On Correctness in RDF stream processor benchmarking

Daniele Dell'Aglio[1], Jean-Paul Calbimonte[2], Marco Balduini[1], Oscar Corcho[2] and Emanuele Della Valle[1]

[1]Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico of Milano
daniele.dellaglio@mail.polimi.it, balduini@elet.polimi.it,emanuele.dellavalle@polimi.it
[2]Ontology Engineering Group, Universidad Politécnica de Madrid
jp.calbimonte@upm.es,ocorcho@fi.upm.es

Abstract. Two complementary benchmarks have been proposed so far for the evaluation and continuous improvement of RDF stream processors: SRBench and LSBench. Each of them puts a special focus on different features of the evaluated systems, including coverage of the streaming extensions of SPARQL supported by each processor, query processing throughput, and an early analysis of query evaluation correctness, based on comparing the results obtained by different processors for a set of queries. However, none of them has analysed the operational semantics of these processors in order to assess the correctness of query evaluation results. In this paper, we first provide examples of the differences in the operational semantics of some existing RDF stream processors, namely CQELS, C-SPARQL and SPARQL$_{stream}$. Then we propose a characterization of their operational semantics, adapting well-known models used in the stream processing engine community: CQL and SECRET. Through this formalization, we address correctness in RDF stream processor benchmarks, allowing to determine the multiple correct answers that systems should provide. Finally, we present CSRBench, an extension of SRBench to address query result correctness verification using an automatic method.

1 Introduction

Ranging from weather station observations to real-time patient health monitoring, data streams are increasingly made available in our world and used by a wider range of applications. Data streams are characterized by their dynamicity (data changes continuously) and their variety (given by the heterogeneous set of data structures, schemas, and formats of the data made available by sensing devices or other data stream producers). Therefore, similarly to what we are experiencing nowadays with many datasets of a more static nature (including those available in relational databases, spreadsheets, etc.), several efforts [1–4] have focused on exposing data streams in RDF-aware formats or on making them available through SPARQL-based query languages, so that data streams can be more easily integrated with other data sources and queried in a uniform manner.

The systems that are being made available for this purpose (in this paper we refer to them as RDF stream processors or engines) support different forms of RDF streaming, including different query languages, constructs and operators, and different evaluation semantics for constructs that are syntactically similar. This is relatively normal in an area where standardization has not started yet. But at the same time it is important to describe clearly and homogeneously all these characteristics, so that such systems can be better understood by third parties and be improved continuously. This is the reason why benchmarking is a relevant activity in this context.

There are good examples of benchmarking activities both in the RDF world (e.g., LUBM [5] and BSBM [6]) and in the data streaming world (e.g., LinearRoad [7]). And some early efforts have been done as well in the context of RDF Streaming benchmarking: LSBench [8] and SRBench [9]. LSBench is mainly focused on understanding the throughput of existing RDF Stream processors and checking correctness by comparing the results of different processors and quantifying the mismatch among them. SRBench is mainly focused on understanding coverage for SPARQL constructs. However, they do not consider the different operational semantics of the benchmarked systems (stream-to-relation, relation-to-stream, etc.). In our work, we focus on correctness and propose a characterization of the operational semantics of RDF stream processors, so as to allow checking if results for a query are correct w.r.t. the proposed model. To summarize the contributions of this paper:

- We motivate the need to understand the operational semantics of RDF stream processors using simple examples (Section 2).
- We **define a model for describing such operational semantics** (Section 3), adapting well-known models used in the data stream management systems (DSMS) area: CQL [10] and SECRET [11].
- We **characterize existing RDF stream processors** according to the operational semantics model (Section 4).
- We **identify new dimensions** to be considered in RDF stream processor benchmarking, especially w.r.t. correctness evaluation, which are complementary to those from LSBench and SRBench (Section 5.1).
- We propose **CSRBench**, an extension of SRBench to address **correctness verification** (Section 5.2).
- We describe how to design (Section 6.1) and implement (Section 6.2) an **oracle to automatically check the correctness of results** given the operational semantics of the benchmarked system .
- We report on the results obtained applying CRSBench to C- SPARQL [1], SPARQLstream [2], CQELS [3] (Section 6.3).
- We elaborate on the lessons learned that could be used by RDF stream processor developers to improve their systems (Section 7).

2 Motivation

As discussed in the introduction, two main benchmarking activities for RDF stream processors have been proposed in the past years (LSBench and SRBench).

While these two evaluation efforts provide relevant contributions to the state of the art, one common limitation is that they do not consider checking the output produced by RDF stream processors. SRBench defines only functional tests in order to verify the query language features supported by the engines, while LSBench does not verify the correctness of the answers, but limits the analysis of correctness to the number of outputs. In sum, both benchmarks make two assumptions: 1) the tested systems work *correctly*, and 2) the tested systems have the *same* operational semantics. However, these assumptions do not always hold for all RDF stream engines, and hence these benchmarks may supply misleading information about them.

