Towards Enriching CQELS with Complex Event Processing and Path Navigation

Minh Dao-Tran[1] and Danh Le-Phuoc[2]

[1] Institute of Information Systems, Vienna University of Technology
Favoritenstraße 9-11, A-1040 Vienna, Austria
dao@kr.tuwien.ac.at

[2] Insight Centre for Data Analytics, National University of Ireland, Galway
danh.lephuoc@nuigalway.ie

Abstract. The increasing popularity of RDF Stream Processing has led to the developments of several RSP engines. Among these, CQELS has been developed with a native and adaptive approach, which gives it a performance advantage over other engines. However, it currently does not support two important features, namely Complex Event Processing and RDFS reasoning. We propose in this paper an extension of the CQELS query language and semantics towards enriching CQELS with these attractive functionalities.

Keywords: RDF Stream Processing, Complex Event Processing, Path Navigation

1 Introduction

Following the trend of using RDF as a unified data model for integrating diverse web data sources across systems under different controls, RDF Stream Processing (RSP) extends RDF to addresses the challenges in querying heterogeneous data streams coming from dynamic data sources of emerging ICT systems such as IoT and Smart Cities. This research field has being increasingly getting more attention with the developments of several RSP engines such as C-SPARQL [5], CQELS [17], EP-SPARQL [3], SPARQL$_{\text{Stream}}$ [7], and Sparkwave [13] which mainly aim at providing stream processing functionalities.

Among these engines, CQELS engine was designed to achieve the strict performance requirements of stream processing engines [20] with native physical query operators. The current implementation of CQELS engine provides high throughput physical operators to cover the combination of CQL operators and SPARQL 1.1 with CQELS query language (CQELS-QL). It has been shown to have a performance advantage over other RSP engines [18].

However, CQELS currently does not support two important features, namely Complex Event Processing (CEP) and RDFS reasoning. The former play an important role in handling and analyzing complex relation over high volume of dynamic data [1], while the latter allows reasoning about types and brings additional expressiveness for RSP. Having them in CQELS will enable a new range of

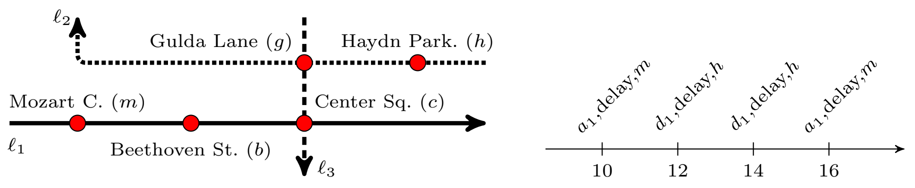

Fig. 1: Public Transportation Scenario

applications that require not only high performance but also high expressiveness, for example, as in the motivating scenario below.

Towards a more powerful CQELS, in this paper, we enrich the current query language of CQELS to support the desired features. Our contributions are:

– extending the RDF stream processing model to work on time intervals instead of time points;
– proposing an extension of the CQELS-QL to handle navigational paths and complex event processing. For the former, we make use of nSPARQL [16] and SPARQL 1.1 property paths; for the latter, we start the sequence operator SEQ, which is the most popular CEP operator in practice;
– giving a semantics of the new language in the SPARQL style, by lifting the join and sequence-related operators to work on sets of mappings with intervals, and introducing evaluation functions that make use of these operators.

The proposed new language features and semantics will be illustrated with the following scenario.

Motivating Scenario. Suppose that the traffic center at city of Vienna wants to improve the quality of public transportation by means of smart services. For this purpose, the center has background datasets regarding available vehicles such as subways, trams, buses, and connections between every pair of consecutive stops, including the stop names, types of vehicles and the duration of time needed to travel between them. Moreover, the center receives sensor data reporting the arrival and delay of vehicles with respect to stops. Passengers who register to the smart service also provide their locations as input streams.

On top of these background and streaming data, the center can offer smart services (i) to aid traffic officers in monitoring and quickly identifying potential problems in the transportation network or (ii) to notify passengers of potential traffic delays and recommend alternative routes.

Example 1 To make sure smooth connections to the city center, a continuous query can be placed to *notify about recent repeated delays of subways following by no arrival at stops that according to the network plan can be reached by subways*. When such information is instantly available to an officer, he/she can immediately react by triggering rerouting or providing complementary vehicles to solve the traffic.

Take a simplified public transportation map in Figure 1 where ℓ_1 and ℓ_2 are subway lines, and ℓ_3 is a tram line. Assume that delays and arrivals of vehicles wrt. stops are reported along the time lines. At time point 14, a repeated delay for the subway d_1 on the way to h is detected, but this is not reported as getting from h to c needs to use the tram line ℓ_3. At time point 16, the repeated delay for the subway a_1 at m is however reported as one can get from m to c by subways. ■

2 Preliminaries

This section briefly reviews the basic building blocks of this work, namely RDF, SPARQL, RDF stream processing, the navigation language nSPARQL, and CEP.

2.1 RDF and SPARQL

RDF is a W3C recommendation for data interchange on the Web [8]. It models data as directed labeled graphs whose nodes are resources and edges represent relations among them. Each node can be a named resource (identified by an IRI), an anonymous resource (a blank node), or a literal. We denote by I, B, L the sets of IRIs, blank nodes, and literals, respectively. Let $IB = I \cup B$, $IBL = I \cup B \cup L$.

