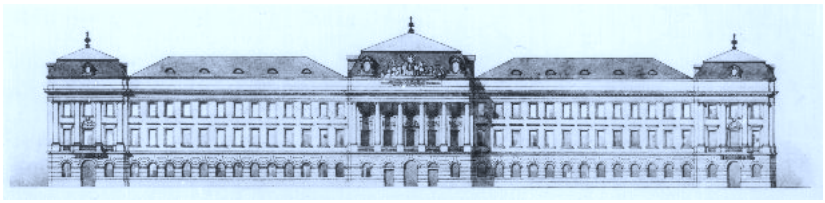
INFSYS
RESEARCH
REPORT



INSTITUT FÜR INFORMATIONSSYSTEME

ARBEITSBEREICH WISSENSBASIERTE SYSTEME

A QUERY MODEL TO CAPTURE EVENT PATTERN MATCHING IN RDF STREAM PROCESSING QUERY LANGUAGES

DANIELE DELL'AGLIO MINH DAO-TRAN JEAN-PAUL CALBIMONTE
DANH LE PHUOC EMANUELE DELLA VALLE

INFSYS RESEARCH REPORT 16-03

OCTOBER 2016

Institut für Informationssysteme
AB Wissensbasierte Systeme
Technische Universität Wien
Favoritenstraße 9-11
A-1040 Wien, Austria
Tel: +43-1-58801-18405
Fax: +43-1-58801-18493
sek@kr.tuwien.ac.at
www.kr.tuwien.ac.at

INFSYS RESEARCH REPORT
INFSYS RESEARCH REPORT 16-03, OCTOBER 2016

A QUERY MODEL TO CAPTURE EVENT PATTERN MATCHING IN RDF STREAM PROCESSING QUERY LANGUAGES

Daniele Dell'Aglio [1] [2] Minh Dao-Tran[3] Jean-Paul Calbimonte[4]
Danh Le Phuoc[5] Emanuele Della Valle[2]

Abstract. The current state of the art in RDF Stream Processing (RSP) proposes several models and implementations to combine Semantic Web technologies with Data Stream Management System (DSMS) operators like windows. Meanwhile, only a few solutions combine Semantic Web and Complex Event Processing (CEP), which includes relevant features, such as identifying sequences of events in streams. Current RSP query languages that support CEP features have several limitations: EP-SPARQL can identify sequences, but its selection and consumption policies are not all formally defined, while C-SPARQL offers only a naive support to pattern detection through a timestamp function. In this work, we introduce an RSP query language, called RSEP-QL, which supports both DSMS and CEP operators, with a special interest in formalizing CEP selection and consumption policies. We show that RSEP-QL captures EP-SPARQL and C-SPARQL, and offers features going beyond the ones provided by current RSP query languages.

[1]Department of Informatics, University of Zurich, Zurich, Switzerland; email: dellaglio@ifi.uzh.ch
[2]Dipartimento di Elettronica, Informatica e Bioingegneria, Politecnico of Milano, Milano, Italy; email: {daniele.dellaglio,emanuele.dellavalle}@polimi.it
[3]Institut für Informationssysteme, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Vienna, Austria; email: dao@kr.tuwien.ac.at.
[4]Institute of Information Systems, HES-SO Valais-Wallis and LSIR, EPFL, Switzerland; email: jean-paul.calbimonte@epfl.ch
[5]Technical University of Berlin, Berlin, Germany; email: danh.lephuoc@tu-berlin.de

Acknowledgements: This research has been supported by the Austrian Science Fund (FWF) project P26471, the Nano-tera.ch D1NAMO project, and the Marie Sklodowska-Curie Programme H2020-MSCA-IF-2014 under Grant No. 661180.

Copyright © 2016 by the authors

1 Introduction

Processing heterogeneous and dynamic data is a challenging research topic and has a wide range of applications in real-world scenarios. Different models, languages, and systems have been proposed in the last years to handle streams on the Web, combining Semantic Web technologies with Complex Event Processing (CEP) [18] and Data Stream Management Systems (DSMS) [5] features. These languages and systems, commonly labeled under the RDF Stream Processing (RSP) name, are solutions that extend SPARQL with stream processing features, based on either the CEP or DSMS paradigm.

A problem that recently emerged is the heterogeneity of those solutions [13, 11]. Every RSP engine has unique features that are not replicable by others; moreover, even when the same feature is supported by two or more engines, the behavior and the produced output can be different and hardly comparable. In our previous work, namely RSP-QL [14] and LARS [7], we developed models to capture the RSP features inspired by the DSMS paradigm, e.g., time-based sliding windows and aggregations over streams.

In this paper, we study the integration of the currently available CEP features in RSP engines into RSP-QL, by investigating the research question: *"Is it possible to extend RSP-QL to enable the detection of expressive event patterns over RDF streams?"* We give an answer with RSEP-QL, an RSP query model that incorporates CEP at its core.

RSEP-QL is a reference model[1] and has several possible uses: (a) to provide a common framework to explain the behavior of existing RSP solutions, enabling their comparison; (b) to support software architects to design new RSP implementations; testers in designing benchmarks and evaluations; and researchers to have a general model to develop new research; (c) to act as a formal model to define a standardized language that embraces the most prominent features of existing RSP languages.

Combining CEP and DSMS features in a unique model is a step towards filling the gap between RSP and stream processing engines available on the non-semantically-aware systems on the market (e.g., Oracle Event Processor, ESPER, IBM InfoSphere Streams) [10]. There are indeed several motivations behind combining DSMS and CEP. It is clearly possible to mix different DSMS and CEP languages to achieve the desired tasks, but there are drawbacks, e.g., the need to learn multiple languages, the limited possibility for query optimizations, the potential higher amount of resources.

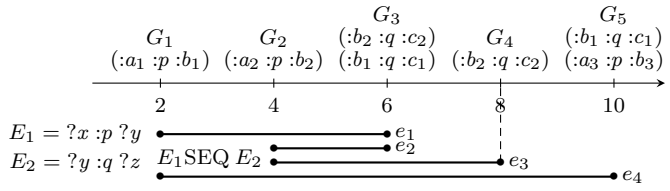
Our contributions are: (1) We elicit a set of requirements to design an RSP query model that supports both DSMS and CEP features. (2) We adapt our model to process RDF graphs as stream elements, following the current guidelines of the W3C RSP Community Group (RSP-CG).[2] (3) We introduce event patterns to capture CEP features of existing RSP engines, most notably the sequencing operator, and provide syntax and semantics as extensions of SPARQL. (4) We formally define selection and consumption policies, to capture the operational semantics of the CEP-inspired RSP engines, contrary to current approaches that consider policies at the implementation level.

