An Adaptive Framework for RDF Stream Processing

Qiong Li[1,3], Xiaowang Zhang[1,3,*], and Zhiyong Feng[2,3]

[1] School of Computer Science and Technology,Tianjin University, Tianjin 300350, P. R. China,
[2] School of Computer Software,Tianjin University, Tianjin 300350, P. R. China
[3] Tianjin Key Laboratory of Cognitive Computing and Application, Tianjin 300350, P.R. China
[*] Corresponding author.
{liqiong, xiaowangzhang, zyfeng}@tju.edu.cn

Abstract. In this paper, we propose a novel framework for RDF stream processing named PRSP. Within this framework, the evaluation of C-SPARQL queries on RDF streams can be reduced to the evaluation of SPARQL queries on RDF graphs. We prove that the reduction is sound and complete. With PRSP, we implement several engines to support C-SPARQL queries by employing current S-PARQL query engines such as Jena, gStore, and RDF-3X. The experiments show that PRSP can still maintain the high performance by applying those engines in RDF stream processing, although there are some slight differences among them. Moreover, taking advantage of PRSP, we can process large-scale RDF streams in a distributed context via distributed SPARQL engines, such as gStoreD and TriAD. Besides, we can evaluate the performance and correctness of existing S-PARQL query engines in processing RDF streams in a unified way, which amends the evaluation of them ranging from static RDF data to dynamic RDF data.

Keywords: RDF Stream; RSP; SPARQL; C-SPARQL

1 Introduction

RDF stream, as a new type of dataset, can model real-time and continuous information in a wide range of applications, e.g. environmental monitoring and smart cities [15,16]. RDF streams have played an increasingly important role in many application domains such as sensors, feeds, and click streams [6]. SPARQL is recommended by W3C as the standard query language for RDF data. Though SPARQL engines are capable of querying integrated repositories and collecting data from multiple sources [7], the large knowledge bases now accessible via SPARQL are static, and knowledge evolution is not adequately supported.

For RDF stream processing (RSP), there are many works by extending SPARQL to support queryies over RDF streams such as C-SPARQL [4], EP-SPARQL [2], CQELS [12], SPARQLstream [6] etc. Continuous SPARQL (C-SPARQL) [4] extends SPARQL by adding window operators to manage RDF streams. Event Processing SPARQL (EP-SPARQL) [2] extends SPARQL by introducing ETALIS (a rule-based event language) to reason with events. Continuous Query Evaluation over Linked Streams (C-QELS) [12] introduces three operators, namely, *window operator*, *relational operator*, and *streaming operator* to manage both RDF streams and RDF graphs. SPARQLstream [6] is an extension of SPARQL by introducing *window-to-stream operators* which are used to convert a stream of windows into a stream of tuples to process RDF streams. Besides, SPARQLstream can also provide the ontology-based streaming data access service since sources link their data content to ontologies through S_2O mappings.

Though those existing languages can represent many expressive continuous queries for RDF streams, there are a few prototype implementations for processing RDF streams [11] (e.g., S4[9], CQELS-Cloud [13], and WAVES [10]) due to the complicacy of implementations and the requirement of highly efficiency in query evaluation [16]. On the other hand, there are many popular and efficient SPARQL query engines only supporting static RDF graphs such as Jena [7], RDF-3X [17], gStore [20]. Since those continuous query languages extend SPARQL by adding some extra operators to manage streams, it becomes an interesting problem to employ current SPARQL query engines to evaluate continuous queries. Francesco Barbieri et al. [5] employs Apache Jena [7] (a SPARQL query engine) to evaluate C-SPARQL queries in their implementation. Especially, those popular distributed SPARQL query engines (e.g., TriAD[8], gStoreD[19]) for large-scale RDF data could become very helpful to process large-scale RDF streams, which is a big challenge so far [11,16].

In this work, we propose a novel framework for RDF stream processing named *PRSP*, which is briefly introduced in [14], where the evaluation of continuous queries on RDF streams can be reduced to the evaluation of SPARQL queries on RDF graphs. For conveying our idea simply, we mainly discuss C-SPARQL queries in this paper. We argue that our framework could also support most of continuous query languages extending SPARQL, such as EP-SPARQL and CQELS. Our major contributions are summarised as follows:

- We formalize a C-SPARQL query as a 5-tuple to ensure the soundness and completeness of our reduction from the evaluation of C-SPARQL queries over RDF streams to the evaluation of SPARQL queries over RDF data. Besides, we present the semantics of C-SPARQL in a refined way.
- We develop an adaptive framework named PRSP for processing RDF stream by constructing four new modules, namely, *query parser* (to obtain parameters of windows and core patterns), *trigger* (to call queries and capture windows periodically), *data transformer* (to transform RDF graphs from RDF streams), and *SPARQL API* (to support various SPARQL endpoints to process RDF streams).
- We implement PRSP and evaluate on the YABench [11] by applying three centralized SPARQL endpoints, namely, Jena, gStore, and RDF-3X and two distributed S-PARQL endpoints, namely, gStoreD and TriAD. The experiments show that PRSP can still maintain the high performance of those engines in RDF stream processing although there are some slight differences among them. Besides, we investigate that, given a C-SPARQL query, the correctness of results is sensitive to its window size and step among those query engines.

The remainder of this paper is structured as follows: the next section recalls RDF stream and C-SPARQL. Section 3 defines our sound and complete formalization of C-SPARQL, and Section 4 designs our framework PRSP. Section 5 presents experiments and evaluations. Finally, we summarize our work in the last section.

2 RDF Stream and C-SPARQL

In this section, we briefly recall RDF stream and C-SPARQL.

2.1 RDF stream

Let I, B, and L be infinite 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 called an *RDF triple*. An *RDF graph* is a finite set of RDF triples.

