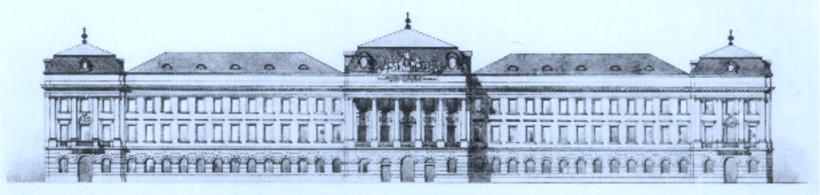
INFSYS
RESEARCH
REPORT



INSTITUT FÜR INFORMATIONSSYSTEME
ARBEITSBEREICH WISSENSBASIERTE SYSTEME

CONTRASTING RDF STREAM PROCESSING SEMANTICS

MINH DAO-TRAN HARALD BECK THOMAS EITER

INFSYS RESEARCH REPORT 15-06
SEPTEMBER 2015

Institut für Informationssysteme
AB Wissensbasierte Systeme
Technische Universität Wien
Favoritenstraße 9-11
A-1040 Wien, Austria
Tel: +43-1-58801-18405
Fax: +43-1-58801-18493
sek@kr.tuwien.ac.at
www.kr.tuwien.ac.at

INFSYS RESEARCH REPORT

INFSYS RESEARCH REPORT 15-06, SEPTEMBER 2015

CONTRASTING RDF STREAM PROCESSING SEMANTICS

Minh Dao-Tran[1] Harald Beck[1] Thomas Eiter[1]

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, CSRBench, and YABench were introduced as attempts to compare different approaches, focusing mainly on the operational aspects. The recent logic-based LARS framework provides a theoretical underpinning to analyze stream processing/reasoning semantics. In this work, we use LARS to compare the semantics of two typical RSP engines, namely C-SPARQL and CQELS, identify conditions when they agree on the output, and discuss situations where they disagree. The findings give insights that might prove to be useful for the RSP community in developing a common core for RSP.

[1] Institut für Informationssysteme, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Vienna, Austria; email: {dao,beck,eiter}@kr.tuwien.ac.at.

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

Contents

1 Introduction

In interconnected information technologies such as Internet of Things and Cyber Physical Systems it is crucial to have simple access to data irrespective of their sources. The Semantic Web's RDF data model was designed to integrate such distributed and heterogeneous data. Recently, RDF Stream Processing (RSP) has been emerging to tackle novel problems arising from streaming data: to integrate querying and processing of static and dynamic data, e.g., information continuously arriving from sensors.

This has led to the development of data models, query languages and processing engines, which diverge in several aspects, ranging from the representation of RDF streams, execution modes [Phuoc et al., 2012], to semantics [Barbieri et al., 2010a, Phuoc et al., 2011, Calbimonte et al., 2010, Bolles et al., 2008, Groppe, 2011]. To deal with this heterogeneity, the RSP community[1] was formed to establish a standard towards 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 two benchmarking systems, namely SRBench [Zhang et al., 2012] and LSBench [Phuoc et al., 2012]. The former defined functional tests to verify the query languages features by the engines, while the latter measured mismatch between the output of different engines, assuming they are sound, i.e., all output produced by them are correct. Later on, CSRBench [Dell'Aglio et al., 2013] 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. YABench [Kolchin and Wetz, 2015] follows the approach by CSRBench with the main purpose of facilitating joint evaluation of functional, correctness, and performance testing. However, this approach allows only partial comparison between engines by referring to their ideal counterparts.

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, [Dell'Aglio et al., 2015] proposed a unifying query model to explain the heterogeneity of RSP systems. It shows the difference between two approaches as represented by representative engines in the RSP community, namely C-SPARQL [Barbieri et al., 2010a], SPARQL$_{\text{Stream}}$ [Calbimonte et al., 2010] and CQELS [Phuoc et al., 2011]. This work identified types of datasets that C-SPARQL/SPARQL$_{\text{Stream}}$ can handle while CQELS cannot, and vice versa. However, it does not point out systematically when and how the engines agree on the output.

In the stream processing community, SECRET [Dindar et al., 2013] was proposed to characterize and analyze the behavior of stream processing engines, but at the operational level.

Latterly, a **L**ogic-based framework for **A**nalyzing **R**easoning over **S**tream (LARS) was introduced [Beck et al., 2015]. LARS can be used as a unifying language which stream processing/reasoning languages can be translated to. 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. Furthermore, the model-based semantics of LARS is a means to formalize the intuition of agreement between not only RSP engines but also engines from other communities, and to identify conditions where this holds.

In this paper, we exploit the capability of LARS to analyze the difference between the semantics of C-SPARQL and CQELS by:

- providing translations that capture the push- and pull- execution modes for general LARS programs,
- providing translations from C-SPARQL and CQELS to LARS,

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

- introducing a notion of *push-pull-agreement* between LARS programs, and
- identifying conditions where C-SPARQL and CQELS agree on their output, by checking whether the translated LARS programs push-pull-agree.

Our findings show that C-SPARQL and CQELS agree on a very limited setting, and give insights on their difference. This result might prove to be useful for the RSP community in developing a common core for RSP. Moreover, we demonstrate how envisaged semantics may be formalized and checked with LARS.

Throughout this paper, for the purpose of a theoretical comparison, we adopt as in [Dell'Aglio et al., 2015] the assumption that execution time of RSP engines is neglectable compared to the data rate of the input streams.

2 Preliminaries

RSP can be intuitively seen as extending querying RDF datasets with SPARQL to querying RDF streams with "continuous SPARQL." LARS is a logic-based framework for analyzing stream reasoning. This section briefly reviews RDF, SPARQL, RSP, and LARS, which will be illustrated using the following running scenario inspired by [Dell'Aglio et al., 2015].

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

2.1 RDF and SPARQL

RDFmodels 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:

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

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, 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 [Pérez et al., 2009].

