RSP-QL Semantics: a Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems

Daniele Dell'Aglio
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico of Milano, Italy

Emanuele Della Valle
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico of Milano, Italy

Jean-Paul Calbimonte
Distributed Information Systems Laboratory, École Polytechnique Fédérale de Lausanne, Switzerland

Oscar Corcho
Ontology Engineering Group, Universidad Politécnica de Madrid, Spain

ABSTRACT

RDF and SPARQL are established standards for data interchange and querying on the Web. While they have been shown to be useful and applicable in many scenarios, they are not sufficiently adequate for dealing with streams of data and their intrinsic continuous nature. In the last years data and query languages have been proposed to extend both RDF and SPARQL for streams and continuous processing, under the name of RDF Stream Processing – RSP. These efforts resulted in several models and implementations that, at a first look, appear to propose alternative syntaxes but equivalent semantics. However, when asked to continuously answer the same queries on the same data streams, they provide different answers at disparate moments due to the heterogeneity of their operational semantics. These discrepancies render the process of understanding and comparing continuous query results complex and misleading. In this work, we propose RSP-QL, a comprehensive model that formally defines the semantics of an RSP system. RSP-QL makes explicit the hidden assumptions of currently available RSP systems, allows defining a formal notion of correctness for RSP query results and, thus, explains why available implementations provide different answers at disparate moments.
Keywords: RDF streams, continuous query processing, conceptual model, correctness

INTRODUCTION

One of the key ingredients of the Ubiquitous Web is the management of data streams, which may be obtained from a range of data sources, from social networks to environmental sensors. These data streams may be expressed using RDF, possibly according to well-established vocabularies, such as the W3C Semantic Sensor Network Ontology (Compton et al., 2012). In these cases, we can talk about RDF streams, which are formally defined as potentially unbounded sequences of time-varying RDF statements or graphs. In recent years, several RDF Stream Processing (RSP) systems have emerged, which allow querying RDF streams using extensions of SPARQL that include operators that take into account the streaming nature of these dynamic data sources (Barbieri, Braga, Ceri, Della Valle, & Grossniklaus, 2010; Calbimonte, Jeung, Corcho, & Aberer, 2012; Phuoc, Dao-Tran, Parreira, & Hauswirth, 2011; Anicic, Fodor, Rudolph, & Stojanovic, 2011). These systems are heterogeneous in terms of syntax and capabilities (due to the choice of operators and syntax selected to extend SPARQL). In addition, they implement different evaluation semantics for a set of constructs that may look similar in principle (for example, they may handle time window operators differently). These engines have different assumptions on how the query processing and delivery of results take place, which makes it difficult to describe, compare, understand and evaluate their behavior.

In this paper, we address the following research question: *is it possible to create a formal RDF stream processing model, including its evaluation semantics, that can be used to describe existing RSP systems?*. For this purpose, we propose RSP-QL, a unifying formal model for representing and processing RDF streams, that reflects the different semantics of existing RSP systems. RSP-QL extends the SPARQL model and also takes into account two existing models coming from the streaming data world: CQL (Arasu, Babu, & Widom, 2006) and SECRET (Botan et al., 2010). CQL is a continuous extension of SQL: its semantics define a formal model with three kinds of operators (S2R, R2R and R2S) that process and transform streams and relations. SECRET is a framework to characterise and analyse the operational semantics of window operators. A second contribution of this paper is to show *how this formal model can be used to test whether an RSP system is correct or not*. RSP-QL extends our previous work (Dell'Aglio, Balduini, & Della Valle, 2013) that was focused on formalising the notion of correctness in RSP query processing and on the development of an oracle – a system that tests whether an RSP implementation works in accordance to the corresponding evaluation semantics of the language that it gives support to. We have shown that this oracle, based on the RSP-QL model can effectively model the behavior of existing engines, and assess the correctness of their results. As a result of our experiments we detected errors in existing implementations, some of which have been now fixed by the corresponding system implementers.

The remainder of the paper is organised as follows. After a brief recap on RDF and SPARQL, we formally define the notion of RDF streams and the evaluation semantics of a generic SPARQL extension that allows handling RDF streams (which we name RSP-QL). Our formal definitions are based on the existing representation and evaluation semantics for RDF and SPARQL. We then show that existing RSP systems can be represented as instances of the RSP-QL query model, highlighting the differences among them, e.g. different strategies to evaluate the continuous queries and different ways to manage the sliding windows. Next, we formally define the notion of correctness in RSP systems, and we explain how to use it to check whether system implementations are computing the correct answers. Finally, we present the conclusion and final remarks.

BACKGROUND: RDF AND SPARQL

RDF is a W3C recommendation for data interchange on the Web (Cyganiak, Wood, & Lanthaler, 2014). RDF data is structured as directed labeled graphs, where the nodes are *resources*, and the edges represent relations among them. Each node of the graph can be a named resource (identified by an IRI), an anonymous resource (a blank node) or a literal. We identify with I, B and L respectively the sets of IRIs, blank nodes and literals. We define an *RDF term* as an element of the set $I \cup B \cup L$.

Definition 1. An **RDF statement** d is a triple $(s, p, o) \in (I \cup B) \times (I) \times (I \cup B \cup L)$. A set of RDF statements is an **RDF graph**.

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

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

Given an RDF graph, it is possible to query it through the SPARQL query language (Harris & Seaborne, 2013), which is another W3C Recommendation. A SPARQL query typically contains one or more triple patterns called a basic graph pattern. Triple patterns are similar to RDF triples except that they may contain variables in place of resources. These patterns may match a subgraph of the RDF data, by substituting variables with RDF terms, resulting in an equivalent RDF subgraph.

Definition 2. A **triple pattern** tp is a triple (sp, pp, op) such that

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

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

Graph patterns in a SPARQL query can include basic graph patterns and other compound patterns defined by different algebraic operators, such as `OPTIONAL`, `UNION` and `FILTER`.

To define the semantics of the evaluation of a SPARQL query, we summarise the notion of solution mappings, and evaluation of SPARQL graph patterns, as detailed in (Harris & Seaborne, 2013; Pérez, Arenas, & Gutierrez, 2009).

Definition 3. A **solution mapping** μ is a partial function $\mu : V \to I \cup B \cup L$. It maps a set of variables to a set of RDF terms. A mapping has a domain $dom(\mu)$ which is the subset of V over which it is defined. We denote as $\mu(x)$ the RDF term resulting by applying the solution mapping to variable x. We denote as ω a **multiset of solution mappings**, and as ψ a sequence of **solution mappings**.

Given a SPARQL query over an RDF graph, a query solution can be represented as a set of solution mappings, each of which assigns terms of RDF triples in the graph, to variables of the query. SPARQL defines its operators in terms of these mappings. For instance the `JOIN` operator is defined as follows:

Definition 4. *L*et ω_1 and ω_2 be multisets of solution mappings. SPARQL 1.1 specification defines `JOIN` as:

$$\begin{aligned} Join(\omega_1, \omega_2) = \{merge(\mu_1, \mu_2)|& \\ \mu_1 \in \omega_1 \wedge \mu_2 \in \omega_2& \\ \wedge\ \mu_1 \text{ and } \mu_2 \text{ are compatible}\}& \end{aligned}$$

Compatibility of mappings is defined as follows: mappings μ_1 and μ_2 are compatible if $\forall x \in dom(\mu_1) \cap dom(\mu_2)$, then $\mu_1(x) = \mu_2(x)$. The definitions of other operators are fully detailed in the SPARQL 1.1 specification (Harris & Seaborne, 2013).

SPARQL queries operate over collections of one or more RDF graphs, namely RDF datasets.

Definition 5. An **RDF dataset** DS is a set:

$$DS = \{g_0, (u_1, g_1), (u_2, g_2), ...(u_n, g_n)\}$$

where g_0 and g_i are RDF graphs, and each corresponding u_i is a distinct IRI. g_0 is called the **default graph**, while the others are called **named graphs**. During the evaluation of a query, the graph from the dataset used for matching the graph pattern is called **active graph**. Multiple graphs can become active during the evaluation.

