What is the Semantics of Your SPARQL Extension?*

Harald Beck, Minh Dao-Tran, and Thomas Eiter

Institute of Information Systems, Vienna University of Technology
Favoritenstraße 9-11, A-1040 Vienna, Austria
{beck,dao,eiter}@kr.tuwien.ac.at

Abstract. We argue for the importance of formal semantics for the exploration and study of new SPARQL extensions; in particular, for RDF stream processing (RSP). As an exemplary show case, we study the languages C-SPARQL and CQELS, which are syntactically similar, but differ substantially in their semantics. To express and compare their semantics in precise terms, we utilize the logic-oriented framework LARS which offers a rule-based semantics similar to Answer Set Programming. Moreover, we emphasize the need for theory to advance language development on more rigorous grounds.

1 Introduction

The W3C recommendation for SPARQL demonstrates the community's success in reaching an agreement on a suitable query language for RDF data. By standardizing a language, one also establishes a common ground for further extensions such as those for querying and reasoning over streaming RDF data.

The well-known paper on the semantics and complexity of SPARQL [7] helped to advance the progress in the development of SPARQL and its standardization by providing a theoretical underpinning to precisely describe its formal semantics. This is useful to clarify previous ambiguities, but also allows for the comparison and the mutual inspiration with other approaches [9].

Nevertheless, similarly as SPARQL itself in its early days, extensions of SPARQL nowadays seem to first avoid a rigorous approach. Often, the syntax is adapted to informally describe the needs of a new use case, and according semantics are then given informally or operationally. From a pure engineering point of view, this is valuable, since certain new use cases can then be solved and empirically evaluated, e.g., with respect to runtime performance. However, from a scientific point of view, an informal description of the new system's behaviour alone is not satisfactory for multiple reasons.

First, in case the actual output of an engine does not correspond with the user's intuition, there is no means to figure out the source of the discrepancy. How could would one even define a bug without a specification, especially for corner cases?

Second, without formal definitions, the limits of a query language cannot be identified clearly. A smart developer might be able to come up with a query to solve a difficult use case. However, some queries might not be expressible at all with the given system.

* This research has been supported by the Austrian Science Fund (FWF) projects P24090, P26471, and W1255-N23.

Only a formal semantics allows one to decide with certainty which queries *cannot* be expressed. For those queries which can be expressed a formal analysis (e.g. complexity) presupposes a formal definition.

Third, maybe even more important for the development of an established query syntax, is the efficient evaluation by means of internal optimizations. Only when we know which queries are equivalent, we can transform a given one into another one which computes the same results, but may be evaluated faster.

Forth, your friend's research group might have similar ideas and come up with their own extension. All the new languages might look very similar but in fact express/compute something quite different. What are the differences, and what exactly do they have in common? How can we compare different approaches scientifically beyond vague conceptual observations?

While the specific capabilities or difference between different languages and systems might be intuitively clear for the expert, it would usually not be apparent to the novice who is looking for the right tool for his job. Moreover, from a scientific point of view, the landscape of knowledge can only be drawn with sharp lines if some formal language is used.

In this note, we give an example how informal query semantics may be formalized in a logic-oriented way, in order to analyze and compare them in precise terms. Similar to [9], which translates (part of) SPARQL to Datalog with negation as failure, we express RDF stream processing semantics in LARS [3], a logic-oriented framework for analyzing reasoning over streams. LARS offers rule-based reasoning similar to Datalog and Answer Set Programming. In particular, we show precisely that the syntactically very similar RDF stream processing approaches of C-SPARQL [2] and CQELS [8] differ strongly in their semantics.

2 RDF Stream Processing

RDF Stream Processing (RSP)[1] can be intuitively seen as extending querying RDF datasets with SPARQL to querying RDF streams with "continuous SPARQL." We use the following scenario to illustrate the above notions.

Illustrating scenario. The Sirius Cybernetics Corporation offers shop owners a real-time geo-marketing solution ($RTGM$) to increase their sales. $RTGM$ provides two services: (i) an application that allows shop owners to push instantaneous discount coupons to a server, and (ii) a free mobile App that fetches the coupons from shops near the phone, matches them with the preferences specified in the user's shopping profile, and delivers the matched coupons to the user. Alice and Bob own shops a and b that sell shoes and glasses, resp. At time point 10, Alice sends out a coupon for a 30% discount for men's MBT shoes. At time 15, Bob sends out a coupon for a 25% discount on Ray-Ban glasses.

Claire has the App installed on her mobile phone and is walking near shops a and b from time 18. She is neither interested in discounts on men's products nor discounts of less than 20%. Therefore, she will get only the discount from shop b.