A triple $(s, p, o) \in IB \times I \times IBL)$ is an *RDF triple*, where s is the subject, p the predicate, and o the object. An *RDF graph* is a set of RDF triples.

Example 2 (cont'd) The background dataset in Example 1 can be represented by the following RDF graph:

$$D = \left\{ \begin{array}{llll}
\texttt{:conn}_1\ \texttt{:beg :m}, & \texttt{:conn}_1\ \texttt{:end :b}, & \texttt{:conn}_1\ \texttt{:means :subway}, & \texttt{:conn}_1\ \texttt{:dur :3m}, \\
\texttt{:conn}_2\ \texttt{:beg :b}, & \texttt{:conn}_2\ \texttt{:end :c}, & \texttt{:conn}_2\ \texttt{:means :subway}, & \texttt{:conn}_2\ \texttt{:dur :2m}, \\
\texttt{:conn}_3\ \texttt{:beg :h}, & \texttt{:conn}_3\ \texttt{:end :g}, & \texttt{:conn}_3\ \texttt{:means :subway}, & \texttt{:conn}_3\ \texttt{:dur :3m}, \\
\texttt{:conn}_4\ \texttt{:beg :g}, & \texttt{:conn}_4\ \texttt{:end :c}, & \texttt{:conn}_4\ \texttt{:means :tram}, & \texttt{:conn}_4\ \texttt{:dur :5m}, \\
 & & \vdots & \\
\texttt{:a}_1\ \texttt{rdf:type :subway}, & & \texttt{:d}_1\ \texttt{rdf:type :subway}, & \\
 & & \vdots &
\end{array} \right\}$$

A *triple pattern* is a tuple $(sp, pp, op) \in (IB \cup V) \times (I \cup V) \times (IBL \cup V)$, where V is a set of variables. A *basic graph pattern* is a set of triple patterns.

SPARQL [12], a W3C recommendation for querying RDF graphs, is essentially a graph-matching query language. A *SPARQL query* is of the form $H \leftarrow B$, where B, the body of the query, is a complex RDF graph pattern composed of basic graph patterns with different algebraic operators such as `UNION`, `OPTIONAL`, etc.; and H, the head of the query, is an expression that indicates how to construct the answer to the query [15].

The semantics of SPARQL is defined via mappings. A *mapping* is a partial function $\mu\colon V \to IBL$. The result of a `SELECT` SPARQL query is a set of mappings that match the query's body, projected to the variables specified in the `SELECT` clause. However, one-shot queries by themselves are not able to give answers under dynamic input as in the running scenario. For this purpose, we need RDF stream processing.

2.2 RDF Stream Processing

RDF Streams and Temporal RDF Graphs. In continuous query processing over dynamic data, the temporal nature of the data is crucial and needs to be captured in the data representation. This applies to both Linked Stream Data and Linked Data, as updates in Linked Data collections are also possible. In [17], *RDF streams* and *instantaneous RDF datasets* were defined by introducing timestamps (time points) to input triples of the former and RDF graphs of the latter. In this paper, we adapt these notions by using *time intervals* as timestamps. Thereby,

1. An *RDF graph at timestamp* t, denoted by $G(t)$, is a set of RDF triples valid at time t and called an *instantaneous RDF graph.* A *temporal RDF graph* is a sequence $G = [G(t)], t \in \mathbb{N}$, ordered by t.
2. An *RDF stream* S is a sequence of elements $\langle g : [t_1, t_2]\rangle$, where g is an RDF graph and $[t_1, t_2]$ is a time interval.

Example 3 (cont'd) The sensor data regarding delay and arrival of vehicles as in Figure 1 can be represented as the following RDF stream.[3]

$$S = \begin{matrix} \langle\{(\mathtt{a}_1, \mathtt{delay}, \mathtt{m})\}, [10, 10]\rangle, & \langle\{(\mathtt{d}_1, \mathtt{delay}, \mathtt{h})\}, [12, 12]\rangle, \\ \langle\{(\mathtt{d}_1, \mathtt{delay}, \mathtt{h})\}, [14, 14]\rangle, & \langle\{(\mathtt{a}_1, \mathtt{delay}, \mathtt{m})\}, [16, 16]\rangle, \ldots \end{matrix} \quad \blacksquare$$

Continuous Queries. Queries in CQELS are inspired by the Continuous Query Language (CQL) [4], where a continuous query is composed from three classes of operators, namely stream-to-relation (S2R), relation-to-relation (R2R), and relation-to-stream (R2S) operators. In CQELS, the former are captured by extending SPARQL 1.1 grammar[4] with a "stream graph" pattern, while the latter are taken care of by SPARQL's operators. For more details on the query syntax, we refer the reader to [17].

Example 4 (cont'd) The following continuous query in the CQELS-QL notifies stops with delays of subways during the last 10 minutes to users:

```
SELECT     ?s
FROM       ex:transportationMap
FROM NAMED WINDOW :W ON ex:publicTransport [RANGE 10m]
WHERE {
 WINDOW :W { ?v :delayAt ?s }
 ?v rdf:type :subway.
}
```

[3] Note that the notification of delay or arrival here is considered atomic event; therefore, the time intervals associated with this data is of the form $[t, t]$. On the other hand, the output of a CQELS query containing non-atomic events (cf. Example 9) can be fed as input stream to another query. In such situation, input events are associated with time interval $[t_1, t_2]$ with $t_1 \leq t_2$.