2 Related Work and Requirements

RSP engines emerged in recent years, with the goal of extending RDF and SPARQL to process RDF streams. They can be broadly divided into two groups. RSPs influenced by CEP reactively process the input streams to identify relevant events and sequences of them. EP-SPARQL [3] is one of the first RSP that adopts some of these complex pattern operators. Other such recent approaches include Sparkwave [17] and Instans [20].

[1] Cf. https://www.oasis-open.org/committees/soa-rm/faq.php.

[2] Cf. https://www.w3.org/community/rsp/



E_1 SEQ E_2	$t = 8$	10
unrestricted	e_1 e_2 e_3	e_4
chronological	e_1 e_2	$\emptyset$
recent	e_2 e_3	$\emptyset$
naive	e_3	$\emptyset$

Figure 1: Illustration of the running example. The stream, on the top left, composes of five items $(G_1, 2) \ldots (G_5, 10)$. Events matched the pattern E_1 SEQ E_2 are depicted below the timeline. The bold lines denote the intervals that justify the events. The table on the right shows the results produced with regards to different policies.

On the other hand, approaches inspired by DSMS exploit sliding window mechanisms to capture a recent and finite portion of the input data, enabling their processing through SPARQL operators [15] in an atemporal fashion. C-SPARQL [6], CQELS [19], and SPARQL$_{stream}$ [9] are representative examples of this group.

Currently, there is so far no RSP language that can combine both paradigms under a clearly defined semantics, leaving a gap for those use cases that require this query expressivity. However, some initial attempts exist. In C-SPARQL, one can access the timestamp of a statement and specify limited forms of temporal conditions. CQELS recently proposed to integrate sequencing and path navigation [12], although it does not include typical selection mechanisms of CEP [10]. In the following, we present a set of requirements to lead the design of RSEP-QL, based on an analysis of the state of the art in RSP, with a particular focus on the CEP features of EP-SPARQL, and C-SPARQL.

[R1] RSEP-QL should process RDF graph-based streams. While in early RSP data models the stream data items are represented by single RDF statements, the recent standardization effort from W3C RSP-CG proposes to adopt RDF graphs as items[1]. The latter model generalizes of the former, as a stream of time-annotated RDF statements can be modeled as a stream of time-annotated RDF graphs, each containing one statement. In this sense, addressing [R1] is important to realize a generic RDF stream query model.

[R2] RSEP-QL must preserve the DSMS features captured by RSP-QL. The introduction of CEP features in the model should not lead to incompatibilities with the RSP models we already captured in RSP-QL [14]. This requirement is important to guarantee that RSEP-QL is generic enough to model the operational semantics of different systems.

[R3] RSEP-QL should capture the CEP features of existing RSP engines. In this work, we focus on the SEQ operator: the most basic building block in CEP. Intuitively, E_1 SEQ E_2 identifies events matching pattern E_1 followed by those matching E_2. Even if it may seem straightforward to formalize this operator, its execution in different engines produces different and hardly comparable results. We, therefore, refine [R3] into two sub-requirements, associated with the two engines we aim at capturing, EP-SPARQL and C-SPARQL. To illustrate our idea, we use the RDF stream depicted in Figure 1.

[R3.1] RSEP-QL should capture the EP-SPARQL SEQ behavior. To the best of our knowledge, EP-SPARQL is the RSP language with the 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 [2]:

- *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.

[1] Cf. `http://goo.gl/pqUSri` (last access: July 7, 2016).

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

The table of Figure 1 shows the different behaviors of these three settings. Assume that there are two evaluations at time points 8 and 10. *Unrestricted* returns e_1, e_2, e_3 at 8 and e_4 at 10. *Chronological* returns only e_1 and e_2 at 8. *Recent* returns only e_2 and e_3 at 8. Furthermore, both *chronological* and *recent* do not return any event at 10 because $(:a_1\ :p\ :b_1)$ were already consumed by the previous evaluation.

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.

[R3.2] RSEP-QL should capture the C-SPARQL SEQ behavior. C-SPARQL is based on DSMS techniques, but it has a naive support to some CEP features. C-SPARQL implements a function, named $timestamp$ that 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. Wrt. the above example, with a sliding window with a length of 7 and that slides of 1 at each step, C-SPARQL outputs e_3 at time 8 and has no output at 10, not because the input triples were consumed, but because it considers only the two triples $(:b_1\ :q\ :c_1)$ and $(:a_3\ :p\ :b_3)$ which do not match the sequencing pattern.

Remarks. While EP-SPARQL is an engine for performing CEP, C-SPARQL is a DSMS-inspired RSP engine that offers a naive support to event pattern matching. As shown above, even with simple event patterns, 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 behavior of both engines. In this sense, satisfying both [R3.1] and [R3.2] is minimal to assess that RSEP-QL is a common framework to describe the semantics of RSP engines.

3 Anatomy of RSEP-QL Queries

A SPARQL query is defined by a signature of the form (E, DS, QF), that indicates the evaluation of an algebraic expression E over a set of data DS to produce an answer formatted according to a query form QF [16]. This section proposes RSEP-QL queries that extend SPARQL's queries with the following features: (1) the capability to take as input not only RDF graphs but also RDF streams; (2) a set of operators to access/process streams; and (3) an evaluation paradigm moving from one-time to continuous semantics.

3.1 Data Model

There are two main kinds of input data in the context of stream processing. The first are streams, defined as sequences of highly dynamic and time-annotated data such as sensor data and micro-posts. The second type is contextual (or background) data, which is usually static or quasi-static and is used to enrich the streams and solve more sophisticated tasks, e.g., sensor locations, user profiles. etc. In RSP, contextual data may be captured by RDF graphs, while streams are captured with RDF streams.

RDF streams. To fulfill [R1], we adopt the notion of time-annotated RDF graphs as elements of RDF streams, following the data model under design by RSP-CG. We define a *timeline* T as an infinite, discrete,

ordered sequence of time instants $(t_1, t_2, \ldots)$, where $t_i \in \mathbb{N}$ and for all $i>0$, it holds that $t_{i+1}-t_i$ is a constant, called the *time unit* of T.

We now extend the definition of RDF graphs with time annotations and then define RDF streams as sequences of them.

