Towards Comparing RDF Stream Processing Semantics*

Minh Dao-Tran, Harald Beck, and Thomas Eiter

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

Abstract. The increasing popularity of RDF Stream Processing (RSP) has led to developments of data models and processing engines which diverge in several aspects, ranging from the representation of RDF streams to semantics. Benchmarking systems such as LSBench, SRBench, and CSRBench were introduced as attempts to compare different approaches. However, these works mainly concentrate on the operational aspects. The recent logic-based LARS framework provides a theoretical underpinning to analyze stream processing/reasoning semantics. Towards comparing RSP engines at the semantic level, in this paper, we pick two representative RSP engines, namely C-SPARQL and CQELS, and propose translations from their languages and execution modes into LARS. We show the faithfulness of the translations and discuss how they can be exploited to provide a formal analysis and comparison of RSP semantics.

Keywords: RDF Stream Processing, Linked-Stream Data, Semantics Comparison

1 Introduction

Within the Semantic Web research area, RDF Stream Processing (RSP) recently emerged to address challenges in querying heterogeneous data streams. This has led to developments of data models and processing engines, which diverge in several aspects, ranging from the representation of RDF streams, execution modes, to semantics [2, 15, 7, 6, 11, 18]. Thus, the RSP community[1] was formed to establish a W3C recommendation.

A standardization must start from seeing the differences between existing approaches and thus comparing RSP engines is an important topic. Initial empirical comparisons were carried out in SRBench [19] and LSBench [16]. The former defined only functional tests to verify the query languages features by the engines, while the latter measured mismatch between the output of different engines. Later on, CSRBench [9] introduced an oracle that pregenerates the correct answers wrt. each engine's semantics, which are then used to check the output returned by the engine. This approach however allows only partial comparison between engines by referring to their ideal counterparts.

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

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

Due to the lack of a common language to express divergent RSP approaches, the three works above could just look at the output of the engines and did not have further means to explain beyond the output what caused the difference semantically.

Recently, [10] proposed a unifying query model to explain the heterogeneity of RSP systems. It shows a difference between two approaches represented by C-SPARQL [2], $\text{SPARQL}_{\text{Stream}}$ [7] and CQELS [15], the representative engines in the RSP community. This work is based on extending SPARQL semantics to the stream setting.

Recently, a **L**ogic-based framework for **A**nalyzing **R**easoning over **S**tream (LARS) was introduced [5]. LARS can be used as a unifying language to which stream processing/reasoning languages can be translated. It may serve as a formal host language to express semantics and thus allows a deeper comparison that goes beyond mere looking at the output of the respective engines. In this paper, towards comparing RSP engines at the semantic level, we pick C-SPARQL and CQELS, and propose

- translations that capture the push- and pull- execution modes for general LARS programs, and
- translations from the query languages of C-SPARQL and CQELS to LARS based on a well-known translation from SPARQL to Datalog [17].

We show the faithfulness of the translations and discuss how they can be exploited to provide a formal analysis and comparison of RSP semantics.

2 Preliminaries

This section briefly reviews RDF, SPARQL, RSP, and LARS, which will be illustrated using the following running scenario inspired by [10].

Example 1 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 discount of less than 20%. Therefore, she will get only the discount from shop b. ∎

2.1 RDF and SPARQL

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

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

Example 2 (cont'd) Information in the scenario of Ex. 1 about products, offers from shops, and Claire's relative locations to shops can be stored in the following RDF graphs:

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

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

SPARQL [12], a W3C recommendation for querying RDF graphs, is essentially a graph-matching query language. A *SPARQL query* is of the form $H \leftarrow B$, where B, the *body* of the query, is a complex RDF graph pattern composed of basic graph patterns with different algebraic operators such as UNION, OPTIONAL, etc.; and H, the *head* of the query, is an expression that indicates how to construct the answer to the query [14]. However, SPARQL is not able to give answers under dynamic input as in the running scenario. For this purpose, we need RSP.

2.2 RDF Stream Processing

RDF Streams and Temporal RDF Graphs. These two notions are introduced to extend Linked Stream Data and Linked Data with temporal aspects:

1. An *RDF graph at timestamp* t, denoted by $G(t)$, is a set of RDF triples valid at time t and called an *instantaneous RDF graph*. A *temporal RDF graph* is a sequence $G = [G(t)], t \in \mathbb{N} = \{0, 1, 2, \ldots\}$, ordered by t.
2. An *RDF stream* $\mathcal{S}$ is a sequence of elements $\langle g : [t] \rangle$, where g is an RDF graph and t is a timestamp.