[4] `http://www.w3.org/TR/sparql11-query/#grammar`

$$
\begin{aligned}
[\![\texttt{self}::a]\!]_G &= \{(a,a)\}\\
[\![\texttt{next}::a]\!]_G &= \{(x,y) \mid \exists z\colon (x,z,y)\in G\}\\
[\![\mathit{axis}^{-1}::a]\!]_G &= \{(x,y) \mid (y,x)\in [\![\mathit{axis}::a]\!]_G\}, \text{ where } \mathit{axis}\in\{\texttt{next},\texttt{node},\texttt{edge}\}\\
&\ \ \vdots
\end{aligned}
$$

Table 1: Formal semantics of nested regular expressions

2.3 Navigating RDF Graphs with nSPARQL

For a data model with graph structure like RDF, being able to navigate through the graphs is fundamentally important. In [16], the authors proposed nSPARQL, a language that incorporates navigational capabilities to a fragment of SPARQL using nested regular expressions. nSPARQL allows to pose interesting and natural queries over RDF data, and has an attractive computational property that nested regular expressions can be efficiently evaluated in polynomial time.

The new SPARQL 1.1 query language introduced *property paths*[5] that covers a fragment of nSPARQL without nesting. In this paper, we augment CQELS-QL with the full functionalities of nSPARQL based on the syntax of SPARQL 1.1. We now briefly recall nSPARQL, the following grammar defines the syntax of nested regular expressions:

$$\mathit{exp} ::= \mathit{axis} \mid \mathit{axis} :: a(a\in \mathit{IBL}) \mid \mathit{axis} :: [\mathit{exp}] \mid \mathit{exp}/\mathit{exp} \mid \mathit{exp}|\mathit{exp} \mid \mathit{exp}^* \mid \mathit{exp}^+,$$

where $\mathit{axis}\in\{\texttt{self},\texttt{next},\texttt{next}^{-1},\texttt{edge},\texttt{edge}^{-1},\texttt{node},\texttt{node}^{-1}\}$. The evaluation of a nested regular expression *exp* in a graph G is formally defined as a binary relation $[\![\mathit{exp}]\!]_G$, denoting the pairs of nodes (x,y) such that y is reachable from x in G by following a path that conforms to *exp*. Table 1 partly shows the formal semantics of nSPARQL on constructs that are used in our running example. For the full semantics, we refer the reader to [16].

Example 5 (cont'd) To find all stops from which one can reach the city center by subway connections, we can use the following expression:

$$\texttt{?s (next}^{-1}\texttt{::beg[next::means/self::subway]/next::end)}^{+}\ \texttt{:c}. \qquad \blacksquare$$

Based on nSPARQL, we define an *extended triple pattern* as either a triple pattern or a triple $(\mathit{sp}, \mathit{exp}, \mathit{op})$, where $\mathit{sp}, \mathit{op} \in I\cup L\cup V$ and *exp* is a nested regular expression. An *extended graph pattern* $\mathcal{P}$ is a set of extended triple patterns.

2.4 Complex Event Processing

Complex Event Processing [14] emerged from publish-subscribe systems [19]. While the latter refer only to single isolated events, CEP aims at timely detecting

[5] http://www.w3.org/TR/sparql11-query/#propertypaths

high-level events as complex patterns of incoming single atomic events whose order is crucial. The defined complex events can in turn be used to compose even more complex events and so forth.

To express ordering between events, CEP languages assign time intervals as timestamps for events and make use of operators rooted from Allen's interval algebra [2]. In this paper, we take the first step to incorporate CEP into CQELS-QL by adopting the sequencing operator SEQ from the SASE system [21] and applies it to time intervals. The version of SEQ with time points in SASE can be briefly described as follows.

Let A_i be an event type. Its semantics represented as $A_i(t)$, is that at a given point t in time, $A_i(t)$ is *True* if an A_i type event occurs at t, and is *False* otherwise. The SEQ operator takes a list of $n > 1$ event types as its parameters, e.g., $\text{SEQ}(A_1, \ldots, A_n)$ and specifies a particular order in which the events of interest should occur. Formally speaking:

$$\text{SEQ}(A_1, \ldots, A_n) \equiv \exists\ t_1 < t_2 < \ldots < t_n \colon A_1(t_1) \land A_2(t_2) \land \ldots \land A_n(t_n).$$

When an event type is negatively specified in the sequence, that is:

$$\text{SEQ}(A_1, \ldots, A_{i-1}, !A_i, A_{i+1}, \ldots, A_n),$$

this corresponds to the SEQ_WITHOUT operator (abbreviated as SEQ_WO in this paper), which intuitively detects sequences of events of types $A_1, \ldots, A_{i-1}$, $A_{i+1}, \ldots, A_n$ where no event of type A_i occurs in between two events of types A_{i-1} and A_{i+1}. Formally:

$$\begin{aligned}&\text{SEQ}(A_1, \ldots, A_{i-1}, !A_i, A_{i+1}, \ldots, A_n) \equiv \\ &\exists t_1 < \ldots < t_{i-1} < t_{i+1} < \ldots < t_n \colon A_1(t_1) \land \ldots \land A_{i-1}(t_{i-1}) \land A_{i+1}(t_{i+1}) \land \ldots \land A_n(t_n) \\ &\qquad\qquad\qquad\qquad \land (\forall t_i \in (t_{i-1}, t_{i+1}) \colon \neg A_i(t_i)).\end{aligned}$$

In this paper, we extend the SEQ operator to work with time intervals and RDF triple patterns, as shown next.