Definition 1 (RDF Stream) *A* timestamped RDF graph *is a pair (G, t), where G is an RDF graph and $t \in T$ is a time instant. An* RDF stream *S is a (potentially) unbounded sequence of timestamped RDF graphs in a non-decreasing time order:*

$$S = (G_1, t_1), (G_2, t_2), (G_3, t_3), (G_4, t_4), \ldots$$

where, for every $i > 0$, (G_i, t_i) is a timestamped RDF graph and $t_i \leq t_{i+1}$.

Other streaming data model profiles exist and are currently under study by the RSP-CG. In this work, we focus on the model where the time annotation is represented by one time instant, as it is a usual case that appears in several scenarios.

Example 1 *Fig.1 illustrates a stream S=$(G_1, 2), (G_2, 4), (G_3, 6), (G_4, 8), (G_5, 10), \ldots$, where each G_i contains the depicted RDF triples.* □

Time-varying graphs. Statements in RDF graphs are atemporal and capture a given situation in a snapshot. We introduce the notion of time-varying graphs to capture the evolution of the graph over time (similar to time-varying relations in [4]).

Definition 2 (Time-varying graph) *A* time-varying graph $\overline{G}$ *is a function that relates time instants $t \in T$ to RDF graphs:*

$$\overline{G}\colon T \rightarrow \{G \mid G \text{ is an RDF graph}\}.$$

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

RDF streams and time-varying graphs differ on the time information: while in the former time annotations are accessible and processable by the stream processing engine, in the latter there is no explicit time annotation. In this sense, t in Def 2 can be viewed as a timestamp denoting the access time of the engine to the graph content.

3.2 RSEP-QL Dataset

A SPARQL dataset is a set of pairs (u, G), where $u \in I \cup \{def\}$[1] is an identifier for an RDF graph G. This section proposes the notion of dataset for RSEP-QL. It differs from SPARQL datasets in the presence of streams, and that RSEP-QL dataset elements may vary over time. Streams are potentially infinite, and the usage of windows allows to have a finite (and usually recent) view of portions of the streams for practical processing. We now introduce a generic notion of window functions, inspired by LARS [7].

Definition 3 (Window function) *A* window function *W with a vector of* window parameters *$\vec{p}$, denoted as $W[\vec{p}]$, takes as input a stream S, a time instant $t \in T$ and produces a* substream *(aka.* window*) S' of S, i.e., a finite subsequence of S.*

[1] $def \notin I \cup L \cup B$ denoting the default graph. See [16] for the definitions of I, L, B.

This generic notion can be instantiated with specific parameters $\vec{p}$ to realize window functions used in practice. In the following, we present a set of window functions that constitute the basis of the operators defined in the next sections.

Time-based (sliding) windows. A *time-based window* function W^τ is defined through $\vec{p} = (\alpha, \beta)$, where α is the width and β is the sliding step. It slides every β time units and filters input graphs of the last α time units. Let $t' = \lfloor \frac{t}{\beta} \rfloor \cdot \beta$, we have that:

$$W^\tau[\vec{p}](S,t) = (G_j, t_j), \ldots, (G_k, t_k),$$

where $[j,k]$ is the maximal interval st. $\forall i \in [j,k]\colon (G_i,t_i) \in S \wedge t' - \alpha < t_i \leq t'$.

Landmark windows. A *landmark window* function W^λ defined through $\vec{p} = (t_0)$ returns the content of the input stream from t_0:

$$W^\lambda[\vec{p}](S,t) = (G_j, t_j), \ldots, (G_k, t_k)$$

where $[j,k]$ is the maximal interval st. $\forall i \in [j,k]\colon (G_i,t_i) \in S \wedge t_0 \leq t_i \leq t$.

As we show below, landmark windows are useful to capture the behaviour of event pattern systems like EP-SPARQL. In fact, they offer views over large portions of the stream, without the eviction mechanism typical of sliding windows.

Identity window. The *identity window* function W^{id} is introduced to give a uniform definition of event patterns evaluation later. It simply returns the input stream, that is:

$$W^{id}[\vec{p}](S,t) = S, \text{ and } \vec{p} \text{ is an empty vector.}$$

Interval windows. The *interval-based* (or fixed) window function $W^\sqcup$ is defined through $\vec{p} = (t', t'')$ and returns the part of the input stream bounded by $[t', t'']$:

$$W^\sqcup[\vec{p}](S,t) = (G_j, t_j), \ldots, (G_k, t_k) \text{ where } \forall i \in [j,k]\colon (G_i,t_i) \in S \wedge t_i \in [t', t''].$$

For simplicity, we often omit the parameters $\vec{p}$ when it is clear from the context and write $W(S,t)$. Notably, window functions can be nested, for example, we can have $W^\sqcup(W^\tau(S,t),t)$. We denote the nesting by the $\bullet$ operator. Formally:

$$W_1 \bullet W_2(S,t) = W_1(W_2(S,t),t).$$

Example 2 *Consider S from Example 1. Here are some results of applying the time-based, landmark, and interval window functions W^τ, W^λ, and $W^\sqcup$ on this stream:*

$$W^\lambda[(1)](S,8) = (G_1,2),(G_2,4),(G_3,6),(G_4,8)$$
$$W^\tau[(5,1)](S,8) = (G_2,4)(G_3,6),(G_4,8)$$
$$W^\sqcup[(0,5)] \bullet W^\lambda[(1)](S,8) = (G_1,2),(G_2,4).$$

Dataset. We now formally define RSEP-QL datasets, as sets of pairs of an identifier $u \in I \cup \{def\}$ and either a window function applied to a stream or a time-varying graph.

Definition 4 (RSEP-QL Dataset) *An* RDF streaming dataset *SDS is a set consisting of an (optional) default time-varying graph $\overline{G}_0$, $n \geq 0$ named time-varying graphs, and $m \geq 0$ named window functions applied to a set of streams $\mathbf{S} = \{S_1, \ldots, S_k\}$:*

$$SDS = \{(def, \overline{G}_0)\} \cup \{(g_i, \overline{G}_i) \mid i \in [1,n]\} \cup$$
$$\{(w_j, W_j(S_\ell)) \mid j \in [1,m],\ \ell \in [1,k]\},\ \textit{where}$$

– *$\overline{G}_0$ is the default time-varying graph,*

– $g_i \in I$ *is the identifier of the time-varying graph* $\overline{G}_i$,

– $w_j \in I$ *is the identifier of the named window function* W_j *over the RDF stream* $S_\ell \in \mathbf{S}$.

We denote by $ids(SDS) = \{def\} \cup \{g_1, \ldots, g_n\} \cup \{w_1, \ldots, w_m\}$ *the set of symbols identifying the time-varying graphs and windows in* SDS.