Example 3 (cont'd) An input stream $\mathcal{S}$ of our running scenario is a sequence of elements $\langle g_i : [t_i] \rangle$, where g_i, representing an offer, is of the form in Ex. 2 and t_i can be either (i) the time point when the offer is announced by a shop owner (application time), or (ii) the time point when g_i arrives at an RSP engine (system time). ■

Continuous Queries. Continuous queries are registered on a set of input streams and 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 follow the approach by the Continuous Query Language (CQL) [1], in which queries are composed of three classes of operators, namely stream-to-relation (S2R), relation-to-relation (R2R), and relation-to-stream (R2S) operators. In the context of RSP, S2R operators are captured by windows on RDF streams, R2R operators are resorted to SPARQL operators, and R2S operators converts "pure" SPARQL output after R2R into output streams.

As CQL is based on SQL, the background data tables and input streams all have schemas. This makes it crystal 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 [10]:

(B_1) C-SPARQL merges snapshots of the input streams into the default graph,
(B_2) CQELS directly accesses the content of the input streams by introducing a new "stream graph" pattern in the body of the query.

Example 4 (cont'd) A continuous query to notify Claire with instantaneous coupons matching her preferences can be expressed in C-SPARQL and CQELS as follows. 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

2.3 Logic-oriented view on Streams, Windows and Time Reference

We will gradually introduce the central concepts of LARS [5] tailored to the considered fragment. We distinguish *extensional atoms* $\mathcal{A}^{\mathcal{E}}$ for input data and *intensional atoms* $\mathcal{A}^{\mathcal{I}}$ for derived information. By $\mathcal{A} = \mathcal{A}^{\mathcal{E}} \cup \mathcal{A}^{\mathcal{I}}$, we denote the set of *atoms*.

Definition 1 (Stream) *A stream $S = (T, \upsilon)$ consists of a* timeline *T, an interval in $\mathbb{N}$, and an* evaluation function *$\upsilon\colon \mathbb{N} \mapsto 2^{\mathcal{A}}$. The elements $t \in T$ are called* time points*.*

Intuitively, a stream S associates with each time point a set of atoms. We call S a *data stream*, if it contains only extensional atoms.

Example 5 (cont'd) The offers in the running scenario (Ex. 1) can be modeled as a data stream $D = (T_D, \upsilon_D)$ with a timeline $T_D = [0, 50]$ whose time unit is minutes, and the evaluation function $\upsilon_D(10) = \{\mathit{offer}(a, \text{“}\mathit{mbt}\text{”}, 30)\}$, $\upsilon(15)$=$\{\mathit{offer}(b, \text{“}\mathit{rayban}\text{”}, 25)\}$, $\upsilon_D(18) = \{\mathit{isNear}(a), \mathit{isNear}(b)\}$ and $\upsilon_D(t) = \emptyset$ for all $t \in T_D \setminus \{10, 15, 18\}$. The evaluation function υ_D can be equally represented as

$$\upsilon_D = \left\{ \begin{array}{l} 10 \mapsto \{\mathit{offer}(a, \text{“}\mathit{mbt}\text{”}, 30)\}, 15 \mapsto \{\mathit{offer}(b, \text{“}\mathit{rayban}\text{”}, 25)\}, \\ 18 \mapsto \{\mathit{isNear}(a), \mathit{isNear}(b)\} \end{array} \right\}.$$

To cope with the amount of data, one usually considers only recent atoms. Let $S = (T, \upsilon)$ and $S' = (T', \upsilon')$ be two streams s.t. $S' \subseteq S$, i.e., $T' \subseteq T$ and $\upsilon'(t') \subseteq \upsilon(t')$ for all $t' \in T'$. Then S' is called a *substream* of S.

Definition 2 (Window function) *A (computable)* window function *w_ι of* type *ι takes as input a stream $S = (T, \upsilon)$, a time point $t \in T$, called the* reference time point*, and a vector of* window parameters *$\boldsymbol{x}$ for type ι and returns a substream S' of S.*

Important are *tuple-based* and *time-based* window functions. The former select a fixed number of latest tuples while the latter select all atoms appearing in last n time points.

Window operators $\boxplus$. Window functions can be accessed in formulas by window operators. That is, an expression $\boxplus\alpha$ has the effect that α is evaluated on the "snapshot" of the stream delivered by its associated window function $w_{\boxplus}$.

By dropping information based on time, window operators specify temporal *relevance*. For each atom in a window, we control the semantics by some temporal *reference*.

Time Reference. Let $S = (T, \upsilon)$ be a stream, $a \in \mathcal{A}$ and $\mathcal{B} \subseteq \mathcal{A}$ static *background data*. Then, at time point $t \in T$,

- a holds, if $a \in \upsilon(t)$ or $a \in \mathcal{B}$;
- $\Diamond a$ holds, if a holds at some time point $t' \in T$;
- $\Box a$ holds, if a holds at all time points $t' \in T$; and
- $@_{t'}a$ holds, if $t' \in T$ and a holds at t'.

Next, the set $\mathcal{A}^+$ of *extended atoms* is given by the grammar

$$a \mid @_t a \mid \boxplus @_t a \mid \boxplus\Diamond a \mid \boxplus\Box a,$$

where $a \in \mathcal{A}$ and t is any time point. Expressions of form $\boxplus \star a$, where $\star \in \{@_t, \Diamond, \Box\}$, are called *window atoms*.

