C-SPARQL Extension for Sampling RDF Graphs Streams

Amadou Fall Dia, Zakia Kazi-Aoul, Aliou Boly, Yousra Chabchoub

C-SPARQL extension for sampling RDF graphs streams

Amadou Fall Dia, Zakia Kazi-Aoul, Aliou Boly and Yousra Chabchoub

Abstract Our daily use of Internet and related technologies generates continuously large amount of heteregenous data flows. Several RDF Stream Processing (RSP) systems have been proposed. Existing RSP systems benefit from the advantages of semantic web technologies and traditional data flow management systems.
C-SPARQL, CQELS, SPARQL$_{stream}$, EP-SPARQL, and Sparkwave extend the semantic query language SPARQL and are examples of those systems. Considering that the storage and processing of all these streams become expensive, we propose a solution to reduce the load while keeping data semantics, and optimizing treatments. In this paper, we propose to extend C-SPARQL for continuously generating samples on RDF graphs. We add three sampling operators (UNIFORM, RESERVOIR and CHAIN) to the C-SPARQL query syntax. These operators have been implemented into Esper, the C-SPARQL's data flow management module. The experiments show the performance of our extension in terms of execution time and preserving data semantics.

1 Introduction

Today we produce more data than resources to process them. Our daily use of social networks (Facebook, Twitter, Linkedin, etc.), contents of social multime-

Amadou Fall Dia
ISEP, Paris 75006, France, e-mail: amadou.dia@isep.fr

Zakia Kazi-Aoul
ISEP, Paris 75006, France, e-mail: zakia.kazi@isep.fr

Aliou Boly
Cheikh Anta Diop University, BP 5005 Dakar-Fann, Senegal, e-mail: aliou.boly@ucad.edu.sn

Yousra Chabchoub
ISEP, Paris 75006, France e-mail: yousra.chabchoub@isep.fr

dia platforms (YouTube, Flickr, iTunes, etc.), sensor networks (observation, remote reading, etc.), internet of things (geolocation, triggering real-time alarms, etc.), etc. produces continuous data streams. Several research groups are interested in semantic web technologies application to real time stream processing. Like DSMSs (Data Stream Management Systems), several extensions to SPARQL have been proposed for processing RDF streams. The six (6) major propositions languages and/or sytems are Streaming SPARQL [Bolles et al., 2008], Continuous SPARQL (C-SPARQL) [Barbieri et al., 2010], CQELS [Le-Phuoc et al., 2011], SPARQL$_{stream}$ [Calbimonte et al., 2010], EP-SPARQL [Anicic et al., 2011a] and Sparkwave [Komazec et al., 2012]. They all extend SPARQL but adopt different approaches to deal with continuous RDF streams.

As the volume and the unpredictable speed of incoming data increase, processing the entire contents of a stream is difficult. Systems need therefore techniques for (i) dynamic resources allocation [Vijayakumar et al., 2010] and [Cao et al., 2012] or (ii) reduction of the input data load. Regarding this last point, when input stream rate is high (exceeds capabilities of RSP engines), systems will be overloaded. For instance, a high query execution time compared to the input stream rate will cause overload and thus loss of important data and latency in processing. To keep pace of data arrival, the system would shed some of the load according to a given method. None of existing SPARQL extensions implement continuous data summaries or sampling mechanisms.

In this paper, we propose to extend C-SPARQL engine by adding sampling operators. These operators will allow us to reduce, on the fly, input RDF data, while preserving semantic links. We consider first a new data input format. Indeed, C-SPARQL takes as input a sequence of pairs, where each pair is made of an RDF triple and its timestamp. This form of representation does not guarantee the semantic links between data within the sample. In the context of RDF data streaming, events are frequently captured by a set of triples but not by only one triple. Thus, instead of RDF triple format (<*subject*, *predicate*, *object*>, *timestamp*), we adopt a graph oriented approach where each graph represents an event formed by a set of temporal RDF triples. We then extend the C-SPARQL query syntax and the continuous execution module of its architecture (Esper) by adding new sampling operators.

