching percentage value, to help the analysis of results. Additionally, we plan to take into account also the verification of quality of service metrics, such as the fact that the systems may provide results with a maximum delay from the theoretical output time.

Additionally, we are interested in studying the behaviour of systems that are not captured by RSP-QL, such as CEP-enabled RSP engines and systems using interval-based timestamps in RDF streams. While RSEP-QL is a solution for the former, further extensions are required to cope with the latter point. It is important to understand the operational semantics of those systems and discover the similarities and the differences between them and the systems we targeted in this work.

CHAPTER 10

Conclusions

[...] we need a theory of semantic processing *of massive sets of complex and highly dynamic data. This will include the development of knowledge representation languages, performance metrics and a systematic roadmap about how to process massive, dynamic, ordered data.*

The sentence, stated by Della Valle, Schlobach, Krötzsch, Bozzon, Ceri and Horrocks in [46], sets one of the key points in the Stream Reasoning research agenda. This thesis develops along this direction, contributing in the definition of a theory for semantic stream processing.

We are not the first in proposing data and query models for semantic stream processing, but our novelty is given by the different approach we took with regards to other works in the area. We focused on filling the gap among existing solutions, on *unifying* the models proposed in state of the art. That means, we identified the common features of available models and solutions and, more important, the peculiarities that make them unique. As result, we built RSEP-QL, a model to describe the continuous query processing of semantic streams.

Such a model sets a milestone on the Stream Reasoning roadmap, as it offers a framework to compare, benchmark and let interoperate the current solutions. Moreover, RSEP-QL can contribute to the standardization of a language to process query streams, as the one under development at the RSP W3C Community Group[1].

This thesis proposes different contributions that we can group in three blocks: (i) *unified query model for semantic streams*, (ii) *unified reasoning over semantic streams* and (iii) *engineering stream reasoning processors*. We analyse them in the next section.

10.1 Review of the Contributions

The main contributions of this activity are consequences of the research questions that we introduced in Section 1.

Unified Query Model for Semantic Streams. The first question, **RQ.1**, asked: *How can the behaviour of existing RDF Stream Processing systems be captured and compared when reasoning processes are not involved?* Our investigation lead to the definition of two reference query models, RSP-QL and RSEP-QL.

RSP-QL, presented in Chapter 5, extends SPARQL by adding support for operators typical of the Data Stream Processing area, such as sliding windows and streaming operators, used respectively to consume and produce RDF streams. RSP-QL moves the evaluation semantics of the model from one time (as in SPARQL) to continuous. That means, RSP-QL produces sequences (*streams*) of answers, computed at different time instants, to cope with the fact that the data on the stream is dynamic and changes over time. RSP-QL captures the evaluation semantics of Data Stream Processing inspired RSP models, e.g. C-SPARQL, CQELS and SPARQL$_{stream}$, as shown in Section 9.1.

RSEP-QL, described in Chapter 6, extends RSP-QL by adding operators for event matching, with the goal of capturing the RSP engine operators inspired by Complex Event Processing solutions, as the ones of EP-SPARQL and the timestamp function of C-SPARQL. These operators enable the search of sequences of events joined through time constraints (e.g. join two events if the former happens before the latter). As result, RSEP-QL supplies in a unique model the operators used by RSP models inspired to either Data Stream Processing or Complex Event Processing, as discussed in Chapter 8.

[1]Cf. `www.w3.org/community/rsp/`.

Unified Reasoning over Semantic Streams. The second part of the thesis is lead by the Research Question **RQ.2**: *What is the correct behaviour of a continuous query engine while processing a semantic stream under entailment regimes?*

One of the main outcomes of RSEP-QL is to define a rigorous evaluation semantics, in the sense that, given a continuous query over a set of input streams and background data sets, it identifies one and only one correct answer.

This fact poses a solid basis to build a reasoning layer on RSEP-QL. In Chapter 7, we defined the entailment regimes in RSEP-QL, extending the SPARQL ones. The presence of streams in the dataset and event patterns in the query language required to expand the meaning of answer under entailment regimes as defined in SPARQL. We introduced three levels, to consider the entailment regime on the merge of the window content, the window or the whole stream. We shown that each level may produce a different answer, and we studied some properties among them.