[1] `https://www.w3.org/community/rsp/`

Using RDF, information in the running scenario can be represented as RDF graphs, i.e., set of triples of the form $(subject, predicate, object)$ in Turtle format[2] as follows:

$$
\begin{aligned}
G &= \{\texttt{:mbt :g\_classify :1. :rayban :g\_classify :0. ...}\} \\
g_1 &= \{\texttt{:a :offers :c}_1\texttt{. :c}_1\texttt{ :on :mbt. :c}_1\texttt{ :reduce :30.}\} \\
g_2 &= \{\texttt{:b :offers :c}_2\texttt{. :c}_2\texttt{ :on :rayban. :c}_2\texttt{ :reduce :25.}\} \\
g_3 &= \{\texttt{:claire :isNear :a. :claire :isNear :b.}\}
\end{aligned}
$$

Here, triples in G represent the static dataset about products. The first triple says that MBT shoes in the stores are men product and the second one means Ray-Ban glasses here are for female. On the other hand, g_1, g_2 contains information regarding the discounts pushed to $RTGM$ at time 10 and 15, respectively, while triples in g_3 inform the server about the shops that Claire is near to at time 18.

Now assume that that product information is stored at <http://products>, and triples in $g_1 \cup g_2$ are collected at <http://coupons-snapshot>. The following SPARQL query computes the relevant coupons for Claire at a single time point.

```
SELECT ?shop ?product ?percent
FROM <http://products> <http://coupons-snapshot>
WHERE { ?shop    :offers      ?coupon.   ?coupon  :reduce  ?percent.
        ?product :g_classify  ?gender.   ?coupon  :on      ?product.
        FILTER (?percent >= 20 && ?gender != 1) }
```

Q_0: One-shot query expressed in SPARQL

This query only works on static data, that is, when triples in $g_1 \cup g_2$ are stored at <http://coupons-snapshot>. To be able to query streaming data as in the illustrating scenario, RSP was introduced. The RSP community is growing and one can see several engines being developed in divergent approaches. We pick here the two representative engines in RSP, namely C-SPARQL [2] and CQELS [8], to show the differences.

Let us first talk about the data model of RSP. In principle, RSP augments the RDF data model with the temporal aspect by associating incoming data (RDF graphs) with timestamps. The community is discussing on a detail data model that allows different types of timestamps to be attached with an RDF graph.[3] For the purpose of illustrating the divergence in RSP engines, we are safe to use a simple data model where RDF streams can be seen as sequences of elements $\langle g : [t] \rangle$, where g is an RDF graph and t is a timestamp, ordered by the timestamps.

On top of this data model, *continuous queries* are defined to be registered on a set of input input streams and a background data, and continuously send out the answers as new input arrives at the streams. There are two modes to execute such queries: In *pull-based* mode, the system is scheduled to execute periodically independent of the arrival of data and its incoming rate; in *push-based* mode, the execution is triggered as soon as data is fed into the system. Continuous queries in C-SPARQL and CQELS are inspired by the Continuous Query Language (CQL) [1]. As CQL is based on SQL, the background data tables and input streams all have schemas. This makes it crystal

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

[3] https://github.com/streamreasoning/RSP-QL/blob/master/Semantics.md

clear to see which input tuple comes from which stream. On the other hand, as RDF is schema-less, it is not straightforward to get this distinction; RSP engines use different approaches to build the snapshot datasets for R2R evaluation [5]:

- C-SPARQL merges snapshots of the input streams into the default graph,
- CQELS directly accesses the content of the input streams by introducing a new "stream graph" pattern in the body of the query.

A continuous query to notify Claire with instantaneous coupons matching her preferences can be expressed in C-SPARQL and CQELS as follows.[4] For readability, we write `<coupons>` instead of `<http://coupons>`, etc.

```
SELECT ?shop ?product ?percent
FROM <products>
  STREAM <coupons>   [RANGE 30m]
  STREAM <locations> [RANGE 5m]
WHERE {
  ?shop    :offers     ?coupon.
  ?coupon  :reduce     ?percent.
  ?coupon  :on         ?product.
  ?user    :isNear     ?shop.
  ?product :g_classify ?gender.
  FILTER
  (?percent >= 20 && ?gender != 1)}
```

Q_1: Notification query in C-SPARQL

```
SELECT ?shop ?product ?percent
FROM <products>
WHERE {
  STREAM <coupons> [RANGE 30m] {
    ?shop    :offers   ?coupon.
    ?coupon  :reduce   ?percent.
    ?coupon  :on       ?product. }
  STREAM <locations> [RANGE 5m] {
    ?user    :isNear   ?shop. }
  ?product :g_classify ?gender.
  FILTER
  (?percent >= 20 && ?gender != 1)}
```

Q_2: Notification query in CQELS

Compared to Q_0 in SPARQL, here we take into account the input stream regarding the locations of the user. The `FROM` clause of Q_1 now applies a window of range $30m$ to a stream of coupons instead of fetching input from a static RDF graph. This means that all streaming triples which arrived in the last 30 minutes will be considered for querying. Similarly, a window for the last 5 minutes is applied on the stream of users' locations. These windows produce a so-called *snapshot* of incoming data for computation. Query Q_2 puts the streams of coupons, users' locations and the corresponding window to the `WHERE` clause and relates the streams with the patterns for matching with them.