Example 3 (cont'd) Assume that a snapshot of coupons in the last 30 minutes contains triples in $g_1 \cup g_2$ from Example 2 and is collected in an RDF store identified by the IRI `<http://coupons-snapshot>`. Furthermore, suppose that product information is stored at `<http://products>`. The following SPARQL query computes the relevant coupons for Claire at a single time point. For simplicity, the geographical proximity between the location of the user and that of the shops is not considered.

```
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

The semantics of SPARQL is defined via mappings. A *mapping* μ is a partial function from V to $I \cup B \cup L$. The result of a `SELECT` SPARQL query is a set of mappings that match the query's body, projected to the variables specified in the `SELECT` clause. For example, Q_0 in Ex. 3 evaluated under the data in Ex. 2 returns a set of a single mapping:

$$\{\{?\texttt{shop} \mapsto b,\ ?\texttt{product} \mapsto \text{“}\texttt{rayban}\text{”},\ ?\texttt{percent} \mapsto 25\}\}.$$

However, one-shot queries by themselves are not able to give answers under dynamic input as in the running scenario. For this purpose, we need RDF stream processing.

2.2 RDF Stream Processing

Temporal RDF Graphs and RDF Streams. In *continuous query processing* over dynamic data, the temporal nature of the data is crucial and needs to be captured in the data representation. For this purpose, temporal RDF graphs and RDF streams are defined by generalizing the standard definition of RDF graphs as follows.

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 4 (cont'd) The 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 Example 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 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) [Arasu et al., 2006], 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. They are reflected in the context of RSP as follows (See also Fig. 1, page 10).

(i) S2R operators, also called *windows*, yield relational snapshots of streaming tuples for pure SPARQL processing. They include time-based and tuple-based windows.

(ii) R2R: Next, finite sets of mappings or triples (if windows did no pattern matching) are processed under SPARQL operators such as AND, OPTIONAL, MINUS, etc., based on the graph pattern specified in the body of the query.

(iii) R2S: The operators `RStream`, `IStream`, and `DStream` are used after R2R operators to convert the "pure" SPARQL output into an output stream. While `RStream` reports all output triples that can be computed based on the snapshots provided by S2R operators at the current execution, `IStream`/`DStream` only reports the inserted/deleted triples in comparison with the previous execution.

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 [Dell'Aglio et al., 2015]:

(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 5 (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

Compared to Q_0 in Ex. 3, 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.

Note that in [Barbieri et al., 2010b], to implement C-SPARQL, the authors translates a C-SPARQL query into an Operator Graph (O-Graph), where the decision on which input sources (RDF streams or RDF graph) go to which operator/pattern matching is made, but the details were not shown. Furthermore, this distinction is done at the implementation level and has never been mentioned at the semantics level of any C-SPARQL publication. This paper proposes a clear view to this issue at the semantics level.

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

We will introduce the central concepts of LARS [Beck et al., 2015] tailored to the considered fragment. Throughout, 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. The *projection* of a stream S to a predicate p is defined as $S|_p = (T, \upsilon|_p)$, where $\upsilon|_p(t) = \{p(\boldsymbol{c}) \mid p(\boldsymbol{c}) \in \upsilon(t)\}$. Here, $p(\boldsymbol{c})$ is any atom with predicate p and arguments (constants) $\boldsymbol{c}$. By $Ats(S) = \bigcup_{t \in T} \upsilon(t)$, we denote the set of all atoms appearing in S.

Example 6 (cont'd) The offers in the running scenario (Example 1) can be modeled as a data stream $D = (T_D, \upsilon_D)$ with a timeline $T_D = [0, 50]$ whose time unit is minute, 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 *window* 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 $S = (T, v)$ 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 $a \in v(t)$ 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 $t' \in T$;
- $@_{t'} a$ holds, if $t' {\in} T$ and a holds at t'

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

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

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 7 The window atom $\boxplus_{\tau}^{30} \Diamond \mathit{offer}(Sh, Pr, Pe)$ takes a snapshot of the last 30 time units (which are minutes in our running scenario) 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, $\boxplus_{\tau}^{5} \Diamond \mathit{isNear}(Sh)$ 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 [Beck et al., 2015].

Syntax. A *rule* r is of the form $a \leftarrow \beta(r)$, where $H(r) = a$ is the *head* and the *body* of r is

$$\beta(r) = \beta_1, \ldots, \beta_j, \mathrm{not}\, \beta_{j+1}, \ldots, \mathrm{not}\, \beta_n,$$

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 8 (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 5, under the input streams in a format as in Example 6.

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