Engineering Stream Reasoning Processors. As claimed above, the design of RSEP-QL enables further research. To support the claim, in this thesis we used RSEP-QL in the context of three problems related to realization and improvement of stream reasoning processors.

In Chapter 9, we applied RSP-QL in the context of benchmarking. It is at the basis of CSRBench, an extension of SRBench for correctness assessment. Given an RSP engine, RSP-QL is used to determine the set of expected correct answers may be produced. We enabled the possibility to automatically verifies if the current answer of the engine matches an expected answer, through the oracle, an open source framework we wrote.

In parallel, in Chapter 4 we presented Triple Wave, a framework to publish RDF streams on the Web. The work fills an important gap in in RDF Stream Orocessing, as it provides flexible mechanisms for plugging in diverse Web data sources, and for consuming the streams in both push and pull mode.

10.2 Future directions

In the previous chapters, we discussed possible extensions of the defined contributions. In this section, we discuss three future directions we envision for this work.

10.2.1 Towards an RSEP-QL Engine

RSEP-QL provides a well-defined semantics to process RDF streams. A natural direction on which this work can be extended is an implementation. As we discussed above, we developed components relative to the algorithms we developed, but a complete proof of concept engine that shows the feasibility of a comprehensive RSEP-QL engine is missing.

The road to an RSEP-QL engine is not just related to implementation: there are several problems we did not tackle. A prototype would support the investigation of query optimization of RSEP-QL queries including the MATCH clause. As explained in Chapter 6, we described the naïve strategy where the MATCH clause is evaluated at the end, in order to compute the correct event mappings. This solution can generate a high number of intermediate mappings. It is possible to study algebraic equivalences to move operators in the MATCH clause, and prove (theoretically and experimentally) the advantages. Additional problems we did not consider are out-of order arrival of data items, arbitrary delays in query operators and other issues that may arrive in real-life systems when overloaded.

We believe that a convenient way to implement the engine is to extend a Semantic Web-related framework, as Jena or Sesame, rather than starting a Data Stream Processing or an Complex Event Processing system. The main reason is that the former provides a native support to RDF, SPARQL and reasoners, as well as APIs to manage them. Extending such a framework allows a full control over the streaming data model and the RSEP-QL operators.

10.2.2 RDF Streams: Models and Publishing

In the future, we envision a widespread adoption of RSP-based solutions in different domains, specially under the umbrella of the Internet of Things (IoT). RSEP-QL helps providing the foundations for well-defined query processors that can interoperate through common query interfaces, even if they follow different architectural approaches. This allows creating an ecosystem of RSEP-QL compatible systems that take advantage of semantics and Linked Data to interact.

An open direction that deserves attention is related to the data model. In this work, we assumed a data model where streams are composed by sequences of RDF graphs annotated with single application time instants. As we briefly discussed in Chapter **??**, this model can be enriched with further information, as the time annotation can have different semantics (e.g. instantaneous validity or beginning of the validity interval), also in relation with the stream item (e.g. states and actions). Moreover, this data model admits alternatives, as streams where there are two or more time instants associated to stream items, each of them with different meanings.

While a definition of a vocabulary to specify the semantics of the time annotation and of the stream items can solve the problem at the data level, it creates several problems at query level. In fact, the query processor should process the vocabulary and consequently take different actions on the data it receives. This problem affects both the query answering and reasoning layers.

Moreover, we did not tackle the problem of blank nodes in the stream: is there any relation two blank nodes with the same label appearing in two different stream items? In RDF, a blank node is defined in the context of a RDF graph, so the same label on two graphs identifies two different unnamed resources. However, in the context of an RDF stream, there are cases where is needed to describe the same (unnamed) resource at different time instants.

