Query Rewriting in RDF Stream Processing

Jean-Paul Calbimonte[1], Jose Mora[2], and Oscar Corcho[2]

[1]Faculty of Computer Science and Communication Systems, EPFL, Switzerland.
firstname.lastname@epfl.ch
[2] Ontology Engineering Group, Universidad Politécnica de Madrid, Spain.
j.mora@upm.es, ocorcho@fi.upm.es

Abstract. Querying and reasoning over RDF streams are two increasingly relevant areas in the broader scope of processing structured data on the Web. While RDF Stream Processing (RSP) has focused so far on extending SPARQL for continuous query and event processing, stream reasoning has concentrated on ontology evolution and incremental materialization. In this paper we propose a different approach for querying RDF streams over ontologies, based on the combination of query rewriting and stream processing. We show that it is possible to rewrite continuous queries over streams of RDF data, while maintaining efficiency for a wide range of scenarios. We provide a detailed description of our approach, as well as an implementation, StreamQR, which is based on the kyrie rewriter, and can be coupled with a native RSP engine, namely CQELS. Finally, we show empirical evidence of the performance of StreamQR in a series of experiments based on the SRBench query set.

1 Introduction

Streams are currently one of the main sources of data on the Web, and are used in a number of applications, ranging from wearable devices for health monitoring to geospatial and environmental sensing. Some of the main challenges of managing this very dynamic type of data are linked to the *velocity* of the inputs, and the need for reactive processing. While these have been addressed to a large extent by the database community, through *Data Stream Management Systems* (DSMS) and *Complex Event Processors* (CEP), there is still a need for tackling the issues of heterogeneity, integration and interpretation of data streams on the Web. Semantic Web technologies and standards have provided fundamental concepts and tools to address these issues, such as ontologies, RDF triple stores and reasoners, although most of these are targeted towards stored data. The goal of RDF stream processing (RSP) is to apply and extend the Semantic Web models and languages for processing RDF data streams. Previous works have presented RSP engines [4, 16, 1, 7, 24], focusing on different aspects of query processing. However, most of them provide limited or no reasoning capabilities, nor use an ontology model (TBox) to provide inferences during query processing. Other works have also explored the field of *stream reasoning*, but they have centered their attention mainly on the materialization of streaming axioms in ontologies [23, 17] and, to some extent, to query processing that takes

into account materialization rules [5]. The problem of answering queries over an ontology can be solved in different ways, and an important technique to do so is *query rewriting*. It is based on the idea of transforming the original query into an expanded query that captures the information of the ontology TBox [21]. Then this expanded query is evaluated over the ABox, providing answers that extract implicit knowledge of the data. This technique has been successfully used in scenarios such as OBDA (Ontology based data access) where data is stored in relational databases [22].

In this paper we address the problem of providing query answering over ontologies in RSP, through a novel approach that combines query rewriting techniques and RDF stream query processing. While most of the focus on query rewriting has been on OBDA for relational databases, we show that it is possible to use it for rewriting continuous queries over streams of RDF data. RDF streams, understood as potentially infinite flows of timestamped triples, require reactive processing of queries, and therefore most RSP engines have focused on efficiency and high throughput. Our approach demonstrates that these engines can be coupled with a query rewriter, and still be efficient for a large range of scenarios. Furthermore we implemented our solution, called StreamQR, extending the CQELS query processor with the kyrie [18] rewriting engine.

The paper is organized as follows. In Section 2 we introduce the notions related to RDF stream processing and query rewriting. In Section 3 we present our proposal for ontology query answering over data streams. Then, in Section 4 we describe the implementation of StreamQR. Section 5 provides details on the experimentation and applicability of this approach. In Section 6 we analyze and compare previous work in the areas of stream reasoning, query rewriting and RDF stream processing. Finally we draw our conclusions in Section 7 and identify future areas of research.

2 Preliminaries

2.1 RDF Stream Processing

Data streams are infinite and time-varying sequences of data values [2], and can also be seen as more complex events whose patterns can be queried and processed [12]. In the case of RDF stream processing (RSP) the elements of the stream are RDF data, typically annotated with a timestamp. Managing streaming data differs significantly from classical stored data, given the potentially infinite nature of streams and the need for continuous evaluation of queries. Continuous processing changes the usual query execution model, since it is the data arrival that initiates query processing, and produces results as soon as streaming data matches the query criteria.

Most RSP systems define a data model based on timestamped triples to represent RDF streams. As in [4, 7], we define an RDF stream S as a sequence of pairs (T, t) where T is a triple $\langle s, p, o\rangle$ and t is a timestamp in the infinite set of non-decreasing timestamps $\mathbb{T}$:

$$S = \{(\langle s, p, o\rangle, t) \mid \langle s, p, o\rangle \in ((I \cup B) \times I \times (I \cup B \cup L)), t \in \mathbb{T}\}$$

where I, B and L are sets of IRIs, blank nodes and literals, respectively. The pairs (T, τ) are called a *timestamped triples*. An RDF stream can be identified by an IRI, which allows referencing a particular stream of tagged triples.