Example 6 The window atom $\boxplus^{30}\Diamond \mathit{offer}(Sh, Pr, Pe)$ takes a snapshot of the last 30 minutes 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, $\boxplus^{5}\Diamond \mathit{isNear}(Sh)$ does the same job to take a snapshot of size 5 minutes of the shops near the user. ∎

2.4 LARS Programs

We present a fragment of the formalism in [5].

Syntax. A *rule* r is of the form $\alpha \leftarrow \beta(r)$, where $H(r) = \alpha$ is the *head* and the *body* of r is $\beta(r) = \beta_1, \ldots, \beta_j, \mathrm{not}\, \beta_{j+1}, \ldots, \mathrm{not}\, \beta_n$. Here, α is of form a or $@_t a$, where $a \in \mathcal{A}^{\mathcal{I}}$, and each β_i is either an ordinary atom or a window atom.

Let $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_i \mid 1 \leq i \leq j\}$ is the *positive* and $B^-(r) = \{\beta_i \mid j < i \leq n\}$ is the *negative body* or r. A (LARS) *program* P is a set of rules. A program is *positive*, if none of its rules has a negative body atom.

Example 7 (cont'd) Suppose we are given static background data $\mathcal{B}$ that contains product information in a predicate of 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 queries in Example 4, under the input streams in a format as in Example 5.

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

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. ∎

Semantics. Let P be a LARS program. For a data stream $D = (T_D, \upsilon_D)$, any stream $I = (T, \upsilon) \supseteq D$ that coincides with D on $\mathcal{A}^{\mathcal{E}}$ is an *interpretation stream* for D. A tuple $M = \langle T, \upsilon, W, \mathcal{B} \rangle$ is an *interpretation* for D, where W is a set of window functions $w_{\boxplus}$ such that the corresponding window operator $\boxplus$ appears in P, and $\mathcal{B}$ is the background knowledge. Throughout, we assume W and $\mathcal{B}$ are fixed and thus also omit them.

Satisfaction by M at $t \in T$ is as follows: $M, t \models \alpha$ for $\alpha \in \mathcal{A}^+$, if α holds in (T, υ) at time t; $M, t \models r$ for rule r, if $M, t \models \beta(r)$ implies $M, t \models H(r)$, where $M, t \models \beta(r)$, if (i) $M, t \models \beta_i$ for all $i \in \{1, \ldots, j\}$ and (ii) $M, t \not\models \beta_i$ for all $i \in \{j{+}1, \ldots, n\}$; and $M, t \models P$ for program P, i.e., M is a *model* of P (for D) at t, if $M, t \models r$ for all $r \in P$. Moreover, M is *minimal*, if in addition no model $M' = \langle T, \upsilon', W, \mathcal{B} \rangle \neq M$ of P exists such that $\upsilon' \subseteq \upsilon$.

Definition 3 (Answer Stream) *An interpretation stream $I = (T, \upsilon)$ for a data stream $D \subseteq I$ is an* answer stream *of program P at time t, if $M = \langle T, \upsilon, W, \mathcal{B} \rangle$ is a minimal model of the* reduct $P^{M,t} = \{r \in P \mid M, t \models \beta(r)\}$. *By $\mathcal{AS}(P, D, t)$ we denote the set of all such answer streams I.*

Since RSP queries return just a single deterministic answer (which of course can contain multiple rows) at a time point, we consider in this paper LARS programs that have a single answer stream. By $AS(P, D, t)$, we directly refer to the single element of $\mathcal{AS}(P, D, t)$.

Example 8 (cont'd) Consider background data $\mathcal{B}$ that contains product information as in Ex. 2. That is, $\mathcal{B} = \{\ldots, g\_classify(\text{“}mbt\text{”}, 1), g\_classify(\text{“}rayban\text{”}, 0), \ldots\}$. Take the data stream D from Ex. 5 and let P be the LARS program consisting of the single rule in Ex. 7. Then, $I = (T_I, \upsilon_I)$ is the only answer stream of P wrt. D and $\mathcal{B}$ at time $t = 18$, where $T_I = T_D$ and $\upsilon_I = \upsilon_D \cup \{18 \mapsto \{ans(b, \text{“}rayban\text{”}, 25)\}\}$. ∎

3 Modeling RSP Queries

Section 2.2 shows a divergence in realizing continuous queries in C-SPARQL and CQELS. To be able to capture and analyze the difference between the two approaches, we need to have a common starting point. This section proposes a formal model of RSP queries that captures this common starting point idea, and then classifies C-SPARQL and CQELS on the model.