3 Complex Events with Graph Pattern Matching and Navigational Path

This section proposes the first step to extend CQELS with CEP and navigational capabilities by augmenting its syntax and semantics with the sequence operator SEQ and nested regular expressions from nSPARQL.

3.1 Extending CQELS Query Language

Let *TrTemp* be the short-cut for the *TriplesTemplate* pattern in SPARQL 1.1. The syntax for *triple sequence patterns TSP*, *window specifications WinSpec*, and *event clauses EC* is defined by the following grammar:

$$\begin{array}{rcl}
\mathit{TSP} & ::= & \mathit{TrTemp} \\
 & & \mid \mathit{SEQ}\ \text{‘(’}\ \mathit{TrTemp}\ (\text{‘,’}\ \mathit{TrTemp})^*\ (\text{‘,!’}\ \mathit{TrTemp})\ (\text{‘,’}\ \mathit{TrTemp})^*\ \text{‘)’} \\
 & & \mid \mathit{SEQ}\ \text{‘(’}\ (\text{‘,’}\ \mathit{TrTemp})^*\ (\text{‘,!’}\ \mathit{TrTemp})\ (\text{‘,’}\ \mathit{TrTemp})^*\ \mathit{TrTemp}\ \text{‘)’} \\
 & & \\
\mathit{WinSpec} & ::= & \text{‘}\mathit{WINDOW}\text{’}\ :\mathit{WName}\ \mathit{ON}\ \mathit{VarOrIRIref}\ \text{‘[’}\mathit{Window}\text{‘]’}\ \text{‘\{’}\ \mathit{TSP}\ \text{‘\}’} \\
 & & \\
\mathit{EC} & ::= & \mathit{WName} \\
 & & \mid \text{SEQ}\ \text{‘(’}\ \mathit{WName}\ (\text{‘,’}\ \mathit{WName})^*\ (\text{‘,!’}\mathit{WName})?\ (\text{‘,’}\ \mathit{WName})^*\ \text{‘)’} \\
 & & \mid \text{SEQ}\ \text{‘(’}\ (\mathit{WName}\ \text{‘,’})^*\ (\text{‘!’}\mathit{WName}\ \text{‘,’})?\ (\mathit{WName}\ \text{‘,’})^*\ \mathit{WName}\ \text{‘)’}
\end{array}$$

To add the navigational capabilities as in nSPARQL to the CQELS-QL, we extend the grammar of the SPARQL 1.1 property paths with one more case for nested path, namely $elt ::= elt[elt]$, where elt is a path element. We call the new query language CQELS-CEP.

Example 6 (cont'd) The following continuous query in CQELS-CEP identify stops (i) from which one can travel to the city center by subways, and (ii) report repeated delays of a subway during the last 10 minutes.

```
 1 SELECT      ?s
 2 FROM        ex:transportationMap
 3 FROM NAMED  WINDOW :W ON ex:publicTransport [RANGE 10m]
 4 WHERE {
 5  WINDOW :W {
 6   SEQ({?v :delayAt ?s}, {?v :delayAt ?s}, !{?v :arriveAt ?s})
 7  }
 8  ?v rdf:type :subway.
 9  ?s (^:beg/[:means :subway]/:end)+ :c.
10 }
```

Line 6 uses operator SEQ to specify an event pattern in which two delays of the same vehicle `?v` wrt. a stop `?s` was reported following by no arrival of `?v` at `?s`. Line 9 is the expression in Example 5 in SPARQL 1.1 syntax extended with nested regular expression described above. ■

3.2 Modeling

This section presents a formal model for the syntax proposed in Section 3.1. Let $P_1, \ldots, P_\ell$ be basic graph patterns. A *triple sequence pattern TSP* is either a graph pattern P_j or a sequence of graph patterns having at most one element negated by '!', i.e., $\mathit{TSP} = \mathit{SEQ}(P_1, \ldots, P_{i-1}, !P_i, P_{i+1}, \ldots, P_\ell)$.

A *window specification* is a tuple $W = (\mathbf{s}, \omega, \mathit{TSP})$ where $\mathbf{s}$ is an IRI identifying an input stream $S_\mathbf{s}$, ω is a window expression, and TSP is a triple sequence pattern. Intuitively, ω specifies how a snapshot of recent input is extracted from the (potentially infinite) stream $\mathbf{s}$. However, unlike traditional stream processing approaches that drop temporal information after the application of windows, this information is kept in our setting. The pattern TSP is then carried out based on the temporal information of valid triples according to the window applications. Given a window specification W, its negated version is denoted by $!\,W$.