Most of the state-of-the-art RSP query languages and systems are to a large extent based on Data stream management systems (DSMS) and complex event processing (CEP) extensions to the standard SPARQL query language. These extensions include operators such as windows, the ability to input and output RDF streams, and the declaration of continuous queries. Examples of such RSP systems include C-SPARQL [4], $\text{SPARQL}_{\text{Stream}}$ [7] and CQELS [16], which incorporate these notions, although with some differences in syntax and semantics. As an example, the CQELS query in Listing 1 requests the maximum temperature and the sensor that reported it in the last hour, from the stream identified as `http://example.org/stream` (prefixes are omitted for brevity).

```
SELECT (MAX(?temp) AS ?maxtemp) ?sensor
WHERE {
  STREAM <http://example.org/stream> [RANGE 1 HOUR] {
  ?obs ssn:observationResult ?result;
       ssn:observedProperty cf-property:air_temperature;
       ssn:observedBy ?sensor.
  ?result ssn:hasValue ?obsValue.
  ?obsValue qu:numericalValue ?temp. }
} GROUP BY ?sensor
```

Listing 1. Query the maximum temperature in the last hour in CQELS.

Sliding windows such as the one in the previous example are available in almost all RSP engines. They limit the scope of triples to be considered by the query operators. In particular, a *window* can be defined as a function that takes a stream S and a time instant $t \in \mathbb{T}$ and produces an RDF graph of triples. A more particular case, the time window W is defined by parameters σ, δ where σ is the size and δ is the slide, such that: $W(S,t) = \{T_j \mid (T_j, t_j) \in S \text{ and } t - \sigma \leq t_j < t\}$ and $t_{\ell+1} - t_\ell = \delta$ for two consecutive windows $W(S, t_\ell)$ and $W(S, t_{\ell+1})$.

2.2 Query Rewriting

Query answering using ontologies provides the capability of extracting explicit and implicit knowledge from a data source. In general, an ontology $\mathcal{O}$ is composed of a *TBox* $\mathcal{T}$ containing intensional knowledge and an *ABox* $\mathcal{A}$ containing extensional knowledge [9]. Query rewriting can help answering queries over ontologies, by transforming –or rewriting– the original query into an expanded query that takes into account the TBox, and using this rewritten query to evaluate it against the ABox part of the ontology. This technique has been used in the past in different scenarios, most notably Ontology-based Data Access (OBDA). In OBDA, data from one (or more) data sources can be queried in terms of a high-level ontological model, hiding from the users the internal schema and storage details of the data sources [21].

In general, a *query rewriting* algorithm works in the following way [9]. First, given an input query q and an ontology $(\mathcal{T}, \mathcal{A})$, it transforms q using the TBox $\mathcal{T}$ into a query q', such that for every ABox $\mathcal{A}$, the set of answers that q' obtains from $\mathcal{A}$ is equal to the set of answers that are entailed by q over $\mathcal{T}$ and $\mathcal{A}$. The rewritten query q' can normally be unfolded and expressed as a union of conjunctive queries, as long as some restrictions are imposed on the expressivity of the ontology language. As an example, consider the query $q(x) \leftarrow \texttt{Sensor}(x)$, requesting instances of `Sensor`, and the TBox axiom: $\texttt{HumiditySensor} \sqsubseteq \texttt{Sensor}$. Using this TBox assertion, query rewriting will produce the union of the following conjunctive queries, which will also request for all instances of `HumiditySensor`:

$$q'(x) \leftarrow \texttt{Sensor}(x) \qquad\qquad q''(x) \leftarrow \texttt{HumiditySensor}(x)$$

An important property for query rewriting is FOL-reducibility [9] or FO-rewritability [14], meaning that rewritten queries are first-order queries, what allows converting them to languages like SQL without using advanced features like recursion. It has been shown that the combined complexity of satisfiability on some of these logics is polynomial, while data complexity is AC^0 [3, 14].

The ontology TBox can be described using different languages or logics. Depending on their expressiveness, the query rewriting process can be more or less expensive in terms of computation. A logic with special relevance in our case is $\mathcal{ELHIO}$, one of the most expressive logics currently used for query rewriting. $\mathcal{ELHIO}$ is not FOL-reducible (in the presence of certain cyclic axioms the rewriting produces a recursive datalog program) but remains tractable (PTIME-complete [21]) for the rewriting process. For the description of our TBoxes we will use acyclic $\mathcal{ELHIO}$.

3 RSP Query Rewriting

Our approach for querying RDF streams over ontologies stems from the combination of the query rewriting techniques described previously and RDF stream query processing. As described in Section 2.2, the first lack support for continuous query processing and the second, in general, do not take into account the TBox of an ontology and hence do not perform any inferences during querying.

3.1 RSP Query Evaluation using the Ontology TBox

Most of the existing RSP engines do not make use an ontology TBox during query evaluation (excepting a materialization technique described in [5] which is not currently available in the C-SPARQL implementation). Consequently it might be the case that there are queries that return a reduced number of answers, or no answers at all, in contrast to what could be expected if TBox assertions were taken into account. For example, let's consider the following RDF graph G with triples of the form:

```
:obs1 rdf:type ssn:Observation .
:obs2 rdf:type ssn:Observation .
:obs3 rdf:type ssn:Observation .
```

If we pose the following query, requesting all instances that are observed by some other entity (e.g. a sensor):

$$q_G(x) \leftarrow \texttt{ssn:observedBy}(x, y)$$

Without any other information, the evaluation of this query over the stream would produce no results, since we do not have any match to a corresponding triple in the stream, for the `ssn:observedBy` property. However if we take into account TBox assertions, the situation may change. For instance, the presence of the following TBox axiom:

$$\texttt{ssn:Observation} \sqsubseteq \exists\ \texttt{ssn:observedBy}$$

would mean that all subjects of the triples in the stream (i.e. `obs1,obs2,obs3`) are also implicitly observed by someone. By rewriting $q(x)$ with this TBox, we would obtain an expansion consisting of the two following queries:

$$q'_G(x) \leftarrow \texttt{ssn:observedBy}(x, y)$$
$$q''_G(x) \leftarrow \texttt{ssn:Observation}(x)$$

While q' is just the original queries and will not produce answers, q'' (containing `ssn:Observation(x)`) will match during the evaluation over the RDF stream.