In fact, RDF stream processors do not always adhere to their operational semantics, as shown in [12]. Furthermore, even when RDF stream engines comply with their own semantics, these may differ from each other and therefore produce *different but correct* results. This means that it is considerably more difficult to compare these engines than those that process static SPARQL queries. Not only are correct answers determined by the input stream and the query, applying a given SPARQL extended algebra, but also by the operational semantics of each system. As an example, let's consider a simplified version of the scenario described in [3]: an RDF stream $\mathbb{S}$ reports the presence of persons in two rooms at a given time. More precisely $\mathbb{S}$ contains timestamped triples of the form $\langle p_i$ `:detectedAt` $r_j\rangle{:}t$, where p_i and r_j are person and room individuals and t is a timestamp, as depicted in Figure 1. The C-SPARQL query in Listing 1 asks for the room where two individuals m_1, m_2 are detected in a time window of 10s.

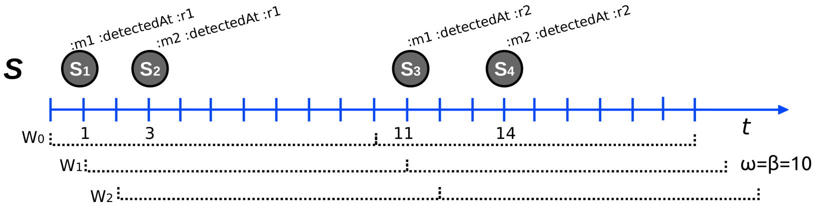

Fig. 1: Stream S of person p_i detected in room r_j, queried with window (ω, β).

```
REGISTER QUERY query AS SELECT ?room
FROM STREAM <http://www.example.org/stream1> [RANGE 10s STEP 10s]
WHERE { <http://ex.org/m1> <http://ex.org/detectedAt> ?room .
        <http://ex.org/m2> <http://ex.org/detectedAt> ?room }
```

Listing 1: C-SPARQL query retrieving the rooms where $m_1\, m_2$ are detected

Intuitively the expected answer would be `:r1` for the first window, and `:r2` for the second. This case is true if the window starts at time 0, as in W_0 in Figure 1, but it could also be the case that the engine starts windowing at $t = 1$ or $t = 2$ as illustrated by W_1 and W_2. For instance in the latter case the result is empty because m_1 and m_2 are never detected in the same window.

Other noticeable differences are reflected on the policies used by RDF stream engines for reporting and outputting query results. Consider the stream $\mathbb{S}$ in

Figure 2, and the same query of Listing 1 (and its equivalent queries in CQELS and SPARQL$_{stream}$). Even if they all start windowing at time 0, they produce outputs differently, as depicted in the figure. CQELS reports as soon as the content of the window changes, while the others do it only when closing the window. And systems like C-SPARQL may produce an output even if it is empty, while the others may produce no output at all in such cases. These and other differences, already spotted in [12], show the need to provide a formal framework for explaining the operational semantics of RDF stream processors.

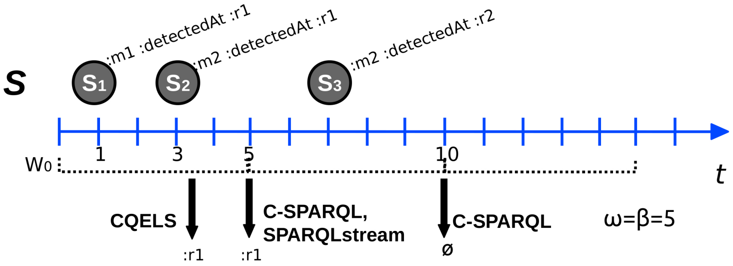

Fig. 2: Different RDF stream engine results report policies of q over Stream S.

These types of behavioral differences have been also detected in non-RDF stream processing engines. The SECRET [11] framework addressed this problem by formalizing a set of parameters to explain the variations in query execution, particularly those related to stream-to-relation operators, as we will describe in Section 3. Formally characterizing RDF stream processors using a model like SECRET is a key element to allow checking the correctness of query results.

3 RDF Stream Processor Characterization

Most RDF stream engines, including C-SPARQL, CQELS or SPARQL$_{stream}$, provide a processing model that can be described through the three main abstract operators defined in the classical DSMS [10] model, as illustrated in Figure 3. A portion of the infinite input stream is selected through an S2R (*stream-to-relation*) operator, typically a window, that produces (using SPARQL algebra terminology) a set of mappings. Then, the R2R operators can process the finite set of mappings (which would be named relation in the DSMS area) and transform it into another one, usually through the evaluation of a query (following the SPARQL 1.0/1.1 algebra) over the window content. Finally, the resulting set of mappings can be converted into a stream by an R2S operator (*relation-to-stream*, e.g. Rstream, Istream or Dstream [10]) and yield as the query output. It is worth noting that if the SPARQL query is a CONSTRUCT/DESCRIBE, the answer is an RDF graph, and the R2S operator will convert it into (part of) an RDF stream, which could be used as input for another RDF stream processor. On the contrary, if the R2R operator executes a SELECT/ASK query, the output will not be a RDF stream, but a relational stream.