An RDF stream S is defined as an (possibly infinite) ordered sequence of pairs which are quadruples, and each pair is made of an RDF triple (s_i, p_i, o_i) and a timestamp τ_i ($i \in \mathbb{Z}$, i.e., an integer) as follows:

$$S(t) = \{(\langle s_i, p_i, o_i \rangle, \tau_i) \mid t \leq \tau_i \leq \tau_{i+1}\}. \tag{1}$$

Note that $S(t)$ is the prefix of S ending at t.

Example 1. Let us consider an RDF stream S_{Sensor} generated in the YABench Benchmark [11], . It is associated with the temperature values from the environmental monitoring sensors. Table 1 shows the pairs of S_{Sensor}.

Table 1. S_{Sensor}: An RDF stream of sensors

Subject (sub)	Predicate (pre)	Object (obj)	Timestamp
...	...	...	...
observation:A2-1	om-owl:observedProperty	weather:AirTemperature	1483850233586
observation:A2-1	om-owl:procedure	sensor:A2	1483850233586
observation:A2-1	om-owl:result	measure:A2-1	1483850233586
measure:A2-1	om-owl:floatValue	73.0^^xsd:float	1483850233586
observation:A1-2	om-owl:observedProperty	weather:AirTemperature	1483850237596
observation:A1-2	om-owl:procedure	sensor:A1	1483850237596
observation:A1-2	om-owl:result	measure:A1-2	1483850237596
measure:A1-2	om-owl:floatValue	69.0^^xsd:float	1483850237596
...	...	...	...

In Table 1, every sensor is identified via its id, e.g., A1; every temperature value is taken as an object; and the timestamp is represented as a 13-bit integer.

2.2 C-SPARQL

C-SPARQL (Continuous SPARQL) [4] extends SPARQL by adding new operators, namely, *registration* and *windows*, to support processing RDF streams. For simplification, we mainly introduce the basic aspects of new operators. We follow the formalization of C-SPARQL [4].

Query registration C-SPARQL queries should be continuously registered to provide continuous querying services. It can be indicated in the following REGISTRATION QUERY clause.

Registration → 'REGISTRATION QUERY' QueryName
['COMPUTED EVERY' *Number TimeUnit*] 'AS' Query
TimeUnit → 'ms' | 's' | 'm' | 'h' | 'd'

The optional COMPUTED EVERY clause identifies the frequency of the update of the query. If it's not been assigned, it depends on the system the frequency of the query automatically computes. Every registered C-SPARQL query yields continuous results whose type and form are similar to standard SPARQL query. Apart from this, the output of C-SPARQL allows new RDF streams through the following REGISTRATION STREAM clause.

Registration → 'REGISTRATION STREAM' QueryName
['COMPUTED EVERY' *Number TimeUnit*] 'AS' Query

Window C-SPARQL does not process a whole RDF stream but its snapshots every time. For this reason, C-SPARQL introduces the notion of *window*, storing snapshots of RDF streams. *Window* can be defined via the FROM clause.

FromStrClause → 'FROM' ['NAMED']'STREAM' *StreamIRI* '[RANGE' Window']'
Window → *LogicalWindow* | *PhysicalWindow*
LogicalWindow → *Number TimeUnit WindowOverlap*
WindowOverlap → 'STEP' *Number TimeUnit* | 'TUMBLING'
PhysicalWindow → 'TRIPLES' *Number*

Note that windows depend upon two parameters, namely, the window size (RANGE: the maximal time-interval of RDF stream quadruples) and the window step (STEP: the frequency of updates of windows).

Example 2. Let Q_{Sensor} denote a C-SPARQL query *SensorQuery* shown in the table:

```
REGISTER QUERY SensorQuery AS
PREFIX om-owl: ⟨http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#⟩
PREFIX weather: ⟨http://knoesis.wright.edu/ssw/ont/weather.owl#⟩
SELECT ?sensor ?obs ?value
FROM STREAM SensorStreams [ RANGE 5s   STEP 4s ]
WHERE { ?obs observedProperty AirTemperature ;
                om-owl:procedure ?sensor ;
                om-owl:result [om-owl:floatValue ?value] . }
```

Thus *SensorQuery* is registered. And the window size of Q_{Sensor} is 5s and the window step of Q_{Sensor} is 4s. Besides, the WHERE clause together with the SELECT clause is the core part of *SensorQuery*.

Analogously, the semantics of C-SPARQL is defined in terms of sets of so-called *mappings* which are simply total functions $\mu\colon V \to U$ on some finite set V' of variables. Formally, let Q be a C-SPARQL query, S an RDF stream, and t a time (taken as the initial time). We can use $[\![Q]\!]_{S(t)}$ to denote the semantics of Q over S at t (where $S(t)$: the prefix of S ending at t) defined via both its indicated window [4] and the core patterns as SPARQL patterns [18].

Furthermore, let k be a natural number. We use $[\![Q]\!]_{S(t,k)}$ to denote the semantics of Q over the k-th window of S starting at t. Intuitively, $[\![Q]\!]_{S(t,k)}$ is a local semantics restricted on a window. In other words, $[\![Q]\!]_{S(t,k)}$ is a subset of $[\![Q]\!]_{S(t)}$. In this sense, $[\![Q]\!]_{S(t,0)}$ denotes the semantics of Q over the *initial window* starting at t.

In Example 1 and Example 2, consider an initial time $t = 1483850232556$. We have $[\![Q_{\text{Sensor}}]\!]_{S_{\text{Sensor}}(t,0)} = \{\mu_0\}$ and $[\![Q_{\text{Sensor}}]\!]_{S_{\text{Sensor}}(t,1)} = \{\mu_1\}$, where μ_0 and μ_1 are shown in Table 2.