Finally, Triple Wave sets a milestone in making available RDF streams on the Web, but additional work is required. Even if formats like JSON-LD can be adapt to serialize the RDF stream, the provider should publish not only the stream, but also the vocabulary and, if available, the graph describing the schema describing the streaming data (i.e. the Stream TBox Mapper value as defined in Chapter 7). Next, a protocol to let the continuous query processor to access this information should be defined. It is worth noting this protocol should also manage the fact that the schema can change over time.

Bibliography

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, 2008.

[2] D. Allemang and J. Hendler. *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123735564, 9780123735560.

[3] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Commun. ACM*, 26 (11):832–843, 1983.

[4] D. Anicic. *Event Processing and Stream Reasoning with ETALIS*. PhD thesis, Karlsruhe Institute of Technology, 2011.

[5] D. Anicic and P. Fodor. EP-SPARQL : A Unified Language for Event Processing and Stream Reasoning. pages 635–644, 2011.

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

[7] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic. Stream reasoning and complex event processing in ETALIS. *Semantic Web*, 3(4):397–407, 2012.

[8] 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 M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, editors, *VLDB*, pages 480–491. Morgan Kaufmann, 2004. ISBN 0-12-088469-0.

[9] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006. doi: 10.1007/s00778-004-0147-z.

[10] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[11] F. Baader, S. Brandt, and C. Lutz. Pushing the el envelope. In *IJCAI*, pages 364–369, 2005.

[12] F. Baader, S. Brandt, and C. Lutz. Pushing the EL envelope. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 364–369, Edinburgh, Scotland, 2005. Professional Book Center. URL `http://ijcai.org/papers/0372.pdf`.

[13] F. Baader, C. Lutz, and B. Suntisrivaraporn. CEL - A polynomial-time reasoner for life science ontologies. In *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 287–291. Springer, 2006.

[14] F. Baader, S. Brandt, and C. Lutz. Pushing theEL envelope further. In *OWL: Experiences and Directions (OWLED)*, pages 1–10, 2008. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.158.9530`.

[15] B. Babcock, S. Babu, and M. Datar. Models and issues in data stream systems. *... of database systems*, pages 1–16, 2002. URL `http://dl.acm.org/citation.cfm?id=543615`.

[16] S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3):109–120, 2001.

[17] M. Balduini, E. Della Valle, D. Dell’Aglio, M. Tsytsarau, T. Palpanas, and C. Confalonieri. Twindex fuorisalone: Social listening of milano during fuorisalone 2013. In P. Cimiano, M. Fernández, V. Lopez, S. Schlobach, and J. Völker, editors, *The Semantic Web: ESWC 2013 Satellite Events - ESWC 2013 Satellite Events, Montpellier, France, May 26-30, 2013, Revised Selected Papers*, volume 7955 of *Lecture Notes in Computer Science*, pages 327–336. Springer, 2013. doi: 10.1007/978-3-642-41242-4_59. URL `http://dx.doi.org/10.1007/978-3-642-41242-4_59`.

[18] M. Balduini, E. Della Valle, D. Dell’Aglio, M. Tsytsarau, T. Palpanas, and C. Confalonieri. Social listening of city scale events using the streaming linked data framework. In H. Alani, L. Kagal, A. Fokoue, P. T. Groth, C. Biemann, J. X. Parreira, L. Aroyo, N. F. Noy, C. Welty, and K. Janowicz, editors, *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II*, volume 8219 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2013. doi: 10.1007/978-3-642-41338-4_1. URL `http://dx.doi.org/10.1007/978-3-642-41338-4_1`.

[19] M. Balduini, I. Celino, D. Dell’Aglio, E. Della Valle, Y. Huang, T. K. Lee, S. Kim, and V. Tresp. Reality mining on micropost streams - deductive and inductive reasoning for personalized and location-based recommendations. *Semantic Web*, 5 (5):341–356, 2014. doi: 10.3233/SW-130107. URL `http://dx.doi.org/10.3233/SW-130107`.

[20] D. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-sparql: a continuous query language for rdf data streams. *International Journal of Semantic Computing*, 4(3), 2010. URL `http://www.worldscientific.com/doi/abs/10.1142/S1793351X10000936`.