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 operators $\boxplus$ 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.

As RSP queries return just a single answer 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 9 (cont'd) Consider background data $\mathcal{B}$ that contains product information as in Example 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 Example 6 and let P be the LARS program consisting of the single rule in Ex. 8. 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, which concerns the same static datasets, the same input streams, and the same algebraic expression on top of the snapshot at an execution. 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 [Polleres, 2007], 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 input stream patterns. Roughly, $\mathcal{S}$ is a set of tuples of the form (s, ω, g), where 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.

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

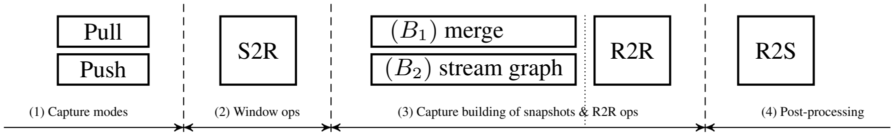


Figure 1: A strategy to capture RSP queries with LARS

Example 10 Queries Q_1 and Q_2 in Ex. 5 stem from $Q = (V, P, \mathcal{D}, \mathcal{S})$, where

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

This query covers all common aspects of Q_1 and Q_2: They both access the static dataset identified by the IRI `<http://products>` and the input streams at `<http://coupons>` and `<http://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 input stream 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) $(\textsf{STREAM}\ \mathbf{s}_i\ \omega_i\ g_i)$, or (ii) $((\textsf{STREAM}\ \mathbf{s}_i\ \omega_i\ g_i)\ \textsf{UNION}\ g_i)$. 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 Example 5 is the brave CQELS query of Q.

4 Capturing RSP Queries Using LARS

The building blocks of RSP queries presented in Section 2.2 recommend a strategy, as depicted in Figure 1, to capture different RSP approaches using 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 [Polleres, 2007].

(4) Finally, post-processing can be carried out to mimic `IStream` and `DStream` based on `RStream`. Note that LARS semantics corresponds to `RStream`, and 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 $\rhd(P)$ and $\lhd(P, U)$ 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

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

Notably, the translation $\lhd$ 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 $\boxplus_{\mathit{NOW}}$ (or equivalently $\boxplus^0_\tau$) before, only the current time point will be selected. The following proposition shows that $\rhd$ and $\lhd$ 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 $\mathcal{AS}(\rhd(P), D, t) = \mathcal{AS}(P, D, t)$, else $\mathcal{AS}(\rhd(P), D, t) = \{D\}$.

(2) If $t \,\%\, U{=}0$, then $\mathcal{AS}(\lhd(P, U), D, t) = \mathcal{AS}(P, D, t)$, else $\mathcal{AS}(\lhd(P, U), D, t){=}\{D\}$.

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 [Beck et al., 2014a,b, 2015].

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

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

4.3 Translate RSP Queries to LARS Programs

For CQL, R2R operators are simply captured by SQL operators. Similarly, capturing R2R operators of continuous SPARQL queries can exploit an existing translation from SPARQL to Datalog rules [Polleres, 2007]. The difference in our setting is the 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 [Polleres, 2007] 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 [Polleres, 2007], 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 the operational semantics of SPARQL 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 can be accessed by a fact $\texttt{triple}(s, p, o, \mathcal{D})$,
- each triple (s, p, o) arriving at a stream s at time t contributes to the evaluation function v at t under a predicate `striple`, that is, $\texttt{striple}(s, p, o, \mathsf{s}) \in v(t)$.

The extension of τ in [Polleres, 2007] with a parameter $\mathcal{S}$ representing the input streams is shown in Fig. 2.

The translation $LT(\cdot)$ is taken from [Polleres, 2007], which is based on the rewriting defined by Lloyd and Topor [Lloyd and Topor, 1984].

[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 \textsf{ AND } P_2),\mathcal{D},\mathcal{S},i) &= \tau(\mathit{vars}(P_1),P_1,\mathcal{D},\mathcal{S},2i) \cup \\
&\quad \tau(\mathit{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{\mathit{vars}(P_1)},\mathcal{D},\mathcal{S}), \\
&\quad\ \qquad\qquad\qquad \mathtt{ans}_{2i+1}(\overline{\mathit{vars}(P_2)},\mathcal{D},\mathcal{S}). \\
\tau(V,(P_1 \textsf{ UNION } P_2),\mathcal{D},\mathcal{S},i) &= \tau(\mathit{vars}(P_1),P_1,\mathcal{D},\mathcal{S},2i) \\
&\quad \tau(\mathit{vars}(P_2),P_2,\mathcal{D},\mathcal{S},2i+1) \cup \\
&\quad\ \mathtt{ans}_i(\overline{V[(V \setminus \mathit{vars}(P_1)) \to \textsf{null}]},\mathcal{D},\mathcal{S}) \leftarrow \mathtt{ans}_{2i}(\overline{\mathit{vars}(P_1)},\mathcal{D},\mathcal{S}). \\
&\quad\ \mathtt{ans}_i(\overline{V[(V \setminus \mathit{vars}(P_2)) \to \textsf{null}]},\mathcal{D},\mathcal{S}) \leftarrow \mathtt{ans}_{2i+1}(\overline{\mathit{vars}(P_2)},\mathcal{D},\mathcal{S}). \\
\tau(V,(P_1 \textsf{ MINUS } P_2),\mathcal{D},\mathcal{S},i) &= \tau(\mathit{vars}(P_1),P_1,\mathcal{D},\mathcal{S},2i) \cup \\
&\quad \tau(\mathit{vars}(P_2),P_2,\mathcal{D},\mathcal{S},2i+1) \cup \\
&\quad\ \mathtt{ans}_i(\overline{V[(V \setminus \mathit{vars}(P_1)) \to \textsf{null}]},\mathcal{D},\mathcal{S}) \leftarrow \mathtt{ans}_{2i}(\overline{\mathit{vars}(P_1)},\mathcal{D},\mathcal{S}), \\