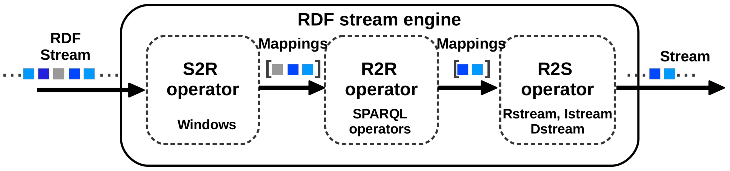

Fig. 3: RDF stream processor model: abstract operators.

In this work we focus mainly on the S2R and R2S operators, as the analysis of R2R operators is well addressed by existing benchmarks. Additionally, to characterize the window operator we refer to the model supplied by the SECRET [11] framework, originally designed to support the task of integrating streaming data processors, by means of explaining the different behavior of existing stream processing engines (SPEs). As we will see in the following, SECRET can also be used to model the behavior of the S2R operator of RDF stream processors.

Basic concepts. SECRET defines some basic concepts that we briefly present in this section. When appropriate, we indicate how they are applicable for RDF stream processors.

The *time domain* $\mathbb{T}$ is a discrete, linearly ordered, countably infinite set of time instants t. A *stream* $\mathbb{S}$ is a set of countably infinite elements. A *stream element* s is a triple[1] $\langle v, t^{app}, \tau^{sys} \rangle$, where v is a tuple (for RDF stream processors, it is an RDF statement) conforming to the schema S; t^{app} and τ^{sys} are time instants in $\mathbb{T}$ that represent the application time and system time, respectively. Unlike other works where only one timestamp is defined [10], SECRET takes into account two timestamps. t^{app} indicates the time instant associated to the event represented by the stream element. It can be shared by multiple elements (introducing the notion of *contemporaneity*) and consequently defines a partial order among the stream elements. τ^{sys} is the “wall-clock” time, which must be unique (i.e. no elements can share the same τ^{sys}), thus introducing a total order in the stream.

Even if the application time is the relevant information from a conceptual point of view, it is important to take into account the system time to explain the correct behavior of stream processors. The existence of two relevant timestamps and the related issues were defined and formalized in [13]. However, stream processor models usually refer only to application time. It is also worth noting that the timestamps are not part of the schema S, so they are usually not considered in the condition clause while writing queries. Anyway, some systems such as C-SPARQL relax this constraint by offering a `timestamp` function to get the application timestamp and include conditions over them in the queries.

The window operator. A *window* over a stream $\mathbb{S}$ is a finite subset W of $\mathbb{S}$. SECRET considers only time-based windows, even if other types exist in SPEs as well [10]. A *time-based window* over a stream $\mathbb{S}$ is a window with a time range

[1] The additional *batchid* parameter originally included in SECRET is not used in RDF stream processors to the best of our knowledge, so we omit it.

$[o, c)$ $(o, c \in \mathbb{T})$ that contains all and only the elements $s \in \mathbb{S}$ with application time t^{app} such that $o \leq t^{app} < c$. A window is *open* at time t if $t \leq c$, otherwise it is *closed*. An *active* window is the open window with the earliest starting time. Figure 4 shows an example of windows over a stream: at time $t = 6$ window W_1 has a range of $[1, 4)$, so it is closed. On the contrary, both W_2 (with range $[3, 6)$) and W_3 (with range $[5, 6)$) are open, and additionally W_2 is the active window. Windows are usually defined through the size and slide parameters. The window *size* (ω) can be computed as $\omega = c - o$. The *slide* (β) is the distance between two consecutive windows: for two consecutive windows $W_n[o_n, c_n)$ and $W_m[o_m, c_m)$, $\beta = o_m - o_n (\neq 0)$. The window in Figure 4 has size $\omega = 3$ and slide $\beta = 2$. When a time-based window slide is equals to the size, it is named *tumbling*. In this particular setting no stream elements are shared between windows.

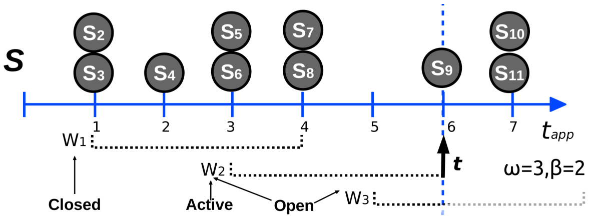

Fig. 4: Windows over a stream $\mathbb{S}$

The SECRET model describes the behavior of time windows through four functions: Scope, Content, Report and Tick.

Scope is a function that associates a time instant t to the time interval $[o_{aw}, c_{aw})$ of the active window aw at t. To compute the scope, it is necessary to provide the t_0 parameter, which is the application timestamp when the first active window starts. It is an absolute time and it depends on both the query (its registration time) and the RDF stream processor (how it processes the query and instantiates the window).

The *Content* function receives as input a pair $\langle t, \tau \rangle$ and identifies the set of elements of $\mathbb{S}$ in the active window at t with a system timestamp earlier than τ. This function depends not only on the scope (and consequently the application time), but also on the system time: it means that asking for the content of the active window at the same application time at two different system time instants can produce two different results.