The previous rewriting examples can be extended to handle continuous evaluation of RDF data streams. For instance, we can consider the following stream based on the dataset of the previous example[1]:

```
:obs1 rdf:type ssn:Observation .  [1]
:obs2 rdf:type ssn:Observation .  [3]
:obs3 rdf:type ssn:Observation .  [6]
...
```

As the stream is potentially infinite, the complete answer to a query theoretically takes infinite time to evaluate, but partial answers may be computed in a continuous fashion, e.g., applying a sliding window W of 3 time units $\sigma = 3, \delta = 3$. Then we can compute continuous queries of the form $q_{W(S,t)}$ for every evaluation time t, computed over the instantaneous graph produced by $W(S, t)$:

$$q_{W(S,t)}(x) \leftarrow \texttt{ssn:observedBy}(x, y)$$

Then, for each instantaneous graph, all other SPARQL operators can be applied on top of the resulting operation. The time window operator is already incorporated into existing RDF stream languages such as CQELS, C-SPARQL, or $\text{SPARQL}_{\text{Stream}}$, as described in Section 2.1.

3.2 Query Answering Semantics

We describe in this section the semantics of our approach of query answering for data streams, over $\mathcal{ELHIO}$ ontologies, one of the most expressive logics that can be currently handled in query rewriting. An ontology $\mathcal{O}$ is composed of a TBox

[1] Turtle `http://www.w3.org/TR/turtle/`, extended with timestamps in square brackets.

$\mathcal{T}$ and an ABox $\mathcal{A}$, i.e. $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$. Given the inference capabilities provided by the TBox, the results to the queries that are posed to the ontology are called *certain answers*. These can be seen intuitively as the kind of answers that users would expect, i.e. the answers matching triples explicitly stated in the database and those entailed by the ontology. We constrain the queries in this work to *conjunctive queries* (CQ) of the form:

$$q_h(\boldsymbol{x}) \leftarrow p_1(\boldsymbol{x_1}) \wedge \ldots \wedge p_n(\boldsymbol{x_n})$$

where q_h is the head predicate of the query, $\boldsymbol{x}$ is a tuple of distinguished variables, $\boldsymbol{x_1} \ldots \boldsymbol{x_n}$ are tuples of variables or constants, and $p_i(\boldsymbol{x_i})$ are unary or binary atoms in the body of the query. Every variable $x_j \in \boldsymbol{x}$ in the head of the query also appears in the body of the query. The certain answers for this type of queries can be defined as the set:

$$cert(q, \mathcal{O}) = \{\boldsymbol{\alpha} \mid q \cup \mathcal{T} \cup \mathcal{A} \models q_h(\boldsymbol{\alpha})\}$$

where q_h is the head predicate of the query q over an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$.

The query rewriting process uses the TBox $\mathcal{T}$ to rewrite the query q into a new query q' such that:

$$cert(q, \mathcal{O}) = \{\boldsymbol{\alpha} \mid q' \cup \mathcal{A} \models q_h(\boldsymbol{\alpha})\}$$

In this work, q' is in fact a *union of conjunctive queries* (UCQ), which is a set of CQ with the same head. In the following, we describe: (i) the query rewriting semantics to obtain the UCQ q', and (ii) the adaptation of this semantics to a continuous evaluation.

$\mathcal{ELHIO}$ axiom	antecedent → consequent
$A \sqsubseteq \{a\}$	`?x rdf:type ex:A → ?x = ex:a`
$A1 \sqsubseteq A2$	`?x rdf:type ex:A1 → ?x rdf:type ex:A2`
$A1 \sqcap A2 \sqsubseteq A3$	`?x rdf:type ex:A1 ; rdf:type ex:A2 → ?x rdf:type ex:A3`
$A \sqsubseteq \exists P$	`?x rdf:type ex:A → ?x ex:P []:fx`
$A1 \sqsubseteq \exists P.A2$	`?x rdf:type ex:A1 → ?x ex:P [rdf:type ex:A1]`
$A \sqsubseteq \exists P^-$	`?x rdf:type ex:A → []:fx ex:P ?x`
$A1 \sqsubseteq \exists P^-.A2$	`?x rdf:type ex:A1 → [ex:P ?x ; rdf:type ex:A2]`
$\exists P \sqsubseteq A$	`?x ex:P ?y → ?x rdf:type ex:A`
$\exists P.A1 \sqsubseteq A2$	`?x ex:P [rdf:type ex:A1] → ?x rdf:type ex:A2`
$\exists P^- \sqsubseteq A$	`?y ex:P ?x → ?x rdf:type A`
$\exists P^-.A1 \sqsubseteq A2$	`[ex:P ?x ; rdf:type A1] → ?x rdf:type A2`
$P \sqsubseteq S$, $P^- \sqsubseteq S^-$	`?x ex:P ?y → ?x ex:S ?y`
$P \sqsubseteq S^-$, $P^- \sqsubseteq S$	`?x ex:P ?y → ?y ex:S ?x`

Table 1. $\mathcal{ELHIO}$ axioms as rules over RDF triples. Converted from Table 2 in [20].