&\quad\ \qquad\qquad\qquad\qquad\qquad\qquad \text{not } \mathtt{ans}'_{2i}(\overline{\mathit{vars}(P_1) \cap \mathit{vars}(P_2)},\mathcal{D},\mathcal{S}). \\
&\qquad\ \mathtt{ans}'_{2i}(\overline{\mathit{vars}(P_1) \cap \mathit{vars}(P_2)},\mathcal{D},\mathcal{S}) \leftarrow \mathtt{ans}_{2i+1}(\overline{\mathit{vars}(P_2)},\mathcal{D},\mathcal{S}). \\
\tau(V,(P_1 \textsf{ OPT } P_2),\mathcal{D},\mathcal{S},i) &= \tau(V,(P_1 \textsf{ AND } P_2),\mathcal{D},\mathcal{S},i) \cup \\
&\quad \tau(V,(P_1 \textsf{ MINUS } P_2),\mathcal{D},\mathcal{S},i) \\
\tau(V,(P \textsf{ 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{\mathit{vars}(P)},\mathcal{D},\mathcal{S}), R.) \\
\tau(V,(\textsf{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}
$$

Figure 2: Extending translation τ in [Polleres, 2007] with input streams $\mathcal{S}$

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

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

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

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

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:

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

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 $\tau/\tau_i(V,P,\mathcal{D},\mathcal{S},1)$.

Example 11 Take Q_1 from Example 5, then $\tau_1(Q_1)$ is the following LARS program.

$$\begin{aligned}
\texttt{ans}_1(Pe, Pr, Sh, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{ans}_2(C, Ge, Pe, Pr, Sh, U, \mathcal{D}, \mathcal{S}),\ Pe \geq 20, Ge \neq 1. && (r_1)\\
\texttt{ans}_2(C, Ge, Pe, Pr, Sh, U, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{ans}_4(C, Pe, Pr, Sh, \mathcal{D}, \mathcal{S}),\\
&\ \texttt{ans}_5(Ge, Pr, Sh, U, \mathcal{D}, \mathcal{S}). && (r_2)\\
\texttt{ans}_4(C, Pe, Pr, Sh, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{ans}_8(C, Sh, \mathcal{D}, \mathcal{S}), \texttt{ans}_9(C, Pe, Pr, \mathcal{D}, \mathcal{S}). && (r_3)\\
\texttt{ans}_8(C, Sh, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{triple}(Sh, \texttt{"offers"}, C, \mathcal{D}). && (r_4)\\
\texttt{ans}_8(C, Sh, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{triple}(Sh, \texttt{"offers"}, C, \mathcal{S}). && (r_5)\\
\texttt{ans}_9(C, Pe, Pr, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{ans}_{18}(C, Pe, \mathcal{D}, \mathcal{S}), \texttt{ans}_{19}(C, Pr, \mathcal{D}, \mathcal{S}). && (r_6)\\
\texttt{ans}_{18}(C, Pe, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{triple}(C, \texttt{"reduce"}, Pe, \mathcal{D}). && (r_7)\\
\texttt{ans}_{18}(C, Pe, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{triple}(C, \texttt{"reduce"}, Pe, \mathcal{S}). && (r_8)\\
\texttt{ans}_{19}(C, Pr, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{triple}(C, \texttt{"on"}, Pr, \mathcal{D}). && (r_9)\\
\texttt{ans}_{19}(C, Pr, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{triple}(C, \texttt{"on"}, Pr, \mathcal{S}). && (r_{10})\\
\texttt{ans}_5(Ge, Pr, Sh, U, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{ans}_{10}(Ge, Pr, \mathcal{D}, \mathcal{S}), \texttt{ans}_{11}(Sh, U, \mathcal{D}, \mathcal{S}). && (r_{11})\\
\texttt{ans}_{10}(Ge, Pr, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{triple}(Pr, \texttt{"g\_classify"}, Ge, \mathcal{D}). && (r_{12})\\
\texttt{ans}_{10}(Ge, Pr, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{triple}(Pr, \texttt{"g\_classify"}, Ge, \mathcal{S}). && (r_{13})\\
\texttt{ans}_{11}(Sh, U, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{triple}(U, \texttt{"isNear"}, Sh, \mathcal{D}). && (r_{14})\\
\texttt{ans}_{11}(Sh, U, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{triple}(U, \texttt{"isNear"}, Sh, \mathcal{S}). && (r_{15})\\
\texttt{triple}(S, P, O, \mathcal{S}) &\leftarrow \boxplus^{30} \Diamond \texttt{striple}(S, P, O, \mathrm{s}_1). && (r_{16})\\
\texttt{triple}(S, P, O, \mathcal{S}) &\leftarrow \boxplus^{5} \Diamond \texttt{striple}(S, P, O, \mathrm{s}_2). && (r_{17})
\end{aligned}$$

Take Q_2 from Example 5, then $\tau_2(Q_2)$ is the following LARS program.

$$\begin{aligned}
\texttt{ans}_1(Pe, Pr, Sh, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{ans}_2(C, Ge, Pe, Pr, Sh, U, \mathcal{D}, \mathcal{S}),\ Pe \geq 20, Ge \neq 1. && (r_{18})\\
\texttt{ans}_2(C, Ge, Pe, Pr, Sh, U, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{ans}_4(C, Pe, Pr, Sh, \mathcal{D}, \mathcal{S}),\\
&\ \texttt{ans}_5(Ge, Pr, Sh, U, \mathcal{D}, \mathcal{S}). && (r_{19})\\
\texttt{ans}_4(C, Pe, Pr, Sh, \mathcal{D}, \mathcal{S}) &\leftarrow \boxplus^{30} (\Diamond \texttt{striple}(Sh, \texttt{"offers"}, C, \mathrm{s}_1) \wedge\\
&\phantom{\leftarrow \boxplus^{30} (}\Diamond \texttt{striple}(C, \texttt{"reduce"}, Pe, \mathrm{s}_1) \wedge\\
&\phantom{\leftarrow \boxplus^{30} (}\Diamond \texttt{striple}(C, \texttt{"on"}, Pr, \mathrm{s}_1)). && (r_{20})\\
\texttt{ans}_5(Ge, Pr, Sh, U, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{ans}_{10}(Ge, Pr, \mathcal{D}, \mathcal{S}), \texttt{ans}_{11}(Sh, U, \mathcal{D}, \mathcal{S}). && (r_{21})\\
\texttt{ans}_{10}(Ge, Pr, \mathcal{D}, \mathcal{S}) &\leftarrow \texttt{triple}(Pr, \texttt{"g\_classify"}, Ge, \mathcal{D}). && (r_{22})\\
\texttt{ans}_{11}(Sh, U, \mathcal{D}, \mathcal{S}) &\leftarrow \boxplus^{5}_{\tau} \Diamond \texttt{striple}(U, \texttt{"isNear"}, Sh, \mathrm{s}_2). && (r_{23})
\end{aligned}$$

where $\mathrm{s}_1 = \texttt{"http://coupons"}$ and $\mathrm{s}_2 = \texttt{"http://locations"}$.

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

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