This paper is organized as follows. Section 2 introduces RDF graphs streams concept by deducting the triples format. We provide a brief state of the art on existing RDF Stream Processing (RSP) systems in section 3. Section 4 gives the three sampling algorithms used for the implementation of our sampling operators while we present in section 5 the extended C-SPARQL syntax and architecture. Section 6 presents our performance evaluation and results. Finally, we conclude and give our future works in section 7.

2 RDF graphs streams

In this section, we present the notion of RDF graph streams adopted for sampling process in C-SPARQL.

2.1 RDF streams: triple based

RDF (Resource Description Framework) is the formal W3C recommendation for semantic data representation. The base element of the RDF model is the triple: $<\mathbf{s},\mathbf{p},\mathbf{o}> \in (I \cup B) \times I \times (I \cup B \cup L)$ where I is a set of IRIs (Internationalized Resource Identifiers), B is a set of blank nodes and L is a set of literals. **s**,**p**, and **o** represent information and are respectively called the subject, the predicate and the object of the RDF triple. However, RDF model lacks of temporal dimension which is compulsory in the context of streaming. Thus, applying semantic web technologies on streaming data has given rise to the notion of RDF streams.

Golab and Özsu [2003] define a data stream as follow: "*a data stream is a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items. It is impossible to control the order in which items arrive, nor is it feasible to locally store a stream in its entirety*". Then RDF Streams are introduced as natural extension of RDF model in streaming context. An RDF stream S can be defined as a temporal ordered sequence of pairs, where each pair consists of an RDF triple and its timestamp: $S = \langle < t_r, \tau_i > \rangle$, where t_r is a triple observed or arrived at time τ_i. Integers τ_i are monotonically non-decreasing and not strictly increasing ($\forall i$, $\tau_i \leq \tau_{i+1}$).

$$\langle < subject_i, predicate_i, object_i >, \tau_i \rangle$$
$$\langle < subject_{i+1}, predicate_{i+1}, object_{i+1} >, \tau_{i+1} \rangle$$
$$...$$

Given that streams are intrinsically infinite, data are usually read through windows upon streams using the CQL [Arasu et al., 2004b] window concept. Queries over RDF streams deal with triples in time-based window (all the triples which occur during a given time interval) or element-based window (a given number of triples).

Most of the existing SPARQL extensions for streaming RDF streams take as input a succession of RDF triples. This representation model allows widely continuous process of streams but consumes only triples, ignoring the graph structure of RDF data. In this case, each event is distributed into a set of triples. Considering stream of successive triples, we can not capture boundaries on a set of triples within different events. Therefore, this succession of triples which belongs to a same event needs to be grouped into a single graph and annotated with the same timestamp.

2.2 RDF streams: graph based

As preliminaries of our work, we consider an RDF graph stream format by extending the definition of RDF stream format for this purpose. Events within streams are naturally captured by a set of triples with the same timestamp or not. Hence, a graph represents those triples with the same source and unique timestamp.

An RDF graph may be constructed from one or a set of RDF triples or statements. More formally, given a set of triples st_r, we define $G\langle st_r\rangle$ as a directed labelled graph where each vertex consists of triple's subject (s) or object (o) and each edge consists of triple's predicate (p). The notion of connectivity in RDF graph definition is important for all triples which compute an event. In fact, triples within an event share between each other a unique or multipath. A connectivity in RDF graph $G\langle st_r\rangle$ is a sequence of edges (predicates) $p_1, ..., p_n$ such as $\forall 1 \leq i < n$, p_i share a vertex (subject or object) with p_{i+1}.