Table 2. R_{Sensor}: The results of the example.

No	?sensor	?obs	?value
μ_0	sensor:A2	observation:A2-1	73.0^^xsd:float
μ_1	sensor:A1	observation:A1-2	69.0^^xsd:float

3 From C-SPARQL to SPARQL

In this section, we present a formal specification of C-SPARQL query and then reduce the evaluation of C-SPARQL query to the evaluation of SPARQL query equivalently.

3.1 Formal specification of C-SPARQL query

In Section 2.2, it can be seen that through adding new production, i.e., windows (including the window size, i.e., RANGE, and the frequency of updates of windows, i.e., STEP), registration and so on, to the standard grammar of SPARQL to extend into C-SPARQL in order to process RDF streams.

Definition 1. *Formally, a C-SPARQL query Q can be taken as a 5-tuple of the form:*

$$Q = [\mathrm{Req}, S, \mathrm{w}, \mathrm{s}, \rho(Q)] \tag{2}$$

where

- Req*: the registration;*
- S*: the RDF stream registered;*
- w*:* RANGE*, i.e., the window size;*
- s*:* STEP*, i.e., the updating time of windows;*
- $\rho(Q)$*: a SPARQL query.*

For convenience, let Q be a C-SPARQL query. We use $\mathrm{Req}(Q)$, $S(Q)$, $\mathrm{w}(Q)$, $\mathrm{s}(Q)$, and $\rho(Q)$ to denote the registration, the RDF stream registered, RANGE, STEP, and the SPARQL query of Q, respectively. Now, we consider an example:

Example 3. Consider the *SensorQuery* Q_{Sensor} in Example 2. Based on Equation (2), we can find:

- $\mathrm{Req}(Q_{\text{Sensor}}) =$ **REGISTER QUERY** *SensorQuery* AS;
- $S(Q_{\text{Sensor}}) =$ *SensorStreams*;
- $\mathrm{w}(Q_{\text{Sensor}}) = 5s$;
- $\mathrm{s}(Q_{\text{Sensor}}) = 4s$;
- $\rho(Q_{\text{Sensor}})$ is a SPARQL query as follows:

```
PREFIX om-owl: ⟨http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#⟩
PREFIX weather: ⟨http://knoesis.wright.edu/ssw/ont/weather.owl#⟩
SELECT ?sensor ?obs ?value
WHERE { ?obs observedProperty AirTemperature ;
                om-owl:procedure ?sensor ;
                om-owl:result [om-owl:floatValue ?value] . }
```

By comparing Example 3 with Example 2, we have the following observations:

Remark 1. Comparing SPARQL, C-SPARQL has some slight differences.

- The registration of C-SPARQL query ensures the continuous recall of C-SPARQL query periodically. However, SPARQL query does not support such a continuous mechanism for recalling query. So we need to design an extra trigger to provide the mechanism (discussed later, see the next section).
- C-SPARQL can support RDF streams but SPARQL cannot due to the timestamp of tuples of RDF streams. We note that the timestamp of quadruples in a window of an RDF stream can be ignored. In this sense, SPARQL can characterize the core pattern of C-SPARQL in a window of an RDF stream.
- C-SPARQL query consists of RDF streams which are to be processed. Indeed, C-SPARQL query processes periodical windows of RDF streams and windows can be stored in the present RDF dataset which can be evaluated by SPARQL queries.

Based on discussions above, *window* is an important notion in transforming C-SPARQL to SPARQL.

Let k be a natural number. A k-th (logical) *window*, denoted by $\mathrm{W}(S, \mathrm{w}, \mathrm{s}, t, k)$, for an RDF stream S, a window size (RANGE) w, an updating time of windows (STEP) s, and an initial time t are a collection of quadruples defined as follows:

$$\mathrm{W}(S, \mathrm{w}, \mathrm{s}, t, k) = \{(\langle s, p, o\rangle, \tau) \in S \mid t + k\mathrm{s} - \mathrm{w} \leq \tau \leq t + k\mathrm{s}\}. \tag{3}$$

Intuitively, a window is a snapshot of a stream. Accordingly, when $k = 0$, it is the initial window.

For instance, in Example 3, the initial window $\mathrm{W}(S_{\text{Sensor}}, 5s, 4s, 1483850232556, 0)$ is shown in Table 3 and the first window $\mathrm{W}(S_{\text{Sensor}}, 5s, 4s, 1483850232556, 1)$ is shown in Table 4:

Table 3. W_0: The initial window of S_{Sensor} starting at t

Subject	Predicate	Object	Timestamp
observation:A2-1	om-owl:observedProperty	weather:AirTemperature	1483850233586
observation:A2-1	om-owl:procedure	sensor:A2	1483850233586
observation:A2-1	om-owl:result	measure:A2-1	1483850233586
measure:A2-1	om-owl:floatValue	73.0^^xsd:float	1483850233586

Table 4. W_1: The 1st window of S_{Sensor} starting at t

Subject	Predicate	Object	Timestamp
observation:A1-2	om-owl:observedProperty	weather:AirTemperature	1483850237596
observation:A1-2	om-owl:procedure	sensor:A1	1483850237596
observation:A1-2	om-owl:result	measure:A1-2	1483850237596
measure:A1-2	om-owl:floatValue	69.0^^xsd:float	1483850237596

The C-SPARQL query evaluation on a stream can be reduced to the evaluation on windows. Then we can have the following:

Lemma 1. *Let Q be a C-SPARQL query. For any RDF stream S and any initial time t, if S is registered in Q then for any natural number k, $[\![Q]\!]_{S(t,k)} = [\![Q]\!]_{W_k}$, where $W_k = \mathrm{W}(S, \mathrm{w}, \mathrm{s}, t, k)$.*

Proof (Skecth). We mainly discuss the function of RANGE. This equation holds since the evaluation of C-SPARQL queries is only restricted in those tuples of RDF streams within RANGE from those queries, which are already in the window W by definition.