The following result shows that our translation preserves the translation in [Polleres, 2007].

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 [Polleres, 2007], and differ due to two approaches by C-SPARQL and CQELS in extending SPARQL to deal with streaming input. So far, it has not been clear under which conditions the two engines will return the same output. Tackling this question now becomes possible at a formal level using LARS.

Given an RDF triple (s, p, o) and a basic graph pattern g, we say (s, p, o) *sub-matches* g, denoted by $sm(s, p, o, g)$, iff there exists a triple pattern $(S, P, O) \in g$ s.t. $[\![(S, P, O)]\!]_{\{(s,p,o)\}} \neq \emptyset$, where the notion of subgraph matching $[\![.]\!]$ is defined in [Pérez et al., 2009]. Given a graph pattern P, let $trp(P)$ be the set of triple patterns appearing in P.

The following result identifies a class of RSP queries where the answer streams of the translated LARS programs by τ_1 and τ_2 coincide on the output predicate $\mathtt{ans}_1$.

Theorem 3 *Let $Q = (V, P, \mathcal{D}, \mathcal{S})$ be an RSP query where $\mathcal{D} = (G, G_n)$ contains a default graph G and a set G_n of named graphs, and $\mathcal{S} = \{(\mathrm{s}_1, \omega_1, g_1), \ldots, (\mathrm{s}_m, \omega_m, g_m)\}$. Let $P_1 = \tau_1(cs(Q))$, $P_2 = \tau_2(Q')$, for any $Q' \in cq(Q)$, D be a data stream, and t be a time point. If*

$$\forall g \neq g' \in \{\{trp(P) \setminus \bigcup g_i\}\} \cup \{g_1, \ldots, g_m\} \colon g \cap g' = \emptyset, \tag{$\star$}$$

$$\begin{aligned}&\forall \mathtt{striple}(s, p, o, \mathrm{s}_i) \in Ats(D) \colon sm(s, p, o, g_i) \textit{ and} \\ &\qquad \forall g_j \neq g_i \in \mathcal{S} \colon \neg sm(s, p, o, g_j) \textit{ and } \neg sm(s, p, o, trp(P) \setminus \bigcup g_j)\end{aligned} \tag{$\star\star$}$$

then $AS(P_1, D, t)|_{\mathtt{ans}_1} = AS(P_2, D, t)|_{\mathtt{ans}_1}$.

Condition ($\star$) requires that the graph patterns wrt. the static dataset and the input streams do not share triple patterns while ($\star\star$) makes sure that triples arrived at stream s_i are not allowed to enter any other stream or to stay in the static dataset. Combining these two conditions intuitively means that all input streams and static dataset have disjoint input. Then, the two approaches in building snapshots correspond as the distinction of input due to stream graph patterns in CQELS also happens for C-SPARQL. Thus, the answer streams produced by two translated LARS programs coincide on the output predicate.

5 RSP Semantics Analysis Based on LARS

Section 4.3 introduces translations from RSP queries on either C-SPARQL or CQELS branches into LARS programs. Under condition ($\star$) and ($\star\star$) in Theorem 3, the two translated LARS programs from a C-SPARQL and a CQELS queries, rooted from the same RSP query, produce the same output predicate $\mathtt{ans}_1$ (thus on `RStream` operator) when they are evaluated at the same time point.

However, C-SPARQL and CQELS are based on two different execution modes: push-based and pull-based, which are captured in Section 4.1 for general LARS programs. In order to theoretically analyze and compare the semantics of C-SPARQL and CQELS, we need to combine the above two results, together with taking into account the difference between `IStream` and `RStream` operators. But first of all, we must clarify what we mean by saying "C-SPARQL and CQELS agree on the output."

5.1 Agreement between C-SPARQL and CQELS

We propose a characterization of agreement between C-SPARQL and CQELS using LARS. For the core notion, we concentrate on the agreement on the resulted mappings after non-aggregate SPARQL operators such as AND, UNION, etc. Extending to aggregate will be discussed in Section 6.

Intuitively, the two semantics are considered to agree on a timeline T with a pulling period U, if (1) they both start at the same time point 0, and (2) for every interval $(i \cdot U, (i+1) \cdot U] \in T$, where $i \geq 0$, the union of outputs produced by CQELS in the interval coincides with the output produced by C-SPARQL at the right-end of the interval. To formalize the conditions for agreement, we need the notion of trigger time points and incremental output presented next.

Trigger Time Points. Let $t_1 < t_2$ be two time points. The set of *trigger time points* in a data stream D in the interval $(t_1, t_2]$ is defined as $ttp(t_1, t_2, D){=}\{t \in (t_1, t_2] \mid \upsilon_D(t){\neq}\emptyset\}$. For a time point $t \in T_D$ such that $t > 0$, the *previous trigger point* of t with respect to D is $prev(t, D) = \max(ttp(0, t-1, D))$ if $ttp(0, t-1, D) \neq \emptyset$ and is 0 otherwise.

Incremental Output. Next, we capture the incremental output strategy, i.e., the `IStream` operator by means of the difference between answer streams of two consecutive trigger time points. Let $I_t = AS(P, D, t)$. Then, the *incremental output* $inc(P, t)$ at a trigger time point t (i.e., $\upsilon_D(t) \neq \emptyset$) is $Ats(I_t \setminus I_{prev(t)})$ if $t > 0$, and $inc(P, 0) = Ats(I_0)$. Here, the difference between two streams $S_1 = (T, \upsilon_1)$ and $S_2 = (T, \upsilon_2)$ is defined as $S' = S_1 \setminus S_2 = (T, \upsilon')$ s.t. for all $t' \in T$, we have that $\upsilon'(t') = \upsilon_1(t') \setminus \upsilon_2(t')$.

Based on this, we define when two LARS programs, executed on push- and pull- modes, *agree* on an interval of time.