An important difference that emerges comparing the SPARQL and the RSEP-QL dataset is that the former contains RDF graphs and is fixed in the sense that SPARQL datasets are composed according to the query (e.g. FROM clauses), and the set of elements included in a dataset does not vary over time. On the other hand, RSEP-QL datasets contain RDF streams and time-varying graphs that are updated as time proceeds.

Example 3 *Let* W_1^λ *and* W_2^τ *be a landmark and a time-based window functions with respective parameters* $\vec{p}_1{=}(1)$ *and* $\vec{p}_2{=}(5,1)$. *Then,* $SDS{=}\{(w_1, W_1^\lambda(S)), (w_2, W_2^\tau(S))\}$ *is an RDF streaming dataset, where* S *is from Example 1.* □

3.3 RSEP-QL Patterns

To fulfill [R2] and [R3], we introduce RSEP-QL operators to enable DSMS and CEP features. We then extend SPARQL graph patterns to support these operators on streams.

In SPARQL, the construction of the query relies on graph patterns. The elementary building block for building graph patterns is *Basic Graph Patterns* (BGP), i.e. sets of triple patterns $(t_s, t_p, t_o) \in (I \cup B \cup L \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$. More complex patterns are recursively defined on top of BGP using operators such as join and union[1].

Concerning DSMS operations, we introduce the *window graph pattern*, defined as an expression $(\text{WINDOW } w_j \ P)$, where P is a SPARQL graph pattern and $w_j \in I$ is an IRI. Intuitively, WINDOW indicates that P should be evaluated over the content of the window identified by w_j in the dataset (similarly to the SPARQL GRAPH operator).

To support CEP features, we introduce *event patterns* as follows.

(1) If P is a Basic Graph Pattern, $w \in I$, then the expressions $(\text{EVENT } w \ P)$ is an event pattern, named *Basic Event Pattern* (BEP)[2];

(2) If E_1 and E_2 are event patterns, then the expressions $(\text{FIRST } E_1)$, $(\text{LAST } E_1)$, $(E_1 \text{ SEQ } E_2)$ are event patterns;

To relate graph and event patterns, we define the *event graph pattern* as $(\text{MATCH } E)$ where E is an event pattern.

3.4 Query definition

Having all building blocks, it is now possible to define RSEP-QL queries.

Definition 5 *An* RSEP-QL query Q *is defined as* (SE, SDS, ET, QF), *where* SE *is an RSEP-QL algebraic expression,* SDS *is an RDF streaming dataset,* ET *is the sequence of time instants on which the evaluation occurs, and* QF *is the Query Form.*

[1] Cf. `https://www.w3.org/TR/sparql11-query` for the whole list.

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

The continuous evaluation paradigm is captured in the query signature through the set ET of execution times. Intuitively, this set represents the time instants on which the algebraic expression evaluation may occur. Note that this set is not explicitly defined by the query and in general it may be unknown at query registration time (as it can depend on the streaming content). In practice, ET can be expressed through report policies [8], which define rules to trigger the query evaluation. For example, C-SPARQL can be captured by a window close report policy, i.e., evaluations are periodically and determined by the window definition. EP-SPARQL and CQELS are regulated by content change report policy, i.e., evaluations occur every time a new item appears on the stream.

Example 4 *This example presents an RSEP-QL query with CEP features. The* MATCH *clause describes an event pattern* (E_1 SEQ E_2)*, where the BEPs* E_1 *and* E_2 *are defined on the respective landmark and time-based windows from Example 3. Their patterns are:* E_1=EVENT w_1 ($?x$:p $?y$) *and* E_2=EVENT w_2 ($?y$:q $?z$).

```
SELECT ?x ?z
FROM NAMED :S  WIN [LND   9] AS :w1
FROM NAMED :S  WIN [RANGE 5] AS :w2
EVENT ON :w1 { ?x :p ?y. } AS E1
EVENT ON :w2 { ?y :q ?z. } AS E2
WHERE { MATCH { E1 SEQ E2 } }
```

4 RSEP-QL Semantics

We now proceed to define the evaluation semantics of the operators introduced in Section 3.3. Section 4.1 and 4.2 present the semantics of the graph pattern and event pattern operators, respectively. Section 4.3 and 4.4 address CEP selection and consumption policies to completely capture settings such as *chronological recent* of EP-SPARQL, or the naive sequencing of triples based on last their appearances like in C-SPARQL.

4.1 Graph Pattern Evaluation Semantics

To cope with graph-based RDF streams, we adapt the graph pattern evaluation semantics from [14]. There, the evaluation semantics of a SPARQL operator is defined as a function that takes as input a graph pattern P and a SPARQL dataset DS having a default RDF graph G, and produces bags of solution mappings: partial functions that map variables to RDF terms. It is usually denoted as $[\![P]\!]_{DS(G)}$.

The RSEP-QL evaluation semantics of graph patterns considers the evaluation time instants and redefines the active graph notion. Given an RSEP-QL dataset SDS and an identifier $\iota \in ids(SDS)$ of one of its elements, we name *temporal sub-dataset*, denoted by SDS_ι, the active element of the dataset. The active element is $SDS_\iota = \overline{G}_i$ if $(\iota = g_i, \overline{G}_i) \in SDS$, or $SDS_\iota = W_j(S_\ell)$ if $(\iota = w_j, W_j(S_\ell)) \in SDS$.

Definition 6 (Graph Pattern Evaluation Semantics) *Given an RSEP-QL pattern P, an active time-varying graph or window identified by $\iota \in ids(SDS)$ of a streaming dataset SDS, and an evaluation time instant t, we define*

$$[\![P]\!]^t_{SDS_\iota}$$

as the evaluation *of P at t over the active element ι in SDS.*

We now briefly summarize the evaluation semantics of the graph patterns available in SPARQL, with a special focus on BGP and window graph patterns from Section 3.3.

Basic Graph Pattern. BGP evaluation in SPARQL is one of the few cases in which there is an actual access to the data stored in the active RDF graph. The idea behind the evaluation of BGPs in RSEP-QL is to exploit the SPARQL evaluation semantics. To make it possible, it is necessary to move from the active element ι of

SDS and the evaluation time instant t to an RDF graph over which the BGP can be evaluated. We name this RDF graph the *snapshot of a temporal sub-dataset* at t, and it is defined as:

$$SDS_{g_i}(t) = \overline{G}_i(t) \qquad \text{and} \qquad SDS_{w_j}(t) = \bigcup_{(G_k,t_k)\in W_j(S_\ell,t)} G_k$$

By exploiting the snapshot of the temporal sub-dataset, it is possible to obtain an RDF graph given a streaming dataset and an active element. This RDF graph is the one over which the BGP has to be evaluated, following the SPARQL semantics.