Report is a function that receives as input a pair $\langle t, \tau \rangle$ and defines the required conditions to pass the window content to the R2R operator. SECRET identifies four reporting strategies (SPEs may use combined strategies as well):

- *Content change:* system reports if the content changes.
- *Window close:* system reports if the active window closes.
- *Non-empty content:* system reports if the active window is not empty.
- *Periodic:* system reports only at regular intervals.

The *Tick* function defines the condition under which the input can be added to the window, becoming processable by the query engine. SECRET defines different strategies: tuple-driven and time-driven. Systems with *tuple-driven* tick strategy add input tuples in the window when they arrive, while systems with *time-driven* tick strategy add sets of tuples to the window at each (application) time instant.

The R2S operator. When the window operator reports, the active window content is processed by the R2R operators, producing a timestamped set of mappings. The list of timestamped set of mappings is transformed then into a data stream (the system output), using the R2S operators defined by CQL [10]: Rstream, Istream and Dstream.

Rstream streams out the computed timestamped set of mappings at each step. Rstream answers can be verbose as the same mapping could be in different portions of the output stream computed at different steps. It is suitable when it is important to have the whole SPARQL query answer at each step, e.g., discover popular topics in the last time period in a social network.

Istream streams out the difference between the timestamped set of mappings computed at the last step and the one computed at the previous step. Answers are usually short (they contain only the difference) and consequently this operator is used when data exchange is expensive. Istream is useful when the focus in on the *new* mappings that are computed by the system, e.g., discover new relevant topics in a social network.

Dstream does the opposite of Istream: it streams out the difference between the computed timestamped set of mappings at the previous step and at the last step. Dstream is normally considered less relevant than Rstream and Istream, but it can be useful, e.g., to retrieve topics that are not relevant anymore.

Finally, we discuss the problem of the *empty answers*. During query execution, the output set of mappings could be empty, e.g., the R2R operator returns an empty answer and the R2S is an Rstream, or the R2S operator is Istream and the set of mappings at the last step is equal to the one at the previous step. In these cases RDF stream processors can either output an empty answer, or do not stream it out. This feature is different than the non-empty content report strategy (presented above), which is strictly related to the window: it imposes that the SPARQL query can be evaluated if the window content is not empty. This feature is related to the R2S operator: it works on the set of mappings that has to be streamed out by the system, so it implies that previously the window operator reported and the SPARQL query was evaluated.

4 Classification of RDF stream processors

In the previous section we presented a model to describe the operational semantics of existing RDF stream processors. The behavior of an RDF stream processor depends on three elements: the inner features of the system, the query, and the input data streams (e.g., it is not possible to determine the Content of a window if the input data is unknown). In this section we focus on the RDF stream processors features that can be described independently of the query and the input, and we classify RDF stream processors systems according to these parameters.

We consider C-SPARQL, CQELS and SPARQL$_{stream}$. C-SPARQL [1] is an RDF stream processor built on top of Esper[2] and Jena[3]: the first is used to manage the streams and the windows over them, while the second executes the SPARQL queries. CQELS [3] has a completely native implementation aimed at achieving higher performance. Finally, SPARQL$_{stream}$ [14] adopts an ontology-based data access to stream processing engines through query rewriting.

The systems are implemented in different ways, but their operational semantics can be explained by the model in Section 3. These descriptions are important not only to foresee how the systems have to work (and consequently to compute the expected correct results), but also to highlight the differences between them. We report in Table 1 the summary of our classification of RDF stream processors.

Feature	**Operator**	**C-SPARQL**	**CQELS**	**SPARQL$_{stream}$**
Report strategy	S2R	Window close and Non-empty content	Content-change	Window close and Non-empty content
Tick	S2R	Tuple-driven	Tuple-driven	Tuple-driven
Output operator	R2S	Rstream	Istream	Rstream, Istream and Dstream
Empty relation notification	R2S	Yes	No	No
Time unit		seconds	hundreds milliseconds	hundreds milliseconds

Table 1: Classification of the RDF stream processors.

Even if the Report function does not depend only on the RDF stream processor, the reporting strategies are strictly related to the system. C-SPARQL and SPARQL$_{stream}$ follow the window-close and non-empty content strategies: they evaluate the boolean query only if the active window is non-empty and when it closes. These strategies allow a strict control on the output rate (the window closes periodically) and a lower dependency on the input stream (only the presence of data in the window, regardless of its rate, is one of the conditions for reporting). On the other hand, CQELS adopts a content-change policy: it evaluates the boolean query on the active window every time its content changes (i.e., when new input data are added and when the active window slides and existing triples are removed). In this way the window operator is more dependent on the data and the system can be highly reactive, but it offers no output rate control. In the general case, if a burst arrives to the upstream system, the down stream system will also suffer it.

Regarding the Tick, the three systems behave the same: they adopt a tuple-driven approach, and move new data into the window as soon as available.

The C-SPARQL engine adopts an Rstream operator, outputting the empty answer when it computes it. CQELS implements the Istream operator, and it does not output empty answers. Finally, SPARQL$_{stream}$ implements all the three R2S operators and, similarly to CQELS, does not output empty answers.