[21] D. F. Barbieri and E. Della Valle. A proposal for publishing data streams as linked data - A position paper. In C. Bizer, T. Heath, T. Berners-Lee, and M. Hausenblas, editors, *Proceedings of the WWW2010 Workshop on Linked Data on the*

Web, LDOW 2010, Raleigh, USA, April 27, 2010, volume 628 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010. URL `http://ceur-ws.org/Vol-628/ldow2010_paper11.pdf`.

[22] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. Querying RDF streams with C-SPARQL. *SIGMOD Record*, 39(1):20–26, 2010.

[23] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, Y. Huang, V. Tresp, A. Rettinger, and H. Wermser. Deductive and inductive stream reasoning for semantic social media analytics. *IEEE Intelligent Systems*, 25(6):32–41, 2010.

[24] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *CIDR*, pages 363–374, 2007.

[25] H. Beck, M. Dao-Tran, T. Eiter, and M. Fink. LARS: A logic-based framework for analyzing reasoning over streams. In *AAAI*, pages 1431–1438. AAAI Press, 2015.

[26] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[27] T. Berners-Lee, C. Bizer, and T. Heath. Linked data-the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[28] A. Bolles, M. Grawunder, and J. Jacobi. Streaming SPARQL - Extending SPARQL to Process Data Streams. In S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis, editors, *ESWC*, volume 5021 of *Lecture Notes in Computer Science*, pages 448–462. Springer, 2008. ISBN 978-3-540-68233-2.

[29] I. Botan, R. Derakhshan, and N. Dindar. Secret: A model for analysis of the execution semantics of stream processing systems. *PVLDB*, 3(1):232–243, 2010. URL `http://dl.acm.org/citation.cfm?id=1920874`.

[30] J. G. Breslin, S. Decker, A. Harth, and U. Bojars. SIOC: an approach to connect web-based communities. *IJWBC*, 2(2):133–142, 2006.

[31] J.-P. Calbimonte, O. Corcho, and A. J. G. Gray. Enabling ontology-based access to streaming data sources. In *ISWC*, pages 96–111, 2010. ISBN 364217745X.

[32] J.-P. Calbimonte, H. Jeung, Ó. Corcho, and K. Aberer. Enabling Query Technologies for the Semantic Sensor Web. *Int. J. Semantic Web Inf. Syst.*, 8(1):43–63, 2012.

[33] G. Carothers, A. Seaborne, C. Bizer, and R. Cyganiak. RDF 1.1 TriG. W3C recommendation, W3C, Feb. 2014.

[34] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 577–589, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc. ISBN 1-55860-150-3. URL `http://dl.acm.org/citation.cfm?id=645917.672169`.

[35] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[36] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 379–390. ACM, 2000. ISBN 1-58113-218-2.

[37] M. Compton, P. M. Barnaghi, L. Bermudez, R. Garcia-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. A. Henson, A. Herzog, V. A. Huang, K. Janowicz, W. D. Kelsey, D. L. Phuoc, L. Lefort, M. Leggieri, H. Neuhaus, A. Nikolov, K. R. Page, A. Passant, A. P. Sheth, and K. Taylor. The SSN ontology of the W3C semantic sensor network incubator group. *J. Web Sem.*, 17:25–32, 2012.

[38] G. Cugola and A. Margara. TESLA: a formally defined event specification language. In *DEBS*, pages 50–61. ACM, 2010.

[39] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15, 2012.

[40] R. Cyganiak, D. Wood, and M. Lanthaler. RDF 1.1 Concepts and Abstract Syntax. http://www.w3.org/TR/rdf11-concepts/, 2014. URL `http://www.w3.org/TR/rdf11-concepts/`.

[41] M. Dao-Tran and D. Le-Phuoc. Towards Enriching CQELS with Complex Event Processing and Path Navigation. In *HiDeSt*, pages 2–14, 2015.