Similarly as in [17], we ignore solution modifiers and formalize an ***RSP query*** as a quadruple $Q = (V, P, \mathcal{D}, \mathcal{S})$, where V is a result form, P is a graph pattern, $\mathcal{D}$ is a dataset,[2] and $\mathcal{S}$ is a set of stream graph patterns. Roughly, $\mathcal{S}$ is a set of tuples of the form $(\mathbf{s}, \omega, g)$, where $\mathbf{s}$ is a stream identifier, ω is a window expression, and g is a basic RDF graph pattern. Given a result form V, we denote by $\overline{V}$ the tuple obtained from lexicographically ordering the set of variables in V.

Example 9 Queries Q_1 and Q_2 in Ex. 4 stem from $Q = (V, P, \mathcal{D}, \mathcal{S})$, where

[2] For simplicity, we omit instantaneous background datasets, which can be extended in a straightforward way.

$$V = \{?\texttt{shop}, ?\texttt{pname}, ?\texttt{percent}\}$$
$$P = (P_1 \cup P_2 \cup P_3)\ \mathsf{FILTER}\ R$$
$$P_1 = \left\{ \begin{array}{lll} ?\texttt{shop} & \texttt{:offers} & ?\texttt{coupon}. \\ ?\texttt{coupon} & \texttt{:on} & ?\texttt{product}. \\ ?\texttt{coupon} & \texttt{:reduce} & ?\texttt{percent}. \end{array} \right\}$$
$$P_2 = \{\, ?\texttt{user} \quad \texttt{:isNear} \quad ?\texttt{shop}. \,\}$$
$$P_3 = \{\, ?\texttt{product} \quad \texttt{:g\_classify} \quad ?\texttt{gender}. \,\}$$
$$R = (?\texttt{percent} \geq 20\ \texttt{\&\&}\ ?\texttt{gender} \neq 1)$$
$$\mathcal{D} = \{\texttt{<products>}\}$$
$$\mathcal{S} = \left\{ \begin{array}{l} (\texttt{<http://coupons>}, \texttt{[RANGE 30m]}, P_1), \\ (\texttt{<http://locations>}, \texttt{[RANGE 5m]}, P_2) \end{array} \right\}.$$

This query covers all common aspects of Q_1 and Q_2 which both access the static dataset at <products> and the input streams at <coupons> and <locations> with a window of range 30 and 5 minutes, respectively. On top of the snapshot from the input streams together with the static dataset, a pattern matching is carried out on the graph pattern P. ■

Next, we show how this RSP query model captures the divergent C-SPARQL and CQELS queries. Consider an RSP query Q.

- The corresponding C-SPARQL query, denoted by $cs(Q)$, can be obtained from Q by setting the graph patterns in all stream graph patterns in $\mathcal{S}$ to $\emptyset$. This goes along with the idea of C-SPARQL to merge patterns on the input streams into the default graph.
- A corresponding CQELS query, however, can be obtained from Q at different levels of cautiousness: for every part of P that contains g_i s.t. $(\mathbf{s}_i, \omega_i, g_i) \in \mathcal{S}$, replace it with either (i) ($\mathsf{STREAM}\ \mathbf{s}_i\ \omega_i\ g_i$), or (ii) ((STREAM $\mathbf{s}_i\ \omega_i\ g_i$) UNION g_i). The former is a *brave* approach when one can make sure that the static dataset and the stream $\mathbf{s}_i$ do not share patterns, while the latter is more *cautious* when one is not sure and rather expects triples matching g_i come from either the static dataset or the input streams. Therefore, Q is corresponding to a set $cq(Q)$ of $2^{|\mathcal{S}|}$ CQELS queries, including a brave one, a cautious one, and the ones in between. Note that Q_2 in Ex. 4 is the brave CQELS query of Q.

4 Capturing RSP Queries Using LARS

We will use the following strategy to capture different RSP approaches with LARS:

(1) First, the two push- and pull-based execution modes can be applied to LARS programs in general via two straightforward translations.
(2) Then, window expressions in RSP are translated into window operators in LARS.
(3) Next, R2R operators and the approaches in building the datasets to be evaluated by R2R operators are captured by two slightly different translations τ_1 and τ_2, based on the translation from SPARQL to Datalog rules in [17].

(4) Finally, post-processing can be carried out to mimic R2S operators. Note that the post-processing can be done operationally. Therefore, it is not of our theoretical interest and will not be considered in this paper.

We now go into details of (1)-(3).

4.1 Push- and Pull-Based Execution Modes for LARS Programs

This section provides two translations that capture the push- and pull-based execution modes by means of LARS itself. Given a LARS program P and a pulling period $U > 0$, the translations $\triangleright(P)$ and $\triangleleft(P, U)$ encode the push- and pull-mode by LARS rules, respectively. Intuitively, we add to the body of each rule in P an ordinary atom `trigger`. Then, rules to conclude `trigger` are added depending on the mode. For push-based mode, `trigger` will be concluded per new incoming input triple. For pull-based mode, the condition is that the current time point is a multiple of U.