Definition 4 *Given two LARS programs P_1, P_2, a data stream $D = (T_D, \upsilon_D)$, and two time points $t_1{<}t_2$ of T_D, let $A_1{=}\bigcup_{t\in ttp(t_1,t_2,D)} inc(P_1, t) \cup \bigcap_{t\in ttp(t_1,t_2,D)\cup\{t_2\}} Ats(I_t)$ and $A_2 = Ats(AS(P_2, D, t_2))$. Let $R = \{p_1, \ldots, p_n\}$ be a set of predicates. We say P_1 and P_2* push-pull-agree *on D and R*

(i) during the interval $(t_1, t_2]$, *denoted by $P_1 \equiv^{D,R}_{t_1,t_2} P_2$, iff $A_1|_R = A_2|_R$;*

(ii) with pulling period U, *denoted by $P_1 \equiv^{D,R}_{U} P_2$, iff $P_1 \equiv^{D,R}_{t_1,t_2} P_2$, where $t_1, t_2 \in T_D$ such that there exists some $i \in \mathbb{N}$, where $t_1 = i \cdot U$ and $t_2 = (i+1) \cdot U$.*

Intuitively, push-pull-agreement during $(t_1, t_2]$ is established by comparing the answer stream evaluated at t_2 with the union of incremental answer computed at all trigger time points in the interval. The term $\bigcap_{t\in ttp(t_1,t_2,D)\cup\{t_2\}} Ats(I_t)$ ensures that for programs that always produce some output $p(\boldsymbol{c})$ at every time point, this output is also counted in comparing the incremental result and the result at t_2.

5.2 Agreement Conditions

Given an RSP query $Q = (V, P, \mathcal{D}, \mathcal{S})$, let $Q_1 = cs(Q)$ and $Q_2 \in cq(Q)$. We want to identify conditions guaranteeing that the LARS programs $\tau_1(Q_1)$ and $\tau_2(Q_2)$ agree on the output predicate $\mathtt{ans}_1$, that is $\tau_2(Q_2) \equiv^{D,\mathtt{ans}_1}_{t_1,t_2} \tau_1(Q_1)$.

Let $D = (T_D, \upsilon_D)$ be a data stream. The projection of D on an input stream identified by an IRI s is $D|_{\mathrm{s}} = (T_D, \upsilon_D|_{\mathrm{s}})$, where for all $t \in T_D$, we have that $\upsilon_D|_{\mathrm{s}}(t) = \{\mathtt{striple}(S, P, O, \mathrm{s}) \in \upsilon_D(t)\}$. That is, we keep only facts with s as the stream identifier. The *snapshot* of D with respect to $\mathcal{S}$ at time point t is defined as:

$$sn(D, \mathcal{S}, t) = \bigcup_{(s,\omega,g)\in\mathcal{S}} w_{\tau(\omega)}(D|_{\mathrm{s}}, t).$$

Intuitively, for each input stream pattern $(\mathrm{s}, \omega, g) \in \mathcal{S}$, we apply the window function $w_{\tau(\omega)} {=} w_{\boxplus}$ to the projection of D on s. Note that $\tau(\omega)$ translates the window expression ω to a window operator $\boxplus$ (Table 1), and $w_{\boxplus}$ is the window function behind $\boxplus$. The union of all substreams extracted by the window functions give us the snapshot. The following result identifies sufficient conditions where C-SPARQL and CQELS agree.

Theorem 4 *Let $Q{=}(V, P, \mathcal{D}, \mathcal{S})$ be an RSP query, where P contains neither* MINUS *nor* FILTER NOT EXISTS*, $\mathcal{D} = (G, G_n)$ contains a default graph G and a set G_n of named graphs, and $\mathcal{S} = \{(\mathbf{s}_1, \omega_1, g_1), \ldots, (\mathbf{s}_m, \omega_m, g_m)\}$ contains only time-based windows of the form* `[RANGE L]`*. Let $Q_1 = cs(Q)$, $Q_2 \in cq(Q)$, and $t_1 < t_2$. If $(\star)$ and $(\star\star)$ hold, and additionally*

$$\bigcup_{t \in ttp(t_1, t_2, D)} sn(D, \mathcal{S}, t) = sn(D, \mathcal{S}, t_2) \qquad (\star\star\star)$$

then

$$\tau_2(Q_2) \equiv_{t_1,t_2}^{D,\mathtt{ans}_1} \tau_1(Q_1).$$

This result can be straightforwardly extended to check whether $\tau_1(Q_1) \equiv_{U}^{D,\mathtt{ans}_1} \tau_2(Q_2)$, but is omitted due to space reason. The theorem shows that having agreement between C-SPARQL and CQELS is not easy to achieve, as discussed in the next section.

6 Discussion and Conclusion

Theorem 4 identifies sufficient conditions on which C-SPARQL and CQELS agree on their output, including (i) no MINUS or FILTER NOT EXISTS operator, (ii) only time-based windows with sliding size 1, (iii) only "disjoint" patterns and data in the static datasets and the input streams, and (iv) having the same snapshot collected in the interval as at the right end of the interval.

While (i)-(iii) correspond to useful fragments of queries for practical purposes, (iv) cannot be guaranteed in case of high throughput. The reason is that with dense input streams, the snapshots taken at time points near the left end of an interval will have high chances to collect more triples than the snapshot at the right end of the interval. Thus, having C-SPARQL and CQELS agreeing in practice is very unlikely, due to the strong semantic implications of push/pull-based querying. Consequently, data independent agreement conditions are unlikely to be found for queries that go beyond pure SPARQL.

One can easily find a counter example for the agreement when relaxing any of (i)-(iii) and keeping other conditions unchanged. For example, when FILTER NOT EXISTS or MINUS is allowed, the translated LARS programs are not positive. This takes away the monotonic property, i.e., having more input one any side (push- or pull-based) might lead to shrinking the output facts and introducing disagreement on the output.

For sliding windows with sliding size greater than 1, C-SPARQL can produce output that CQELS cannot, even when $(\star\star\star)$ is satisfied. Intuitively, this is because the window only slides after a certain amount of time and might miss some most recent input. In this case, we think that pull-based is preferable over push-based execution.