Query Rewriting in $\mathcal{ELHIO}$ Query rewriting in our approach uses a non-recursive $\mathcal{ELHIO}$ ontology to rewrite the conjunctive query q into a union of conjunctive queries (UCQ) q' such that $cert(q, \mathcal{O}) = \{\boldsymbol{\alpha} \mid q' \cup \mathcal{A} \models q_h(\boldsymbol{\alpha})\}$. The details of the inference steps for the rewriting can be found in [18]. The transformation of the ontology to first order logic is performed according to the rules described in [21]. Afterwards, the UCQ is then converted into a union of basic graph patterns (BGPs). Since this conversion is merely syntactical, the semantics of the rules in $\mathcal{ELHIO}$ can be expressed over transformations on triple patterns, as detailed in Table 1.

Query Rewriting for Continuous Queries In the context of this work, the ontology is not considered to be static at query evaluation time, but it is available as a stream. Assuming that that the ABox of the ontology is dynamic and that the TBox does not change at query time, we can define an *instantaneous ABox* as $\mathcal{A}(t)$, for a given time t. An instantaneous ABox contains all assertions in the stream timestamped at time t. Then we can represent the stream as a sequence of ABoxes over time: $\mathcal{A}(0), \mathcal{A}(1), ..., \mathcal{A}(t_i), ...$ where t_i represents a time point. Furthermore, given a time window w with starting and ending times s and e, we can define $\mathcal{A}_w$ as the ABox consisting of the union of the instantaneous ABoxes $\mathcal{A}(t)$ such that $s \leq t < e$.

To provide continuous answers to queries on this stream of ABoxes, each query is executed over a sliding time window that limits the number of assertions of the stream. A slide parameter indicates how often this window is computed (as described in Section 2.1, but without loss of generality we can assume that the window size is equal to the slide. Then, if we assume t_0 as the initial evaluation time, given a query q_w over a window of size δ, it will be evaluated at times $t_0, t_0 + \delta, ..., t_0 + k\delta, ...$, with $k \in \mathbb{N}$. The certain answers for this query, for the k-th window can be defined as:

$$cert(q_{w_k}, \langle \mathcal{T}, \mathcal{A}_{w_k} \rangle) = \{\boldsymbol{\alpha} \mid q_w \cup \mathcal{T} \cup \mathcal{A}_{w_k} \models q_h(\boldsymbol{\alpha})\}$$

where the start and end times of the k-th window w_k define the contents of the ABox $\mathcal{A}_{w_k}$. This instantaneous query evaluation is compatible with the previous definition of certain answers given above, as we are referring to instantaneous snapshots of the ABox stream. Therefore, the query rewriting algorithms for static data sources can be used to compute the inferences on this instantaneous query. For the k-th window, the corresponding query q_{w_k} will be rewritten into a new query q'_{w_k} such that:

$$cert(q_{w_k}, \langle \mathcal{T}, \mathcal{A}_{w_k} \rangle) = \{\boldsymbol{\alpha} \mid q'_w \cup \mathcal{A}w_k \models q_h(\boldsymbol{\alpha})\}$$

As we will see in the next section, the time-window based query q' can be concretely implemented using an RDF stream processor that natively includes the window operator, as we have seen in Section 2.1. Given that queries are continuously evaluated, the rewriting process does not need to be performed on every window evaluation. Assuming that $\mathcal{T}$ does not change, the query rewriting can be executed only once.

4 RSP Rewriting Implementation

In this section we describe StreamQR, an implementation of the query rewriting approach presented in Section 3. This prototype is available as open-source code[2], and is based on two main components: (i) the query rewriter kyrie [18], which rewrites queries using $\mathcal{ELHIO}$ ontologies, and (ii) the RSP query engine CQELS [16]. The whole process is divided in a series of steps that are graphically summarized in Figure 1.

[2] Github StreamQR: `https://github.com/jpcik/streapler/tree/streamQR`

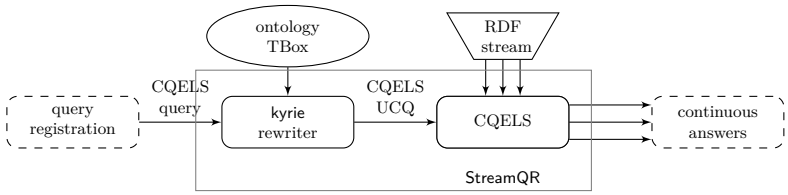


Fig. 1. High level architecture of StreamQR. The original query is rewritten by kyrie into a union of CQELS queries based on the ontology TBox. The rewritten queries are evaluated by the CQELS engine, on the incoming RDF stream.

RDF streams continuously feed StreamQR, specifically through the CQELS interface for consuming incoming streams, and allows registering queries specified in the CQELS language. In the following we describe the implementation of the rewriting and continuous execution.