To sum up, RSP engines are different in several aspects. The two representative ones diverge in the way they build the snapshot datasets for R2R evaluation, and more crucially, the execution modes (pull- vs. push-based). This makes comparing them not trivial. A logic-based approach to stream reasoning (presented next) allows us to fulfill this goal more conveniently.

3 LARS: A Logic-based Framework for Analyzing Reasoning over Streams

3.1 LARS in a Nutshell

LARS [3] is a logic-based approach recently introduced for the purpose of analyzing stream reasoning. In a nutshell, LARS provides a logic with window and temporal

[4] Note that a unifying syntax for RSP queries is still work in progress. Again, for illustrating purposes, we choose to use a stable, old syntax.

operators to handle streaming data. On top of this, a rule-based language is defined which can be seen as extension of Answer Set Programming [6] for streams.

Window operators $\boxplus$ collect a recent finite portion of the input stream (recent in terms of time or counting tuples) for further computation. Three temporal operators allow fine-grained control for temporal reference. Given a formula φ, $\Diamond\varphi$ (resp. $\Box\varphi$) holds, if φ holds at some (resp. all) time point(s) in the selected window, and $@_t\varphi$ holds if φ holds exactly at time point t in the window. Window operators can be nested, thus together with temporal operators they provide a compact means to express complex conditions on the input streams. In the following, we will demonstrate LARS by building up a LARS rule for the illustrating scenario. Technical details on LARS can be found in [3].

Let $\mathit{offer}(Sh, Pr, Pe)$ be a ternary predicate about discounts offered by shops, and $isNear(Sh)$ be a unary predicate providing information that the user is near a shop Sh. The window atom $\boxplus^{30}_{\tau}\Diamond \mathit{offer}(Sh, Pr, Pe)$ takes a snapshot of the last 30 time units of a stream and uses the $\Diamond$ operator to check whether an offer from shop Sh on product Pr with a discount of $Pe\%$ appeared in the stream during this period. Similarly, $\boxplus^{5}_{\tau}\Diamond isNear(Sh)$ does the same job to take a snapshot of size 5 time units of the shops near the user. Here, τ is the window type, specifying that the underlying window function applied on the input stream is a time-based window.

An example of nesting of window operators is $\boxplus^{30}_{\tau}\Box\, \boxplus^{5}_{\tau}\, \Diamond isNear(a)$. It holds if in the last 30 time units, the user is near shop a in every subinterval of 5 time units. This might indicate that she is very interested in some product provided by this shop.

Suppose we are given static background data that contains product information in a predicate of the form $g\_\mathit{classify}(Pr, Ge)$, where $Ge = 0$ (resp. 1) marks that product Pr is for women (resp., men). The following LARS rule amounts to the RSP queries in the previous section.

$$\begin{aligned} ans(Sh, Pr, Pe) \;\leftarrow\; & \boxplus^{30}_{\tau}\Diamond \mathit{offer}(Sh, Pr, Pe), \boxplus^{5}_{\tau}\Diamond isNear(Sh), \\ & g\_\mathit{classify}(Pr, Ge), Pe \geq 20, Ge \neq 1. \end{aligned}$$

This rule works as follows: the two window atoms provide offers announced in the last 30 minutes and the shops near the user within the last 5 minutes. Together with the gender classification of products provided by g_classify, only products not for men ($Ge \neq 1$) and have discount rate from 20% are concluded at the head with predicate ans.

The semantics of LARS programs (a set of LARS rules) is provided in terms of *answer streams*. Intuitively, the input to a LARS program is a data stream containing of extensional facts associated with time points. An answer stream at a time point must include the data stream, and might add timestamped intensional facts due to the application of LARS rules. The formal definition of answer streams [3] involves applying the FLP reduct on a LARS program based on a guess, as in ASP. There can be multiple answer streams at a time point as we allow default negation in the body of LARS rules.

As in our scenario, the LARS program having the single LARS rule above has one answer stream where at time point 18, $ans(b, rayban, 25)$ is concluded.