Let W be a window. We use G(W) to denote the collection of RDF triples in W. In this sense, G(W) is an RDF graph by removing the timestamp of quadruples. As a result, we can reduce the evaluation of C-SPARQL queries on a window to the evaluation of SPARQL queries on an RDF graph generated from that window by removing timestamp.

The following property shows that the reduction from one of window from an RDF stream to its RDF graph is equivalent.

Lemma 2. *Let Q be a C-SPARQL query. For any RDF stream S and any initial time t, if S is registered in Q then for any natural number k, $[\![Q]\!]_{W_k} = [\![\rho(Q)]\!]_{G(t)}$, where $W_k = \mathrm{W}(S, \mathrm{w}, \mathrm{s}, t, k)$ and $G(t, k) = \mathrm{G}(W_k)$.*

By Lemma 1 and Lemma 2, we can conclude the main result of this paper:

Theorem 1. *Let Q be a C-SPARQL query. For any RDF stream S and any initial time t, if S is registered in Q then for any $k = 0, 1, 2, \ldots$, $[\![Q]\!]_{S(t,k)} = [\![\rho(Q)]\!]_{G(t,k)}$, where $G(t, k) = \mathrm{G}(\mathrm{W}_k)$.*

In Example 1 and 3, considering an initial time $t = 1483850232556$, we have that $[\![\rho(Q_{\text{Sensor}})]\!]_{G(W_0)} = \{\mu_0\}$ and $[\![\rho(Q_{\text{Sensor}})]\!]_{G(W_1)} = \{\mu_1\}$, where μ_0 and μ_1 are already stated in Table 2. Clearly, $[\![Q_{\text{Sensor}}]\!]_{W_k} = [\![\rho(Q_{\text{Sensor}})]\!]_{G(t,k)}$ $(k = 0, 1)$. That is, $[\![Q_{\text{Sensor}}]\!]_{S_{\text{Sensor}}(t,k)} = [\![\rho(Q_{\text{Sensor}})]\!]_{G(t,k)}$ $(k = 0, 1)$.

Theorem 1 ensures that the evaluation problem of C-SPARQL queries over RDF streams can be equivalent to the evaluation problem of SPARQL queries over RDF graphs. Moreover, Theorem 1 can show that the evaluation problem of C-SPARQL has the same computational complexity as SPARQL [4].

4 A framework for RDF stream processing

In this section, we introduce PRSP, an adaptive framework for processing RDF stream.

The framework of PRSP is shown in Fig. 1, which contains four main modules: query parser, trigger, data transformer, and SPARQL API. We give a detailed description about the framework below. Both C-SPARQL queries and RDF streams as the input of PRSP, they are transformed by query parser module and data transformer module, respectively. And the output of query parser can be of immediate use or can be processed by trigger module, which is to call SPARQL queries and produce window selector periodically. After that, data transformer module generates RDF graphs periodically via the input of RDF streams and window selector. Meanwhile, SPARQL API module is capable of getting RDF graphs and SPARQL queries obtained from the former modules as inputs and producing the final results. PRSP is an adaptive framework for RDF stream processing since it can apply various SPARQL query engines to process RDF stream. And the right box, consisting of any SPARQL query engine, is used as a black box to evaluate RDF graphs.

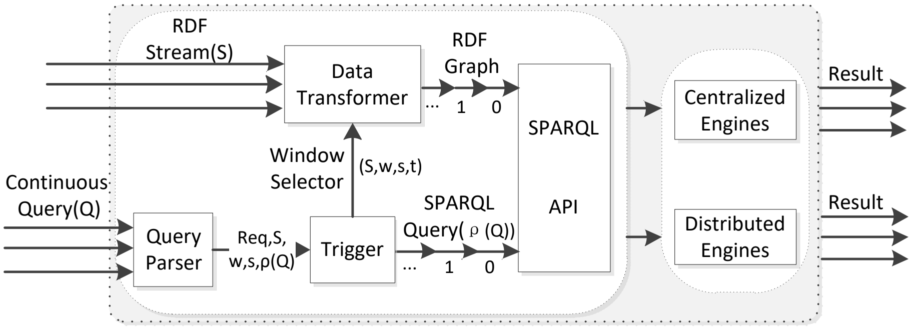


Fig. 1. The framework of PRSP

Query Parser. The query parser module replies on the information captured by Denotational Graph (i.e., D-Graph) [5] which is defined as a view on the O-Graph [18], to obtain parameters of windows and core patterns from the input, i.e., continuous query (Q). The output of this module is the 5-tuple (i.e., $\text{Req}, S, \text{w}, \text{s}, \rho(Q)$) of a Q defined in section 3, and they can be addressed in trigger module.

Trigger. The 5-tuple (i.e., $\mathrm{Req}, S, \mathrm{w}, \mathrm{s}, \rho(Q)$) of a C-SPARQL ($Q$) is as the input of the trigger module which is to call SPARQL queries ($\rho(Q)$) and produce window selector ($S, \mathrm{w}, \mathrm{s}, t$) periodically. Let t_0 be an initial time. In Section 3, for any $k = 0, 1, \ldots$, we have $t = t_0 + k\mathrm{s}$. And let k denote the k-th window of RDF stream S starting at t_0 over Q and the update frequency of SPARQL queries $\rho(Q)$ parsing from query parser module. The window selector ($S, \mathrm{w}, \mathrm{s}, t$) is captured by data transformer module, and SPARQL query is pushed into SPARQL API to be executed in a query engine.