As a running example, consider the following ontology TBox, including assertions about observations. For instance, temperature observations are observed by a temperature sensor, a thermistor is a type of temperature sensor, etc[3]:

$$\texttt{met:TemperatureObservation} \sqsubseteq \exists\ \texttt{ssn:observedBy.aws:TemperatureSensor}$$
$$\texttt{met:AirTemperatureObservation} \sqsubseteq \texttt{met:TemperatureObservation}$$
$$\texttt{met:ThermistorObservation} \sqsubseteq \texttt{ssn:TemperatureObservation}$$
$$\texttt{aws:Thermistor} \sqsubseteq \texttt{aws:TemperatureSensor}$$
$$\texttt{aws:CapacitiveBead} \sqsubseteq \texttt{aws:TemperatureSensor}$$

Registration. In this stage StreamQR takes as input an ontology TBox and a registered CQELS query, to produce a union of conjunctive queries (UCQ). For instance the following query asks for all instances observed by a temperature sensor in the last 10ms.

```
PREFIX ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX aws: <http://purl.oclc.org/NET/ssnx/meteo/aws#>
CONSTRUCT { ?o a :ObservedTemperature. }
WHERE {
  STREAM :stream1 [RANGE 10ms]  {
   ?o ssn:observedBy ?t .
   ?t a aws:TemperatureSensor.  }
}
```

Listing 2. CQELS query for the latest 10ms of observations of temperature sensors.

Pre-processing. When this query is registered, the rewriting process is launched using the kyrie [18] module. kyrie uses $\mathcal{ELHIO}$ as the language for the ontologies, as it is one of the most expressive DLs that can be currently handled in query

[3] For brevity we use the prefixes aws: http://purl.oclc.org/NET/ssnx/meteo/aws#, met:http://purl.org/env/meteo#, ssn:http://purl.oclc.org/NET/ssnx/ssn#

rewriting (see Section 6 for details). While the ontology is typically an OWL file, kyrie ignores assertions that go beyond the expressivity of $\mathcal{ELHIO}$. It also converts the ontology to Horn clauses and performs additional pre-processing. At this point, the system produces a conjunctive query from the basic graph patterns in the original CQELS query, while the time window definitions of the query (the query context) are preserved. For example, the following conjunctive query clauses can be obtained from the query in Listing 2:

```
Q(?0)  <-  aws:TemperatureSensor(?1), ssn:observedBy(?0,?1)
```

Saturation and expansion. These clauses with the conjunctive query form a logic program that is saturated by using resolution with free selection (RFS) and including a series of optimizations, as described in [18]. This results in a number of clauses that is usually smaller than those produced by other similar techniques (e.g. REQUIEM [21]). After two saturation stages, we can remove functional terms to obtain a Datalog program or expand that program into a UCQ (as in our case without recursion). As an example, the previous conjunctive query is expanded to a union of the following conjunctive queries:

```
Q(?0)  <-  met:TemperatureObservation(?0)
Q(?0)  <-  met:AirTemperatureObservation(?0)
Q(?0)  <-  met:ThermistorObservation(?0)
Q(?0)  <-  aws:TemperatureSensor(?1), ssn:observedBy(?0,?1)
Q(?0)  <-  aws:Thermistor(?1), ssn:observedBy(?0,?1)
Q(?0)  <-  aws:CapacitiveBead(?1), ssn:observedBy(?0,?1)
```

Back to CQELS. The UCQ is then syntactically re-transformed back in CQELS using the context information from the original query. This includes the window definition, the query form and other modifiers. We refer to this query as the CQELS UCQ to avoid ambiguity with the original CQELS query. Following our example, after finishing the rewriting process we obtain the following union of queries (Listing 3), which already takes into account the axioms in the TBox. Given that CQELS does not allow unions on the stream clause, in practice this query is split into different CQELS queries with the same window and the union content as the pattern inside the stream clause.

```
PREFIX ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX aws: <http://purl.oclc.org/NET/ssnx/meteo/aws#>
PREFIX met: <http://purl.org/env/meteo#>
CONSTRUCT { ?o a :ObservedTemperature. }
WHERE { STREAM :stream1 [RANGE 10ms] {
  { ?o a met:TemperatureObservation } UNION
  { ?o a met:AirTemperatureObservation } UNION
  { ?o a met:ThermistorObservation } UNION
  { ?s a aws:TemperatureSensor . ?o ssn:observedBy ?s } UNION
  { ?s a aws:Thermistor . ?o ssn:observedBy ?s } UNION
  { ?s a aws:CapacitiveBead . ?o ssn:observedBy ?s }
}
```

Listing 3. The query of Listing 2 after rewriting.

The general description of the process is depicted in Figure 2.

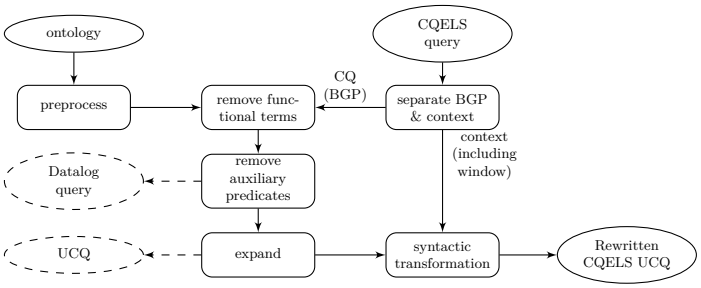


Fig. 2. Main steps of the query rewriting algorithm: preprocess, remove functional terms, auxiliary predicates and unfold (adapted from [18]). Added steps for syntactic conversion between CQELS and UCQ.

