Towards a Unified Language for RDF Stream Query Processing

Daniele Dell'Aglio[1](✉), Jean-Paul Calbimonte[2](✉), Emanuele Della Valle[1], and Oscar Corcho[3]

[1] DEIB, Politecnico of Milano, Milano, Italy
{daniele.dellaglio,emanuele.dellavalle}@polimi.it
[2] EPFL, Lausanne, Switzerland
jean-paul.calbimonte@epfl.ch
[3] Ontology Engineering Group, Universidad Politécnica de Madrid, Madrid, Spain
ocorcho@fi.upm.es

Abstract. In recent years, several RDF Stream Processing (RSP) systems have emerged, which allow querying RDF streams using extensions of SPARQL that include operators to take into account the velocity of this data. These systems are heterogeneous in terms of syntax, capabilities and evaluation semantics. Recently, the W3C RSP Group started to work on a common model for representing and querying RDF streams. The emergence of such a model and its accompanying query language is expected to take the most representative, significant and important features of previous efforts, but will also require a careful design and definition of its semantics. In this work, we present a proposal for the query semantics of the W3C RSP query language, and we discuss how it can capture the semantics of existing engines (CQELS, C-SPARQL, SPARQL$_{stream}$), explaining and motivating their differences. Then, we use RSP-QL to analyze the current version of the W3C RSP Query Language proposal.

1 Introduction

RDF Stream Processing (RSP) systems allow querying streams of RDF data, extending the SPARQL language with operators that can handle the highly dynamic and volatile nature of these data sources [1,3,6,10]. 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. However, 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.

Initiatives have started with the goal of proposing a common model and query language for processing RDF Streams, converging in the RSP Community Group of the W3C[1]. The emergence of such a model is expected to take the

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

F. Gandon et al. (Eds.): ESWC 2015, LNCS 9341, pp. 353–363, 2015.
DOI: 10.1007/978-3-319-25639-9_48

most representative, significant and important features of previous efforts, but will also require a careful design and definition of its semantics. In this context, it is essential to lay down the foundations of formal semantics for the standardized RSP query model, such that we consider beforehand the notions of correctness, continuous evaluation, evaluation time, and operational semantics, to name a few.

To address this challenge, we have previously proposed RSP-QL [9], a unifying formal model for representing and querying 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 [2] and SECRET [4]. This model, which already explains the heterogeneous semantics of existing RSP systems, can be used as a basis for the current RSP Group standardization effort. In this paper, we show that the new language proposed in the RSP Group are covered by the RSP-QL model, therefore providing a well-founded semantics for it. We also show that this new language allows covering cases that previous RSP languages are unable or partially able to address.

As running example, consider a social network micro-blogging stream, which contains microposts emitted by users on different topics. Such stream contains timestamped sets of RDF triples that represent posts, their authors, topics, etc., as in the following RDF stream snippet:

```
1 :post12 sioc:has_creator :susan [10]
2 :post12 sioc:topic :rsp2015 [10]
3 :post12 sioc:content ‘‘Workshop just started" [10]
```

Listing 1. RDF stream elements: each RDF stream is enriched with a time instant representing its validity time.

The task we aim at solving is the search of the emerging topics on this stream. One way of characterizing emerging topics is by finding out those which are frequently appearing lately, and less before. This apparently simple query contains some interesting elements that reveal differences among existing RSP languages, and challenge some of their capabilities. This is first, due to the fact that it requires looking at the same stream from two different perspectives: in the one hand it needs to keep track of very recent topics, while on the other hand needs to be aware of a longer time span, so that it can make sure that the new topics were not present before. Moreover, as we will see later, current RSP languages have implicit assumptions on how the results of a continuous query are streamed out, and how they react to changes on the sliding windows. We showed previously [9] that the RSP-SQL model is capable of covering these cases, while – as we will see next – current systems cannot. We also show that existing RSP languages present limitations that are partially solved by the current proposed language of the RSP Group, and that the latter is also covered by the RSP-QL model.

The remainder of this paper is structured as follows. We introduce the RSP-QL model in Sect. 2, including its main definitions. In Sect. 3, we provide a summary of the main semantic differences between existing RSP languages. Afterwards, we compare the syntactical limitations of these languages, compared to RSP-QL model, in Sect. 4. Section 5 is dedicated to explaining how the language proposed by the W3C RSP Group covers some of these limitations, and we show that it is also covered by the RSP-QL model. Finally, we conclude and provide final remarks on Sect. 6.