An *event clause* EC is either a window specification W or a sequence of them having at most one element negated by '!', i.e., of the form

$$EC = \text{SEQ}(W_1, \dots, W_{i-1}, !\,W_i, W_{i+1}, \dots, W_n).$$

In [9], the authors model an RSP query as a tuple $Q = (V, P, \mathcal{D}, \mathcal{S})$ where V is a result form, P is a graph pattern, $\mathcal{D}$ is a static background dataset, and $\mathcal{S}$ is a set of input stream schemas. Under the extension of CQELS queries with triple sequence patterns, window specifications, and event clauses, we extend this idea and model CQELS queries with CEP and navigation path as tuples of the form $Q = (V, \mathcal{P}, \mathcal{D}, \mathcal{EC})$, where V is a result form, $\mathcal{P}$ is an extended graph pattern (RDF graphs with nested regular expressions, cf. Section 2.3), $\mathcal{D}$ is a set of instantaneous RDF datasets, and $\mathcal{EC}$ is a set of event clauses. To concentrate on the main contribution of this work on CEP and navigation features, we assume that the size of $\mathcal{D}$ is 1 and write D to refer to the single element of $\mathcal{D}$.

Example 7 (cont'd) The query in Ex. 6 can be modeled as $Q = (V, \mathcal{P}, D, \mathcal{EC})$, where

- $V = \{$`?s`$\}$,
- $D =$ `ex:transportationMap`,
- $\mathcal{P} = \{$`?v rdf:type :subway. ?s (^:beg/[:means :subway]/:end)+ :c.`$\}$,
- $\mathcal{EC} = \{($`ex:publicTransport`, `[RANGE 10m]`$, \text{SEQ}(P_1, P_1, !P_2))\}$,

where $P_1 = \{$`?v :delayAt ?s`$\}$ and $P_2 = \{$`?v :arriveAt ?s`$\}$. ■

4 Semantics

We now give a semantics in the SPARQL style [15] for the proposed language. Section 4.1 adapts the notion of window functions to work on input streams whose elements are associated with time intervals. In Section 4.2 we associate time intervals with SPARQL's traditional mappings, thus work on sets of mappings with intervals. We extend the join, union operators in [15] and introduce sequence-related operators to work on this input. Based on these basic operators, Section 4.3 defines an evaluation function that captures the semantics of the proposed language.

4.1 Window Functions

A *window function* w_ι takes as input an RDF stream S, time point t, a collection of parameters for type ι, and returns a sub-bag of S. For example, for time-based the window function with sliding size 1, the parameter is a range T of how far the window looks back in time to collect valid input. It can be formally stated as follows.

$$\begin{aligned} w_{RANGE}(S, t, T) \;=\; \{\langle g, [t'_1, t'_2]\rangle \mid \langle g, [t_1, t_2]\rangle \in S \wedge \\ t'_1 = \max(t_1, t - T) \wedge t'_2 = \min(t_2, t)\}. \end{aligned}$$

Each window expression then corresponds to a window function with its parameters fully specified. Let ω be a window expression, we denote by $\mathit{wtype}(\omega)$ the window type and by $\mathit{wparams}(\omega)$ the parameters of the corresponding window function. For example, with $\omega =$ `[RANGE 10m]`, we have that $\mathit{wtype}(\omega) = \mathit{RANGE}$ and $\mathit{wparams}(\omega) = 10m$.

4.2 Operators on Mappings with Intervals

A *mapping with interval* is a pair $\langle \mu, [t_1, t_2] \rangle$ where μ is a mapping and $[t_1, t_2]$ is a time interval satisfying that $t_1 \leq t_2$. Note that two mappings are called compatible, denoted by $\mu_1 \cong \mu_2$ iff $\forall x \in \mathit{dom}(\mu_1) \cap \mathit{dom}(\mu_2)\colon \mu_1(x) = \mu_2(x)$.

Let $\Omega_1, \Omega_2, \Omega_3$ be sets of mappings with intervals. We define the following operators on these sets:

$$\begin{aligned}
\text{JOIN}(\Omega_1, \Omega_2) &= \{\langle \mu_1 \cup \mu_2, [\max(t_1, t_3), \min(t_2, t_4)] \rangle \mid \\
&\qquad \langle \mu_1, [t_1, t_2] \rangle \in \Omega_1 \wedge \langle \mu_2, [t_3, t_4] \rangle \in \Omega_2 \wedge \\
&\qquad \mu_1 \cong \mu_2 \wedge \max(t_1, t_3) \leq \min(t_2, t_4)\} \\
\text{OR}(\Omega_1, \Omega_2) &= \{\langle \mu, [t_1, t_2] \rangle \mid \langle \mu, [t_1, t_2] \rangle \in \Omega_1 \vee \langle \mu, [t_1, t_2] \rangle \in \Omega_2\} \\
\text{SEQ}(\Omega_1, \Omega_2) &= \{\langle \mu_1 \cup \mu_2, [t_1, t_4] \rangle \mid \langle \mu_1, [t_1, t_2] \rangle \in \Omega_1 \wedge \langle \mu_2, [t_3, t_4] \rangle \in \Omega_2 \\
&\qquad \wedge \mu_1 \cong \mu_2 \wedge t_2 < t_3\} \\
\text{SEQ\_WO}(\Omega_1, \Omega_2, \Omega_3) &= \{\langle \mu_1 \cup \mu_2, [t_1, t_6] \rangle \mid \langle \mu_1, [t_1, t_2] \rangle \in \Omega_1 \wedge \langle \mu_3, [t_5, t_6] \rangle \in \Omega_3 \\
&\qquad \wedge \mu_1 \cong \mu_2 \wedge t_2 < t_5 \wedge \\
&\qquad (\nexists \langle \mu_2, [t_3, t_4] \rangle \in \Omega_2\colon \mu_2 \cong \mu_1 \cup \mu_3 \wedge [t_3, t_4] \subset [t_2, t_5])\} \\
\text{NO\_HEAD}(\Omega_1, \Omega_2) &= \{\langle \mu_2, [t_3, t_4] \rangle \in \Omega_2 \mid \nexists \langle \mu_1, [t_1, t_2] \rangle \in \Omega_1\colon \mu_1 \cong \mu_2 \wedge t_2 < t_3\} \\
\text{NO\_TAIL}(\Omega_1, \Omega_2, t) &= \{\langle \mu_1, [t_1, t_2] \rangle \in \Omega_1 \mid \nexists \langle \mu_2, [t_3, t_4] \rangle \in \Omega_2\colon \mu_1 \cong \mu_2 \wedge t_2 < t_3 \wedge t_4 \leq t\}
\end{aligned}$$