5 Evaluation

In this section we evaluate the performance of StreamQR in terms of the throughput of the query answering process, considering different input streaming rates and simultaneous continuous queries. More concretely, we compute the throughput of the system in terms of triples processed per unit of time, comparing StreamQR with rewriting enabled, and StreamQR running CQELS without any rewriting. Then, we evaluate the throughput under different input data loads, considering that the query answering process depends not only on the input rate but also on the number of query matches produced in a given time span. Furthermore, we compare different queries whose rewriting produces different number of UCQs. Finally, we compare StreamQR with TrOWL [23], a state-of-the-art stream reasoner, which provides incremental reasoning over ontology streams, although not targeted towards query answering. We used a modified version of the SRBench [28] benchmark queries, as well as an ontology based on AWS (Ontology for Meteorological sensors[4]), which extends the W3C SSN ontology [11]. The ontology describes sensors, observations, features of interest, ans other weather-related concepts. The ontology is available online [5]. We have taken the SRBench queries and adapted them according to our extended ontology. The full set of queries is available in the Github repository, and additional information about the experiments can be found there as well[6]. All the experiments were performed on an Intel Core i7 3.1 GHz, 16 GB.

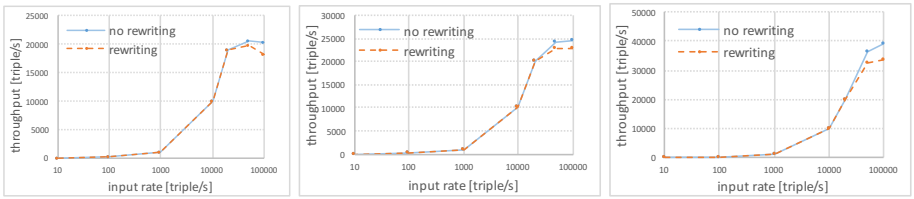
Comparing with CQELS without rewriting. A key indicator in stream processing is the system throughput, which can be measured in terms of the number of input elements processed per unit of time. Input rates, number of matching results and the number of concurrent queries are some of the main factors that impact throughput. In most evaluations concerning query rewriting, the rewriting time is also relevant. This is also the case in RSP, although to a lesser degree, because

[4] AWS: `http://www.w3.org/2005/Incubator/ssn/ssnx/meteo/aws`

[5] `http://jpcik.github.io/streapler/ontology/envsensors.owl`

[6] `https://github.com/jpcik/streapler/wiki/StreamQR-experiments`

for continuous queries the rewriting is performed once, so its cost is not critical in the long run. In the first experiment we compared the throughput of StreamQR with rewriting enabled, to only evaluating the query with CQELS. We tested under different input rates, ranging from 10 to 100K triples/s, as depicted in Fig. 3(a,b,c). We also tested under three different load conditions, i.e. in such a way that only 10%, 50% and 90% of the input data matches the continuous query. As it can be seen in the results, the rewriting of StreamQR performs exactly as without rewriting for most of the input rates. Only under very high rates there is a considerable difference. This is the case for the three types of loads, although we can see considerable changes depending on the percentage of matching input triples. Notice that in these cases, up to 200K triples/s., the behavior reaches the maximum expected throughput.



(a) 10% of input matches (b) 50% of input matches (c) 90% of input matches

Fig. 3. Throughput in StreamQR with and without rewriting.

Variations in input matching. As we saw in the previous experiment the number of matching triples affects the overall throughput. The more matches, the more time the engine spends on evaluation. We performed a series of experiments under different input loads, with a varying distribution of the types of triples, in such a way that 10, 20, 50, 80 and 90% of the triples match the query. As we can see in Figure 4, up to 10K triples/s., in almost all cases StreamQR is capable of handling all the input. Beyond that, the throughput degrades until it reaches a limit. Notice that for each run, as StreamQR produces a UCQ, several queries are running simultaneously.

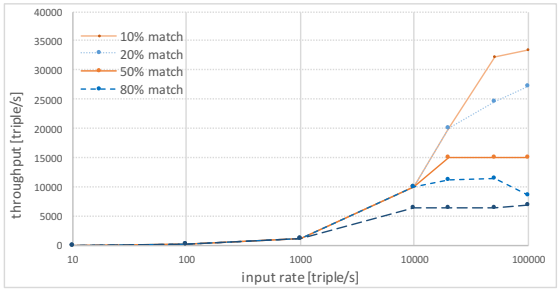


Fig. 4. Throughput in StreamQR for different distributions of input triples.

Query rewritings. Different queries may produce a UCQ with a different number of sub-queries. In this experiment we launched nine distinct queries that produce from 2 to over 180 sub-queries. As it was expected, in general the throughput decreases for queries that produce more rewritten queries (Figure 5). Although

this can also be affected by the complexity of the query, it is a limiting factor on the overall throughput. Existing techniques used in query rewriting and OBDA can be used to alleviate this, for instance by pruning queries that may not match any input. In stream processing this can be feasible in many cases as the data is often repetitive in terms of structure and can be deduced in the long run. Even then, for around 1K triples/s, it still reaches maximum throughput.

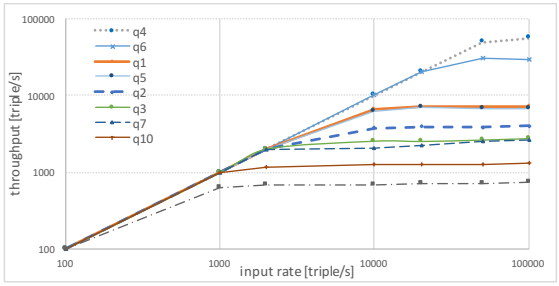


Fig. 5. Throughput for different queries with multiple rewritings, respectively 2, 2, 16, 18, 31, 36, 88, 51 and 185 sub-queries.