[42] S. Dehghanzadeh, D. Dell'Aglio, S. Gao, E. Della Valle, A. Mileo, and A. Bernstein. Approximate continuous query answering over streams and dynamic linked data sets. In P. Cimiano, F. Frasincar, G. Houben, and D. Schwabe, editors, *Engineering the Web in the Big Data Era - 15th International Conference, ICWE 2015, Rotterdam, The Netherlands, June 23-26, 2015, Proceedings*, volume 9114 of *Lecture Notes in Computer Science*, pages 307–325. Springer, 2015. doi: 10.1007/978-3-319-19890-3_20. URL `http://dx.doi.org/10.1007/978-3-319-19890-3_20`.

[43] S. Dehghanzadeh, D. Dell'Aglio, S. Gao, E. Della Valle, A. Mileo, and A. Bernstein. Online view maintenance for continuous query evaluation. In A. Gangemi, S. Leonardi, and A. Panconesi, editors, *Proceedings of the 24th International Conference on World Wide Web Companion, WWW 2015, Florence, Italy, May 18-22, 2015 - Companion Volume*, pages 25–26. ACM, 2015. doi: 10.1145/2740908.2742761. URL `http://doi.acm.org/10.1145/2740908.2742761`.

[44] E. Della Valle, S. Ceri, D. F. Barbieri, D. Braga, and A. Campi. A first step towards stream reasoning. In *FIS*, volume 5468 of *Lecture Notes in Computer Science*, pages 72–81. Springer, 2008.

[45] E. Della Valle, S. Ceri, F. Van Harmelen, and D. Fensel. It's a Streaming World! Reasoning upon Rapidly Changing Information, 2009. ISSN 15411672. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5372206`.

[46] E. Della Valle, S. Schlobach, M. Krötzsch, A. Bozzon, S. Ceri, and I. Horrocks. Order matters! harnessing a world of orderings for reasoning over massive data. *Semantic Web*, 4(2):219–231, 2013.

[47] D. Dell'Aglio. Ontology-based top-k query answering over massive, heterogeneous, and dynamic data. In L. Aroyo and N. F. Noy, editors, *Proceedings of the Doctoral Consortium co-located with 12th International Semantic Web Conference (ISWC 2013), Sydney, Australia, October 20, 2013.*, volume 1045 of *CEUR Workshop Proceedings*, pages 17–24. CEUR-WS.org, 2013. URL `http://ceur-ws.org/Vol-1045/paper-03.pdf`.

[48] D. Dell'Aglio and E. Della Valle. Incremental reasoning on RDF streams. In A. Harth, K. Hose, and R. Schenkel, editors, *Linked Data Management.*, pages 413–435. Chapman and Hall/CRC, 2014. URL `http://www.crcnetbase.com/doi/abs/10.1201/b16859-22`.

[49] D. Dell'Aglio, J. Calbimonte, M. Balduini, Ó. Corcho, and E. Della Valle. On correctness in RDF stream processor benchmarking. In H. Alani, L. Kagal, A. Fokoue, P. T. Groth, C. Biemann, J. X. Parreira, L. Aroyo, N. F. Noy, C. Welty, and K. Janowicz, editors, *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II*, volume 8219 of *Lecture Notes in Computer Science*, pages 326–342. Springer, 2013. doi: 10.1007/978-3-642-41338-4_21. URL `http://dx.doi.org/10.1007/978-3-642-41338-4_21`.

[50] D. Dell'Aglio, E. Della Valle, J. Calbimonte, and Ó. Corcho. RSP-QL semantics: A unifying query model to explain heterogeneity of RDF stream processing systems. *Int. J. Semantic Web Inf. Syst.*, 10(4):17–44, 2014. doi: 10.4018/ijswis.2014100102. URL `http://dx.doi.org/10.4018/ijswis.2014100102`.

[51] D. Dell'Aglio, M. Balduini, and E. Della Valle. Applying Semantic Interoperability Principles to Data Stream Management. In A. Harth, K. Hose, and R. Schenkel, editors, *Data Management in Pervasive Systems.*, pages 135–166. Springer, 2015.