Finally, if the static datasets and the input streams share patterns by which triples are matched for R2R operators, the result of C-SPARQL and CQELS will be different. For instance, if these datasets share the same pattern and the static dataset contains some triples matching this pattern, then C-SPARQL can produce output even when no input triple arrives at the stream, as it cannot distinguish where a triple comes from. On the other hand, the stream graph pattern of CQELS has no mapping due to empty input, and thus produces no output. However, this situation should not happen often in practice as merged input streams will usually be distinguishable by an implicit schema. For instance, in our running example, the pattern `?user :isNear ?shop` can only match triples in the stream `<locations>`, since predicate `:isNear` will not be streamed in `<coupons>`.

The core notion of agreement does not consider aggregate. When considering aggregate, we observe that only certain types of aggregate allow for tracing agreement between pull- and push-based executions. For example, for COUNT, we can say that CQELS agrees with C-SPARQL in an interval $(t_1, t_2]$ iff the sum of output values produced by the former during the interval equal to the output value returned by the latter at t_2. Similar extension can be done for MAX, MIN. However, with MEDIAN or AVG, one cannot reproduce the result from CQELS' output to match that from C-SPARQL. In general, we can only give agreement notion for aggregates that can be recursively defined.

Conclusion and Outlook. This paper utilizes LARS to give insights on the contrast between two RSP semantics implemented in two representative engines, namely C-SPARQL and CQELS. Compared to the closest work in [Dell'Aglio et al., 2015], we made another important step forward by introducing a notion of agreement between the engines and identifying conditions for it to hold.

The theoretical result is based on the assumption that engine execution time is neglectable to the input rate. For further practical comparison, we envision future work where this condition is dropped. Implementing the proposed translations is also on our agenda. In another direction, we are investigating equivalence for general LARS programs. Once this result is available, one can have an automatic equivalence checker which takes any two translated LARS programs of two continuous queries from any two stream processing languages, tell whether the two original queries are equivalent, and possibly even enumerate their different outputs due to our model-based approach.

References

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

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, 2010a.

D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus. An execution environment for C-SPARQL queries. In *EDBT*, pages 441–452, 2010b.

H. Beck, M. Dao-Tran, T. Eiter, and M. Fink. Towards Ideal Semantics for Analyzing Stream Reasoning. In *ReactKnow*, 2014a.

H. Beck, M. Dao-Tran, T. Eiter, and M. Fink. Towards a Logic-Based Framework for Analyzing Stream Reasoning. In *OrdRing*, pages 11–22, 2014b.

H. Beck, M. Dao-Tran, T. Eiter, and M. Fink. LARS: A logic-based framework for analyzing reasoning over streams. In *AAAI*, 2015.

A. Bolles, M. Grawunder, and J. Jacobi. Streaming SPARQL - extending SPARQL to process data streams. In *ESWC*, pages 448–462, 2008.

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.

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.

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.

N. Dindar, N. Tatbul, R. J. Miller, L. M. Haas, and I. Botan. Modeling the execution semantics of stream processing engines with SECRET. *VLDB J.*, 22(4):421–446, 2013.

S. Groppe. *Data Management and Query Processing in Semantic Web Databases*. Springer, 2011.

M. Kolchin and P. Wetz. Demo: YABench - Yet Another RDF Stream Processing Benchmark. In *RSP Workshop*, 2015.

J. W. Lloyd and R. W. Topor. Making Prolog more Expressive. *J. Log. Program.*, 1(3):225–240, 1984.

J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34:16:1–16:45, September 2009. ISSN 0362-5915.

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.

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.

A. Polleres. From SPARQL to rules (and back). In *WWW 2007*, pages 787–796, 2007.

Y. Zhang, P. Minh Duc, O. Corcho, and J. P. Calbimonte. SRBench: A Streaming RDF/SPARQL Benchmark. In *ISWC*, pages 641–657, 2012.

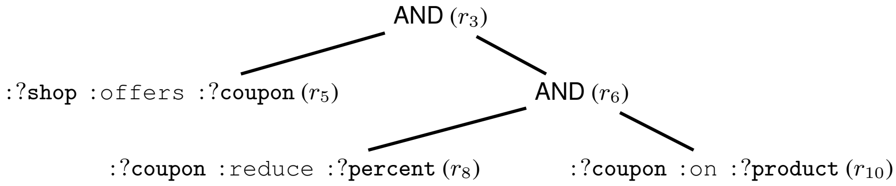

AND (r_3)
:?shop :offers :?coupon (r_5)
AND (r_6)
:?coupon :reduce :?percent (r_8)
:?coupon :on :?product (r_{10})

Figure 3: Structural analysis of rules constructed by τ_1 that are in correspondence with (r_{20}) by τ_2

A Proofs

A.1 Proof Sketch of Proposition 1

The rules concluding `trigger` mimics the pushed- and pull-modes. When `trigger` is concluded, the original LARS program P is evaluated. Therefore the "if" parts of (1) and (2) hold.

When `trigger` is not concluded, no rule in the translated LARS programs fires; therefore, no intensional fact will be concluded and thus the input stream D is the only answer stream. Therefore, the "else" part of (1) and (2) hold.

A.2 Proof Sketch of Proposition 2

The two translations τ_1 and τ_2 are established by (i) adding another argument $\mathcal{S}$ on τ and the predicates $\mathtt{ans}_i$ from [Polleres, 2007], and (ii) introducing new rules with τ'. When $\mathcal{S} = \emptyset$, it is easy to see that chopping off this argument in τ_1, τ_2 and the predicate $\mathtt{ans}_i$ gives us exactly τ. Furthermore, the additional rules in τ' will never fire during evaluating $\tau_1(cs(Q))$, $\tau_2(Q')$ since there is no facts to support them. Therefore, Proposition 2 holds.

A.3 Proof Sketch of Theorem 3

The conditions ($\star$) and ($\star\star$) intuitively means that all streams and static datasets provide disjoint input, both on schematic and data levels. Under these conditions, certain rules generated translation τ_1 can be disregarded; for example, rules (r_4), (r_7), (r_9), (r_{13}), and (r_{14}) in Example 11. These rules will never fire since they try to import triples of the form not provided by the respective sources.