JOIN and OR are natural extensions of the AND and OR operators in the [15] with time intervals. Note that we use the name JOIN instead of AND in order not to be confused with operator AND in ETALIS [3] where it means taking the smallest interval that covers both $[t_1, t_2]$ and $[t_3, t_4]$, i.e., $[\min(t_1, t_2), \max(t_3, t_4)]$. We, on the other hand, take the "intersection" of $[t_1, t_2]$ and $[t_3, t_4]$.

Operator SEQ will be use to combine mappings with time intervals in a sequencing order. SEQ_WO$(\Omega_1, \Omega_2, \Omega_3)$ combines compatible sequences of mappings in Ω_1 and Ω_3 without a compatible mapping in Ω_2 (the negative element). The last two operators NO_HEAD and NO_TAIL are two special cases of SEQ_WO where the sequence starts or end with a negative element.

Example 8 (cont'd) Let $\Omega_3 = \emptyset$ and

$$\begin{aligned}
\Omega_1 &= \{\langle \{\texttt{?v} \mapsto a_1, \texttt{?s} \mapsto m\}, [10, 10] \rangle, \langle \{\texttt{?v} \mapsto d_1, \texttt{?s} \mapsto h\}, [12, 12] \rangle\} \\
\Omega_2 &= \{\langle \{\texttt{?v} \mapsto a_1, \texttt{?s} \mapsto m\}, [16, 16] \rangle, \langle \{\texttt{?v} \mapsto d_1, \texttt{?s} \mapsto h\}, [14, 14] \rangle\}
\end{aligned}$$

We have that

$$\Omega_4 = \text{NO\_TAIL}(\text{SEQ}(\Omega_1, \Omega_2), \Omega_3, 16) = \left\{ \begin{array}{l} \langle \{\texttt{?v} \mapsto a_1, \texttt{?s} \mapsto m\}, [10, 16] \rangle, \\ \langle \{\texttt{?v} \mapsto d_1, \texttt{?s} \mapsto h\}, [12, 14] \rangle \end{array} \right\} \quad \blacksquare$$

4.3 Evaluation Function

Let $Q = (V, \mathcal{P}, D, \mathcal{EC})$ be a query, $EC_1, \ldots, EC_m$ be event clauses, W, $W_1, \ldots,$ W_n, be window specifications, $P, P_1, \ldots, P_\ell$ be graph patterns, and t be a time point. The *evaluation* of Q at t is recursively defined by the function $[\![., .]\!]$ as follows.

Case (1) breaks the evaluation into evaluating the event clauses on streaming data and the extended graph pattern on the background data. Case (2) uses JOIN and the evaluation of single event clause to evaluate a set of event clauses.

$$[\![Q, t]\!] = [\![(V, \mathcal{P}, D, \mathcal{EC}), t]\!] = \{\langle \mu|_V, [t_1, t_2]\rangle \mid \langle \mu, [t_1, t_2]\rangle \in \text{JOIN}([\![\mathcal{EC}, t]\!], [\![\mathcal{P}, t]\!]_D)\} \quad (1)$$

$$[\![\{EC_1, EC_2, \ldots, EC_m\}, t]\!] = \text{JOIN}([\![EC_1, t]\!], [\![\{EC_2, \ldots, EC_n\}, t]\!]) \quad (2)$$

The evaluation of an event clause is shown in (3)-(8). Cases (3) and (4) are resp. the inductive and base cases for positive sequences of window specifications.

$$[\![\text{SEQ}(W_1, W_2, \ldots, W_n), t]\!] = [\![\text{SEQ}(W_1, \text{SEQ}(W_2, \ldots, W_n)), t]\!] \quad (3)$$

$$[\![\text{SEQ}(W_1, W_2), t]\!] = \text{SEQ}([\![W_1, t]\!], [\![W_2, t]\!]) \quad (4)$$

Cases (5)-(7) and (8) are the inductive and base cases for sequences of window specifications with one negative element, respectively. Note that when the negative element is at the beginning or the end of the sequence, we need the special operators NO_HEAD and NO_TAIL.

$$[\![\text{SEQ}(W_1, \ldots, W_{i-1}!W_i, W_{i+1}, \ldots, W_n), t]\!] =$$
$$[\![\text{SEQ\_WO}(\text{SEQ}(W_1, \ldots, W_{i-1}), W_i, \text{SEQ}(W_{i+1}, \ldots, W_n), t]\!] \quad (5)$$

$$[\![\text{SEQ}(!W_1, W_2, \ldots, W_n), t]\!] = \text{NO\_HEAD}([\![W_1, t]\!], [\![\text{SEQ}(W_2, \ldots, W_n), t]\!]) \quad (6)$$

$$[\![\text{SEQ}(W_1, \ldots, W_{n-1}, !W_n), t]\!] = \text{NO\_TAIL}([\![\text{SEQ}(W_1, \ldots, W_{n-1}), t]\!], [\![W_n, t]\!]) \quad (7)$$

$$[\![\text{SEQ\_WO}(W_1, W_2, W_3), t]\!] = \text{SEQ\_WO}([\![W_1, t]\!], [\![W_2, t]\!], [\![W_3, t]\!]) \quad (8)$$