[2] Cf. `http://esper.codehaus.org/`

[3] Cf. `http://jena.apache.org/`

The last parameter we consider is the (application) time unit. In the model presented in Section 3, the time is defined as a ordered set of discrete time instants. Anyway, to classify the systems, we need a piece of information: the minimal time unit that systems can support. In other words, what is the minimal difference between two timestamps that guarantee the correct behavior of the system. It is very complicated to determine this value, it depends on several factors, e.g., number of input streams managed by the system, data input rate, number of registered queries and their complexity, etc. In the environment we set up for our experiments, the time unit for C-SPARQL is in the order of seconds, while for CQELS and SPARQL$_{stream}$ it is in the order of hundreds milliseconds.

5 Extending RDF stream benchmarks

We have seen how existing RDF stream processors can be classified through the model we described in Section 3. We exploited the SECRET model to characterize the operational semantics of the window operators implemented by the engines. In this section we will use the model to address the problem of describing the execution of a query, given an input stream and a system previously characterized. This allows us to check the correctness of RDF stream engines, by comparing the modeled output and the system actual output. We first identify in Section 5.1 the main dimensions that we consider, and the we show how we extended the SRBench benchmark for checking correctness in Section 5.2.

5.1 Problem Dimensions

There are three main dimensions that affect query results in RDF streaming query processing: system, query and input stream data. For the system dimension, we consider:

- *Report* and *tick* policies, as described in Section 3.
- Window's initial time t_0, which is used by the system to determine the window scope (and consequently the content). While this parameter is usually not configurable, it can be inferred in a post-execution analysis.
- Empty relation notification policy.
- Input stream timestamp policy.

For the query dimension, we mainly consider the S2R and R2S operators, largely neglected in previous benchmarking efforts. Checking correctness with different window configurations is one of the key elements of the proposed extensions. In particular we consider:

- *Window size*. Varying the window sizes (e.g. 10 s, 1 s, 100 ms, etc.) will result in different scopes, and consequently different window content.
- *Window slide*. Variations on the window slide (e.g. slide every 1 second, every 10 milliseconds, etc.) also produce differences on the scope. Window slides equal to the window size are used to indicate that a window is *tumbling*, while a slide smaller than the size produces a *sliding* window. We propose testing different combinations of these parameters.

– R2S operator. While some RDF stream systems provide only one default operator for the output, others allow explicitly indicating the type of R2S that is expected in the results.

For the input stream dimension, we can briefly mention the input data rate (e.g. triples per second), the window content size (e.g. number of triples), and the data stream distribution (constant, normal, bursts in the input data, etc.).

5.2 CSRBench: Correctness Extensions of SRBench

CSRBench is the extended benchmark for correctness checking, based on SRBench, taking into account the properties described in the previous section, and including the queries described below. To do so we have not modified the input data in any sort, but we have mainly modified the benchmark queries in order to stress the S2R operators, adding the following three types of queries to the existing of SRBench. Because the queries are parametrized, the different combinations are useful to produce a set of concrete queries that cover a wide range of cases.

Parametrized window size and slide. By varying the window size and slide, the query[4] in Listing 2 allows testing different cases: different window sizes and window slides (e.g. 10, 100, 1000 ms., etc.).

Therefore, the value assigned to these two parameters will allow obtaining sliding windows (slide is smaller than size) or tumbling windows (slide is equal to size).

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
SELECT ?sensor ?tempvalue ?obs
FROM NAMED STREAM <http://cwi.nl/SRBench/observations> [NOW - %WSIZE% MS SLIDE %WSLIDE% MS]
WHERE { ?obs om-owl:observedProperty weather:_AirTemperature ;
             om-owl:procedure ?sensor ;
             om-owl:result [om-owl:floatValue ?tempvalue] .
        FILTER(?tempvalue > %TEMP%) }
```

Listing 2: Parametrized window slide and size example in SPARQL$_{stream}$

Parametrized aggregate query. Aggregate queries are commonly used in stream processing, as the focus is sometimes on data trends and summarization rather than on individual data points. Aggregates pose challenges to the computation of the window content, and depending on the streaming processor report and tick policies, the results of a sum, average or other function may greatly vary. This type of issues are often overlooked when querying single stream triples.

[4] Full query descriptions in the three languages available at: http://www.w3.org/wiki/CSRBench.

Joins of triples in different timestamps. The previous queries include graph pattern matching of triples that are typically received at the same timestamp (or nearly): e.g. an observation and its value, its type, etc. However, there are cases where queries including joins at different timestamps may be relevant. This is more challenging for query engines and correctness checking. For instance the query in Listing 3 asks for sensor stations that record a high atmospheric temperature variation, in a time window.

It is worth noting that this query produces answers when the temperature increases or decreases (there is no control about the order of the observations, so `?value1` could be before or after `?value2`). If we would like to write the query that looks for increasing temperature values, we should write a multi-window query, or we need a mechanism to put constraints on the application timestamps in the query, such as C-SPARQL's `timestamp` function.

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
REGISTER QUERY q AS SELECT ?sensor ?ob1 ?value1 ?obs
FROM NAMED STREAM <http://cwi.nl/SRBench/observations>[RANGE %WSIZE% S STEP %WSLIDE% S]
WHERE { ?ob1 om-owl:procedure ?sensor ;
             om-owl:observedProperty weather:_AirTemperature ;
             om-owl:result [om-owl:floatValue ?value1].
        ?ob2 om-owl:procedure ?sensor ;
             om-owl:observedProperty weather:_AirTemperature ;
             om-owl:result [om-owl:floatValue ?value2].
        FILTER(?value1-?value2 > %VARIATION_THRESHOLD%)  }
```