[52] D. Dell'Aglio, J. Calbimonte, E. Della Valle, and Ó. Corcho. Towards a unified language for RDF stream query processing. In F. Gandon, C. Guéret, S. Villata, J. G. Breslin, C. Faron-Zucker, and A. Zimmermann, editors, *The Semantic Web: ESWC 2015 Satellite Events - ESWC 2015 Satellite Events Portorož, Slovenia, May 31 - June 4, 2015, Revised Selected Papers*, volume 9341 of *Lecture Notes in Computer Science*, pages 353–363. Springer, 2015. doi: 10.1007/978-3-319-25639-9_48. URL `http://dx.doi.org/10.1007/978-3-319-25639-9_48`.

[53] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12):1495–1539, 2008.

[54] M. Fitting. *First-order logic and automated theorem proving.* Springer Science & Business Media, 2012.

[55] C. Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982.

[56] M. Gebser, O. Sabuncu, and T. Schaub. An incremental answer set programming based system for finite model computation. *AI Commun.*, 24(2):195–212, 2011.

[57] M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, and T. Schaub. Stream reasoning with answer set programming: Preliminary report. In *KR*. AAAI Press, 2012.

[58] B. Glimm and M. Krötzsch. SPARQL beyond subgraph matching. In *International Semantic Web Conference (1)*, volume 6496 of *Lecture Notes in Computer Science*, pages 241–256. Springer, 2010.

[59] B. Glimm, C. Ogbuji, S. Hawke, I. Herman, B. Parsia, A. Polleres, and A. Seaborne. SPARQL 1.1 Entailment Regimes. W3C recommendation, W3C, June 2013.

[60] W. O. W. L. W. Group. OWL 2 Web Ontology Language Document Overview (Second Edition). W3c recommendation, W3C, 2012.

[61] C. Gutiérrez, C. A. Hurtado, and A. A. Vaisman. Temporal RDF. In A. Gómez-Pérez and J. Euzenat, editors, *ESWC*, volume 3532 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005. ISBN 3-540-26124-9.

[62] C. Gutierrez, C. A. Hurtado, and A. Vaisman. Introducing time into RDF. *IEEE Transactions on Knowledge and Data Engineering*, 19(2):207–218, 2007.

[63] S. Harris and A. Seaborne. SPARQL 1.1 Query Language. http://www.w3.org/TR/sparql11-query/, 2013.

[64] P. Hayes and P. Patel-Schneider. RDF 1.1 TriG. W3C recommendation, W3C, Feb. 2014.

[65] Z. Huang and H. Stuckenschmidt. Reasoning with multi-version ontologies: A temporal logic approach. In *International Semantic Web Conference*, volume 3729 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.

[66] Z. Huang and H. Stuckenschmidt. Reasoning with multi-version ontologies: A temporal logic approach. In *ISWC*, pages 398–412, 2005.

[67] Y. Kazakov, M. Krötzsch, and F. Simancik. The incredible ELK - from polynomial procedures to efficient reasoning with el ontologies. *J. Autom. Reasoning*, 53(1): 1–61, 2014.

[68] E. Kharlamov, S. Brands, M. Giese, E. Jimenez-Ruiz, S. Lamparter, C. Neuenstadt, Ö. L. Özçep, C. Pinkel, A. Soylu, D. Zheleznyakov, M. Roshchin, S. Watson, and I. Horrocks. Semantic Access to Siemens Streaming Data: the OPTIQUE Way. In *Proc. of International Semantic Web Conference ISWC, Posters & Demonstrations Track*, 2015.

[69] S. Komazec, D. Cerri, and D. Fensel. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In F. Bry, A. Paschke, P. T. Eugster, C. Fetzer, and A. Behrend, editors, *DEBS*, pages 58–68. ACM, 2012. ISBN 978-1-4503-1315-5.

[70] M. Krötzsch and S. Rudolph. Conjunctive Queries for EL with Composition of Roles. In D. Calvanese, E. Franconi, V. Haarslev, D. Lembo, B. Motik, A.-Y. Turhan, and S. Tessaris, editors, *Description Logics*, volume 250 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[71] M. Krötzsch, S. Rudolph, and P. Hitzler. Conjunctive Queries for a Tractable Fragment of OWL 1.1. In *ISWC/ASWC*, pages 310–323, 2007.