2 RSP-QL Semantics

The main difference between RDF Stream Processing and traditional RDF/SPARQL processing is given by the time dimension. In RSP, time plays a main role, and it has to be taken into account in both the data and query models. In the following, we present extensions of those models, and we will use them in the remaining of the paper to analyze existing languages.

Data Model. The RDF data model does not take into account the time, as stated in the RDF 1.1 recommendation [13]:

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

For this reason, the RDF data model has to be extended to take into account the time dimension. We propose two different extensions, that bring data to be roughly classified in two classes: *RDF stream* and *background data*.

An **RDF stream** is a sequence of timestamped data items (d_i, t_i), where each d_i is a RDF statement[2] and t_i is the time instant associated to d_i:

$$S = ((d_1, t_1), (d_2, t_2), \ldots, (d_n, t_n), \ldots)$$

Given a RDF stream S, the time stamps are in a non-decreasing order (i.e. for each i, $t_i \leq t_{i+1}$). Consumers usually access RDF streams through push paradigms: they register themselves to the RDF stream producers, and they start to receive the new streamed data.

Background data identifies the data that does not change (static) or changes very slowly w.r.t. data stream rate (quasi-static), and it is usually used to solve more complex queries (e.g., combining a stream of micro-posts with the graphs of authors) [7]. Background data includes RDF data stored in SPARQL endpoints, RDF repositories and sets of RDF data (that are usually fetched by the query processor). In this case, the time dimension is pushed through the notions of **time-varying** and **instantaneous** graphs. The former captures the dynamic evolution of a RDF graph over time: a time-varying graph G is a function that maps time instants to RDF graphs

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

[2] In this work we consider the case where data items are RDF statements.

The latter is the value of G at a fixed time instant t: the instantaneous graph $G(t)$ identifies an RDF graph.

Query Model. The time dimension also affects the query model, moving the evaluation from one-time paradigm to a continuous one. While SPARQL allows to issue queries that are evaluated once, RSP-QL allows to register continuous queries (i.e. issued once) and evaluated multiple times. The answer of a continuous query is composed by listing the results of each evaluation iteration. We define RSP-QL queries as extension of SPARQL queries, in order to maintain backward compatibility with the SPARQL query model. 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.

A SPARQL query [12] is defined through a triple (E, DS, QF), where E is the algebraic expression, DS is the data set and QF is the query form. We extend this definition for RSP-QL: a RSP-QL query is defined through a quadruple (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, and QF is the Query Form. While the Query Form values are the same of SPARQL (i.e. `SELECT`, `CONSTRUCT`, `DESCRIBE` and `ASK`), dataset and algebraic expression are extended to take into account the time dimension.

ET is the set of time instants on which the evaluation occurs. This notion is useful for modelling the RSP-QL query, but it is worth to note that it is hard to use it in practice when designing the RSP-QL syntax or implementing the RSP engines. In fact, the ET sequence is potentially infinite, so the syntax needs a compact representation of this set. Moreover, ET could be unknown when the query is issued: the time instants on which the query has to be evaluated can depend on the data in the RDF stream, e.g. the query should be evaluated every time the window content changes. For this reason, we relate ET to policies, as defined in SECRET [4]. Policies allow to determine when the query has to be evaluated, e.g. evaluation can be periodical or can depend on the status of window content.

A dataset represents the data against which the algebraic expression is evaluated. Given that we moved from RDF graphs to time-varying graphs and RDF streams, the notion of dataset as in SPARQL needs to be extended accordingly. In particular, fixed an evaluation time instant $t \in ET$, we aim at having a SPARQL-compliant data set. That is, we need a way to move from time-varying graphs and RDF streams to RDF graphs. Regarding the former, we already introduced the notion of instantaneous graph, that identifies an RDF graph at time t; regarding the latter, we use the notion of sliding window to determine a subset of the RDF stream to be taken into account at time t. A **time-based sliding window** $\mathbb{W}$ takes as input a stream S and produces a time-varying graph $G_{\mathbb{W}}$. $\mathbb{W}$ 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. A sliding window generates a sequence of windows, i.e., portions of data items in

the stream that can be queried as RDF graphs. We can finally define a RSP-QL dataset SDS as a set composed by an optional default graph G_0, n named graphs (u_i, G_i) and m named sliding windows over $o \leq m$ streams $(w_i, \mathbb{W}_i(S_j))$:

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

An RSP-QL expression uses all the SPARQL operators. As explained above, fixed an evaluation time t, the RSP-QL dataset SDS can be converted in a SPARQL dataset, and consequently the SPARQL operators can be used in order to process it (additional details on the evaluation semantics can be found in [9]). Additionally, a new class of ***streaming** operators is introduced: they transform sequences of solution mappings in sequences of timestamped solution mappings. Those operators are required to prepare the part of the answer to be appended to the output stream. These operators have been first introduced in [2], and are named Rstream, IStream and Dstream. **Rstream** streams out the computed set of mappings at each step; its answers can be verbose as the same mapping could be in different portions of the output stream computed at different steps. It is suitable when it is important to have the whole SPARQL query answer at each step, e.g., discover popular topics in the last time period in a social network. **Istream** streams out the difference between the current set of solution mappings and the one computed at the previous step. In this case, answers are usually shorter than Rstream ones (they contain only the difference) and consequently this operator is used when data exchange is expensive. Finally, **Dstream** does the opposite of Istream: it streams out the difference between the solution mappings computed at the previous step and the current one.

3 Heterogeneity in RSP Engines

Existing RSP query languages have different underlying semantics, and even if their syntax is similar, these differences have fundamental consequences at query evaluation time. This analysis involves the query models of C-SPARQL, SPARQL$_{stream}$ and CQELS, as well as their query language syntaxes. In the case of C-SPARQL [3], the stream processor is built on top of Esper[3] and Jena, combining them to process windows over streams with the first, and SPARQL execution with the second. CQELS [10] has a completely native implementation aimed at achieving higher performance. Finally, SPARQL$_{stream}$ [5] adopts an ontology-based data access to stream processing engines through query rewriting.

[3] Esper: http://esper.codehaus.org/.

All these systems support a subset of SPARQL 1.1 operators [14] and they are heterogeneous in the way they process the RDF streams and report the results.

Some of the differences in RSP engines are reflected in how the query dataset is constructed and how the windows are declared. For instance, CQELS associates a named (time-varying) graph to each window in the query, and the window content is accessed with the `STREAM` clause, analogous to the `GRAPH` in SPARQL. However, it is not possible to declare the sliding window in such a way that its content is included in the default graph of the dataset. On the contrary, C-SPARQL does not allow to name the time-varying graphs computed by the sliding windows, but all the graphs computed by the sliding windows are merged and set as the default graph. Similarly, in SPARQL$_{stream}$ named stream graphs can be declared but not used inside the query body. This allows writing simpler queries in C-SPARQL and SPARQL$_{stream}$, as all sliding windows are declared before the `WHERE` clause and the data from the streams is available in the default graph. Nevertheless, this does not allow defining more complex queries, such as those with multiple sliding windows over the same stream, which is possible in CQELS.

Regarding the evaluation time of windows, the query models of C-SPARQL, SPARQL$_{stream}$ and CQELS allow controlling the width and slide of windows. However, they provide no way to determine the time when the first window opens (known as t^0 in [8]), as this parameter is managed internally by the systems. Another important but diverging aspect in available RSP systems, is related to the report policy and strategy, which are implementation-dependent. This is a major source of heterogeneity, as these systems do not allow explicitly specifying control policies and strategies in the query syntax. As analyzed in [8], 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, evaluating the query every time new statements enter the window. Finally, another important feature that is supported differently is the streaming operator, i.e. Rstream, Istream and Dstream. Only SPARQL$_{stream}$ actually supports them in its syntax. C-SPARQL implicitly uses only the Rstream operator, streaming out the whole output at each evaluation, while CQELS works only in Istream mode. As a result, C-SPARQL answers can be more verbose, as the same solutions can be present in the output stream, computed at different evaluation times. Conversely, CQELS streams out the difference between the set of mappings computed at the last and previous evaluation steps.