Data Transformer. Via Esper or another DSMS, the data transformer module transforms RDF streams S specified in continuous query Q to capture snapshots based on the window selector ($S, \mathrm{w}, \mathrm{s}, t$) obtained from query parser module, and then convert to RDF graphs by means of removing timestamps of quadruples in windows. Therefore, it can be to generate RDF graphs periodically, and submit to SPARQL API to process.

SPARQL API. Our proposed adaptive framework for RDF stream processing (PRSP) is designed an unified interface for running various SPARQL query engines. In the current version of PRSP, we have only deployed five SPARQL engines, including three centralized engines, Jena, RDF-3X and gStore, and two distributed engines, gStoreD and TriAD. Those SPARQL query engines are relatively high-performance or newer. And we believe that it is easy and convenient to apply other SPARQL query engines. Through SPARQL API, both RDF graphs $G(t, k)$ and SPARQL query $\rho(Q)$ as the input of PRSP, it will output the continuous and real-time query results that users need. Because the most systems are considered scientific prototypes and work in progress, there is no doubt that they can't support all capabilities and querying services. For instance, gStoreD merely provide query model of BGP (basic graph pattern), and can't support the operators of filter and so on.

5 Experiments and Evaluations

5.1 Experimental setup

Implementations and running environment. All centralized experiments, including exploiting Jena, RDF-3X and gStore to process RDF streams, were carried out on a machine running Ubuntu 14.04.5 LTS, which has 4 CPUs with 6 cores (E5-4607) and 64GB memory. And a cluster of 5 machines (1 master and 4 workers) with the same performance as the former were used for distributed experiments. All query systems, including three centralized engines (Jena, RDF-3X, and gStore) and two distributed systems (TriAD and gStoreD), are SPARQL query engines for subgraph matching.

Dataset and continuous queries. For evaluation, we utilized YABench RSP benchmark [11], which provided a real world dataset describing different water temperatures captured by sensors spread throughout the underground water pipeline systems. In our experiments, we perform tumbling windows with a 5-seconds-window which slides every 5 seconds, and sliding windows with a 5-seconds-window which slides every 4 seconds. The experiments were carried out under five different input loads (i.e., *s =500/1000/1500/2000/2500 sensors*) for windows using the query template Q_{Sensor}. The complexity of the scenarios was in the ascending order, from the least complex configuration (*s = 500 sensors*) that loaded roughly 42,000 triples to the most complex configuration (*s = 2500 sensors*) that injected more than 210,000 triples. For comparison, consider that different SPARQL query engines have different capabilities, and some systems can not support complex queries (e.g., filter operator, aggregation operator), so the experiments chose a BGP query template Q_{Sensor} with four triple patterns.

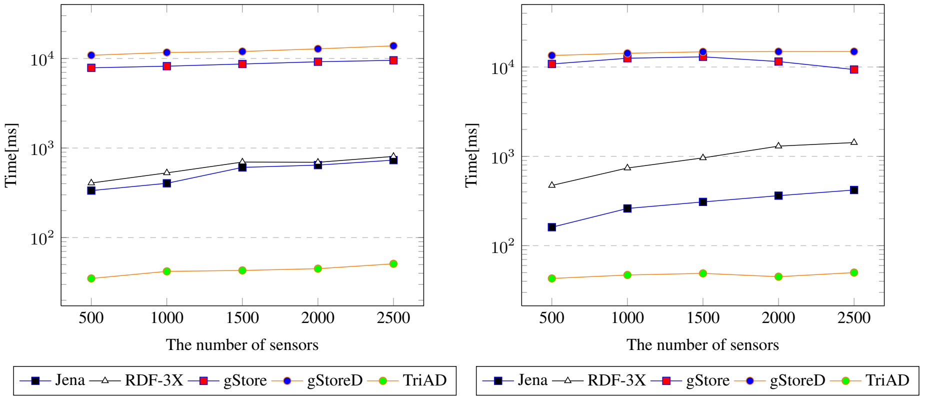
5.2 Performance analysis

When processing RDF streams, it can be considered as three procedures: triples load time (*TLT*), query response time(*QRT*), and engine execution time (*EET*). *TLT* indicates the total time of RDF graphs obtained from data transformer module loading to SPARQL query engines. *QRT* denotes the registered query response time. *EET* means the total execution time of a SPARQL engine while processing all windows, containing *TLT* and *QRT*. We evaluate performance in terms of average time of the three procedures from the whole streams whose duration set at 30 seconds, respectively.

Figs.2 – 4 compare the different systems within PRSP under the five scenarios, and corresponding the three processes (*TLT*, *QRT*, and *EET*) are depicted in Figs.5(a) – 5(d), respectively. For most of the cases, all processes increase varying degrees with the increase in the amount of RDF dataset in windows, but there are slightly different. In addition to gStore, the centralized SPARQL query engines outperform the distributed engines by orders of magnitude under the lower load owing to the unnecessary for centralized systems to communicate with other nodes. Moreover, the gaps among the three procedures between gStoreD and TriAD decrease slightly along with the increase of the dataset, because they are designed to handle large datasets. It also reveals when the input load ranges from *s = 500 sensors* to *s = 1000 sensors* (i.e., the lower load), RDF-3X has a better performance than Jena and gStore, whereas Jena outperforms both RDF-3X and gStore under the higher load (i.e., *s = 2500 sensors*). TriAD is superior to gStoreD under the five scenarios. Besides that, the *TLT* of both gStore and gStoreD occupy a large rate of *EET*, resulting in their lower efficiency for processing RDF streams.