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

$$query \to \ \mathsf{SELECT}\ v_1, ..., v_n\ \mathsf{WHERE}\ gp$$

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

There are other constructs such as *solution modifiers* (e.g. `DISTINCT`, `ORDER BY`, `LIMIT`) that are applied after pattern matching. These and other modifiers can be found in the SPARQL Query Language specification (Harris & Seaborne, 2013). With all these concepts at hand, we can define a SPARQL query as follows.

Definition 6. A **SPARQL query** is a tuple (E, DS, QF), where E is a SPARQL algebra expression, DS is an RDF dataset, and QF is a query form.

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

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

The evaluation semantics of a SPARQL query algebra expression w.r.t. an RDF dataset is also defined for every operator of the algebra.

Definition 7. The **SPARQL evaluation semantics** of an algebra expression E is denoted as $eval(DS(g), E)$, where $DS(g)$ is the dataset DS with active graph g.

For example, given two graph patterns P_1 and P_2 the evaluation of the join operator is given as:

$$eval(D(g), Join(P_1, P_2)) = \\ Join(eval(D(g), P_1), eval(D(g), P_2))$$

STREAMING EXTENSION OF THE RDF MODEL

Now that we have introduced the basic concepts that define RDF and SPARQL, we move in this section into how the RDF model can be extended to deal with data streams. One of the first steps towards this goal consists in considering the temporal dimension that RDF data in such data streams must have. Several works studied how to add such temporal dimension to RDF (Gutiérrez, Hurtado, & Vaisman, 2005; Pugliese, Udrea, & Subrahmanian, 2008; Motik, 2012), for instance by adding a fourth element to the triple to express the validity time. In this section, we formalise the notion of RDF stream associating a time instant to each RDF statement, as in (Barbieri et al., 2010; Calbimonte et al., 2012; Phuoc et al., 2011; Komazec, Cerri, & Fensel, 2012; Urbani, Margara, Jacobs, van Harmelen, & Bal, 2013). We start by defining the notion of time as in (Arasu et al., 2006).

Definition 8. The **time** T is an infinite, discrete, ordered sequence of time instants $(t_1, t_2, \ldots)$, where $t_i \in \mathbb{N}$. A **time unit** is the difference between two consecutive time instants $(t_{i+1} - t_i)$ and it is constant.

It is now possible to extend the definition of RDF statement with a temporal annotation, and consequently define RDF streams as sequences of them.

Definition 9. A **timestamped RDF statement** is a pair (d, t), where d is an RDF statement and $t \in T$ is a time instant. An **RDF stream** S is a (potentially) unbounded sequence of timestamped RDF statements in non-decreasing time order:

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

where, for every $i > 0$, (d_i, t_i) is a timestamped RDF statement and where $t_i <= t_{i+1}$.

Example 3. *(cont'd). The Sirius Cybernetics Corporation offers for free-download a mobile App that delivers instantaneous discount coupons to shoppers while they are near by shops like A and B. The shoppers Carl, Diana and Eve have such an App on their mobiles. When they are within 200 meters from A or B, the App records it on the RDF stream S_{nearby} with the following timestamped RDF statements (we use a turtle-like notation, where the statements are enriched with the time relative instant):*

```
:diana  :isNearby   :a  [2] .
:eve    :isNearby   :b  [2] .
:carl   :isNearby   :a  [5] .
:eve    :isNearby   :a  [7] .
:diana  :isNearby   :b  [12] .
```

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

Before moving to the extension of the query language to process RDF streams, we introduce the time notion in RDF graphs. As explained in the RDF 1.1 primer:

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

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

Definition 10. A **time-varying graph** G is a function that relates time instants $t \in T$ to RDF graphs:

$$G : T \rightarrow \{g \mid g \text{ is an RDF graph}\}$$

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

Figure 1 helps in understanding the difference between the two concepts. We indicate with the lowercase letter g $(g, g_1, g_2, \ldots)$ the RDF graphs, and with the capital letter G the time-varying graphs $(G, G_1, G_2, \ldots)$. The time-varying graph G associates time instants to RDF graphs; for each time instant t for which G is defined, $G(t)$ refers to an instantaneous RDF graph; being G a function, each time instant is associated to one and only one RDF graph. It is worth to note that the instantaneous graph contains RDF statements (without any timestamp). It follows that instantaneous graphs can be queried through the SPARQL query language without any continuous extension.

Example 4. *(cont'd). The Sirius Cybernetics Corporation manages to convince both Alice and Bob to use its instantaneous discount coupon*

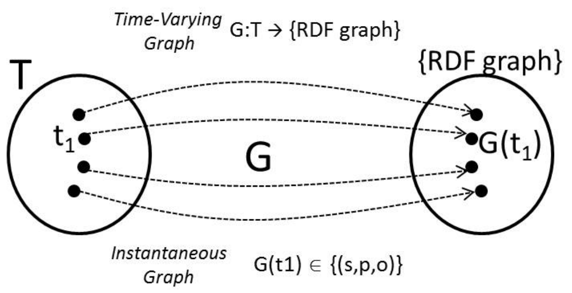


Figure 1. : Time-varying and instantaneous graph

service from the time instant 2 to the time instant 13. After that, Alice leaves the service, and only Bob keeps using it. The time-varying graph g_{shops}, which captures the shops using the instantaneous coupon service, is built as follows:

- *at time $t_1 < 2$, $G_{shops}(t_1)$ is the empty graph;*
- *at time $t_2 \in [2, 13]$, $G_{shops}(t_2)$ is the graph presented in Example 1 (g_{shops}) including both shops;*
- *at time $t_3 > 13$, $G_{shops}(t_3)$ is the following RDF graph g'_{shops} including only shop B:*

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

CONTINUOUS EXTENSION OF THE SPARQL QUERY LANGUAGE

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

One of the main differences between SPARQL and RSP-QL is the way in which queries are evaluated. Adopting the DSMS nomenclature (Babu & Widom, 2001; Chen, DeWitt, Tian, & Wang, 2000), SPARQL allows to issue **one-time queries**, queries that are evaluated once by the SPARQL engine. In contrast, RSP-QL allows to register **continuous queries**, queries issued once and continuously evaluated (Babu & Widom, 2001), i.e. they are evaluated multiple times, and the answer is composed by listing the results of each evaluation iteration.

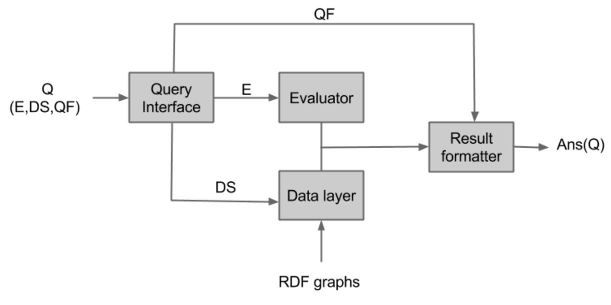
Example 5. *(cont'd). The instantaneous discount coupon service offered by the Sirius Cybernetics Corporation allows shop owners to propose discounts to shoppers nearby, who use the App associated to the service, e.g., when their shops are empty. Those instantaneous coupons are streamed out in the RDF stream* S_{coupon}*:*

```
:alice :offers "10% discount on ..."  [8]  .
:bob   :offers "free coffee at ..."   [15] .
```

The first statement states that Alice offers an instantaneous discount coupon at time instant 8, while the second reports on a offer from Bob at the time instant 15.