Example 5 *Take SDS from Example 3. We have*

$$SDS_{w_2}(12) = \bigcup_{(G_k,t_k)\in W_2^T[(5,1)](S,12)} G_k = G_4 \cup G_5 = \{:a_3 :p :b_3,\ :b_1 :q :c_1,\ :b_2 :q :c_2\}.$$

We can now define the evaluation of a basic graph pattern P as:

$$[\![P]\!]^t_{SDS_\iota} = [\![P]\!]_{SDS_\iota(t)} = \{\mu \mid dom(\mu) = var(P) \text{ and } \mu(P) \in SDS_\iota(t)\}. \tag{1}$$

Other SPARQL Graph Patterns. For other graph patterns, we maintain the idea of SPARQL of defining them recursively [16]. For example, the graph pattern $P_1\ Join\ P_2$:

$$[\![P_1 Join\ P_2]\!]^t_{SDS_\iota} = [\![P_1]\!]^t_{SDS_\iota} \bowtie [\![P_2]\!]^t_{SDS_\iota} \tag{2}$$

where SDS_ι indicates the active time-varying graph or window in the RSEP-QL dataset SDS and P_1, P_2 are graph patterns. The evaluation of $P_1 Join\ P_2$ consists of joining the two multisets of solution mappings computed by evaluating P_1 and P_2 at time t with regards to the active part SDS_ι of the RDF streaming dataset SDS.

Window Graph Pattern. Finally, we define the evaluation semantics of the window graph patterns. Given a window identifier w_j and a graph pattern P, we have that:

$$[\![\text{WINDOW } w_j\ P]\!]^t_{SDS_\iota} = [\![P]\!]^t_{SDS_{w_j}} \tag{3}$$

The following example shows the application of Equations (3) and (1).

Example 6 *Take SDS from Ex. 3 and its sub-temporal-dataset $SDS_{w_2}(12)$ from Ex. 5, let $P = \{?x :p\ ?y\}$. We have that:*

$$[\![\text{WINDOW } w_2\ P]\!]^{12}_{SDS_{def}} = [\![P]\!]^{12}_{SDS_{w_2}} = [\![?x :p\ ?y]\!]_{SDS_{w_2}(12)} = \{\{?x \mapsto :a_3, ?y \mapsto :b_3\}\}.$$

4.2 Event Pattern Evaluation Semantics

Similarly to Section 4.1, we define the evaluation semantics of event pattern operators by decomposing complex patterns into simple ones. The main difference 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 subpattern is processed. We address (i) by defining the notion of *event mapping* as a triple (μ, t_1, t_2) composed by a solution mapping and two time instants t_1 and t_2, representing the initial and final time instants that justify the matching, respectively. We assume that a partial order $\prec$ to compare timestamps is given. Depending on particular applications, specific ordering can be chosen. Regarding (ii), we associate the evaluation with an active window function that sets the boundaries of the valid ranges for evaluating event patterns.

Definition 7 (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*

$$\langle\!\langle E \rangle\!\rangle^t_W$$

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

Different from graph pattern evaluation semantics, in this case there is no explicit reference to data. This information is carried in the basic event patterns defined below.

Basic Event Patterns. Similar to BGPs, Basic Event Patterns (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 snapshot of a temporal sub-dataset (identified by w_j), restricted by the active window function W:

$$\langle\!\langle \text{EVENT } w_j \ P \rangle\!\rangle^t_W \ = \{(\mu, t_k, t_k) \mid \mu \in [\![P]\!]_{G_k} \wedge (G_k, t_k) \in W \bullet W_j(S_\ell, t)\} \tag{4}$$

Example 7 *We show how to evaluate $\langle\!\langle E_2 \rangle\!\rangle^8_{W^{id}}$ for $E_2 = (\text{EVENT } w_2 \ (?y :q \ ?z))$ from Example 4. First of all, from Example 2, we have that*

$$W^{id} \bullet SDS_{w_2}(8) = W^{id} \bullet W^{\mathcal{T}}_2(S, 8) = W^{\mathcal{T}}_2(S, 8) = (G_1, 2), (G_2, 4)(G_3, 6), (G_4, 8).$$

Now we evaluate $[\![?y :q \ ?z]\!]_{G_k}$ for $1 \leq k \leq 4$. Only G_3 and G_4 have matches, which are $\mu_2 = \{?y \mapsto :b_1, ?z \mapsto :c_1\}$ and $\mu'_2 = \{?y \mapsto :b_2, ?z \mapsto :c_2\}$. Combining with the timestamps 6 and 8 when G_3 and G_4 respectively appear in S, we have:

$$\langle\!\langle E_2 \rangle\!\rangle^8_{W^{id}} = \{(\mu_2, 6, 6), (\mu'_2, 6, 6), (\mu'_2, 8, 8)\}.$$

It is worth comparing the evaluation semantics of a BEP with the one of a BGP as defined in Section 4.1. 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.

Other Event Patterns. 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 $\prec$.

$$\langle\!\langle \text{FIRST } E \rangle\!\rangle^t_W = \{(\mu, t_1, t_2) \in \langle\!\langle E \rangle\!\rangle^t_W \mid \nexists (\mu', t_3, t_4) \in \langle\!\langle E \rangle\!\rangle^t_W : (t_3, t_4) \prec (t_1, t_2)\} \tag{5}$$

$$\langle\!\langle \text{LAST } E \rangle\!\rangle^t_W = \{(\mu, t_1, t_2) \in \langle\!\langle E \rangle\!\rangle^t_W \mid \nexists (\mu', t_3, t_4) \in \langle\!\langle E \rangle\!\rangle^t_W : (t_1, t_2) \prec (t_3, t_4)\} \tag{6}$$

Let us now consider the SEQ operator. The evaluation of E_1 SEQ E_2 is defined as:

$$\langle\!\langle E_1 \text{ SEQ } E_2 \rangle\!\rangle^t_W = \\ \{(\mu_1 \cup \mu_2, t_1, t_4) \mid (\mu_2, t_3, t_4) \in \langle\!\langle E_2 \rangle\!\rangle^t_W \wedge (\mu_1, t_1, t_2) \in \langle\!\langle \mu_2(E_1) \rangle\!\rangle^t_{W^{\sqcup}[0, t_3 - 1] \bullet W}\} \tag{7}$$