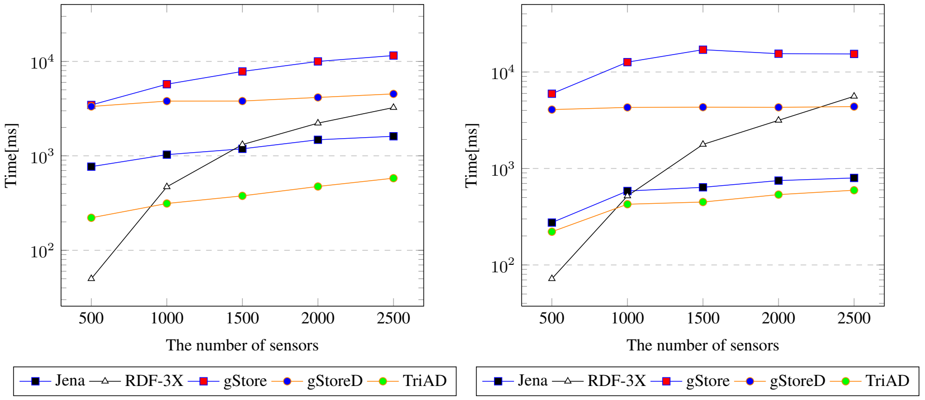
Jena stores its data in main memory and its schema takes the strategy space for time. Resource URIs and simple literal values are stored directly in the statement table in Jean. RDF-3X stores everything in a clustered $B^{+} - Tree$. And triples, sorted in lexicographical order, can be compressed well, which makes them efficiently scan and fast lookup if prefix is known. TriAD combines join-ahead pruning by using a novel form of RDF graph summarization with a locality-based, horizontal partitioning of RDF triples into a grid-like, i.e., distributed index structure. But both gStore and gStoreD parse the RDF graphs to construct indexes, i.e., $VS*tree$, which consumes more time, in order to get results faster.

To assess the scalability performances between the two distributed engines (i.e., gStoreD and TriAD), the experimentation is based on four different set-ups by increasing the number of nodes: 2 nodes, 3 nodes, 4 nodes, and 5 nodes. The graph in Fig.6(d) indicates the engines execution time for the windows over RDF streams for different number of nodes under $s = 1000sensors$. Owing to the lack of available and ready-to-use distributed RSP engines, we just compare the two distributed systems. It is noticeable that until the two engines with 5 nodes in distribution model, the *EET* reduces along with the increase of the nodes. It implies the communication of the master with slaves with fewer nodes occupies a large rate of *EET*.



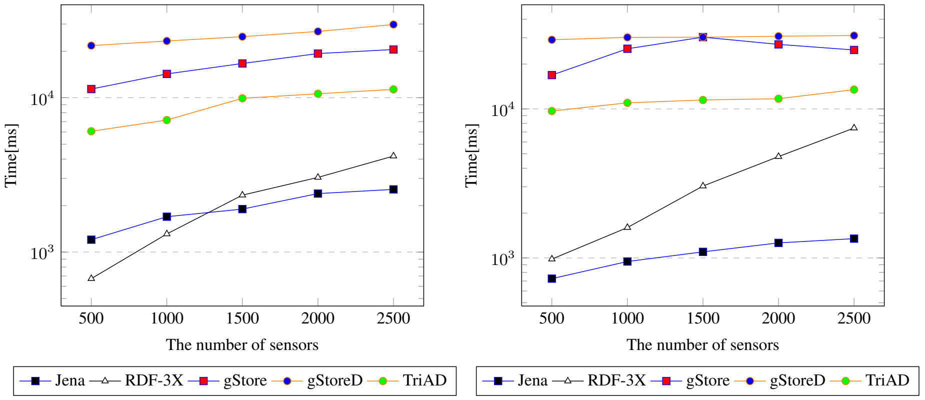
(a) The triples load time over tumbling window (b) The triples load time over sliding window

Fig. 2. Triples load time in different scenarios within PRSP



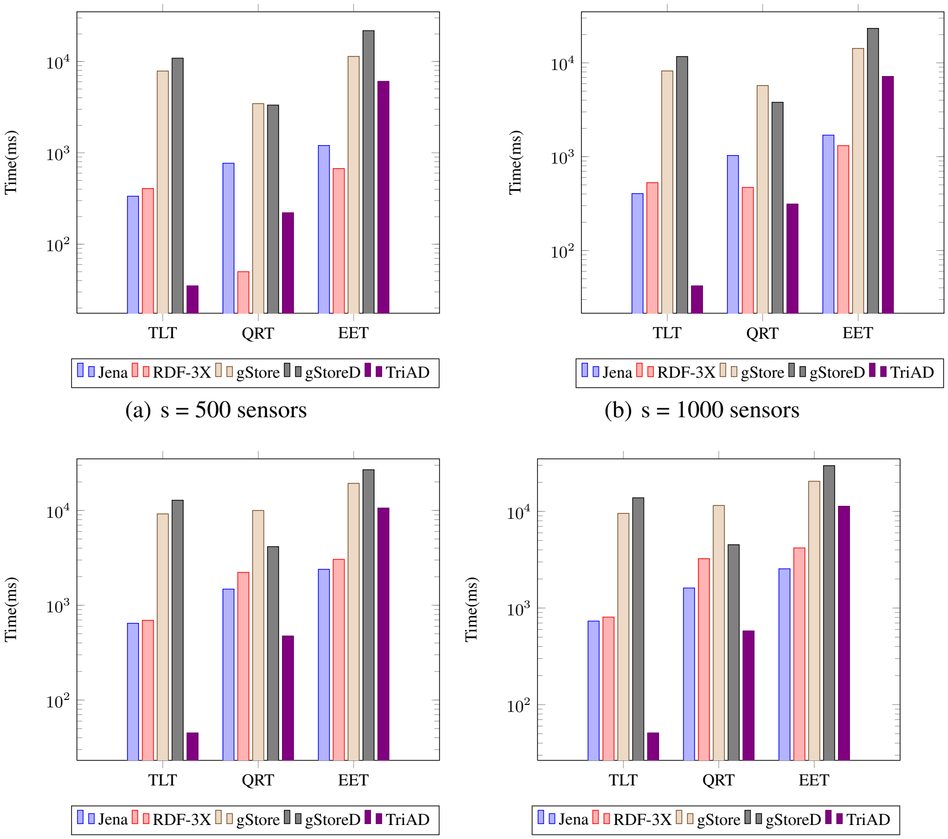
(a) Query response time over tumbling window (b) Query response time over sliding window