[72] W. Kuo, W.-T. K. Chien, and T. Kim. *Reliability, Yield, and Stress Burn-in: A Unified Approach for Microelectronics Systems Manufacturing & Software Development.* Springer, 1998.

[73] D. Le-Phuoc, M. Dao-Tran, J. Xavier Parreira, and M. Hauswirth. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. F. Noy, and E. Blomqvist, editors, *ISWC*, volume 7031 of *Lecture Notes in Computer Science*, pages 370–388. Springer, 2011. ISBN 978-3-642-25072-9.

[74] F. Lécué. Diagnosing changes in an ontology stream: A DL reasoning approach. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.*, 2012. URL `http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/4988`.

[75] F. Lécué. Scalable maintenance of knowledge discovery in an ontology stream. In *IJCAI*, pages 1457–1463. AAAI Press, 2015.

[76] D. C. Luckham. *The power of events - an introduction to complex event processing in distributed enterprise systems.* ACM, 2005. ISBN 978-0-201-72789-0.

[77] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995. ISSN 0098-5589. doi: 10.1109/32.464548.

[78] A. Margara, J. Urbani, F. van Harmelen, and H. E. Bal. Streaming the web: Reasoning over dynamic data. *J. Web Sem.*, 25:24–44, 2014.

[79] A. Mauri, J. Calbimonte, D. Dell'Aglio, M. Balduini, E. Della Valle, and K. Aberer. Where are the RDF streams?: On deploying RDF streams on the web of data with triplewave. In S. Villata, J. Z. Pan, and M. Dragoni, editors, *Proceedings of the ISWC 2015 Posters & Demonstrations Track co-located with the 14th International Semantic Web Conference (ISWC-2015), Bethlehem, PA, USA, October 11, 2015.*, volume 1486 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015. URL `http://ceur-ws.org/Vol-1486/paper_95.pdf`.

[80] Y. Mei and S. Madden. ZStream: a cost-based query processor for adaptively detecting composite events. In *Proc. ACM SIGMOD International Conference on Management of data*, pages 193–206. ACM, 2009.

[81] A. Mileo, A. Abdelrahman, S. Policarpio, and M. Hauswirth. Streamrule: A non-monotonic stream reasoning system for the semantic web. In *RR*, volume 7994 of *Lecture Notes in Computer Science*, pages 247–252. Springer, 2013.

[82] B. Motik. Representing and querying validity time in RDF and OWL: A logic-based approach. *J. Web Sem.*, 12:3–21, 2012.

[83] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz, et al. OWL 2 web ontology language: Profiles (Second Edition). W3C recommendation, 2012.

[84] S. Muñoz, J. Pérez, and C. Gutierrez. Simple and efficient minimal rdfs. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):220–234, 2009.

[85] Ö. L. Özçep and R. Möller. Ontology based data access on temporal and streaming data. In *Reasoning Web*, volume 8714 of *Lecture Notes in Computer Science*, pages 279–312. Springer, 2014.

[86] Ö. L. Özçep, R. Möller, and C. Neuenstadt. A stream-temporal query language for ontology based data access. In *Description Logics*, volume 1193 of *CEUR Workshop Proceedings*, pages 696–708. CEUR-WS.org, 2014.

[87] P. Patel-Schneider and B. Motik. OWL 2 web ontology language. mapping to RDF graphs (second edition). W3C recommendation, W3C, Dec. 2012. URL `http://www.w3.org/TR/owl2-mapping-to-rdf/`.

[88] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.

[89] 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 *International Semantic Web Conference (2)*, volume 7650 of *Lecture Notes in Computer Science*, pages 300–312. Springer, 2012.

[90] E. Prud'Hommeaux, A. Seaborne, and Others. SPARQL query language for RDF. *W3C recommendation*, 2008.