Listing 3: Query joining triples of different timestamps example in C-SPARQL

6 An Oracle for Correctness Checking

Once the benchmark is defined, we need a way to check if the results, provided by a system to the benchmark queries and input, correspond to the expected ones according to the system operational semantics. For this we propose an *oracle* that generates and compares results of RDF stream processors and check their correctness. The oracle works as follows: given a stream $\mathbb{S}$, a continuous query q, and an operational semantics M that describes how the target system should work (Section 4), it produces a result r^o. Furthermore, given a system result r^s for the same query q and input $\mathbb{S}$, the oracle checks the correctness of r^s by comparing it to the theoretical answer r^o (see Figure 5).

6.1 Oracle design

The oracle is based on the off-line execution of a continuous query q, translated to a static query q', over an RDF dataset that simulates an RDF stream. The oracle operates in two main stages: (i) the setup of the dataset, and (ii) the execution and comparison of the results.

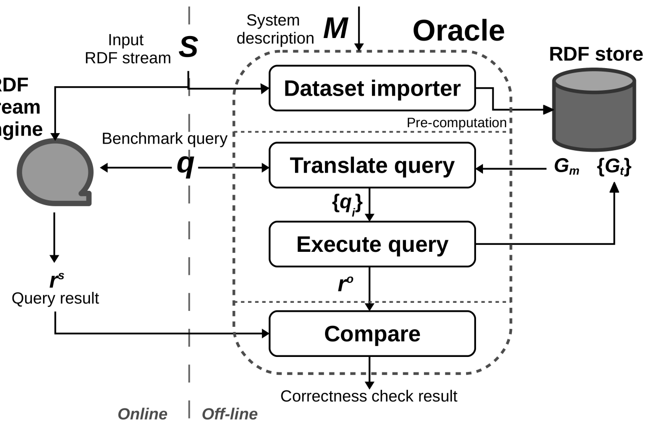

Fig. 5: Oracle for RDF Stream query results correctness checking.

The main goal of the dataset setup is to produce an RDF dataset that simulates the real RDF stream input, and is composed of a metadata RDF graph G_m and a set of data RDF graphs $\{G_t\}$, defined as follows. The original RDF stream $\mathbb{S}$ is composed of timestamped triples of the form $(\langle s, p, o\rangle, t)$, where t is the application timestamp, following the model of [15]. For each timestamp t in $\mathbb{S}$, a corresponding data graph G_t is created, and also the following metadata triple is added in G_m: $\langle G_t$:$\mathsf{hasTimestamp}$ $t\rangle$. Finally, for each t, the triples of $\mathbb{S}$ with timestamp t are imported into G_t, i.e. $G_t = \{\langle s, p, o\rangle \mid (\langle s, p, o\rangle, t) \in \mathbb{S}\}$.

Once the RDF dataset is set up, the oracle can execute the benchmark continuous queries over it, by translating them into a set of plain SPARQL queries that will be successively executed over one or more of the G_t graphs. This successive execution simulates a continuous query over a limited amount of time. The translation performed by the oracle in this work mainly considers the window operators of the continuous query, and can be summarized as follows.

1. the oracle computes the set of values to be assigned to t_0: $\{t_0^1, t_0^2, \ldots, t_0^k\}$;
2. it sets t_0 to the next value t_0^i;
3. it determines the *scope* of the next window that will report. To do it, the oracle considers the report strategy of the target system and it computes the time interval $[o, c)$ of the window;
4. next, the scope $[o, c)$ is used to determine the window *content*. The oracle selects which of the G_t graphs contains the data of the active window: given the scope $[o, c)$, the relevant graphs are $\{G_t \mid \langle G_t, \text{:}\,\mathsf{hasTimestamp}, t\rangle \in G_m \text{ and } o \leq t < c\}$;
5. the window content is evaluated by the query engine. The oracle executes a plain SPARQL query over the graphs determined by the previous step. The SPARQL query preserves the graph patterns and output modifiers of the original query q, only omitting the named window operators;
6. the answer of the query is a timestamped set of mappings, and it is added to the oracle output stream $r^o(t_0^i)$. The conversion depends on the R2S operator definition contained in the target system description;
7. if $r^o(t_0^i)$ is not complete, the oracle moves to step 3.

When the oracle results $r^o(t_0^i)$ is completely streamed out (the window slided over the whole stream $\mathbb{S}$), the oracle compares it with the RDF system output r^s. If they match, the experiment is successful, i.e., the system output is correct w.r.t. its operational semantics M, and for input $\mathbb{S}$ and query q. Otherwise, the oracle computes a new output stream starting from step 2. If none of the output streams generated by the oracle $r^o(t_0^1), r^o(t_0^2), \ldots, r^o(t_0^k)$ matches the output stream r^s of the system, the experiment fails.