Now consider evaluating window specification W, Case (9) brings it to evaluating the triple sequence pattern of W under the window expression ω and the input stream $S_{\mathbf{s}}$ identified by $\mathbf{s}$. Then, the evaluation of a sequence of extended graph patterns are shown in Cases (9)-(10) for positive sequences, and in Cases (12)-

(15) for sequences with one negative element.

$$[\![W, t]\!] = [\![(\mathbf{s}, \omega, \mathit{TSP}), t]\!] = [\![\mathit{TSP}, t]\!]^{\omega}_{S_{\mathbf{s}}} \tag{9}$$

$$[\![\mathrm{SEQ}(P_1, P_2, \ldots, P_\ell), t]\!]^{\omega}_{S} = [\![\mathrm{SEQ}(P_1, \mathrm{SEQ}(P_2, \ldots, P_\ell)), t]\!]^{\omega}_{S} \tag{10}$$

$$[\![\mathrm{SEQ}(P_1, P_2), t]\!]^{\omega}_{S} = \mathrm{SEQ}([\![P_1, t]\!]^{\omega}_{S}, [\![P_2, t]\!]^{\omega}_{S}) \tag{11}$$

$$\begin{aligned}&[\![\mathrm{SEQ}(P_1, \ldots, P_{i-1}, !P_i, P_{i+1}, \ldots, P_\ell), t]\!]^{\omega}_{S} = \\ &\qquad [\![\mathrm{SEQ\_WO}(\mathrm{SEQ}(P_1, \ldots, P_{i-1}), P_i, \mathrm{SEQ}(P_{i+1}, \ldots, P_\ell)), t]\!]^{\omega}_{S}\end{aligned} \tag{12}$$

$$[\![\mathrm{SEQ}(!P_1, P_2, \ldots, P_\ell), t]\!]^{\omega}_{S} = \mathrm{NO\_HEAD}([\![P_1, t]\!]^{\omega}_{S}, [\![\mathrm{SEQ}(P_2, \ldots, P_\ell), t]\!]^{\omega}_{S}) \tag{13}$$

$$[\![\mathrm{SEQ}(P_1, P_2, \ldots, !P_\ell), t]\!]^{\omega}_{S} = \mathrm{NO\_TAIL}([\![\mathrm{SEQ}(P_1, \ldots, P_{n-1}), t]\!]^{\omega}_{S}, [\![P_\ell, t]\!]^{\omega}_{S}) \tag{14}$$

$$[\![\mathrm{SEQ\_WO}(P_1, P_2, P_3), t]\!]^{\omega}_{S} = \mathrm{SEQ\_WO}([\![P_1, t]\!]^{\omega}_{S}, [\![P_2, t]\!]^{\omega}_{S}, [\![P_3, t]\!]^{\omega}_{S}) \tag{15}$$

Finally, (16) and (17) show how (extended) graph patterns are evaluated. The former evaluates graph patterns on an input stream S and a window expression ω, by taking into account from S only valid input triples at the time point t according to the window function of type $\mathit{wtype}(\omega)$ with the parameters $\mathit{wparams}(\omega)$, and then applying standard SPARQL pattern matching. The latter evaluates an extended graph pattern, i.e., with nested regular expressions, on an instantaneous background data D. We take the data in D at t, and assign the time label $[t_1, t]$ to the output mappings, where D has not changed since t_1.

$$[\![P, t]\!]^{\omega}_{S} = \left\{ \begin{aligned} \langle \mu, [t_1, t_2] \rangle \mid\ & \mathit{dom}(\mu) = \mathit{var}(P) \wedge \\ & \mu(P) \subseteq g \wedge \\ & \langle g, [t_1, t_2] \rangle \in w_{\mathit{wtype}(\omega)}(S, t, \mathit{wparams}(\omega)) \end{aligned} \right\} \tag{16}$$

$$[\![\mathcal{P}, t]\!]_{D} = \left\{ \begin{aligned} \langle \mu, [t_1, t] \rangle \mid\ & \mu \in [\![\mathcal{P}]\!]_{D(t)} \wedge \\ & \forall t' \in [t_1, t] \colon D(t') = D(t) \wedge \\ & (t_1 = 0 \vee D(t_1 - 1) \neq D(t)) \end{aligned} \right\} \tag{17}$$

Example 9 (cont'd) Consider evaluating the query $Q = (V, \mathcal{P}, D, \mathcal{EC})$ in Ex. 7 at time point 16 on the background dataset D from Ex. 2 and the input stream S from Ex. 3. The evaluation uses the equations (1), (9), (13), (16), (17).