Formally speaking, for a LARS rule r, a LARS program P, a pulling period U, let

$$\begin{aligned} \mathit{trigger}(r) &= H(r) \leftarrow B(r), \texttt{trigger}. \\ \mathit{trigger}(P) &= \{\mathit{trigger}(r) \mid r \in P \wedge B(r) \neq \emptyset\} \\ \triangleright(P) &= \mathit{trigger}(P) \cup \{\texttt{trigger} \leftarrow \boxplus_{NOW}\, p(\boldsymbol{X}). \mid p \in \mathcal{A}^{\mathcal{I}}\} \\ \triangleleft(P, U) &= \mathit{trigger}(P) \cup \{\texttt{trigger} \leftarrow \boxplus_{NOW}\, @_T \mathit{true}, T \,\%\, U = 0.\} \end{aligned}$$

Notably, the translation $\triangleleft$ for the pull-based mode needs to acquire the current time point, which is achieved as follows. The logical constant true always holds, and thus $@_T \mathit{true}$ holds for all considered time points T. By applying window operator $\boxplus_{NOW}$ (or equivalently $\boxplus^0$) before, only the current time point will be selected. The following proposition shows that $\triangleright$ and $\triangleleft$ faithfully capture the execution modes.

Proposition 1 *Let P be a LARS program, U be a positive integer, and $D = (T_D, \upsilon_D)$ be an input stream. For every $t \in T_D$, it holds that*

(1) If $\upsilon_D(t) \neq \emptyset$, then $\mathcal{AS}(\triangleright(P), D, t) = \mathcal{AS}(P, D, t)$
(2) If $\upsilon_D(t) = \emptyset$, then $\mathcal{AS}(\triangleright(P), D, t) = \{D\}$
(3) If $t \,\%\, U = 0$, then $\mathcal{AS}(\triangleleft(P, U), D, t) = \mathcal{AS}(P, D, t)$
(4) If $t \,\%\, U \neq 0$, then $\mathcal{AS}(\triangleleft(P, U), D, t) = \{D\}$.

4.2 Translate RSP Window Expressions to LARS Window Operators

Table 1 presents a translation from windows in RSP to window operators in LARS. Given a window expression ω in RSP, $\tau(\omega)$ returns a LARS window operator which corresponds to a window function that provides the same functionalities as ω [4, 3, 5].

4.3 Translate RSP Queries to LARS Programs

For capturing R2R operators of continuous SPARQL queries we can exploit an existing translation from SPARQL to Datalog rules [17]. The difference in our setting is the

Window expression ω	$\tau(\omega)$
`[RANGE L]`	$\boxplus^{L}$
`[RANGE L SLIDE D]`	$\boxplus^{L,0,D}$
`[ROWS N]`	$\boxplus^{N}_{\#}$
`[NOW]`	$\boxplus^{0}$ or $\boxplus_{NOW}$
`[RANGE UNBOUNDED]`	$\boxplus^{\infty}$

Table 1: Translating window expressions ω to LARS' window operators

streaming input and how RSP engines take snapshots of the stream to build datasets for SPARQL evaluation.

We propose two strategies (T_1) and (T_2) to extend the translation in [17] to capture R2R operators and the ways to build snapshot datasets (B_1), (B_2) (cf. Section 2.2):

(T_1) For (B_1), we just need to make sure that the triples from the input streams are collected into the default graph.

(T_2) For (B_2), we introduce one more case for translating a stream graph pattern to LARS rules.

Towards formally presenting our translations, we start with a review of the translation from SPARQL to Datalog in [17], which has two parts:

(i) The first part imports RDF triples from the dataset into a 4-ary predicate of the form $\texttt{triple}(S, P, O, G)$, where (S, P, O) covers RDF triples and G holds a graph identifier. This can be done with the Answer Set Programming solver dlvhex.[3]

(ii) For the second part, a function τ takes as input a result form V, a graph pattern P, a dataset $\mathcal{D}$, an integer $i{>}0$ and translates the input into a Datalog program, recursively along P. The base case is a single RDF triple pattern, i.e., $P = \{(S, P, O)\}$. Intuitively, τ converts SPARQL operators to declarative rules.

Our purpose is to provide a translation for *theoretical analysis* rather than for practical implementation of RSP queries. Thus, we concentrate on (ii). For (i), we assume that

- each triple (s, p, o) from the static dataset $\mathcal{D}$ can be accessed by $\texttt{triple}(s, p, o, \mathcal{D})$,
- each triple (s, p, o) arriving at a stream $\mathbf{s}$ at time t contributes to the evaluation function υ at t under a predicate `striple`, that is, $\texttt{striple}(s, p, o, \mathbf{s}) \in \upsilon(t)$.

Figure 1 shows the extension of τ in [17] with a parameter $\mathcal{S}$ representing the input streams. The translation $LT(\cdot)$ is taken from [17], which is based on the rewriting defined by Lloyd and Topor [13].