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

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



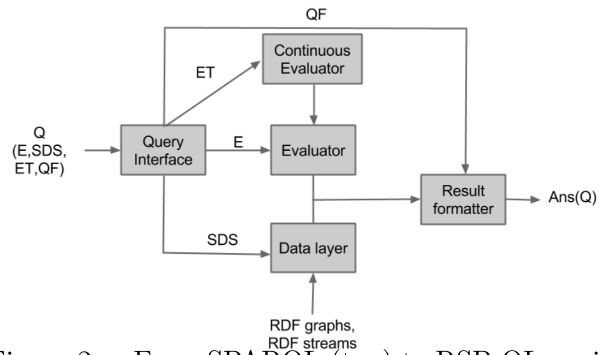


Figure 2. : From SPARQL (top) to RSP-QL engine (bottom)

Definition 11. An **RSP-QL query** Q is defined as (SE, SDS, ET, QF) where

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

The continuous-evaluation paradigm influences the definition of RSP-QL query. We go in depth in the remaining of the section, but we provide now some intuitions about this model.

First, the dataset has to take into account time, both to manage the RDF streams and to cope with time-varying RDF graphs.

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

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

Assumptions

In the rest of this section, we make the following assumptions. The first assumption is related to the time required for query evaluation: we assume that the time required to evaluate the query over the current input and to produce the portion of answer is lower than the time unit. It is a common assumption made in this kind of work (Cugola & Margara, 2012), so as to guarantee that the stream processor works without accumulating delays in the continuous-evaluation process.

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

RSP-QL dataset

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

Definition 12. A **window** $W(S)$ is a set of RDF statements extracted from a stream S. A **time-based window** is a window defined through two time instants o, c (respectively named opening and closing time instants) such that:

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

Example 6. *(cont'd) The time-based window W over S_{nearby} with opening time $o = 4$ and $c = 8$ contains the following RDF statements:*

```
:carl   :isNearby   :a  .
:eve    :isNearby   :a  .
```

It is worth noting that the window builds an RDF graph: in fact, the RDF statements do not have any time annotation.

A time-based window selects a portion of the statements contained in the stream. In order to be able to process the content of the stream at different time instants, we need an operator that creates multiple RDF graphs over the stream (i.e. a time-varying graph). This operator is called time-based sliding window, and it operates by creating a set of time-based windows (with different opening/closing time instants) over the stream.

Definition 13. A **time-based sliding window** $\mathbb{W}$ takes as input a stream S and produces a time-varying graph $G_{\mathbb{W}}$ (Figure 3). $\mathbb{W}$ is defined through a set of parameters (α, β, t^0), where:

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

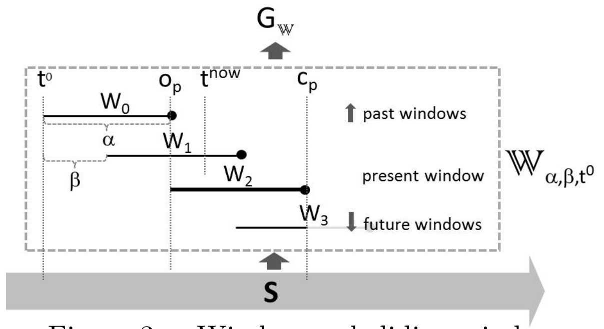


Figure 3. : Window and sliding window.

We denote with $\mathbb{W}(S)$ the application of sliding window $\mathbb{W}$ on the stream S

The above definition explains which are the inputs and outputs of the sliding window. Next, we provide the description of how it works. To distinguish the time-based windows (which produce sets of RDF statements) from the time-based sliding windows (operator) we use the capital letter W, W_1, W_2, ... for the former and the blackboard bold letter $\mathbb{W}$, $\mathbb{W}_1$, $\mathbb{W}_2$, ... for the latter.

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

- the opening time of the first window W_1 is t^0
- each window has width α, i.e. for each window W_i of $\mathbb{W}$ defined through $(o_i, c_i]$, it holds:
$$c_i - o_i = \alpha$$
- the difference between the opening time instants of two consecutive windows is β, i.e. given two windows W_i and W_{i+1} of $\mathbb{W}$, defined respectively in the time intervals $(o_i, c_i]$ and $(o_{i+1}, c_{i+1}]$, holds:
$$o_{i+1} - o_i = \beta$$

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

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

As aforementioned, the output of a time-based sliding window $\mathbb{W}$ is a time-varying graph $G_{\mathbb{W}}$. At each time instant t for which $\mathbb{W}$ is defined, $G_{\mathbb{W}}(t)$ contains the content of the present (sub-)window $W_p^{'} = (o_p, t]$ such that $W_p^{'} \subseteq W_p$, i.e.,

$$\begin{aligned}\mathbb{W}(S,t) = &G_{\mathbb{W}}(t) = \\ &\{d | d \in W_p^{'} \wedge (d, t_d) \in S \wedge t_d \in (o_p, t]\}\end{aligned}$$

Example 7. *(cont'd) The time-based sliding window $\mathbb{W}_1$ over S_{nearby} defined through ($\alpha = 5, \beta = 2, t^0 = 1$) generates the following windows: W_1 between $(1, 6]$, W_2 between $(3, 8]$, W_3 between $(5, 10]$ and so on. If $\mathbb{W}_1$ is evaluated at the time instant 9, the present window is W_3 and it contains:*

```
:eve    :isNearby   :a .
```

To summarise, given a time instant t^{now} and a time-based sliding window $\mathbb{W}$ defined through the parameters (α, β, t^0), $\mathbb{W}$ takes as input a stream S, produces a time-varying graph $G_{\mathbb{W}}$, and $G_{\mathbb{W}}(t^{now})$ denotes an instantaneous graph at time t^{now}. Finally, we can define the concept of RSP-QL dataset.

Definition 14. An **RSP-QL dataset** SDS is a set composed by an (optional) default graph, n ($n \geq 0$) named graphs and m ($m \geq 0$) named time-varying graphs obtained by the application of time-based sliding windows over $o \leq m$ streams:

$$\begin{aligned}SDS = \{&G_0,\\ &(u_1, G_1), \ldots, (u_n, G_n),\\ &(w_1, \mathbb{W}_1(S_1)), \ldots, (w_j, \mathbb{W}_j(S_1)),\\ &(w_{j+1}, \mathbb{W}_{j+1}(S_2)), \ldots, (w_k, \mathbb{W}_k(S_2)),\\ &\ldots\\ &(w_l, \mathbb{W}_l(S_o)), \ldots, (w_m, \mathbb{W}_m(S_o))\}\end{aligned}$$

where

- G_0 is the default time-varying graph
- u_p, w_q are IRIs ($u_p, w_q \in I$) for each $p \in [1, n]$ and $q \in [1, m]$
- (u_p, G_p) identifies a time-varying named graph, for each $p \in [1, n]$
- $(w_q, \mathbb{W}_q(S_r))$ identifies a named time-based sliding window over an RDF stream, for each $q \in [1, m]$ and $r \in [1, o]$

When composing a SPARQL query, it is possible to declare different graphs that will be merged to compose the default graph. In this case, the default graph is composed by merging together the unnamed graphs and unnamed time-based sliding windows applied to the RDF streams.

Example 8. *(cont'd) One of the RSP-QL dataset that can be built to answer the query presented in Example 5 is:*

$$SDS = \{G_0 = G_{shops}, \\ (w_1, \mathbb{W}_1(S_{nearby})), (w_2, \mathbb{W}_2(S_{coupon}))\}$$

where the default graph is the time-varying graph describing the shops that use the instantaneous discount coupon service, w_1 and w_2 identify respectively the sliding window $\mathbb{W}_1$ over the stream S_{nearby} (defined in Example 7), and the sliding window $\mathbb{W}_2$ over the stream S_{coupon}. $\mathbb{W}_2$ is defined as $(\alpha = 2, \beta = 2, t^0 = 0)$.