4 Syntactical Limitations in RSP Languages

The heterogeneity of existing RSP engines described previously is reflected by their syntaxes. Their different design choices brought differences in the RSP engines and in their execution models. In this section, we use the RSP-QL model and the running example described above to highlight those differences. The task we want to solve is the identification of all the most emerging topics in the last 10 min. Emerging topics are identified as those that appear at least a certain amount of times in the latest 10 min, and sensibly less in a longer time span of 120 min.

CQELS. First, we analyze CQELS. In Listing 2, we report the CQELS-QL query that models the task described above in the running example.

```
1  CONSTRUCT {?topic a :EmergingTopic}
2  WHERE{
3    STREAM :in [RANGE 120m STEP 10m] {
4      SELECT ?topic (COUNT(*) AS ?totalLong)
5      WHERE { ?m1 sioc:topic ?topic.}
6      GROUP BY ?tlong}
7    STREAM :in [RANGE 10m STEP 10m] {
8      SELECT ?topic (COUNT(*) AS ?totalShort)
9      WHERE { ?m2 sioc:topic ?topic. }
10     GROUP BY ?tshort}
11   FILTER (totalShort-totalLong/12 > threshold)
12 }
```

Listing 2. CQELS query to find the emerging topics

The query declares two sliding windows over the same input stream `:in`: the first, $\mathbb{W}_l^{CQ}$ (Line 3), has width $\alpha_l = 120\,\text{min}$ and slide $\beta_l = 10\,\text{min}$; the second, $\mathbb{W}_s^{CQ}$ (Line 7), has width and slide $\alpha_s = \beta_s = 10\,\text{min}$ (it is a *tumbling window*). Each sliding window contains a subquery to compute the topics and the total number of their appearances (respectively `?totalLong` and `?totalShort`). The emerging value is computed at Line 11: if this value is greater than a `threshold` value, then the topic is selected as emergent, and it is streamed out according to the `CONSTRUCT` clause at Line 1. The RSP-QL dataset of this query is the following:

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

The syntax of CQELS-QL brings to assign an implicit name to each sliding windows (in the example, $\mathbb{W}_l^{CQ}$ and $\mathbb{W}_s^{CQ}$). In other words, it is not possible to assign explicit identifiers to the sliding windows. In this way, the language gains in usability, but it forbids to add sliding windows contents to the default graph.

Another limit of CQELS is given by the *streaming operator: as explained above, CQELS uses an Istream operator to produce the output. That is, it cannot produce an Rstream with the whole result of each operator. In other words, the algebraic expressions of CQELS-QL always assume Istream as outer element of the algebraic expression.

C-SPARQL. The example query cannot be written in one C-SPARQL query, as the syntax of C-SPARQL does not allow to distinguish among multiple windows defined over the same stream. Let us consider the query in Listing 3, the RSP-QL dataset built by the query is the following:

$$SDS^{CS} = \{G_0 = \{\mathbb{W}_l^{CS}(\texttt{:in}), \mathbb{W}_s^{CQ}(\texttt{:in})\}$$

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

```
1 REGISTER STREAM :out AS
2 CONSTRUCT {?tshort a :EmergingTopic}
3 FROM STREAM :in [RANGE 120m STEP 10m]
4 FROM STREAM :in [RANGE 10m STEP 10m]
5 WHERE{
6   ?m sioc:topic ?topic.
7 }
```

Listing 3. C-SPARQL Query: the triple pattern is evaluated against the union of the two sliding windows

It is actually possible to solve the running example task through a network of three C-SPARQL queries. First, Q_1^{CS} and Q_2^{CS} process the input stream `:in` in order to process the number of topics in the long and in the short windows. Listings 4 shows Q_1^{CS}.

```
1 REGISTER STREAM :longStream AS
2 CONSTRUCT {?topic :totalLong ?totalLong}
3 FROM STREAM :in [RANGE 120m STEP 10m]
4 WHERE{
5   SELECT ?topic ((COUNT(?topic) AS ?totalLong)
6   WHERE{ ?m1 sioc:topic ?topic. }
7   GROUP BY ?topic
8 }
```

Listing 4. C-SPARQL Query Q_1^{CS}: it counts the number of topics in the previous 120 min