Fig. 3. Query response time in different scenarios within PRSP



(a) The execution time over tumbling window (b) The execution time over sliding window

Fig. 4. Execution time in different scenarios within PRSP



(a) s = 500 sensors

(b) s = 1000 sensors

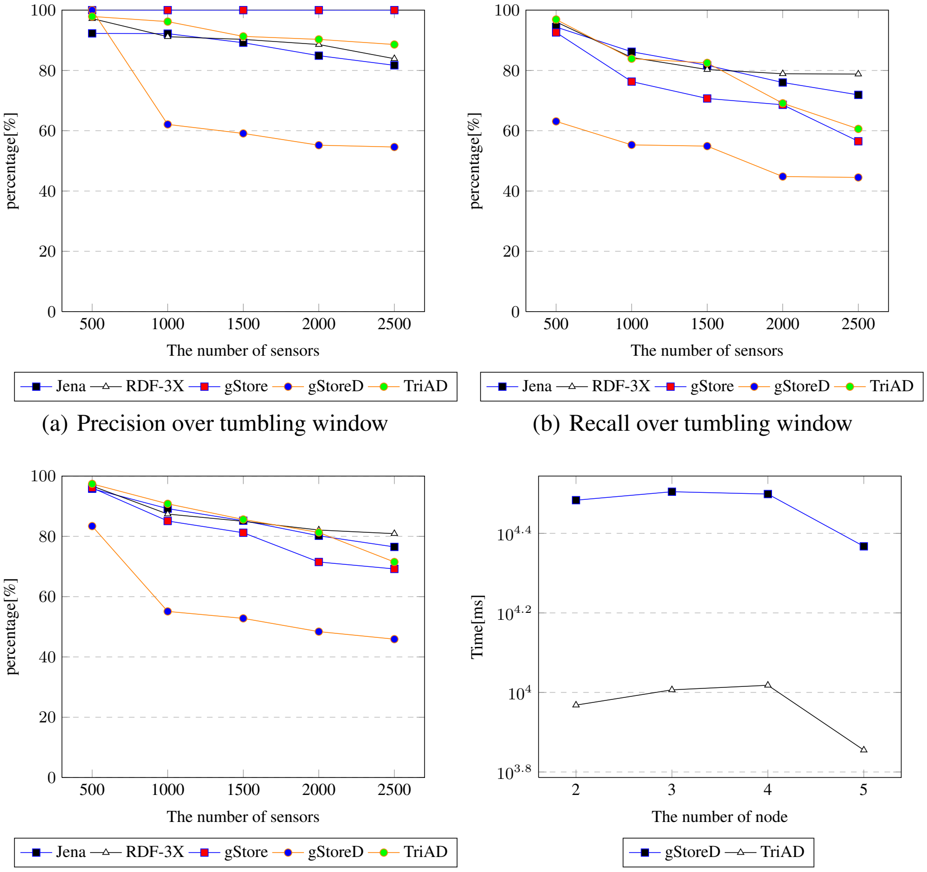
(c) s = 2000 sensors

(d) s = 2500 sensors

Fig. 5. RDF stream for processing time within PRSP

5.3 Correctness analysis

In our experiments, we validated the correctness of the output from our framework PRSP over the five SPARQL query engines, by means of oracle metrics from YABench RSP benchmark. The oracle tests and verifies the effectiveness (i.e., correct and incorrect) of the output from SPARQL query engines by measuring the precision, recall, and f-measure score (i.e., the weighted harmonic mean of precision and recall) which reflecting the overall index. And the correctness analysis help us to find out which system is better to process RDF streams. Figs.6(a) – 6(c) illustrate the results of precision, recall and f-measure scores from the experiments under the five load scenarios within PRSP. Along with more input loads for windows, most of them enjoy lower recalls with relatively high accuracy. We can observe that gStore succeeds to maintain 100% precision even though under higher load (i.e., *s = 2500 sensors*), but achieves lower recall. Generally, as we can see that recall scores are lower than precisions for all the five SPARQL query engines, and the values drop down dramatically when the engines are put under larger loads. Since every SPARQL query engine only can process a certain amount of data at a given time, which leads to lower recall scores under higher load.



(a) Precision over tumbling window

(b) Recall over tumbling window

(c) F-measure over tumbling window

(d) Scalability performance

Fig. 6. (a)–(c), the correctness for PRSP, and (d) the scalability performance between gStoreD and TriAD under s = 1000 sensors

In summary, our experiments show that different query engines in processing RDF streams exhibit different performance, and their correctness in answering is severely sensitive to the window sizes and steps to be selected.