Time-varying collections of solution mappings

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

As explained above, the continuous query evaluation consists in evaluating the query multiple times at different instants. At each iteration, fixed a time instant, the RSP-QL engine can determine on which data the algebraic expression should be evaluated and from now on, the evaluation process is atemporal. In other words, we need to push the time dimension in the data types exchanged by the operators, and we do not need to redefine the existing SPARQL 1.1 operators to work with timestamped data. This approach is different from the one in (Bolles, Grawunder, & Jacobi, 2008), where the authors redefine the SPARQL algebraic operators in order to cope with streaming data.

As explained in the background section, SPARQL algebra operators work on RDF graphs or on collections of solution mappings (Harris & Seaborne, 2013). For example, `BGP`s receive as input the active RDF graphs and produce as output multisets (bags) of solution mappings; `JOIN`, `UNION` and `DIFF` operators consume and produce multisets of solution mappings; and `ORDER BY` and `DISTINCT` operators work on sequences of solution mappings.

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

Definition 15. A **time-varying sequence of solution mappings** Ψ maps time instants $t \in$

T to the set of solution mapping sequences:

$$\Psi : T \rightarrow \{\psi \mid \psi \text{ is a sequence of solution mappings}\}$$

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

A **time-varying multiset of solution mappings** Ω maps the time T to the set of solution mapping multisets

$$\Omega : T \rightarrow \{\omega \mid \omega \text{ is a multiset of solution mappings}\}$$

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

Even if our model ensured that existing SPARQL 1.1 operators continue to work on RDF statements and solution mappings, we need to adjust their definitions to introduce the time-aware collections of inputs (outputs). In the following, we show the extension for the `JOIN` operator, presented in Definition 4. The following definition introduces the instantaneous multisets of solution mappings (differences w.r.t. SPARQL 1.1 `JOIN` definition are underlined).

Definition 16. $\underline{\text{For a given time instant } t}$, let $\underline{\Omega_1(t)}$ and $\underline{\Omega_2(t)}$ be $\underline{\text{instantaneous}}$ multisets of solution mappings. We define **RSP-QL `JOIN`** as:

$$\begin{aligned} Join(\underline{\Omega_1(t)},\underline{\Omega_2(t)}) = \{merge(\mu_1, \mu_2) \mid \\ \mu_1 \in \underline{\Omega_1(t)} \\ \wedge \mu_2 \in \underline{\Omega_2(t)} \\ \wedge \mu_1 \text{ and } \mu_2 \text{ are compatible}\} \end{aligned}$$

Notably, fixed a time instant t, $\Omega_1(t)$ and $\Omega_2(t)$ are two bags of solution mappings: the JOIN operator works on their content (solution mappings) as in the original definition.

Continuous evaluation semantics

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

We now extend the definition of SPARQL evaluation semantics (Definition 7) to take into account the time dimension: we add a third parameter, evaluation time t, in the *eval* function signature.

Definition 17. (RSP-QL evaluation semantics). Given an RSP-QL dataset SDS, an algebraic expression SE and an evaluation time instant t, we define

$$eval(SDS(G), SE, t)$$

as the evaluation of SE at time t with respect to the RSP-QL dataset SDS having active time-varying graph G.

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

Definition 18. The **evaluation of a Basic Graph Pattern** operator is defined in the following way:

$$eval(SDS(G), BGP, t) = eval(SDS(G, t), BGP)$$

The solution of the BGP is computed with regard to the RSP-QL dataset SDS having G as active graph at time t (i.e. $G(t)$ it is an instantaneous graph). The output of the evaluation is an instantaneous multiset of solution mappings $\Omega(t)$.

If G is a time-varying graph, $SDS(G,t)$ refers to the instantaneous graph $G(t)$ at time t; otherwise, if G is the graph obtained through a sliding window $\mathbb{W}$ over a stream S, $SDS(G,t)$ points to the instantaneous graph $G_{\mathbb{W}}(t)$. To sum up:

$$SDS(G,t) = \begin{cases} SDS(G(t)), \\ \text{if G is a time-varying graph} \\ SDS(\mathbb{W}(S,t)), \\ \text{if } \mathbb{W}(S,t) \text{ is obtained by } \mathbb{W}(S) \end{cases}$$

The above evaluation of the BGP is important because it shows that the BGP is evaluated over an RDF graph (identified by an instantaneous graph at evaluation time). The evaluation definitions of the other existing SPARQL 1.1 algebraic operators propagates the evaluation time to the evaluation of the algebraic expressions. The next definition presents the evaluation of the JOIN operator.

Definition 19. The **evaluation of JOIN** is defined as follows:

$$\begin{aligned} \mathit{eval}(SDS(G),\mathit{Join}(P1,P2),t) = \\ \mathit{Join}(\mathit{eval}(SDS(G),P1,t), \\ \mathit{eval}(SDS(G),P2,t)) \end{aligned}$$

where $SDS(G)$ indicates the active time-varying graph G in the RDF stream dataset SDS and $P1, P2$ are graph patterns.

In the algebraic tree the JOIN operators have two children, represented by the two graph patterns $P1$ and $P2$. The evaluation of the JOIN operator consists in applying the JOIN (Definition 16) to the two multisets of solution mappings computed by evaluating $P1$ and $P2$ at time t with regards to the RSP-QL dataset with active graph G.

Streaming operators

In previous sections, we extended the query model of SPARQL to consume dynamic data (data that changes over time) and to process it in a continuous fashion. Now, we need to work on the output of the query: our model should produce not only time-varying RDF graphs, but also RDF streams. To enable this feature, we add a set of *streaming operators, that take as input sequences of solution mappings and produce sets of timestamped solution mappings. As we see above, in the RSP-QL model the timestamped solution mappings can be processed only by the Query Form operators, what means that the *streaming operators are thought to be the outer elements of the algebraic trees. We plan to investigate in future work the introduction of additional operators that are able to process those kinds of data.

Despite other RSP-QL operators that are inherited by SPARQL, the *streaming operators require a time instant as input parameter because they are time-aware: they need to know the current evaluation time in order to produce their outputs. They reintroduce the temporal dimension in the data, appending a time instant on the solution mappings; *streaming operators can be considered the dual operators of sliding windows, that process timestamped RDF statements removing the time annotation by the RDF statements. Those operators were first defined for relational data stream processing in (Arasu et al., 2006). For this reason, we maintain the original names and we redefine them to work in the RSP-QL setting. We start by defining the RStream operator.

Definition 20. Let Ψ be a time-varying sequence of solution mappings and $t \in T$ the evaluation time instant. We define **RStream** in the following way:

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

We define the **RStream evaluation semantics** as follows:

$$\begin{aligned} \mathit{eval}(SDS(G),\mathit{RStream}(L),t) = \\ \mathit{RStream}(\mathit{eval}(SDS(G),L,t),t) \end{aligned}$$

where L is a solution sequence.

The RStream operator is the simplest one among the three that we present in this section.

It takes as input a sequence of solution mappings $\Psi(t)$, and annotates each of them with the evaluation time t. This operator allows streaming out the whole answer produced at each evaluation iteration.

Definition 21. (IStream). Given a time-varying sequence of solution mappings Ψ, and two consecutive time instants t_{j-1} and t_j in the ET sequence (i.e. there is no time instant $t \in ET$ such that $t \in (t_{j-1} < t_j)^2$), we define the **IStream** operator as follows:

$$IStream(\Psi(t_{j-1}), \Psi(t_j), t_j) = \\ \{(\mu, t_j) | \mu \in \Psi(t_j) \wedge \mu \notin \Psi(t_{j-1})\}$$

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

$$eval(SDS(G), IStream(L), t_j) = \\ IStream(eval(SDS(G), L, t_{j-1}), \\ eval(SDS(G), L, t_j), t_j)$$

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

Definition 22. (DStream). Given a time-varying sequence of solution mappings Ψ, and two consecutive time instants t_{j-1} and t_j in the ET sequence, we define the **DStream** operator as follows:

$$DStream(\Psi(t_{j-1}), \Psi(t_j), t_j) = \\ \{(\mu, t_j) | \mu \notin \Psi(t_j) \wedge \mu \in \Psi(t_{j-1})\}$$

We define the **DStream evaluation semantics** as following:

$$eval(SDS(G), DStream(L), t_j) = \\ DStream(eval(SDS(G), L, t_{j-1}), \\ eval(SDS(G), L, t_j), t_j)$$

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

Query Form

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

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

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

Evaluation time instants

We defined ET as the sequence of time instants at which the evaluation occurs. It is an abstract concept which is key to the RSP-QL query model and its continuous-evaluation semantics, but it is hard to use it in practice when designing the RSP-QL syntax. In fact, the ET sequence is potentially infinite, so the syntax needs a compact representation of this set. Moreover, the

ET set could be unknown when the query is composed: the time instants on which the query has to be evaluated could depend on the data that is streaming through the RSP, e.g. the query should be evaluated every time the window content changes. In other words, query designers can be interested in associating the query evaluation to some relevant events, which can be known a priori (e.g. periodical evaluation) or not (e.g. status of the window content).