For (T_1) we modify the base cases of τ for (S, P, O) with

$$\begin{aligned}\tau(V, (S, P, O), \mathcal{D}, \mathcal{S}, i) \;=\; & \{\texttt{ans}_i(\overline{V}, \mathcal{D}, \mathcal{S}) \leftarrow \texttt{triple}(S, P, O, \mathcal{D})\} \cup \\ & \{\texttt{ans}_i(\overline{V}, \mathcal{D}, \mathcal{S}) \leftarrow \texttt{triple}(S, P, O, \mathcal{S})\},\end{aligned}$$

[3] `http://www.kr.tuwien.ac.at/research/systems/dlvhex/`

$$
\begin{aligned}
\tau(V,(S,P,O),\mathcal{D},\mathcal{S},i) &= \mathtt{ans}_i(\overline{V},\mathcal{D},\mathcal{S}) \leftarrow \mathtt{triple}(S,P,O,\mathcal{D})\\
\tau(V,(P_1 \text{ AND } P_2),\mathcal{D},\mathcal{S},i) &= \tau(vars(P_1),P_1,\mathcal{D},\mathcal{S},2i)\cup\\
&\quad\ \tau(vars(P_2),P_2,\mathcal{D},\mathcal{S},2i+1)\cup\\
&\quad\ \mathtt{ans}_i(\overline{V},\mathcal{D},\mathcal{S}) \leftarrow \mathtt{ans}_{2i}(\overline{vars(P_1)},\mathcal{D},\mathcal{S}),\\
&\qquad\qquad \mathtt{ans}_{2i+1}(\overline{vars(P_2)},\mathcal{D},\mathcal{S}).\\
\tau(V,(P_1 \text{ UNION } P_2),\mathcal{D},\mathcal{S},i) &= \tau(vars(P_1),P_1,\mathcal{D},\mathcal{S},2i)\\
&\quad\ \tau(vars(P_2),P_2,\mathcal{D},\mathcal{S},2i+1)\cup\\
&\quad\ \mathtt{ans}_i(\overline{V[(V\setminus vars(P_1))\rightarrow \mathsf{null}]},\mathcal{D},\mathcal{S}) \leftarrow \mathtt{ans}_{2i}(\overline{vars(P_1)},\mathcal{D},\mathcal{S}).\\
&\quad\ \mathtt{ans}_i(\overline{V[(V\setminus vars(P_2))\rightarrow \mathsf{null}]},\mathcal{D},\mathcal{S}) \leftarrow \mathtt{ans}_{2i+1}(\overline{vars(P_2)},\mathcal{D},\mathcal{S}).\\
\tau(V,(P_1 \text{ MINUS } P_2),\mathcal{D},\mathcal{S},i) &= \tau(vars(P_1),P_1,\mathcal{D},\mathcal{S},2i)\cup\\
&\quad\ \tau(vars(P_2),P_2,\mathcal{D},\mathcal{S},2i+1)\cup\\
&\quad\ \mathtt{ans}_i(\overline{V[(V\setminus vars(P_1))\rightarrow \mathsf{null}]},\mathcal{D},\mathcal{S}) \leftarrow \mathtt{ans}_{2i}(\overline{vars(P_1)},\mathcal{D},\mathcal{S}),\\
&\qquad\qquad \text{not } \mathtt{ans}'_{2i}(\overline{vars(P_1)\cap vars(P_2)},\mathcal{D},\mathcal{S}).\\
&\quad\ \mathtt{ans}'_{2i}(\overline{vars(P_1)\cap vars(P_2)},\mathcal{D},\mathcal{S}) \leftarrow \mathtt{ans}_{2i+1}(\overline{vars(P_2)},\mathcal{D},\mathcal{S}).\\
\tau(V,(P_1 \text{ OPT } P_2),\mathcal{D},\mathcal{S},i) &= \tau(V,(P_1 \text{ AND } P_2),\mathcal{D},\mathcal{S},i)\cup\\
&\quad\ \tau(V,(P_1 \text{ MINUS } P_2),\mathcal{D},\mathcal{S},i)\\
\tau(V,(P \text{ FILTER } R),\mathcal{D},\mathcal{S},i) &= \tau(V,P,\mathcal{D},\mathcal{S},2i)\cup\\
&\quad\ LT(\mathtt{ans}_i(\overline{V},\mathcal{D},\mathcal{S}) \leftarrow \mathtt{ans}_{2i}(\overline{vars(P)},\mathcal{D},\mathcal{S}),R.)\\
\tau(V,(\mathsf{GRAPH}\ g\ P),\mathcal{D},\mathcal{S},i) &= \tau(V,P,g,\mathcal{S},i) \text{ for } g\in V\cup I\\
&\quad\ \mathtt{ans}_i(\overline{V},D) \leftarrow \mathtt{ans}_i(\overline{V},g),\mathtt{isIRI}(g),g\neq \mathtt{default}.
\end{aligned}
$$

Fig. 1: Extending translation τ in [17] with input streams $\mathcal{S}$

and add the following rules to import input streaming triples to the default graph:

$$\tau'(\mathcal{S}) = \{\mathtt{triple}(S,P,O,\mathcal{S}) \leftarrow \tau(\omega)\diamond\mathtt{striple}(S,P,O,\mathtt{s}) \mid (\mathtt{s},\omega,g)\in\mathcal{S}\}.$$