[91] A. Pugliese, O. Udrea, and V. S. Subrahmanian. Scaling RDF with time. In J. Huai, R. Chen, H.-W. Hon, Y. Liu, W.-Y. Ma, A. Tomkins, and X. Zhang, editors, *WWW*, pages 605–614. ACM, 2008. ISBN 978-1-60558-085-2.

[92] Y. Ren and J. Z. Pan. Optimising ontology stream reasoning with truth maintenance system. In *CIKM*, pages 831–836. ACM, 2011.

[93] Y. Ren and J. Z. Pan. Optimising ontology stream reasoning with truth maintenance system. In *CIKM*, pages 831–836, 2011.

[94] M. Rinne, E. Nuutila, and S. Törmä. INSTANS: High-Performance Event Processing with Standard RDF and SPARQL. In B. Glimm and D. Huynh, editors, *International Semantic Web Conference (Posters {&} Demos)*, volume 914 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.

[95] R. Rosati. On Conjunctive Query Answering in EL. In D. Calvanese, E. Franconi, V. Haarslev, D. Lembo, B. Motik, A.-Y. Turhan, and S. Tessaris, editors, *Description Logics*, volume 250 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[96] F. Schreiber. Is Time a Real Time? An Overview of Time Ontology in Informatics. In W. Halang and A. Stoyenko, editors, *Real Time Computing*, volume 127 of *NATO ASI Series*, pages 283–307. Springer Berlin Heidelberg, 1994. ISBN 978-3-642-88051-3. doi: 10.1007/978-3-642-88049-0{_}14. URL `http://dx.doi.org/10.1007/978-3-642-88049-0{_}14`.

[97] M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 75–86, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc. ISBN 1-55860-382-4. URL `http://dl.acm.org/citation.cfm?id=645922.673479`.

[98] H. Stuckenschmidt, S. Ceri, E. Della Valle, and F. van Harmelen. Towards expressive stream reasoning. In K. Aberer, A. Gal, M. Hauswirth, K.-U. Sattler, and A. P. Sheth, editors, *Semantic Challenges in Sensor Networks*, number 10042

in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. URL `http://drops.dagstuhl.de/opus/volltexte/2010/2555`.

[99] S. Tallevi-Diotallevi, S. Kotoulas, L. Foschini, F. Lécué, and A. Corradi. Real-time urban monitoring in dublin using semantic and stream technologies. In *International Semantic Web Conference (2)*, volume 8219 of *Lecture Notes in Computer Science*, pages 178–194. Springer, 2013.

[100] J. Urbani, A. Margara, C. J. H. Jacobs, F. van Harmelen, and H. E. Bal. DynamiTE: Parallel Materialization of Dynamic RDF Data. In H. Alani, L. Kagal, A. Fokoue, P. T. Groth, C. Biemann, J. X. Parreira, L. Aroyo, N. F. Noy, C. Welty, and K. Janowicz, editors, *International Semantic Web Conference (1)*, volume 8218 of *Lecture Notes in Computer Science*, pages 657–672. Springer, 2013. ISBN 978-3-642-41334-6.

[101] R. Volz, S. Staab, and B. Motik. Incrementally maintaining materializations of ontologies stored in logic databases. *J. Data Semantics*, 2:1–34, 2005.

[102] O. Walavalkar, A. Joshi, T. Finin, and Y. Yesha. Streaming Knowledge Bases. In *Proceedings of the Fourth International Workshop on Scalable Semantic Web knowledge Base Systems*, October 2008.

[103] K. Walzer, M. Groch, and T. Breddin. Time to the rescue - supporting temporal reasoning in the rete algorithm for complex event processing. In *DEXA*, pages 635–642, 2008.

[104] Y. Zhang, M.-D. Pham, Ó. Corcho, and J.-P. Calbimonte. SRBench: A Streaming RDF/SPARQL Benchmark. In P. Cudré-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J. X. Parreira, J. Hendler, G. Schreiber, A. Bernstein, and E. Blomqvist, editors, *International Semantic Web Conference (1)*, volume 7649 of *Lecture Notes in Computer Science*, pages 641–657. Springer, 2012. ISBN 978-3-642-35175-4.