To address this issues, we introduce in RSP-QL the notion of *policy* to express the time instant set ET. The concept was initially proposed by Botan et al. in SECRET (Botan et al., 2010).

Definition 23. A **policy** P is a combination of one or more boolean conditions (shortly **strategy**, according to the SECRET model) which allows identifying the potentially infinite set of time instants ET. Each strategy is associated to a window and could set constraints to the window content or its parameters. Given a policy P, the **evaluation time instant set** ET_P is the set of time instants on which the policy in P is satisfied, i.e.

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

We can now indicate with $Q = (SE, SDS, ET_P, QF)$ the RSP-QL query where ET is represented through the policy P. In the remaining of this section, we describe the four SECRET strategies:

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

WC Window Close: the window reports if the active window closes.

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

P Periodic: the window reports only at regular intervals.

Example 9. *(cont'd) Let's define the policy P for the query described in Example 5: it should detect the shoppers recently spotted in nearby shops that offer instantaneous discount coupons. The query involves two streams, S_{nearby} and S_{coupon}, and for each of them there is an associated sliding window $\mathbb{W}_1(S_{nearby})$ and $\mathbb{W}_2(S_{coupon})$ with parameters ($\alpha_1 = 5$, $\beta_1 = 2$, $t_1^0 = 1$) and ($\alpha_2 = 2$, $\beta_2 = 2$, $t_2^0 = 0$). As policy P, we set the Window Close and the Non-empty Content to $\mathbb{W}_2(S_{coupon})$:*

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

As result, the ET_P set contains pair time instants such that the window content of $\mathbb{W}_2(S_{coupon})$ is not empty.

RSP-QL query evaluation

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

Definition 24. Let Q a continuous query $Q = (SE, SDS, ET, QF)$, where SE is an RSP-QL algebraic expression, SDS is an RSP-QL dataset (Definition 14), ET is the sequence of evaluation time instants (Definition 23) and QF is the query form. The **continuous evaluation of** Q produces an output $Ans(Q)$, and it is computed in the following way: for each $t \in ET$,

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

$$eval(SDS(G_0), SE, t)$$

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

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

Example 10. *(cont'd) We can now sum up and formalise the query described in Example 5. The query Q^{ex} is defined through the four parameters (SE, SDS, QF, ET). The RSP-QL dataset of Q^{ex} is the one described in Example 8:*

$$\begin{aligned} SDS =& \{G_0 = G_{shops}, \\ & (w_1, \mathbb{W}_1(S_{nearby})), (w_2, \mathbb{W}_2(S_{coupon}))\}, \end{aligned}$$

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

Instead of the RSP-QL algebraic expression, we write the `WHERE` *clause in a SPARQL-like syntax:*

```
Rstream{
  GRAPH w_1 { ?shopper :isNearby ?shop  }
  GRAPH w_2 { ?shop_owner :offers ?coupon }
  ?shop_owner :owns ?shop
}
```

The clause matches shoppers nearby shops whose owner is offering an instantaneous discount coupon. The sequence of solution mappings are streamed out using the Rstream operator.

For simplicity, we set the query form as `SELECT` *and we assume to project all the variables. The evaluation time instant set is defined through the policy:*

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

Now, we can apply Definition 24 and determine which is the unique correct answer of Q^{ex}. First, the ET set is determined; given the policy P, the input data and the sliding window definitions, it follows that the evaluation occurs at time 8 and 16. At these time instants, the algebraic expression is evaluated over the RSP-QL dataset, and the following timestamped solution mappings are computed:

Finally, the timestamped solution mappings are appended at the $Ans(Q^{ex})$ stream.

?shopper	?shop	?shop_owner	?coupon	timestamp
:carl	:a	:adam	"10% discount on ..."	8
:eve	:a	:adam	"10% discount on ..."	8
:diana	:b	:bob	"free coffee at ... "	16

Relaxing the no duplicate data assumption

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

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

EXPLAINING HETEROGENEITY OF RSP SYSTEMS

In this section we study the heterogeneity of the operational semantics of C-SPARQL, SPARQL$_{stream}$ and CQELS using the RSP-QL model, which is generic enough to capture their behaviour. This characterisation provides a basis for formally analyzing RSP engines, e.g.

modeling interoperability among RDF stream processors, or for defining correctness of query results, as shown later in the correctness section. It is worth to note that this analysis involves the query models of C-SPARQL, SPARQL$_{stream}$ and CQELS, as well as their query language syntaxes and their implementations. In fact, there are parameters of the RSP-QL model that are constrained by their query language syntax, while other parameters are encoded in the system implementations.

All those systems support (a subset of) SPARQL 1.1 operators (Zhang, Pham, Corcho, & Calbimonte, 2012). They are heterogeneous in the way they process the RDF streams and report the results. The following table summarises the comparison of the systems and highlights their differences.

Feature	**C-SPARQL**	**CQELS**	**SPARQL** $_{stream}$
Sliding window parameters	α and β	α and β	α and β
RSP-QL dataset	$G_{\mathbb{W}}$s in G_0	named $G_{\mathbb{W}}$s	$G_{\mathbb{W}}$s in G_0
Evaluation time instants	Window close and Non-empty content	Content-change	Window close and Non-empty content
Streaming operator	Rstream	Istream	Rstream, Istream and Dstream

Sliding window operator for C-SPARQL, CQELS and SPARQL$_{stream}$

RSP-QL defines the sliding window operator through three parameters: width (α), slide (β) and t^0. The query models of C-SPARQL, SPARQL$_{stream}$ and CQELS only allow controlling the width and slide of windows. The t^0 parameter is managed internally by systems and the query language does not provide syntactic constructs to constrain it. The query designer cannot determine when the first window of the sliding window opens: each sliding window can start at different time instants, and consequently, the system can produce different outputs.

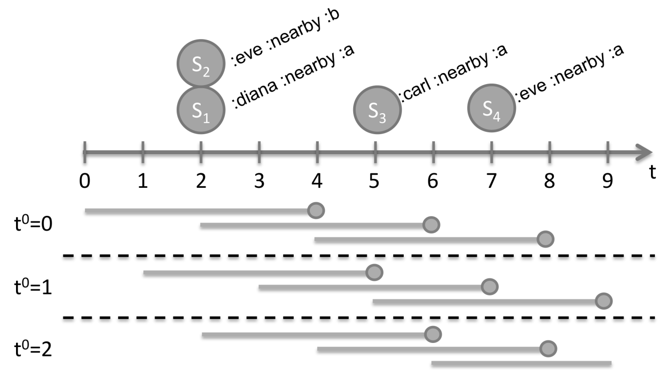


Figure 4. : Sliding windows with different t^0 values

Example 11. *Let consider a C-SPARQL query that defines a sliding window $\mathbb{W}$ through the parameters $(\alpha = 5, \beta = 2)$. The sliding window is applied to the S_{nearby} stream. The t^0 parameter cannot be explicitly defined, and the sliding window can open at different time instants, as shown in Figure 4. This fact influences the portion of the stream that is captured by the sliding window operators: if we focus on the first window of each sliding window operator, we can notice that in the first case ($t^0 = 0$) it contains the elements s_1 and s_2, in the second case ($t^0 = 1$) it contains the elements s_1, s_2 and s_3, while in the third case ($t^0 = 2$), the first window captures s_3 only.*

RSP-QL dataset for C-SPARQL, CQELS and SPARQL$_{stream}$

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

In CQELS, a named time-varying graph is associated to each window; window content can be accessed using the `STREAM` operator that is analogous to the SPARQL `GRAPH` one. It is not possible to put the time-varying graph generated from the sliding window in the default graph of the dataset.