Next, recall from [Phuoc et al., 2011] that the stream graph pattern contains only triple templates. This means when C-SPARQL merges such patterns into the default graph, we can see them as a pattern of triple templates connected by operator AND. Example 10 illustrates this observation: P_3 contains triple patterns connected with AND. It is plugged in the stream graph pattern of P_2 also appears in P_1.

Thus, we can choose an order in applying translation τ_1 on a C-SPARQL query that mimics the order when τ_2 is applied on its CQELS counterpart query. The two LARS programs in Example 11 was constructed in the same order: we split the AND operator between the first three triple templates and the last two (that is P_3 AND P_4) to mimic the split between the stream graph pattern (STREAM s ω P_3) and the pattern P_4 from the default graph.

Under the proposed ordering, the translated rules regarding the static datasets between τ_1 and τ_2 coincides (note the removed rules due to condition ($\star$)).

What we need to argue now is the correspondence between the translations of the pattern related to the input streams. For this, we see that when g is a set of triple templates, i.e., the combination of

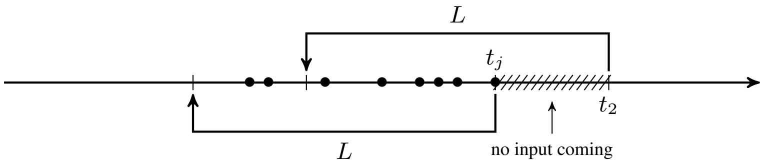


Figure 4: Illustration for proving (ii)

these triple templates by operator AND; the translation $\tau_2(V, (\mathsf{STREAM}\ \mathrm{s}\ \omega\ g), \mathcal{D}, \mathcal{S}, i)$ is a short hand for $\tau_1(V, g, \mathcal{D}, \mathcal{S}, i) \cup \tau'(\mathcal{S})$. The latter simply unfolds the binary tree where intermediate nodes are operators AND and leaf nodes are triple templates in g, while the former translate the whole graph pattern in one go. Figure 3 illustrates this correspondence on the pattern P_3 (Example 10). Furthermore, τ_2 applies the window and $\Diamond$ operators directly in the translation of the stream graph pattern. This is complemented in τ_1 by $\tau'(\mathcal{S})$.

A.4 Proof Sketch of Theorem 4

First, based on Theorem 3, condition $(\star)$ guarantees that Q_1 and Q_2 can be seen as "equivalent" on the output predicate $\mathtt{ans}_1$. To prove Theorem 4, we can start with a LARS program P translated from either Q_1 or Q_2 using the respective translation, and show that

$$P\ {}^{\rhd}{\equiv}^{\lhd}_{(t_1,t_2],\{\mathtt{ans}_1\}}\ P.$$

Now, we have that

$$\begin{aligned} A_1 &= \textstyle\bigcup_{t \in ttp(t_1,t_2)} inc(t) \cup \bigcap_{t \in ttp(t_1,t_2) \cup \{t_2\}} Ats(I_t) \\ A_2 &= Ats(AS(P, D, t_2)). \end{aligned}$$

We need to show two directions: (i) $A_1|_{\mathtt{ans}_1} \subseteq A_2|_{\mathtt{ans}_1}$, and (ii) $A_2|_{\mathtt{ans}_1} \subseteq A_1|_{\mathtt{ans}_1}$. Under the assumption that the graph pattern in Q does not contain MINUS, the translated LARS program P is positive, that is, there is no rule with negative atom in the body. This gives us a nice monotonic property that adding new facts into P does not shrink its model.

We therefore can accomplish the proof with the following intuition: a concluded fact of the predicate $\mathtt{ans}_1$ in A_1 or A_2 must be conducted from a set of supporting input facts from predicates `triple` or `striple`. Our idea to prove (i) and (ii) is to show that whenever such a set of supporing input facts is available to derive a fact ofthe predicate $\mathtt{ans}_1$ in push- or pull- execution mode, the set is also available in the other mode. Since we assume a static background dataset, we will concentrate in the following only on facts from predicate `striple`.

Prove (i). Assume that $\mathtt{ans}_1(\boldsymbol{c}) \in A_1$, we need to show that $\mathtt{ans}_1(\boldsymbol{c}) \in A_2$.

Indeed, suppose that $\mathtt{ans}_1(\boldsymbol{c}) \in I_t$, where $t \in (t_1, t_2]$. This means there exists a set $\{\mathtt{striple}(\boldsymbol{c}_1), \ldots, \mathtt{striple}(\boldsymbol{c}_k)\} \subseteq sn(t)$ that contributes to concluding $\mathtt{ans}_1(\boldsymbol{c})$.

By condition $(\star\star)$, we have $\{\mathtt{striple}(\boldsymbol{c}_1), \ldots, \mathtt{striple}(\boldsymbol{c}_k)\} \subseteq sn(t_2)$, therefore, $\mathtt{ans}_1(\boldsymbol{c}) \in I_{t_2}$.

Prove (ii). Assume that $\mathtt{ans}_1(\boldsymbol{c}) \in A_2$, we need to show that $\mathtt{ans}_1(\boldsymbol{c}) \in A_1$ by pointing out a trigger time point t' s.t. $\{\mathtt{striple}(\boldsymbol{c}_1), \ldots, \mathtt{striple}(\boldsymbol{c}_k)\} \subseteq sn(t')$.

Let t_j be a time point in $ttp(t_1, t_2)$ such that $ttp(t_j, t_2) = \emptyset$. Intuitively, this means that t_j is the maximal time point in the interval with input coming in. Given a stream $S = (T, \upsilon)$, under the condition that Q only has time-based window expression of the form [RANGE L], we have that

$$Ats(w((L,0,1),S,t_2)) \subseteq Ats(w((L,0,1),S,t_j)),$$

where w is the time-based window function defined in [Beck et al., 2015]. Intuitively, as illustrated in Figure 4, the window applied at t_j covers more time points with non-empty input than the application of the same window at t_2. Note that this only holds if the sliding size of the window is 1.

Therefore, we have $\{\mathtt{striple}(\boldsymbol{c}_1),\ldots,\mathtt{striple}(\boldsymbol{c}_k)\} \subseteq sn(t_j)$ and thus $\mathtt{ans}_1(\boldsymbol{c}) \in I_{t_j}$. From here, it is easy to see that $\mathtt{ans}_1(\boldsymbol{c}) \in A_1$.