Intuitively, for each event mapping (μ_2, t_3, t_4) that matches E_2, Equation (7) 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 8 (cont'd) *We show how* $\langle\!\langle E_1 \text{ SEQ } E_2 \rangle\!\rangle^8_{W^{id}}$ *is evaluated. For* $(\mu_2, t_3, t_4) = (\{?y \mapsto :b_1, ?z \mapsto :c_1\}, 6, 6) \in \langle\!\langle E_2 \rangle\!\rangle^8_{W^{id}}$*, we then evaluate:*

$$\langle\!\langle \mu_2(E_1) \rangle\!\rangle^8_{W^{id}} = \langle\!\langle \text{EVENT } w_1 \ (?x :p :b_1) \rangle\!\rangle^8_{W^{\sqcup}[0,5] \bullet W^{id}} = \langle\!\langle \text{EVENT } w_1 \ (?x :p :b_1) \rangle\!\rangle^8_{W^{\sqcup}[0,5]}.$$

Similar to Example 7, we first see that $W^{\sqcup}[0,5] \bullet W_1^{\lambda}(S, 8) = (G_1, 2), (G_2, 4)$*. Then, evaluating* $[\![?x :p :b_1]\!]_{G_k}$ *for* $k = 1, 2$ *matches in only* G_1*. Therefore, the mapping satisfying conditions* (a) *and* (b) *is* $(\mu_1, t_1, t_2) = (\{?x \mapsto :a_1, ?y \mapsto :b_1\}, 2, 2)$*. Finally, Equation* (7) *gives us* $(\{?x \mapsto :a_1, ?y \mapsto :b_1, ?z \mapsto :c_1\}, 2, 6)$*.*

Similarly, with $(\mu'_2, 6, 6)$ *and* $(\mu'_2, 8, 8)$ *from Example 7, we find a compatible and preceding match* $(\{?x \mapsto :a_2, ?y \mapsto :b_2\}, 4, 4)$ *for* E_1*. This gives us two more results:* $(\{?x \mapsto :a_2, ?y \mapsto :b_2, ?z \mapsto :c_2\}, 4, 8)$ *and* $(\{?x \mapsto :a_2, ?y \mapsto :b_2, ?z \mapsto :c_2\}, 6, 8)$*.* □

Event Graph Pattern. Finally, we define the semantics of the MATCH operator. Being a graph pattern, its evaluation semantics is defined through the function in Definition 6. Intuitively, the function acts to remove the time annotations from event mappings and to produce a bag of solution mappings. Thus, the result of this operator can be combined with results of other graph pattern evaluations (i.e., other bags of solution mappings).

$$[\![\text{MATCH } E]\!]^t_{SDS_\iota} = \{\mu \mid (\mu, t_1, t_2) \in \langle\!\langle E \rangle\!\rangle^t_{W^{id}}\} \tag{8}$$

The initial active window function to E is W^{id}, which imposes no time restriction. Such restrictions can appear later by CEP operators like in Eq (7).

Example 9 (cont'd) *Applying* MATCH *on* $(E_1 \text{ SEQ } E_2)$ *from Example 8 returns:*

$$[\![\text{MATCH } (E_1 \text{ SEQ } E_2)]\!]^8_{SDS_{def}} = \{\{?x \mapsto :a_i, ?y \mapsto :b_i, ?z \mapsto :c_i\} \mid i = 1, 2\}.$$

4.3 Event Selection Policies

Evaluating the SEQ operator as in Equation (7) takes into account all possible matches from the two sub-patterns. This kind of evaluation captures only the *unrestricted* behavior 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 behavior, Eq. (9) simply picks the two latest event mappings that match the two sub-patterns and compare their associated timestamps.

$$\begin{aligned}\langle\!\langle E_1 \text{ SEQ}^n E_2 \rangle\!\rangle^t_W = \{(\mu_1 \cup \mu_2, t_1, t_4) \mid (t_1, t_2) \prec (t_3, t_4) \wedge \\ (\mu_1, t_1, t_2) \in \langle\!\langle \text{LAST } E_1 \rangle\!\rangle^t_W \wedge (\mu_2, t_3, t_4) \in \langle\!\langle \text{LAST } E_2 \rangle\!\rangle^t_W\}\end{aligned} \tag{9}$$

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

$$\begin{aligned}\langle\!\langle E_1 \text{ SEQ}^c E_2 \rangle\!\rangle^t_W = \{(\mu_1 \cup \mu_2, t_1, t_4) \mid (\mu_2, t_3, t_4) \in \langle\!\langle E_2 \rangle\!\rangle^t_W \wedge \\ \langle\!\langle \mu_2(E_1) \rangle\!\rangle^t_{W^\star} \neq \emptyset \wedge (\mu_1, t_1, t_2) \in \langle\!\langle \text{FIRST } \mu_2(E_1) \rangle\!\rangle^t_{W^\star} \wedge \\ (\nexists (\mu'_2, t'_3, t'_4) \in \langle\!\langle E_2 \rangle\!\rangle^t_W : \langle\!\langle \mu'_2(E_1) \rangle\!\rangle^t_{W^\star} \neq \emptyset \wedge (t'_3, t'_4) \prec (t_3, t_4))\}.\end{aligned} \tag{10}$$

Compared to (7), Equation (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 $\langle\!\langle \mu_1(E_2) \rangle\!\rangle^t_{W^\star} \neq \emptyset$ 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 $\langle\!\langle \text{FIRST } \mu_2(E_1) \rangle\!\rangle^t_{W^\star}$, which makes sure that it is the first compatible event that appeared before (μ_2, t_3, t_4). Finally, the output event matching $E_1 \text{ SEQ}^c E_2$ is $(\mu_1 \cup \mu_2, t_1, t_4)$.