C-SPARQL does the opposite: its query language does not allow to name the time-varying graphs computed by the sliding windows. As a result, all the graphs computed by the sliding windows are merged and set as default graph. In SPARQL$_{stream}$, similarly, named stream graphs can be declared but not used inside the query body. Therefore, graphs derived by sliding windows are logically merged in the default graph of the query dataset.

Even if a comparison among the expressiveness of the RSP system query languages if out of scope of this paper, we can make some considerations in order to explain how the shape of the dataset influences these systems. On the one hand, it may be considered easier to write queries in C-SPARQL and SPARQL$_{stream}$ than in CQELS: all the sliding windows are declared before the `WHERE` clause and the data from the streams is available in the default graph. On the other hand, CQELS allows to write more complex queries, such as queries with multiple sliding window over the same stream.

Example 12. *In Example 8, the following dataset is defined:*

$$SDS = \{G_0 = G_{shops}, (w_1, \mathbb{W}_1(S_{nearby})), (w_2, \mathbb{W}_2(S_{coupon}))\}$$

This dataset can be created in CQELS: its query language allows accessing window contents through the identifiers w_1 and w_2. C-SPARQL and SPARQL$_{stream}$ do not allow accessing named time-varying graphs generated by sliding windows so the above dataset cannot be managed by these system.

Let's now consider this dataset:

$$SDS = \{G_0 = G_{shops} \cup G_{\mathbb{W}_1} \cup G_{\mathbb{W}_2}\},$$

where $G_{\mathbb{W}_1}$ and $G_{\mathbb{W}_2}$ are the time-varying graphs computed respectively by $\mathbb{W}_1(S_{nearby})$) and $\mathbb{W}_2(S_{coupon})$. In this case, C-SPARQL and SPARQL$_{stream}$ can manage this dataset, while CQELS cannot – it does not allow adding the window contents to the default time-varying graph.

Evaluation time instants in C-SPARQL, CQELS and SPARQL$_{stream}$

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

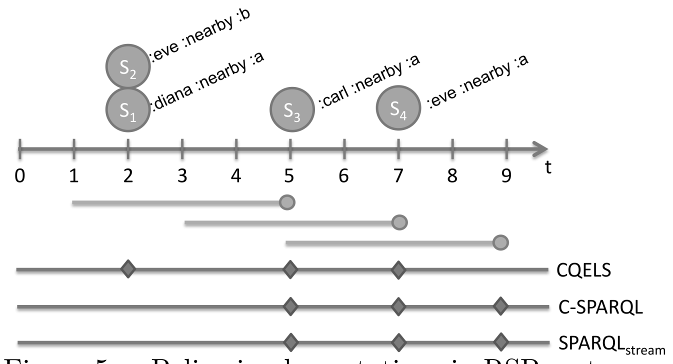


Figure 5. : Policy implementations in RSP systems

As analysed in (Dell'Aglio, Calbimonte, Balduini, Corcho, & Della Valle, 2013), C-SPARQL and SPARQL$_{stream}$ adopt a Window Close and Non-empty Content policy to the windows of the query, while CQELS implements the Content-Change policy (it evaluates the query as every time new statements enter the window). It follows that the systems build different evaluation time instant sets, and consequently they stream out new results at different time instants.

Example 13. *Let's consider the sliding window $\mathbb{W}$ in Figure 5: it is over S_{nearby} and it is defined through $(\alpha = 5, \beta = 2, t^0 = 1)$. The lower part of the figure shows the effect of different policies on the set time instants the evaluation occurs: each diamond represent an evaluation time in the relative RSP system. While CQELS evaluates the query as soon as it receives new timestamped RDF statements, C-SPARQL and SPARQL$_{stream}$ follow a regular pattern: they report every time the windows close (except for the cases where windows are empty).*

Streaming operators

The last feature on which we focus is the streaming operator support. Previously, we defined three streaming operators: Rstream, Istream and Dstream, but only SPARQL$_{stream}$ supports all of them. C-SPARQL permits only the Rstream operator (it streams out the whole output at each evaluation), while CQELS admits only the Istream one (it streams out only the new statements).

C-SPARQL answers can be verbose as the same solution mapping could be in different portions of the output stream computed at different evaluation time instants. It is suitable when it is important to have the whole SPARQL query answer at each step. CQELS streams out the difference between the timestamped set of mappings computed at the last step and the one computed at the previous step. Consequently, answers are usually short (they contain only the difference) and it is a good solution when data exchange is expensive.

CORRECTNESS OF RDF STREAM PROCESSING SYSTEMS

Finally, in this section we address the correctness assessment issue, to show one of the possible uses of RSP-QL. The question we investigate is: *given an RSP system, an input dataset and a query, is its output correct?* In the previous sections, we described all the elements we need to cope with this: First, we defined the RSP-QL model, a query model able to determine a unique answer given a RSP-QL dataset and a RSP-QL query. Next, we explained how existing systems are captured by the RSP-QL model. In particular, some parameters of the RSP-QL model are *hidden* in the systems, due to constraints in the query language syntaxes and in their implementations.

Even if those parameters cannot be controlled, it is necessary to take them into account to determine if the answer is correct. Intuitively, a query q for an RDF Stream Processing system is a *partially defined* RSP-QL query, i.e. some of its parameters are undefined or implicitly defined. Every RSP system can be analyzed, so that it is possible to determine the values that those parameters assume, and consequentially derive a set of RSP-QL queries. Each of those queries generates a different (but correct) answer: if the result of the query q matches one of them, we can state that it is correct (as we will see next). Anyway, to assess whether an RDF Stream Processing system behaves correctly or not, some assumptions and approximations are required, due to the infinite nature of input streams (as we will detail in the correctness in practice subsection). We then conclude by presenting a software developed to automatically assess the correctness of RSP engine answers and carried in the CSRBench suite.

Correctness based on RSP-QL

The RSP-QL model is built to capture the query model and the operational semantics of existing RSP systems. This model is able to reproduce each result of the targeted systems. Anyway, as explained in the previous section, each system C has a different behavior, and the relative queries are not fully-defined RSP-QL queries. It is possible to exploit that analysis to derive a set of well-defined RSP-QL queries and determine if the answer is correct.

Definition 25. Let us denote with C an RSP system (C-SPARQL, CQELS or SPARQL$_{stream}$), with q a continuous query of C, and with $\mathbb{Q}$ the set of RSP-QL queries that can be derived by q (shortly $\mathbb{Q} = rspqlQueries(q, C)$). The answer $Ans_C(q)$, produced by continuously executing q on C, is **correct** with regards to the RSP-QL model iff exists a query $Q \in \mathbb{Q}$ such that $Ans_C(q) = Ans(Q)$.