Comparison with TrOWL. Finally, we compared the performance of StreamQR with TrOWL, which provides incremental reasoning for ABoxes. While the target of TrOWL is not query answering, but materialization, it is a state-of-the-art stream reasoner which can be used to populate an RDF store that can be periodically queried. We compared the throughput in three different settings. First, with TrOWL only consuming the data without performing any reasoning (no-reclassify), then activating the reasoning, but allowing only additions, and finally including removals as well. The removal operation is known to be expensive in incremental materialization. As we can see in Figure 6, StreamQR sustains better throughput under fast input rates, even at the same level as TrOWL without any reasoning. With reasoning enable in TrOWL, this is even more noticeable. Under lower input rates both are able to reach maximum throughput. Given that the goal of TrOWL is not stream query answering, this comparison is only informative, showing that input throughput with materialization is lower than with query rewriting. A more systematic comparison of materialization vs. query rewriting is worth considering, although it is outside of the scope of this paper.

6 Related Work

Different DL languages have been explored and used for query rewriting and several systems have been implemented for these languages. The *DL-Lite* family of languages [9], a first milestone in this area, derived into $DL\text{-}Lite_{\mathcal{R}}$ and $DL\text{-}Lite_{\mathcal{F}}$.$DL\text{-}Lite_{\mathcal{R}}$ includes ISA and disjointness assertions between roles and $DL\text{-}Lite_{\mathcal{F}}$ includes functionality restrictions on roles. These logics are first-order reducible with a tractable complexity [9], as done in Quonto and extended in Presto and Prexto [26, 25]. The $OWL2\ QL$ profile was inspired by the *DL-Lite* family and designed to keep the complexity of rewriting low, considering first-order rewritability. As a summary of a more extensive comparison [3], the main

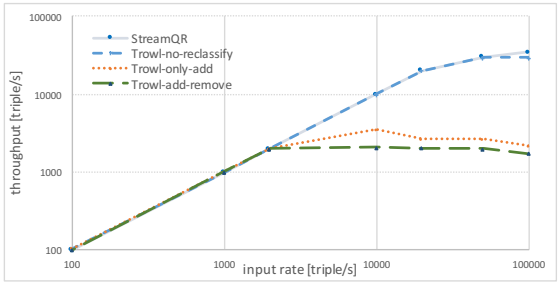


Fig. 6. Comparison with TrOWL, no reclassify, only additions, and with removals.

difference with *DL-Lite* is related with the lack of the unique name assumption (UNA). Among the systems that address this expressiveness we can find Rapid [10].

The $\mathcal{ELHIO}^{\neg}$ logic [21] is more expressive. It extends the expressiveness of $DL\text{-}Lite_{\mathcal{R}}$ by including basic concepts of the form $\{a\}$, $\top$, and $B_1 \sqcap B_2$, as well as axioms of the form $\exists R.B \sqsubseteq C$. This logic does not preserve the first-order rewritability property, what means that depending on the query and the expressiveness in the ontology, the generated Datalog may contain recursive predicates. Thus some queries cannot be expressed as a union of conjunctive queries (UCQ) and must be rewritten to recursive Datalog. In spite of that, the computational complexity of the rewriting process remains tractable (PTIME-complete). Among the systems that can handle this logic we can find REQUIEM [21] and kyrie [18].

Some of the Datalog paradigms that ensure decidability are chase termination, guardedness or stickiness, extended to weak-stickiness by Calì [8]. These paradigms limit the loops that can be present in some Datalog to ensure decidability of the unfolding and thus first-order rewritability. Among the systems that can handle this logic we can find Nyaya [14]. Finally, Horn-$\mathcal{SHIQ}$ includes role hierarchies and inverse roles as $\mathcal{ELHIO}$. It does also include universal restrictions and transitive roles ($\mathcal{S}$) axioms of the form $A \sqsubseteq \forall R.B$ and $trans(R)$. Among the systems that can handle this logic we can find Clipper [13].

With regards to RDF stream processing and reasoning, as described in Section 2.1, different approaches have surfaced in recent years, adding streaming support to SPARQL-based query processors. C-SPARQL [4] takes a hybrid approach that partially relies on a plug-in architecture that internally executes streaming queries with an existing DSMS. CQELS [16] implements a native RDF stream query engine with a focus on the adaptivity of streaming query operators and their ability to efficiently combine streaming and stored data. EP-SPARQL [1] adopts a perspective oriented to complex pattern processing, and includes sequencing and simultaneity operators. Other recent approaches focused on event processing based on the Rete algorithm for pattern matching include Sparkwave [15] and Instans [24]. There has also been a proposal for including rules from a knowledge base into C-SPARQL [5], although these are based on instantaneous materialization only for RDF-S, and are not available

yet in the C-SPARQL software package. Concerning OBDA, the STARQL [19] framework introduced an ABox sequencing strategy which allows it to use unions of conjunctive queries combined with languages such as DL-Lite.

Previous efforts on stream reasoning have focused on ontology maintenance for streams, e.g. using *truth maintenance systems* [23] and approximate reasoning optimized for memory consumption, by eliminating unnecessary intermediate results. Other works have also proposed parallelization techniques for the materialization of inferences in streaming knowledge-bases, although limited only to a fragment of RDFS [27]. On a similar path, works on knowledge evolution [17] have used DL reasoning over *ontology streams* to detect and explain the nature of the changes on the ontology, as well as potential inconsistencies. Concerning theoretical results, the LARS framework [6] proposed a rule-based formalization that captures the semantics of stream reasoning engines.