Equation (11) follows the same principle as Equation (10), except that it selects the last instead of the first event mappings.

$$\begin{aligned} \langle\!\langle E_1 \text{ SEQ}^r E_2 \rangle\!\rangle^t_W = \{(\mu_1 \cup \mu_2, t_1, t_4) \mid (\mu_2, t_3, t_4) \in \langle\!\langle E_2 \rangle\!\rangle^t_W \wedge \\ \langle\!\langle \mu_2(E_1) \rangle\!\rangle^t_{W^\star} \neq \emptyset \wedge (\mu_1, t_1, t_2) \in \langle\!\langle \text{LAST } \mu_2(E_1) \rangle\!\rangle^t_{W^\star} \wedge \\ (\nexists (\mu'_2, t'_3, t'_4) \in \langle\!\langle E_2 \rangle\!\rangle^t_W : \langle\!\langle \mu'_2(E_1) \rangle\!\rangle^t_{W^\star} \neq \emptyset \wedge (t_3, t_4) \prec (t'_3, t'_4))\}. \end{aligned} \tag{11}$$

Example 10 (cont'd) *Continue with the setting in Example 8, one can check that:*

$$\langle\!\langle E_1 \text{ SEQ}^n E_2 \rangle\!\rangle^8_{W^{id}} = \left\{ \; (\{?x \mapsto :a_2, ?y \mapsto :b_2, ?z \mapsto :c_2\}, 4, 8) \; \right\};$$

$$\langle\!\langle E_1 \text{ SEQ}^c E_2 \rangle\!\rangle^8_{W^{id}} = \left\{ \begin{array}{l} (\{?x \mapsto :a_1, ?y \mapsto :b_1, ?z \mapsto :c_1\}, 2, 6) \\ (\{?x \mapsto :a_2, ?y \mapsto :b_2, ?z \mapsto :c_2\}, 4, 6) \end{array} \right\};$$

$$\langle\!\langle E_1 \text{ SEQ}^r E_2 \rangle\!\rangle^8_{W^{id}} = \left\{ \begin{array}{l} (\{?x \mapsto :a_1, ?y \mapsto :b_1, ?z \mapsto :c_1\}, 2, 6) \\ (\{?x \mapsto :a_2, ?y \mapsto :b_2, ?z \mapsto :c_2\}, 4, 8) \end{array} \right\}.$$

4.4 Event Consumption Policies

Selection policies are not sufficient to capture the behavior of EP-SPARQL in the chronological and recent settings. As described in Section 2, 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 $(\text{EVENT } w_j \; P)$.

Consumption policies which determine input for the evaluation will be covered next. Def. 8 is about a possible input for the evaluation while Def. 9 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 in an event pattern.

Definition 8 (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 9 (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 10 (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. For the formal description of $\mathcal{P}^c$ and $\mathcal{P}^r$, we refer the reader to the appendix section.

We now proceed to incorporate consumption policies into event patterns evaluation. The idea is to execute the evaluation function $\langle\!\langle . \rangle\!\rangle$ with a policy function $\mathcal{P}$, i.e., to evaluate an event pattern E with $\langle\!\langle E \rangle\!\rangle^t_{W,\mathcal{P}}$. Then, when the evaluation process reaches a BEP at leafs of the operator tree, $\mathcal{P}$ is used to filter out already consumed input. Formally:

$$\langle\!\langle \text{EVENT } w_j \ P \rangle\!\rangle^{t_i}_{W,\mathcal{P}} = [\![P]\!]_{\mathcal{I}},$$

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

Example 11 *Similar to Example 10, one has*

$$\langle\!\langle E_1 \ \text{SEQ}^c \ E_2 \rangle\!\rangle^8_{W^{id},\mathcal{P}^c} = \left\{ \begin{array}{l} (\{?x \mapsto :a_1, ?y \mapsto :b_1, ?z \mapsto :c_1\}, 2, 6) \\ (\{?x \mapsto :a_2, ?y \mapsto :b_2, ?z \mapsto :c_2\}, 4, 6) \end{array} \right\}.$$

Furthermore, carrying out the evaluation under the chronological policy ($\mathcal{P}^c$) will consume G_1, G_2, and G_3. Then, at time $t = 10$, there is no $(:a_1 \ :p \ :b_1)$ available to match the new coming triple $(:b_1 \ :q \ :c_1)$, and no event of the pattern $E_1 \ \text{SEQ}^c \ E_2$ is produced.

5 Conclusions & Outlook

The evaluation semantics of graph and event patterns presented in this paper constitutes a milestone towards defining a holistic query model for RSP that combines features from DSMS and CEP. We showed in [14] that RSP-QL, the model underlying RSEP-QL, covers the DSMS features of major RSP languages, and in this work, we introduced the CEP features. Moreover, RSEP-QL models both event patterns and their evaluation semantics taking into account the presence of selection and consumption policies. These policies are key to determine the answer that a query should produce for a given input stream. Thus, it is not possible to consider them as only technical/implementation related.

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

Table 1: Coverage of DSMS/CEP features of RSEP-QL compared to EP-SPARQL and C-SPARQL.

We have also shown that RSEP-QL complies with the set of requirements described in Section 2. First, it processes RDF graph-based streams [R1]. It is also capable of capturing the DSMS features of representative RSP languages [R2], as an inheritance from the expressivity of RSP-QL. Moreover, 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. Table 1 shows the equivalence of the main features in RSEP-QL with their counterparts in 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.

Our formalization is able to capture a rich set of operators including time-based sliding windows and event patterns such as sequencing, and combines them. As a result, RSEP-QL offers expressivity beyond the capabilities of current RSPs. For example, RSEP-QL allows to define event patterns over more than one streams, e.g., given E_1 SEQ E_2, E_1 and E_2 can match over different streams. It is not possible to express this with an EP-SPARQL or C-SPARQL query, as the first operates on a unique stream, while the latter merges different input streams in a unique one.

Furthermore, the expressivity of RSEP-QL allows defining complex queries that combine both windows and event patterns. 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. Another example consists in enriching the event pattern matching with information from contextual streaming data and other streams.

Future works include enriching RSEP-QL with more CEP operators, e.g., DURING and NOT, and realizing other selection and consumption policies in CEP, e.g., *strict/partition contiguity*, *skip till next match*, and *skip till any match* [1] in RSEP-QL.

Another important aspect of this work is its compatibility with alternative data models. Even though we chose a particular model based on timestamped graphs, one can see that it can be converted, or in some case, extended if necessary, to other similar models. For example, data streams with interval timestamps can be easily incorporated into the event pattern evaluation semantics. Finally, the RSEP-QL model can also be helpful for the RSP community, as it provides the most comprehensive query processing model for RDF streams so far. We plan to align our model to the latest proposals of the W3C RSP group, as well as study how it can be adapted for the different profiles proposed in the RSP abstract model.