3.2 LARS as a Convenient Means to Compare RSP Engines

Before the introduction of LARS, comparing RSP engines was done at the practical level, meaning that the proposed benchmarking systems just compare the output of the engines without theoretical analysis of the difference in their semantics. The most recent work [5] in the RSP community on this topic just identified the difference in building the snapshot of C-SPARQL and CQELS.

With LARS, different stream processing/reasoning languages (not limited to just RSP languages) can be compared within the same framework. The achieved results can be used to compare existing languages with future ones as long as a translation of the new languages to LARS is materialized. This give us more reusability in contrast to working on an ad-hoc comparison between two RSP engines (say C-SPARQL and CQELS), and then start from scratch when a new engine with new syntax and semantics is introduced.

Furthermore, the model-based approach of LARS allows us to talk about semantics properties and construct formal conditions to identify situation when different engines agree on their output.

In the next section, we summarize our strategy and findings in using LARS to compare C-SPARQL and CQELS. Note that for a theoretical analysis, we adopt as in [5] the assumption that execution time of RSP engines is neglectable compared to the data rate of the input streams.

4 Capturing RSP Semantics with LARS for Analysis

In order to exploit LARS to analyze, compare the semantics of RSP engines, the first step is to provide translations from the respective RSP query languages into LARS. As RSP query languages (C-SPARQL and CQELS in particular here) are constructed based on SPARQL, we make use of a well-known translation from SPARQL to Datalog proposed in [9] and introduce additional LARS rules to deal with the streaming input. As a consequence, we end up with two translations which are slightly different in the way they treat the streaming input to build the snapshots to be evaluated by rules capturing the functionalities of SPARQL operators (rules that originally come from [9]).

Still, the execution modes pull- and push-based need to be captured. For this, we introduced additional, independent LARS rules that mimic these execution modes on any LARS program.

Combining the two above translations allow us to capture the semantics of C-SPARQL and CQELS. The comparison of their semantics can thus be broken down into two phases according to the two translations.

The first phase compares the operation of the two engines at the same time point, having the same input stream fed into them. Once the condition for the two engines to output the same results given this setting is identified, we can move to the next phase.

In the second phase, we just need to analyze the conditions to guarantee that the two pull- and push-based executions agree by finding the conditions that the two modes provide the same input to the engines. It turned out that this is the most tricky part to get the engines agreed. The main reason is that the engines are executed at different time

points. While the pull-based mode triggers one execution after a fix interval of time, the push-based execution can be triggered several times in the same interval according to the arriving of input in the stream.

Therefore, we introduced an *interval-based notion of agreement* between C-SPARQL and CQELS. Intuitively, they are said to agree within an interval iff the output returned by C-SPARQL at the end of the interval coincides with the union of the outputs reported by CQELS during the interval.

The necessary conditions found in [4] show that the engines only agree on a very restricted setting, with a special shape of the input stream: the stream needs to be off frequently at the beginning of the computing interval so that the input collected for pull-based execution at the end of the interval coincides with the input collected by push-based mode at every time point in the interval. This does not hold for dense stream where input tuples/triples arrive at almost every time point.

5 Conclusion

We capture two prominent RSP semantics within the logic-based framework LARS which provides a common base for formal comparisons. Based on the generic rule-based approach one can abstract from particular assumptions with regards to data representation, query syntax, or processing models. We showed that the syntactically similar query languages C-SPARQL and CQELS differ drastically in their output. While these differences might be intuitive for the RSP expert, they now can be understood precisely on formal grounds.

References

1. A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.
2. D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-SPARQL: a continuous query language for rdf data streams. *Int. J. Semantic Computing*, 4(1):3–25, 2010.
3. H. Beck, M. Dao-Tran, T. Eiter, and M. Fink. LARS: A logic-based framework for analyzing reasoning over streams. In *AAAI*, 2015.
4. M. Dao-Tran, H. Beck, and T. Eiter. Contrasting RDF Stream Processing Semantics. In *JIST*, 2015.
5. D. Dell'Aglio, E. D. Valle, J.-P. Calbimonte, and O. Corcho. Rsp-ql semantics: a unifying query model to explain heterogeneity of rdf stream processing systems. *IJSWIS*, 10(4), 2015.
6. T. Eiter, G. Ianni, and T. Krennwallner. Answer Set Programming: A Primer. In *RW*, pages 40–110, 2009.
7. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34:16:1–16:45, September 2009.
8. D. L. Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *ISWC (1)*, pages 370–388, 2011.
9. A. Polleres. From SPARQL to rules (and back). In *WWW 2007*, pages 787–796, 2007.