One can see that Ω_4 in Example 8 is the result of $[\![\mathcal{EC}, 16]\!]$. On the other hand, $\Omega_5 = [\![\mathcal{P}, t]\!]_D = \{\{\texttt{?v} \mapsto a_1, \texttt{?s} \mapsto m\}\}$. The mapping $\{\texttt{?v} \mapsto d_1, \texttt{?s} \mapsto h\}$ is not concluded in Ω_5 because there is no all-subway path from h to c. Joining Ω_4 and Ω_5 and project the output to V gives us a single answer $\texttt{?s} \mapsto m$. ■

5 Related Work

A number of RSP engines have been developed by the RSP community, namely C-SPARQL [5], CQELS [17], EP-SPARQL/ETALIS [3], SPARQL$_{\text{Stream}}$ [7], and Sparkwave [13]. Among them, C-SPARQL provides a function to extract timestamps of the input triples, and by applying comparison on them, one can mimic the sequence operator without negation. However, it cannot capture complicated event patterns as time points-based semantics cannot cover multiple overlapping time intervals. Sparkwave provides RDFS entailment, which is subsumed by the navigational paths proposed in this work.

Only EP-SPARQL/ETALIS explicitly provides CEP functionalities via operators such as SEQ, PAR, DURING, etc. SEQ with negation is not directly offered but can be mimicked by existing operators. EP-SPARQL is actually just a fragment of ETALIS to work with RDF streams. ETALIS offers a Prolog rule-based semantics, and execution of the semantics boils down to backward chaining. On the other hand, we propose here an operational semantics in the SPARQL style. It is interesting to later on have a detail comparison of these two semantics.

Regarding more recent work, [11] proposed a data model for Semantically enabled Complex Event Processing in which RDF is considered as a first-class citizen. This model can be seen as an adaptation of the SASE approach [21] to work with RDF triples. LARS [6] is a logic-based framework for analyzing stream reasoning. It has a high expressiveness as the consequence of having a semantics based on Answer Set Programming [10] with advanced features such as recursion, non-monotonic reasoning, etc. It also provides different ways to refer to time using temporal operators. However, LARS has a similar problem as C-SPARQL in capturing CEP as it bases on time points. Furthermore, LARS has been developed mainly for the purpose of theoretical analysis instead of practical implementation.

6 Conclusions and Outlook

We propose an extension of the CQELS-QL to CQELS-CEP to handle navigational paths and complex event processing, starting with operator SEQ, which requires to enhance the current RSP query model to work with time intervals. We also give a semantics of the extended language in the SPARQL style. Future work needs to be done on both theoretical and practical sides. For the former, we will extend the semantics to handle Kleene closure of CEP operators, and compare our semantics with that of EP-SPARQL/ETALIS. More effort will be spent on analyzing the complexity of CQELS-CEP. Regarding practical work, efficient data structures and algorithms will be designed and implemented to realize the proposed semantics.

Acknowledgment

This work has emanated from research supported in part by research grants from Irish Research Council under Grant No. GOIPD/2013/104, European Commission under Grant No. FP7-ICT-608662 (VITAL), and by the Austrian Science Fund (FWF) project P26471.

References

1. J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 147–160, 2008.
2. J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Commun. ACM*, 26(11):832–843, 1983.
3. D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *WWW*, pages 635–644, 2011.
4. A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.
5. D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-SPARQL: a continuous query language for rdf data streams. *Int. J. Semantic Computing*, 4(1):3–25, 2010.
6. 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, 2015.
7. J.-P. Calbimonte, Ó. Corcho, and A. J. G. Gray. Enabling ontology-based access to streaming data sources. In *ISWC (1)*, pages 96–111, 2010.
8. R. Cyganiak, D. Wood, and M. Lanthaler. RDF 1.1 Concepts and Abstract Syntax. http://www.w3.org/TR/rdf11-concepts/, 2014.
9. M. Dao-Tran, H. Beck, and T. Eiter. Towards Comparing RDF Stream Processing Semantics. submitted to HiDeSt 2015.
10. T. Eiter, G. Ianni, and T. Krennwallner. Answer Set Programming: A Primer. In *RW*, pages 40–110, 2009.
11. S. Gillani, G. Picard, F. Laforest, and A. Zimmermann. Towards an Efficient Semantically Enriched Complex Event Processing and Pattern Matching. In *3rd International Workshop on Ordering and Reasoning*, 2014.
12. S. Harris and A. Seaborne. SPARQL 1.1 Query Language. http://www.w3.org/TR/sparql11-query/, 2013.
13. S. Komazec, D. Cerri, and D. Fensel. Sparkwave: Continuous Schema-Enhanced Pattern Matching over RDF Data Streams. In *DEBS*, pages 58–68, 2012.
14. D. C. Luckham. *The Power of Events - an Introduction to Complex Event Processing in Distributed Enterprise Systems.* ACM, 2005.
15. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.*, 34:16:1–16:45, September 2009.
16. J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *J. Web Sem.*, 8(4):255–270, 2010.
17. D. L. Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *ISWC (1)*, pages 370–388, 2011.
18. D. L. Phuoc, M. Dao-Tran, M.-D. Pham, P. Boncz, T. Eiter, and M. Fink. Linked stream data processing engines: Facts and figures. In *ISWC - ET*, pages 300–312, 2012.
19. D. S. Rosenblum and A. L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *ESEC / SIGSOFT FSE*, pages 344–360, 1997.
20. M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, Dec. 2005.
21. E. Wu, Y. Diao, and S. Rizvi. High-Performance Complex Event Processing over Streams. In *SIGMOD Conference*, pages 407–418, 2006.