A Formalizing Consumption Policies

This section formalizes the policy that consumes all input triples that are used to produce mappings of a MATCH pattern. Such input triples can be found via the notion of *justification*. Intuitively, (i) the justifications of an output event mapping (μ, t_1, t_2) from an event pattern E contains the event mappings that contribute to match E in order to produce (μ, t_1, t_2), while (ii) the justification of an event mapping (μ, t, t) of a basic event pattern (EVENT w_j P) is the event mapping itself. The following equations formally realize

this intuition:

$$just(\text{MATCH } E, I, t) = \bigcup_{(\mu,t_1,t_2)\in[\![E]\!]^t_{W^{id}}} just((\mu, t_1, t_2), E, W, I, t)$$

$$just((\mu, t_1, t_2), \text{EVENT } w_j\ P, W, I, t) = \{(\mu, t_1, t_2)\} \cap I(w_j) \cap \bigcup_{(G_k,t_k)\in W\bullet W_j(S_\ell,t)} G_k$$

$$just((\mu, t_1, t_4), E_1 \text{ SEQ}^c\ E_2, W, I, t) = just((\mu_1, t_1, t_2), E_1, W^\star, I, t) \cup just((\mu_2, t_3, t_4), E_2, W, I, t),$$

where $W^\star = W^{\sqcup}[0, t_3 - 1] \bullet W$, and

- $(\mu_2, t_3, t_4) \in [\![E_2]\!]^t_{W,\mathcal{P}^c}$ such that
  - $[\![\mu_2(E_1)]\!]^t_{W^\star,\mathcal{P}^c} \neq \emptyset$
  - $\nexists(\mu'_2, t'_3, t'_4) \in [\![E_2]\!]^t_{W,\mathcal{P}^c} \colon [\![\mu'_2(E_1)]\!]^t_{W^\star,\mathcal{P}^c} \neq \emptyset \wedge (t'_3, t'_4) \prec (t_3, t_4))$
- $(\mu_1, t_1, t_2) \in [\![\text{LAST } \mu_2(E_1)]\!]^t_{W^\star,\mathcal{P}^c}$

$$just((\mu, t_1, t_4), E_1 \text{ SEQ}^r\ E_2, W, I, t) = just((\mu_1, t_1, t_2), E_1, W^\star, I, t) \cup just((\mu_2, t_3, t_4), E_2, W, I, t),$$

where $W^\star = W^{\sqcup}[0, t_3 - 1] \bullet W$, and

- $(\mu_2, t_3, t_4) \in [\![E_2]\!]^t_{W,\mathcal{P}^r}$ such that
  - $[\![\mu_2(E_1)]\!]^t_{W^\star,\mathcal{P}^r} \neq \emptyset$
  - $\nexists(\mu'_2, t'_3, t'_4) \in [\![E_2]\!]^t_{W,\mathcal{P}^r} \colon [\![\mu'_2(E_1)]\!]^t_{W^\star,\mathcal{P}^r} \neq \emptyset \wedge (t_3, t_4) \prec (t'_3, t'_4))$
- $(\mu_1, t_1, t_2) \in [\![\text{FIRST } \mu_2(E_1)]\!]^t_{W^\star,\mathcal{P}^r}$

We can now formally define the policy functions $\mathcal{P}^c$ and $\mathcal{P}^r$. Without loss of generality, we assume that different streams do not share input triples.[1]

$$\begin{aligned}\mathcal{P}^c(E, t_i, I_{i-1}, \Delta_i) &= \mathcal{P}^r(E, t_i, I_{i-1}, \Delta_i)\\ &= \{(I_{i-1}(w_j) \cup \Delta_i(w_j)) \setminus just(\text{MATCH } E, I_{i-1}, t) \mid w_j \hat{\in} E\}.\end{aligned}$$

[1] When it is the case, one can always prefix the input triples for defining $\mathcal{P}^c$, $\mathcal{P}^r$, and then remove the prefix after that.

References

[1] Agrawal, J., Diao, Y., Gyllstrom, D., Immerman, N.: Efficient pattern matching over event streams. In: SIGMOD. pp. 147–160 (2008)

[2] Anicic, D.: Event Processing and Stream Reasoning with ETALIS. Ph.D. thesis, Karlsruhe Institute of Technology (2011)

[3] Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: WWW. pp. 635–644 (2011)

[4] Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB J. 15(2), 121–142 (2006)

[5] Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: PODS. pp. 1–16. ACM (2002)

[6] Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: a Continuous Query Language for RDF Data Streams. Int. J. Semantic Computing 4(1), 3–25 (2010)

[7] Beck, H., Dao-Tran, M., Eiter, T., Fink, M.: LARS: A Logic-based Framework for Analyzing Reasoning over Streams. In: AAAI. pp. 1431–1438 (2015)

[8] Botan, I., Derakhshan, R., Dindar, N., Haas, L.M., Miller, R.J., Tatbul, N.: SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. PVLDB 3(1), 232–243 (2010)

[9] Calbimonte, J.P., Jeung, H., Corcho, Ó., Aberer, K.: Enabling query technologies for the semantic sensor web. Int. J. Semantic Web Inf. Syst. 8(1), 43–63 (2012)

[10] Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Computing Surveys 44(3), 15:1–15:62 (2011)

[11] Dao-Tran, M., Beck, H., Eiter, T.: Contrasting RDF Stream Processing Semantics. In: JIST. pp. 289–298 (2016)

[12] Dao-Tran, M., Le-Phuoc, D.: Towards Enriching CQELS with Complex Event Processing and Path Navigation. In: HiDeSt. pp. 2–14 (2015)

[13] Dell'Aglio, D., Balduini, M., Della Valle, E.: On the need to include functional testing in RDF stream engine benchmarks. In: The 10th ESWC 2013 Conference Workshops: BeRSys2013, AImWD2013 and USEWOD2013 (2013)

[14] Dell'Aglio, D., Valle, E.D., Calbimonte, J., Corcho, O.: 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)

[15] Gutierrez, C., Hurtado, C., Vaisman, A.: Introducing time into RDF. IEEE Transactions on Knowledge and Data Engineering 19(2), 207–218 (2007)

[16] Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. http://www.w3.org/TR/sparql11-query/ (2013)

[17] Komazec, S., Cerri, D., Fensel, D.: Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In: DEBS, pp. 58–68 (2012)

[18] Luckham, D.C.: The power of events - an introduction to complex event processing in distributed enterprise systems. ACM (2005)

[19] Phuoc, D.L., Dao-Tran, M., Parreira, J.X., Hauswirth, M.: A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In: ISWC. pp. 370–388 (2011)

[20] Rinne, M., Törmä, S., Nuutila, E.: SPARQL-based applications for RDF-encoded sensor data. In: SSN, vol. 904, pp. 81–96 (2012)