6.2 Implementation

We have implemented a prototype of the proposed oracle and made it available as an open source project[5]. The project repository supplies also all the resources required to repeat the experiments: the input stream (with different streamer implementations for the analyzed systems), the queries, and the code to execute them in C-SPARQL, CQELS and SPARQL$_{stream}$.

The oracle is built on the top of the Sesame framework, it implements the algorithm described in the previous section and it is able to verify (off-line) the results produced by the RDF stream processors. It can manage queries with one time-based sliding window over a stream, it supports the whole SPARQL 1.1 query language, and it implements the three R2S operators Rstream, Istream and Dstream. The oracle is configurable and it is possible to change both the input stream and the benchmark queries. In this way it can also be used by RDF stream processor developers to set up testing environments while implementing their systems.

We plan to improve the oracle with several extensions. We aim to improve output mechanism: at the moment the match between an oracle result r^o and the system result r^s provides a boolean answer: true if r^s is contained (or it is equal) to r^o, and false otherwise. We plan to provide a more expressive matching mechanism, through the introduction of a precision rate, to help the analysis of the results. Additionally, we plan to take into account also the verification of quality of service metrics, such as the fact that the systems may provide results with a maximum delay from the theoretical output time.

6.3 Experiments with RDF stream engines

We have used the oracle to check the correctness of the CSRBench queries, for the three representative RDF stream engines already classified in Section 4, and using their latest available implementations (as of April 2013)[6]: C-SPARQL (0.9), CQELS (Aug 2011) and SPARQL$_{stream}$ (1.0.5). Because the benchmark extensions put stress on the window operators, we left out other RDF stream implementations such as EP–SPARQL [4], which do not include them.

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

[6] C-SPARQL: http://streamreasoning.org/download, CQELS: http://code.google.com/p/cqels/, SPARQL$_{stream}$: https://github.com/jpcik/morph-streams

The dataset used for experiments consists of a subset of the LSD dataset of SRBench, which comprises weather observations from hurricanes in the US. Only the data from hurricane Charley has been used, for a total of 3 hours of records. Data is replayed with parametrized input rates. We defined 7 queries, by instantiating the parameters of the three types of queries defined in Section 5:

- *Q1.* Query latest temperature observations and its originating sensor, filtered by a threshold. $\omega = 10s$, $\beta = 10s$, tumbling window.
- *Q2.* Query latest temperature observations and its originating sensor, filtered by a threshold. $\omega = 1s$, $\beta = 1s$, tumbling window.
- *Q3.* Query latest relative humidity observations and its originating sensor, filtered by a threshold. $\omega = 4s$, $\beta = 4s$, tumbling window.
- *Q4.* Query latest average temperature value, filtered by a threshold. $\omega = 4s$, $\beta = 4s$, tumbling window.
- *Q5.* Query latest temperature observations and its originating sensor, filtered by a threshold. $\omega = 5s$, $\beta = 1s$, sliding window.
- *Q6.* Query latest sensors having observations with a variation of temperature values higher than a threshold. $\omega = 5s$, $\beta = 5s$, tumbling window.
- *Q7.* Query latest sensors having observations with higher temperature values than a fixed sensor station. $\omega = 5s$, $\beta = 5s$, tumbling window.

Query	C-SPARQL	CQELS	SPARQL$_{stream}$
Q1	✓	✓	✓
Q2	✓	✓	✓
Q3	✓	✓	✓
Q4	✓	×	×
Q5	×	✓	✓
Q6	✓	×	✓
Q7	✓	×	✓

Table 2: Correctness checking: results for C-SPARQL, CQELS and SPARQL$_{stream}$.

7 Conclusions and Future work

As shown in Table 2, none of the RDF stream engines successfully passes all the tests. This provides an idea of the difficulty of assessing correctness in this type of systems. We now describe the cases where there are some failures.

Queries *Q1*, *Q2* and *Q3* focus on variations of the window size and slide, for the case of tumbling windows. All the systems behave in the correct way and provide the correct answers. These results are for the most part very similar in terms of content, as the graph pattern of the queries operates over (almost) contemporaneous triples. The main difference is on the timing of the output. The report policy of CQELS enables an almost immediate answer after a match is produced. This behavior interestingly hides any difference on the use of a slide, and consequently the results for *Q1* and *Q2* are virtually identical in CQELS. Also, smaller windows such as the one in *Q2* forced to configure the time resolution of the processing engine, in the case of SPARQL$_{stream}$, which otherwise would be unable to slide at the given rate. In the case of SPARQL$_{stream}$, the results in these three queries with C-SPARQL is noticeable on the absence of any output when no matches are produced. This situation must not be confused with the absence of data in the input stream.