The query builds a stream `:longStream`, that brings the topics and the number of appearance of the topics in the last 120 min (according to the sliding window definition at Line 3). Similarly, query Q_2^{CS} (we omit it for brevity, but it is similar to Q_1^{CS} – it changes the window size, the name of the output stream and the property name in the `CONSTRUCT` close) builds a stream `:shortStream` with the topics and their number of appearance in the previous 10 min. Those streams are the input of query Q_3^{CS}, reported in Listing 5, which computes the trending value of the topics, and add the topic in the output stream `:out` if the emerging value is greater than the threshold one (Line 7). In this case, the output contains the whole list of topics, as C-SPARQL uses Rstream as *streaming operator.

```
1 REGISTER STREAM :out AS
2 CONSTRUCT {?tshort a :EmergingTopic}
3 FROM STREAM :longStream [RANGE 10m STEP 10m]
4 FROM STREAM :shortStream [RANGE 10m STEP 10m]
5 WHERE{
6   ?topic :countLong ?totalLong; :countShort ?totalShort.
7   FILTER (?totalShort-?totalLong/12 > threshold)
8 }
```

Listing 5. C-SPARQL Query Q_3^{CS}: computation of the trending topics

SPARQL$_{stream}$. The case of SPARQL$_{stream}$, is similar to the one of C-SPARQL. Named stream graphs can be declared but the names cannot be used inside the query body. Therefore, graphs derived by sliding windows are logically merged in the default graph of the query dataset. As stated before, the Rstream operator can be explicitly indicated in the query.

5 Analysis of the W3C RSP Query Language Proposal

In this section, we briefly analyze the language under development by the W3C RDF Stream Processing community group[4]. Listing 6 shows the query that captures the running example task.

```
1  REGISTER STREAM :out
2  AS CONSTRUCT RSTREAM{ ?tshort a :EmergingTopic }
3  FROM NAMED WINDOW :lwin ON :in [RANGE PT120M STEP PT10M]
4  FROM NAMED WINDOW :swin ON :in [RANGE PT10M STEP PT10M]
5  WHERE{
6    WINDOW :lwin{
7      SELECT ?topic (COUNT(*) AS totalLong)
8      WHERE { ?m1 sioc:topic ?topic. }
9      GROUP BY ?topic }
10   WINDOW :swin{
11     SELECT ?topic (COUNT(*) AS totalShort)
12     WHERE { ?m2 sioc:topic ?tshort. }
13     GROUP BY ?topic }
14   FILTER(?totalShort-?totalLong)/12 > threshold)
15 }
```

Listing 6. The running example modelled through the W3C RSP Query Language

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

First, the new language allows to declare the *streaming operator (Rstream, at Line 2). Moreover, the new language allows to build both CQELS and C-SPARQL data sets: it is possible due to the sliding windows declarations in the `FROM` clause, combined with the use of the `NAMED` keyword (Lines 3 and 5). Next, in the `WHERE` clause, the `WINDOW` keyword is used to refer to the content of the named sliding windows (similarly to the `GRAPH` keyword in SPARQL). The RSP-QL dataset built by the query is:

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

Nevertheless, this syntax is not enough to determine a unique query following th RSP-QL model. As we explained in Sect. 3, there is no explicit information to determine which is the report policy and when the sliding windows start to

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

work (i.e., the t^0 value). A possible solution for the latter problem can be the introduction of a `STARTING AT` command to express the t^0 value. Alternatively, the language could allow to define a pattern to express the t^0 value.

6 Conclusions

In this paper, we presented RSP-QL, a formal query model that extends SPARQL for evaluating continuous queries over RDF streams. We first used the model to inspect the query languages of three RSP engines, namely C-SPARQL, CQELS and SPARQL$_{stream}$. As we discussed, RSP-QL can capture the semantics of those different engines and languages. Having well-defined RSP engine models would enable interoperability through common query interfaces, even if the implementations architectural approaches.

We then used RSP-QL to discuss the language under development at the W3C RSP Community Group. On the one hand, we provided evidence that the new language overcomes some limitations of C-SPARQL, CQELS-QL and SPARQL$_{stream}$; on the other hand, it still lacks some features that could lead in misinterpretations and in different implementations. We strongly believe that those aspects need to be addressed at a syntactic or and semantic level, in order to guarantee that a query is associated to one RSP-QL query. This would guarantee the possibility of determining a unique answer given the query and the data. In this sense, RSP-QL aims at constituting a contribution to ongoing efforts in the Semantic Web community to provide standardized and agreed definition of extensions to RDF and SPARQL for managing data streams.