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

Given an RDF stream processor C and a query q, the function *rspqlQueries* generates a set of RSP-QL queries $\mathbb{Q}$. For each query Q of $\mathbb{Q}$, the algebraic expression and the query form are the same of q. The set of evaluation time instants is set accordingly to the report policy of C.Regarding the dataset, we analysed two main differences between the RSP-QL query model and the RDF stream processor systems. In the previous section we showed that each system has syntactical constraints that limit the shape of the dataset, and then we discussed the fact that t^0 instants are system-related parameters and they are out of control of the query designer. While the former does not influence the number of queries in $\mathbb{Q}$, the latter generates multiple alternative sliding window operators, and consequently, different queries. The sliding windows operators of the queries in $\mathbb{Q}$ are built in the following way:

- for each sliding window $\mathbb{W}_i$ in q, determine the set $T^0_{\mathbb{W}_i}$ of possible t^0 instants – the time instants on which the first window of $\mathbb{W}_i$ opens:
$$T^0_{\mathbb{W}_i} = \{a, a+1, \ldots\},$$
where a is the query registration time.
- compute the set Z with the combinations of the starting time instants of the sliding windows in q:
$$Z = \prod_{i=1}^{n} T^0_{\mathbb{W}_i}$$

Each element of the Z set is a vector z of dimension n (where n is the number of windowing operators in q). The number of vectors in Z is the number of queries that *rspqlQueries* generates. In particular, given the j-th vector $z_j \in Z$, the query Q_j has a dataset with n sliding windows. For each value z_{ji} of the vector z_j ($i \in [1, n]$), a sliding window $\mathbb{W}_i$ is defined as:
$$\mathbb{W}_i(\alpha_i, \beta_i, z_{ji}),$$
where (α_i, β_i) are the width and slide parameters that define the i-th sliding window of q.

Correctness in practice

Definition 25 gives a notion of correctness assessment for RSP systems. The idea is to compare the output of a system with the ones generated through the RSP-QL query model, and check if the answers match. Anyway, the notion is theoretical and it is not feasible in reality:

1. the continuous and infinite nature of data streams do not allow determining whether two answers match. The infinite input lengths imply undecidability in the matching problem – it is possible to determine if inputs are different, but it requires infinite time to determine if they are equal;
2. the *rspqlQueries* function generates a RSP-QL query for each possible combination of t^0 values, but those combinations are infinite (sliding windows can start to work at any time instant).

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

Theorem 1. Let's consider two sliding window operators $\mathbb{W}'$ and $\mathbb{W}''$, defined respectively through (α, β, a) and $(\alpha, \beta, a+n\beta)$, where $a \in T$ is a time instant and n a natural number. The sliding windows are applied to a generic stream S and generate two time-varying graphs $G'_{\mathbb{W}}$ and $G''_{\mathbb{W}}$. For each time instant $t \in T$ on which $G''_{\mathbb{W}}$ is defined, i.e. $t \geq a + n\beta$, it holds:
$$G'_{\mathbb{W}}(t) = G''_{\mathbb{W}}(t)$$

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

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

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

Exploiting the property, we can bound the set:

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

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

We need also to introduce a constraint in Definition 25, to set a time instant on which the correctness assessment starts.

Definition 26. The answer $Ans_C(q)$, produced by continuously executing q on C, is **correct** with regards to the RSP-QL model iff from a time instant t_s, there exists a query $Q \in \mathbb{Q}$ such that $Ans_C(q) = Ans(Q)$. t_s is:

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

This new definition sets a time instant on which the comparison starts. In particular, it cuts off the transient state of the query answering (the registration and the set up of the query), and it focus on the stable phase of the process. It is a suitable assumption, due to the fact that in stream processing it is common to focus on behaviour of the system when system is in a stable state (Arasu et al., 2004).

Regarding the problem of the infinite input length, it is necessary to bind the length of the stream. This bound has to guarantee that the inputs are long enough to discover long-term running problems (burn-in tests (Kuo, Chien, & Kim, 1998)) of the RDF stream processors, e.g. windows misalignment over the data streams.

RSP-QL implementation in CSRBench

RSP-QL is at the basis of the *oracle* provided by CSRBench (Dell'Aglio, Calbimonte, et al., 2013) and used to automatically verify the results of the tests. CSRBench is an extension of SRBench (Zhang et al., 2012) to tackle the correctness assessment of RSP engine outputs. In addition to the oracle, CSRBench supplies data streams and a set of parametrised queries[3] to be used to perform tests.

The oracle is built on the top of the Sesame framework and it is available as open source project[4]. The idea behind its architecture is to simulate the evaluation of a continuous query over a stream by using a SPARQL 1.1 engine and an a RDF store. The goal of the oracle is to verify if the answer provided by an RSP engine is correct: to assess it, the oracle adopts the correctness as defined in Definition 25. The current prototype manages RSP-QL datasets composed by one time-based sliding window over a stream; it supports the whole SPARQL 1.1 query language, and it implements the three R2S operators Rstream, Istream and Dstream. Given the input stream S, a query q, an RSP engine C, and the result $Ans_C(q)$ provided by C, the oracle verifies (off-line) if $Ans_C(q)$ is correct. To do it, the oracle stores the RDF stream S in a RDF store and transforms the continuous query q in set of SPARQL 1.1 queries. Then, it evaluates the queries over the data in the RDF store, simulating the continuous evaluation of q and generating the set of possible answers that can be produced: if one of them matches with $Ans_C(q)$, the result of C is correct.

The oracle operates in two main stages, as depicted in Figure 6: (i) the set up of the dataset, and (ii) the execution and comparison of the results.

The main goal of the first stage is to produce a set of RDF graphs to simulate the real RDF stream S and the sliding windows over it. To do it, the oracle creates a metadata RDF graph g_m and a set of data RDF graphs $\{g_t\}$, defined as follows. The RDF stream S is composed by a sequence of timestamped triples of the form (d, t). For each timestamp t in S, a corresponding RDF graph g_t is created, and the following metadata triple is added in g_m:

$$g_t \text{ :}hasTimestamp\ t$$

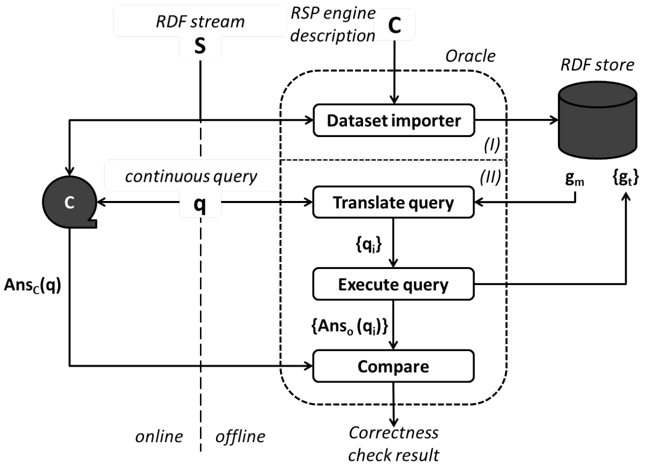


Figure 6. : Oracle for RDF Stream query results correctness checking.

Finally, for each t, the triples of S with timestamp t are imported into g_t, i.e. $g_t = \{d \mid (d,t) \in S\}$.

Once the RDF stream is stored as a set of RDF graphs, the oracle can execute the continuous query q over it in the second step. q is transformed into a set of SPARQL 1.1 queries $\{q_i\}$, one for each possible $z_i \in Z$, i.e. one for each starting time instant (i.e. one for each element of the set Z). Due to the fact that q has only one sliding window $\mathbb{W}$ with slide parameter β, it follows:

$$Z = T^0_{\mathbb{W}} = \{a, a+1, \dots, a+\beta-1\}$$

Next, q_i is executed: q_i is associated with a sequence of RDF datasets representing the window contents: the sequence is defined accordingly to the t^0 value of z_i, to the sliding window definition and the report policy of C. Given the evaluation time t_e, the selection of the current window content is made by selecting one or more graphs g_t, one for each time instant t in the active window at t_e. The results of the evaluation are then combined as required by the streaming operator of q. This execution simulates a continuous query evaluation over a limited amount of time. Finally, the result produced by C is compared with the results produced by the oracle: if there is a query q_i such that $Ans_C(q) = Ans_o(q_i)$, then the oracle produces a positive result.