The other four queries exploited unexpected behaviours of the analysed systems. C-SPARQL does not pass the experiment with *Q5*: this query highlights the use of an explicitly controlled slide, smaller than the window size. The problem is related to the fact that when a query is registered in C-SPARQL, there is a transitory phase on which some open windows are erroneously reported. When the system becomes stable and the first window closes, C-SPARQL starts to behave correctly and works as expected. This wrong behaviour is related to the sliding windows, in case of tumbling windows C-SPARQL works correctly.

SPARQL$_{stream}$ does not behave in the correct way with *Q4*. In general, this query poses challenges in several aspects. The first and most obvious is related to the t_0 parameter (initial windowing time). Because the first window starts at different points in each system, the resulting average values are completely different. The oracle adequately handles these variations, by computing results for different possible values of t_0. Nevertheless, other issues arise on the way aggregates are implemented in the absence of matches in the graph patterns. In this particular case, SPARQL$_{stream}$ outputs a `null` value instead of a 0 average value. This unexpected behavior is the cause of the failure in *Q4*, although in other cases the resulting values are correct. It is also worth mentioning that because SPARQL$_{stream}$ uses a underlying SPE through query rewriting, by changing it with another implementation, its modelled parameters (e.g. report, tick, etc.) could also change. Therefore its operational semantics depend on the underlying system it uses in a particular deployment.

Even CQELS does not provide the correct answer on *Q4*, and additionally it shows wrong behaviours on *Q6* and *Q7*. Both *Q6* and *Q7* focus on the evaluation of joins in triples with different timestamps. In the first case the equality join is on the observation sensor, which is a URI in the dataset, while in the second it is basically a cross-product of a single fixed observation against all observations in the window. In this case, the problem is given by the fact that CQELS does not correctly remove the RDF statements from the active window. As result there are aggregations and joins on elements that should not be in the window anymore, and the system produces additional wrong mappings. This behaviour does not emerge with other queries, thanks to the Istream operator: when queries filters the input stream, the answers are computed looking for triple patterns over data with the same application timestamp. Consequently, only answers obtained from new data entering the window will be output, due to the fact that the data already present in the window produced answers that were output in previous steps.

In this work we made an effort to cover an existing gap in current benchmarks for RDF stream processors. Checking the correctness of streaming query results is complementary to other tests such as functional coverage, performance, scalability, etc., but it is also key to assess how a system complies to its operational semantics. A comparison among this type of systems is not possible if we are unable to judge whether their output is correct or not. To do so we have introduced a parametrized model based on SECRET, that provides a formal way of explaining the operational semantics of RDF stream systems.

We aim to improve our work in several directions. Some extensions to the oracle development were already anticipated in Section 6. Additionally, we are interested in studying the behaviour of systems when the query defines multiple windows. The SECRET framework does not cover this point, and there are no indications on the report strategies in these cases, i.e., when there are two or more windows is necessary to define strategies taking into account the status of all of them. Finally, we would like to extend the approach to verify the correctness of RDF stream processors that do not use windows, such as EP-SPARQL: it is important to understand the operational semantics of those systems, and discover the similarities and the differences between them and the systems we targeted in this work.

References

1. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: A continuous query language for RDF data streams. IJSC **4**(1) (2010) 3–25
2. Calbimonte, J.P., Jeung, H., Corcho, O., Aberer, K.: Enabling Query Technologies for the Semantic Sensor Web. IJSWIS **8**(1) (2012) 43–63
3. Le-Phuoc, D., Dao-Tran, M., Xavier Parreira, J., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: ISWC. (2011) 370–388
4. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: WWW. (2011) 635–644
5. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Journal of Web Semantics **3**(2-3) (2005) 158–182
6. Bizer, C., Schultz, A.: The berlin SPARQL benchmark. IJSWIS **5**(2) (2009) 1–24
7. Arasu, A., Cherniack, M., Galvez, E.F., Maier, D., Maskey, A., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear Road: A Stream Data Management Benchmark. In: VLDB. (2004) 480–491
8. Le-Phuoc, D., Dao-Tran, M., Pham, M.D., Boncz, P., Eiter, T., Fink, M.: Linked stream data processing engines: Facts and figures. In: ISWC. (2012) 300–312
9. Zhang, Y., Duc, P., Corcho, O., Calbimonte, J.P.: SRBench: A Streaming RDF/SPARQL Benchmark. In: ISWC. (2012) 641–657
10. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language : semantic foundations. The VLDB Journal **15**(2) (2006) 121–142
11. Botan, I., Derakhshan, R., Dindar, N.: Secret: A model for analysis of the execution semantics of stream processing systems. PVLDB **3**(1) (2010) 232–243
12. Dell'Aglio, D., Balduini, M., Valle, E.D.: On the need to include functional testing in rdf stream engine benchmarks. In: BeRSys 2013. (2013)
13. Srivastava, U., Widom, J.: Flexible time management in data stream systems. In: PODS, New York, New York, USA (2004) 263
14. Calbimonte, J.P., Corcho, O., Gray, A.J.G.: Enabling ontology-based access to streaming data sources. In: ISWC. (2010) 96–111
15. Gutierrez, C., Hurtado, C., Vaisman, A.: Introducing time into RDF. IEEE Transactions on Knowledge and Data Engineering **19**(2) (2007) 207–218