The RSP-QL model can be used, not only to characterize and define new RDF stream query languages, but also to define and develop new tools and optimizations in RSP systems. As an example, in [8] we use RSP-QL to provide foundations for defining RSP benchmarks that take into account the often disregarded problem of correctness in stream processing. RSP-QL can also be used to understand the behavior and capabilities of RSP engines, from theoretical to practical perspectives.

Several challenges are in the scope of future works around the RSP-QL model. The current version of the model focuses on window-based continuous query languages, but other paradigms can also be studied, such as those inspired in Complex Event Processing [1]. This may include the need for studying intervals on RDF streams and additional operators such as sequences. Furthermore, it might be worth considering the possibility of implementing an engine that follows RSP-QL, and validate the execution model. We also foresee to include stream reasoning in RSP-QL, currently absent in the model, which is one of the key features of Semantic Web systems [11]. We are convinced that a well-defined and unified RSP query language will contribute to the overall goal of establishing a model that is both well-founded and applicable in real RSP systems.

Acknowledgments. This research is partially supported by the IBM Ph.D. Fellowship Award 2014 granted to D. Dell'Aglio.

References

1. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: Proceedings of 20th International Conference on World Wide Web WWW 2011, pp. 635–644 (2011)
2. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language : semantic foundations. VLDB J. **15**(2), 121–142 (2006)
3. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: a continuous query language for RDF data streams. IJSC **4**(1), 3–25 (2010)
4. Botan, I., Derakhshan, R., Dindar, N., Haas, L., Miller, R.J., Tatbul, N.: Secret: a model for analysis of the execution semantics of stream processing systems. PVLDB **3**(1), 232–243 (2010)
5. Calbimonte, J.-P., Corcho, O., Gray, A.J.G.: Enabling ontology-based access to streaming data sources. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 96–111. Springer, Heidelberg (2010)
6. Calbimonte, J.P., Jeung, H., Corcho, O., Aberer, K.: Enabling query technologies for the semantic sensor web. IJSWIS **8**(1), 43–63 (2012)
7. Dehghanzadeh, S., Dell'Aglio, D., Gao, S., Della Valle, E., Mileo, A., Bernstein, A.: Approximate continuous query answering over streams and dynamic linked data sets. In: Cimiano, P., Frasincar, F., Houben, G.-J., Schwabe, D. (eds.) ICWE 2015. LNCS, vol. 9114, pp. 307–325. Springer, Heidelberg (2015)
8. Dell'Aglio, D., Calbimonte, J.-P., Balduini, M., Corcho, O., Della Valle, E.: On correctness in RDF stream processor benchmarking. In: Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J.X., Aroyo, L., Noy, N., Welty, C., Janowicz, K. (eds.) ISWC 2013, Part II. LNCS, vol. 8219, pp. 326–342. Springer, Heidelberg (2013)
9. Dell'Aglio, D., Della Valle, E., Calbimonte, J.P., Corcho, O.: RSP-QL semantics: a unifying query model to explain heterogeneity of RDF stream processing systems. IJSWIS **10**(4), 17–44 (2015)
10. Le-Phuoc, D., Dao-Tran, M., Xavier Parreira, J., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 370–388. Springer, Heidelberg (2011)
11. Margara, A., Urbani, J., van Harmelen, F., Bal, H.E.: Streaming the web: reasoning over dynamic data. J. Web Sem. **25**, 24–44 (2014)
12. Prud'hommeaux, E., Harris, S., Seaborne, A.: SPARQL 1.1 Query Language, March 2013. http://www.w3.org/TR/sparql11-query
13. Wood, D., Lanthaler, M., Cyganiak, R.: RDF 1.1 concepts and abstract syntax, February 2014. http://www.w3.org/TR/rdf11-concepts/
14. Zhang, Y., Duc, P.M., Corcho, O., Calbimonte, J.-P.: SRBench: a streaming RDF/SPARQL benchmark. In: Cudré-Mauroux, P., et al. (eds.) ISWC 2012, Part I. LNCS, vol. 7649, pp. 641–657. Springer, Heidelberg (2012)