The translation for strategy (T_1) is $\tau_1(V,P,\mathcal{D},\mathcal{S},i) = \tau(V,P,\mathcal{D},\mathcal{S},i)\cup\tau'(\mathcal{S})$.
For (T_2), let τ_2 be a function that agrees with τ, and moreover fulfills:

$$\tau_2(V,(\mathsf{STREAM}\ \mathtt{s}\ \omega\ g),\mathcal{D},\mathcal{S},i)=\mathtt{ans}_i(\overline{V},\mathcal{D},\mathcal{S}) \leftarrow \tau(\omega)(\bigwedge_{(S,P,O)\in g}\diamond\mathtt{striple}(S,P,O,\mathtt{s})).$$

When it is clear from context, we will write in the sequel $\tau/\tau_i(Q)$ ($i\in\{1,2\}$), for a query $Q=(V,P,\mathcal{D},\mathcal{S})$ instead of $\tau/\tau_i(V,P,\mathcal{D},\mathcal{S},1)$.

Given an RSP query $Q{=}(V,P,\mathcal{D},\mathcal{S})$, let $Q'{\in}\{cs(Q)\}{\cup}cq(Q)$ and $I_i{=}AS(\tau_i(Q'),D,t)$ for a data stream D and a time point t, where $i\in\{1,2\}$. We denote the set of atoms of predicate $\mathtt{ans}_j$ with the parameter corresponding to $\mathcal{S}$ projected away by

$$chop(I,Q) = \{\mathtt{ans}_j(\overline{V_j},\mathcal{D}) \mid \mathtt{ans}_j(\overline{V_j},\mathcal{D},\mathcal{S})\in I\}\cup(I\setminus\{\mathtt{ans}_j(\overline{V_j},\mathcal{D},\mathcal{S})\in I\}).$$

The following result shows that our translation preserves the translation in [17].

Proposition 2 *Let $Q = (V, P, \mathcal{D}, \emptyset)$ be an RSP query, that is, a SPARQL query, and $cq(Q) = \{Q'\}$. Let I be the single answer set of $\tau(Q)$, $I_1 = AS(\tau_1(cs(Q)), D, t)$, and $I_2 = AS(\tau_2(Q'), D, t)$. It holds that $I = chop(I_1, Q) = chop(I_2, Q)$.*

Translations τ_1 and τ_2 share the core from translation τ in [17], and differ due to two approaches by C-SPARQL and CQELS in extending SPARQL to deal with streaming input. Furthermore, the two engines execute on two different modes, namely pull- and push-based. This makes it non-trivial to analyze situations in which the engines should return the same output. Tackling this question now becomes possible with LARS, which will be discussed next.

5 Utilizing LARS for Comparing RSP Semantics

This section discusses ideas to tackle the comparison between C-SPARQL and CQELS using LARS. The main questions are:

(i) What do we mean by saying "C-SPARQL and CQELS return the same output?"
(ii) When do the two approaches of dealing with streaming input coincide, i.e., merging input streams into the default graph and using stream graph patterns do not make any difference in building the datasets evaluated by the engines?
(iii) Under which conditions will the push- and pull-based executions fulfill (i)?

The most important question is (i), which establishes the whole setup and methodology for the comparison. Since C-SPARQL and CQELS build on a pull- and push-based mode, respectively, it does not make sense to require that the output of the two engines coincides at every single time point. Instead, we need some notion of *agreement* between the two engines. Intuitively, we say that C-SPARQL and CQELS agree on a time interval $[t_1, t_2]$, if the union of outputs returned by CQELS at every time point in $(t_1, t_2]$ (i.e., the total output in that interval) coincides with the output returned by C-SPARQL at t_2. Assume that both two engines start at time 0 of a timeline T and C-SPARQL is executed with a pulling period of U time units. Then, we say that the engines agree on T just if they agree on every interval $[i \cdot U, (i+1) \cdot U] \in T$, for $i \geq 0$. On the translated LARS programs, checking for agreement on the output boils down to checking the agreement on the output facts of the predicate $\texttt{ans}_1$.

Imagine a user staring at his App to wait for notifications. If the two engines agree, then at the end of the pulling period, the user will see the same notifications in both cases; if the pulling period is short enough, she might even not notice any difference between the two modes. This makes the agreement notion reasonable from a practical point of view.

Now we analyze possible sufficient conditions of agreement for C-SPARQL and CQELS, by finding answers for questions (ii) and (iii).

To put (ii) more concretely in the context of LARS, take an RSP query Q and let $Q_1 = cs(Q)$ and $Q_2 \in cq(Q)$; we want to find conditions under which the answer streams of $\tau_1(Q_1)$ and $\tau_2(Q_2)$ at time point t on the input stream D have the same extension of predicate $\texttt{ans}_1$.