7 Conclusions

In this paper we presented an approach for providing query answering over ontologies for RDF stream processors, through a novel approach that combines query rewriting techniques and an RSP engines. Furthermore, we implemented StreamQR, a system that incorporates the kyrie rewriter into an existing RSP engine, and that shows the feasibility of our approach. We also provided evidence that this implementation can still be efficient in terms of throughput, for a large range of scenarios, compared with an RSP engine with no rewriting or inferencing capabilities.

In the future, we plan to study other criteria such as correctness of the query answering process, which is known to be non trivial for data streams. Moreover, we are interested in exploring different expressiveness in order to find a good balance between efficiency and complexity. Finally, we believe that there is still large room for research in approaches that combine rewriting and incremental materialization for stream reasoners and query answering over ontologies.

Acknowledgments Partially supported by the Nano-Tera.ch OpenSense2 and D1namo projects, evaluated by the SNSF. Supported by Ministerio de Economía y Competitividad (Spain) under the project 4V: Volumen, Velocidad, Variedad y Validez en la Gestión Innovadora de Datos (TIN2013-46238-C4-2-R)

References

1. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: WWW, pp. 635–644 (2011)
2. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. The VLDB Journal 15(2), 121–142 (June 2006)
3. Artale, A., Calvanese, D., Kontchakov, R., Zakharyaschev, M.: The DL-Lite family and relations. J. Artif. Int. Res. 36(1), 1–69 (2009)
4. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: SPARQL for continuous querying. In: WWW, pp. 1061–1062 (2009)
5. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Incremental reasoning on streams and rich background knowledge. In: ESWC, pp. 1–15 (2010)

6. Beck, H., Dao-Tran, M., Eiter, T., Fink, M.: Lars: A logic-based framework for analyzing reasoning over streams. In: AAAI (2015)
7. Calbimonte, J.P., Corcho, O., Gray, A.J.G.: Enabling ontology-based access to streaming data sources. In: ISWC, pp. 96–111 (2010)
8. Calì, A., Gottlob, G., Pieris, A.: Query answering under non-guarded rules in datalog+/-. In: Web Reasoning and Rule Systems, vol. 6333, pp. 1–17 (2010)
9. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The DL-Lite family. Journal of Automated Reasoning 39(3), 385–429 (Oct 2007)
10. Chortaras, A., Trivela, D., Stamou, G.: Optimized query rewriting for OWL 2 QL. In: Automated Deduction – CADE-23, vol. 6803, pp. 192–206 (2011)
11. Compton, M., Barnaghi, P., Bermudez, L., García-Castro, R., Corcho, O., Cox, S., Graybeal, J., Hauswirth, M., Henson, C., Herzog, A., Huang, V., Janowicz, K., Kelsey, W.D., Phuoc, D.L., Lefort, L., et al.: The SSN ontology of the W3C semantic sensor network incubator group. J. Web Semantics 17, 25–32 (2012)
12. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Computing Surveys 44(3), 15:1–15:62 (2011)
13. Eiter, T., Ortiz, M., \vSimkus, M., Tran, T.K., Xiao, G.: Query rewriting for horn-SHIQ plus rules. In: AAAI (2012)
14. Gottlob, G., Orsi, G., Pieris, A.: Ontological query answering via rewriting. In: Advances in Databases and Information Systems, pp. 1–18. No. 6909 (Jan 2011)
15. Komazec, S., Cerri, D., Fensel, D.: Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In: DEBS, pp. 58–68 (2012)
16. 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, pp. 370–388 (2011)
17. Lécué, F.: Diagnosing changes in an ontology stream: A DL reasoning approach. In: AAAI (2012)
18. Mora, J., Corcho, O.: Engineering optimisations in query rewriting for OBDA. In: I-SEMANTICS, pp. 41–48 (2013)
19. Özçep, Ö.L., Möller, R., Neuenstadt, C.: A stream-temporal query language for ontology based data access. In: KI, pp. 183–194 (2014)
20. Pérez-Urbina, H.: Tractable query answering for description logics via query rewriting. PhD thesis (2009)
21. Pérez-Urbina, H., Horrocks, I., Motik, B.: Efficient query answering for OWL 2. In: ISWC, vol. 5823, pp. 489–504 (2009)
22. Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Rosati, R.: Linking data to ontologies. Journal on Data Semantics X pp. 133–173 (2008)
23. Ren, Y., Pan, J.Z.: Optimising ontology stream reasoning with truth maintenance system. In: CIKM, pp. 831–836 (2011)
24. Rinne, M., Törmä, S., Nuutila, E.: SPARQL-based applications for RDF-encoded sensor data. In: SSN, vol. 904, pp. 81–96 (2012)
25. Rosati, R.: Prexto: Query rewriting under extensional constraints in DL-Lite. In: ESWC, vol. 7295, pp. 360–374 (2012)
26. Rosati, R., Almatelli, A.: Improving query answering over DL-Lite ontologies. In: KR (2010)
27. Urbani, J., Margara, A., Jacobs, C., van Harmelen, F., Bal, H.: Dynamite: Parallel materialization of dynamic rdf data. In: ISWC, pp. 657–672 (2013)
28. Zhang, Y., Duc, P., Corcho, O., Calbimonte, J.P.: SRBench: A Streaming RDF/SPARQL Benchmark. In: ISWC, pp. 641–657 (2012)