6 Conclusions

In this paper, we present a novel framework for processing RDF streams by reducing RDF streams to RDF data. Taking advantage of the reduction, our framework can ideally support the most SPARQL endpoints including those under construction. Besides, our framework can be used to evaluate the performance and correctness of existing S-PARQL query engines in processing RDF streams in a unified way, which amends the evaluation of them ranging from static RDF graphs to dynamic RDF streams. We also find that the efficiency of evaluations over RDF streams could influence the correctness of querying slightly different to RDF data. In the future work, we will adapt more query engines for RDF data, such as a redesign of database architecture [1] in order to take advantage of modern hardware and a compressed bit-matrix structure BitMat [3] for storing huge RDF graphs.

Table 5. Precision/Recall/F-measure results

		Jena	RDF-3X	gStore	gStoreD	TriAD
sensor=500	precision	92.3%	97.2%	100%	100%	97.9%
	recall	94.4%	96.1%	92.6%	63.1%	96.9%
	F-Measure	95.8%	96.7%	96.1%	83.4%	97.4%
sensor=1000	precision	92.2%	91.2%	100%	62.1%	96.2%
	recall	86.2%	84.3%	76.3%	55.3%	83.9%
	F-Measure	89.2%	87.4%	85.1%	55.1%	90.8%
sensor=1500	precision	89.2%	90.3%	100%	59.1%	91.3%
	recall	81.7%	80.3%	70.7%	54.9%	82.5%
	F-Measure	85.2%	85.0%	81.2%	52.8%	85.6%
sensor=2000	precision	84.9%	88.6%	100%	55.2%	90.3%
	recall	76.0%	78.9%	68.6%	44.8%	69.1%
	F-Measure	80.2%	82.1%	71.5%	48.4%	81.3%
sensor=2500	precision	81.7%	83.9%	100%	54.6%	88.6%
	recall	71.9%	78.8%	56.5%	44.5%	60.6%
	F-Measure	76.5%	80.9%	69.2%	45.9%	71.5%

Acknowledgments

This work is supported by the programs of the National Natural Science Foundation of China (61672377), the National Key Research and Development Program of China (2016YFB1000603), and the Key Technology Research and Development Program of Tianjin(16YFZCGX00210).

References

1. Boncz, P. A., Kersten, M. L., and Manegold, S.: Breaking the memory wall in MonetDB. Commun. ACM, 51(12), 77–85 (2008)
2. Anicic, D., Fodor, P., Rudolph, S., and Stojanovic, N.: EP-SPARQL: A unified language for event processing and stream reasoning. In: Proc. of WWW '11, pp. 635–644 (2011)
3. Atre, A., Chaoji, V., J. Zaki, M., A. Hendler, J.: Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data. In: Proc. of WWW'10, pp.41–50 (2010).
4. Barbieri, D. F., Braga, D., Ceri, S., Della Valle, E., and Grossniklaus, M.: Querying RDF streams with C-SPARQL. SIGMOD REC. 39(1), 20–26 (2010)
5. Barbieri, D. F., Braga, D., Ceri, S., and Grossniklaus, M.: An execution environment for C-SPARQL queries. In: Proc. EDBT '10, pp. 441–452 (2010)

6. Calbimonte, J. P., Corcho, O., and Gray, A. J.: Enabling ontology-based access to streaming data sources. In: Proc. of ISWC'10, pp. 96–111 (2010)
7. Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., and Wilkinson, K.: Jena: Implementing the semantic web recommendations. In: Proc. of WWW '04 (Alternate track papers & posters), pp. 74–83 (2004)
8. Gurajada, S., Seufert, S., Miliaraki, I., and Theobald, M.X.: TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing. In: Proc. of SIGMOD '14, pp. 289–300 (2004)
9. Hoeksema, J., and Kotoulas, S.: High-performance distributed stream reasoning using S4. In: Proc. of Ordring Workshop at ISWC '11 (2011)
10. Khrouf, H., Belabbess, B., Bihanic, L., Kepeklian, G., and Curé, O.: WAVES: Big data platform for real-time RDF stream processing. In: Proc. of SR+SWIT@ISWC '16, pp. 37–48 (2016)
11. Kolchin, M., Wetz, P., Kiesling, E., and Tjoa, A. M.: YABench: A comprehensive framework for RDF stream processor correctness and performance assessment. In: Proc. of ICWE '16, pp. 280–298 (2016)
12. Le-Phuoc, D., Dao-Tran, M., Parreira, J. X., and Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: Proc. of ISWC '11, pp. 370–388 (2011)
13. Le-Phuoc, D., Quoc, H. N. M., Le Van, C., and Hauswirth, M.: Elastic and scalable processing of linked stream data in the cloud. In: Proc. of ISWC'13, pp. 280–297 (2013)
14. Li, Q., Zhang, X., Feng, Z.: PRSP: A plugin-based framework for RDF stream processing. In: Proc. of WWW'17, poster, pp. 815–816 (2017)
15. Margara, A., and Cugola, G.: Processing flows of information: From data stream to complex event processing. In: Proc. of DEBS '11, pp. 359–360 (2011)
16. Margara, A., Urbani, J., Van Harmelen, F., and Bal, H.: Streaming the web: Reasoning over dynamic data. J. Web Sem., 25(1), 24–44 (2014)
17. Neumann, T., and Weikum, G.: The RDF-3X engine for scalable management of RDF data. VLDB J. 19(1), 91–113 (2010)
18. Pérez, J., Arenas, M., and Gutierrez, C.: Semantics and complexity of SPARQL. In: Proc. of ISWC '06, pp. 30–43 (2006)
19. Peng, P., Zou, L., Özsu, M. T., Chen, L., and Zhao, D.: Processing SPARQL queries over distributed RDF graphs. VLDB J. 25(2), 243–268 (2016)
20. Zou, L., Özsu, M.T., Chen, L., Shen, X., Huang, R., and Zhao, D.: gStore: A graph-based SPARQL query engine. VLDB J. 23(4), 565–590 (2014)