Observe that $\tau_1(Q_1)$ and $\tau_2(Q_2)$ differ on dealing with input coming from predicate `striple`. The former program merges facts of `striple` into `triple`, which will

be used later on to conclude $\texttt{ans}_i$; the latter program concludes $\texttt{ans}_i$ directly from `striple`. If the static dataset and the input streams share some patterns, then τ_1 may not be capable of identifying the origin of some `triple`. Even if the input stream does not receive any incoming triple, $\tau_1(Q_1)$ may still conclude some output due to facts staying in the static part. On the other hand, $\tau_2(Q_2)$ returns no output as the stream is not updated. To avoid such confusion, it is required that the static dataset and the input streams do not share patterns. Fortunately, this usually holds in practice. For instance, in our running example, the pattern `?user :isNear ?shop` can only match triples in the stream `<http://locations>`, since predicate `:isNear` will not be streamed in `<http://coupons>`, and the static dataset should not contain such information.

When the conditions for question (ii) are fulfilled, one has a better setting to analyze (iii). Still, for the first step, we need to assume as in [10] that the execution time of the engines is marginal compared to the input rate. Note that with these conditions, we can for a given LARS program P compare the output of $\triangleleft(P, U)$ and $\triangleright(P)$ in time intervals $[t_1, t_2] = [i \cdot U, (i + 1) \cdot U]$. As the rules in P are shared by both translations, intuitively, the total output of $\texttt{ans}_1$ by $\triangleright(P)$ during $[t_1, t_2]$ will coincide with the one by $\triangleleft(P, U)$, if the total input to $\triangleright(P)$ during $[t_1, t_2]$ coincides with the input to the $\triangleleft(P, U)$ at t_2. Here the "input" consists of the snapshots obtained by evaluating the windows on the streams.

However, this condition cannot be guaranteed under high throughput. The reason is that with dense input streams, the snapshots taken at time points near the beginning of an interval will have high chances to collect more input than the snapshot at its end. Thus, in practice it is rather unlikely that C-SPARQL and CQELS will agree, due to the strong semantic implications of push/pull-based querying.

Conclusions and Outlook. This paper establishes first steps towards formally comparing two RSP semantics implemented in two well-known engines, namely C-SPARQL and CQELS, by proposing translations to capture the languages and execution modes of the engines, and discussing idea to formalize a notion of agreement between the two semantics as well as a condition for it to hold. Next steps include working out these ideas formally and implementing the translations to build a comparison benchmarking systems of different stream processing approaches using LARS.

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. Towards a Logic-Based Framework for Analyzing Stream Reasoning. In *OrdRing*, pages 11–22, 2014.
4. H. Beck, M. Dao-Tran, T. Eiter, and M. Fink. Towards Ideal Semantics for Analyzing Stream Reasoning. In *ReactKnow*, 2014.
5. H. Beck, M. Dao-Tran, T. Eiter, and M. Fink. LARS: A logic-based framework for analyzing reasoning over streams. In *AAAI*, 2015.
6. A. Bolles, M. Grawunder, and J. Jacobi. Streaming SPARQL - extending SPARQL to process data streams. In *ESWC*, pages 448–462, 2008.

7. J.-P. Calbimonte, Ó. Corcho, and A. J. G. Gray. Enabling ontology-based access to streaming data sources. In *ISWC (1)*, pages 96–111, 2010.
8. R. Cyganiak, D. Wood, and M. Lanthaler. RDF 1.1 Concepts and Abstract Syntax. http://www.w3.org/TR/rdf11-concepts/, 2014.
9. D. Dell'Aglio, J. Calbimonte, M. Balduini, Ó. Corcho, and E. D. Valle. On Correctness in RDF Stream Processor Benchmarking. In *ISWC 2013*, pages 326–342, 2013.
10. 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.
11. S. Groppe. *Data Management and Query Processing in Semantic Web Databases*. Springer, 2011.
12. S. Harris and A. Seaborne. SPARQL 1.1 Query Language. http://www.w3.org/TR/sparql11-query/, 2013.
13. J. W. Lloyd and R. W. Topor. Making Prolog more Expressive. *J. Log. Program.*, 1(3):225–240, 1984.
14. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34:16:1–16:45, September 2009.
15. 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.
16. D. L. Phuoc, M. Dao-Tran, M.-D. Pham, P. Boncz, T. Eiter, and M. Fink. Linked stream data processing engines: Facts and figures. In *ISWC - ET*, pages 300–312, 2012.
17. A. Polleres. From SPARQL to rules (and back). In *WWW 2007*, pages 787–796, 2007.
18. O. Walavalkar, A. Joshi, T. Finin, and Y. Yesha. Streaming Knowledge Bases. In *SSWS*, 2008.
19. Y. Zhang, P. Minh Duc, O. Corcho, and J. P. Calbimonte. SRBench: A Streaming RDF/S-PARQL Benchmark. In *ISWC*, pages 641–657, 2012.