CONCLUSIONS

Pervasive and ubiquitous computing is gaining widespread adoption, and it is one of the ingredients for enabling the Web of Things. In this spectrum, semantics plays a key role in enabling interoperability and facilitating discovery and accessibility of data coming from sensors, mobile devices, people and other *'things'*. However, the existing standards for representing semantically annotated data on the Web lack the necessary foundations for supporting the dynamic nature of streaming data that is produced in the Internet of Things. In order to solve this problem, several extensions to the RDF model and the SPARQL query language have been produced in the past years, and these extensions have also been implemented by a corresponding number of systems (RSP systems). As we have detailed in this paper, the existing systems follow different operational semantics that make it hard to compare seemingly equivalent queries, and that make it difficult to assess the correctness of an RSP system in general.

In this paper, we presented RSP-QL semantics, a formal query model that extends SPARQL for evaluating continuous queries over RDF streams, whose correctness can be formally assessed and that can actually model existing RSP systems, namely C-SPARQL, CQELS and SPARQL$_{stream}$. The contributions of this work can be summarised as follows. First, we have formally extended RDF to include time annotations, which are necessary to define an RDF stream. Moreover, and beyond previous works, we have introduced the notion of instantaneous graphs, which help bridging the static RDF world with the RDF stream world for query processing. With this RDF stream model, we have defined the semantics of RSP-QL, a query language extension to SPARQL for RSP, whose correctness can be formally assessed and that can capture the processing model of existing systems, constituting a unifying and comprehensive model for RSP. Furthermore, we have shown in theory and in practice how this model can be used not only to explain the operational semantics of existing RSP systems, but also to help assessing the correctness of the results they provide. To do so, we have proposed a formulation for correctness and an abstract design of an *oracle* that uses it to determine if the actual continuous query results of a certain engine are correct or not.

The proposed formalisation constitutes a contribution to ongoing efforts in the semantic web community to provide standardised and agreed definition of extensions to RDF and SPARQL for managing data streams[5]. It also provides foundations for defining RSP benchmarks that take into account the often disregarded problem of correctness, and possible trade-offs with performance, as well as setting the initial steps towards RSP integration.

In the future, we envision a widespread adoption of RSP-based solutions in different domains, specially under the umbrella of the Internet of Things (IoT). The proposed model can certainly help providing the foundations for well-defined query processors that can interoperate through common query interfaces, even if they follow different architectural approaches. This will allow creating an ecosystem of RSP compatible systems that take advantage of semantics and Linked Data to interact. The challenges we foresee are manifold. So far we have only scratched the surface of an RSP-QL model that does not consider reasoning, which is one of the key advantages of using semantic models. Also, IoT data is typically expected to include unavoidable volumes of noisy data, which may perturb the notion of correctness of query answering. Also, in this work we have not taken into account external factors that may have a nontrivial impact over query processing, such as out-of order arrival of data items, arbitrary delays in query operators and other issues that may arrive in real-life systems under stress.

Acknowledgments

Supported by the Nano-Tera.ch Opensense2 project, the IBM faculty-award 2013 granted to prof. E. Della Valle and the IBM PhD Fellowship Award 2014 granted to D. Dell'Aglio.

REFERENCES

Anicic, D., Fodor, P., Rudolph, S., & Stojanovic, N. (2011). EP-SPARQL: a unified language for event processing and stream reasoning. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, & R. Kumar (Eds.), Www (p. 635-644). ACM.

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

Arasu, A., Cherniack, M., Galvez, E. F., Maier, D., Maskey, A., Ryvkina, E., ... Tibbetts, R. (2004). Linear Road: A Stream Data Management Benchmark. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, & K. B. Schiefer (Eds.), Vldb (p. 480-491). Morgan Kaufmann.

Babu, S., & Widom, J. (2001). Continuous Queries over Data Streams. SIGMOD Record, 30(3), 109-120.

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

Bolles, A., Grawunder, M., & Jacobi, J. (2008). Streaming SPARQL - Extending SPARQL to Process Data Streams. In S. Bechhofer, M. Hauswirth, J. Hoffmann, & M. Koubarakis (Eds.), Eswc (Vol. 5021, p. 448-462). Springer.

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

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

Chen, J., DeWitt, D. J., Tian, F., & Wang, Y. (2000). NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In W. Chen, J. F. Naughton, & P. A. Bernstein (Eds.), Proceedings of the 2000 acm sigmod international conference on management of data, may 16-18, 2000, dallas, texas, usa (p. 379-390). ACM.

Compton, M., Barnaghi, P. M., Bermudez, L., Garcia-Castro, R., Corcho, Ó., Cox, S., … Taylor, K. (2012). The SSN ontology of the W3C semantic sensor network incubator group. J. Web Sem., 17, 25-32.

Cugola, G., & Margara, A. (2012). Processing flows of information: From data stream to complex event processing. ACM Comput. Surv., 44(3), 15.

Cyganiak, R., Wood, D., & Lanthaler, M. (2014). RDF 1.1 Concepts and Abstract Syntax. http://www.w3.org/TR/rdf11-concepts/. Retrieved from `http://www.w3.org/TR/rdf11-concepts/`

Dell'Aglio, D., Balduini, M., & Della Valle, E. (2013). On the need to include functional testing in RDF stream engine benchmarks. In The 10th eswc 2013 conference workshops: Bersys2013, aimwd2013 and usewod2013.

Dell'Aglio, D., Calbimonte, J.-P., Balduini, M., Corcho, Ó., & Della Valle, E. (2013). On Correctness in RDF Stream Processor Benchmarking. In H. Alani et al. (Eds.), International semantic web conference (2) (Vol. 8219, p. 326-342). Springer.

Gutiérrez, C., Hurtado, C. A., & Vaisman, A. A. (2005). Temporal RDF. In A. Gómez-Pérez & J. Euzenat (Eds.), Eswc (Vol. 3532, p. 93-107). Springer.

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

Komazec, S., Cerri, D., & Fensel, D. (2012). Sparkwave: continuous schema-enhanced pattern matching over rdf data streams. In F. Bry, A. Paschke, P. T. Eugster, C. Fetzer, & A. Behrend (Eds.), Debs (p. 58-68). ACM.

Kuo, W., Chien, W.-T. K., & Kim, T. (1998). Reliability, yield, and stress burn-in: A unified approach for microelectronics systems manufacturing & software development. Springer.

Motik, B. (2012). Representing and querying validity time in rdf and owl: A logic-based approach. J. Web Sem., 12, 3-21.

Pérez, J., Arenas, M., & Gutierrez, C. (2009). Semantics and complexity of SPARQL. ACM Trans. Database Syst., 34(3).

Phuoc, D. L., Dao-Tran, M., Parreira, J. X., & Hauswirth, M. (2011). A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In L. Aroyo et al. (Eds.), International semantic web conference (1) (Vol. 7031, p. 370-388). Springer.

Pugliese, A., Udrea, O., & Subrahmanian, V. S. (2008). Scaling RDF with time. In J. Huai et al. (Eds.), Www (p. 605-614). ACM.

Urbani, J., Margara, A., Jacobs, C. J. H., van Harmelen, F., & Bal, H. E. (2013). DynamiTE: Parallel Materialization of Dynamic RDF Data. In H. Alani et al. (Eds.), International semantic web conference (1) (Vol. 8218, p. 657-672). Springer.

Zhang, Y., Pham, M.-D., Corcho, Ó., & Calbimonte, J.-P. (2012). Srbench: A streaming rdf/sparql benchmark. In P. Cudré-Mauroux et al. (Eds.), International semantic web conference (1) (Vol. 7649, p. 641-657). Springer.

FOOTNOTES

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

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

[3] The description of the queries is available at: `http://www.w3.org/wiki/CSRBench`

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

[5] Interested readers are invited to learn more visiting `http://www.w3.org/community/rsp/`