With the definition above, RDF graph stream can be simply defined as a sequence of pair $(G(st_r)_i, \tau_i)$, where $G\langle st_r\rangle_i$ is an RDF graph represented as an event and τ_i is the time when this event occurs.

$$(G\langle st_r\rangle_i, \tau_i)$$
$$(G\langle st_r\rangle_{i+1}, \tau_{i+1})$$
$$\dots$$

Two graphs may share a same timestamp value, which means that their events occurs at the same time. Highlight that two triples with the same timestamp are not necessary within the same event. Fig. 1 shows an example of a flow of RDF graphs from sensors deployed in a water distribution network. The data collected concern the pressure, the flow rate, the chlorine content, the temperature, etc.

In particular, sampling RDF streams when events are decomposed into a list of individual RDF triples may break semantic links between them in the sample. In that case, samples may end up with meaningless data and queries over them only observe

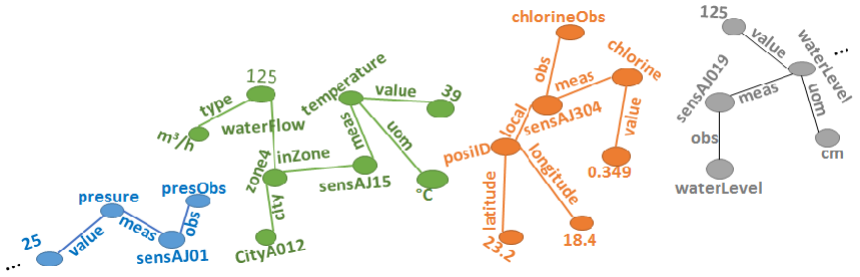

Fig. 1: Example of graph streams from water distribution network

a partial RDF graph and obviously return false or incomplete results. Therefore, we adopt a graph-oriented approach that ensures the preservation of data semantic (i.e. meaning links between subjects and objects) after a sampling operation.

3 RDF stream processing systems

All proposed SPARQL extensions for stream processing share at the basis the same issues in terms of heterogeneity (multiple data sources), and lack of explicit data semantic allowing complex queries and reasoning over data. Several systems exist such as Streaming SPARQL, C-SPARQL, CQELS, SPARQL$_{stream}$, EP-SPARQL and Sparkwave. In this section, we give a brief state of the art of those SPARQL extensions.

Streaming SPARQL [Bolles et al., 2008] extends SPARQL grammar with the capability to explicitly state data streams and define physical or logical windows over them. Authors modify the semantics of SPARQL by first adding the STREAM keyword to allow as input a data stream identified by an IRI. Then, they consider two window types. The time based window is defined with the RANGE (a definition of the window size), SLIDE (a delay after which window is moved) and FIXED (in case which RANGE value equals SLIDE value). The element based window is defined with ELEMS (a number of given elements) keyword. Authors focus only on an extension to cope with window queries over data streams without taking into account performance issues. Since they extend SPARQL 1.0 version, they therefore do not support group by clauses and aggregation which are usually necessary in streaming context. There is no proposed Streaming SPARQL system. The following Streaming SPARQL query returns every minute, pressure values captured in the last 10 minutes.

```
SELECT  ?WaterPressure
FROM STREAM <http://waterdist.org/sens> WINDOW RANGE 10 MINUTE
                                                SLIDE
WHERE {?sensorID ex:hasPressure ?WaterPressure . }
```

Continuous SPARQL or shortly **C-SPARQL** [Barbieri et al., 2010] proposes a continuous query syntax and a framework which reuses independently existing and tested technologies such as data stream management systems like STREAM [Arasu et al., 2004a] or ESPER[1] and triple stores Jena[2] or Sesame[3]. Authors also propose their own operators for windowing (inspired from CQL, the query language used in relational stream systems) and aggregation functions. C-SPARQL also allows periodical query evaluation, temporal function and multiple streams. Authors

[1] http://www.espertech.com/esper/

[2] https://jena.apache.org/

[3] http://rdf4j.org/sesame/

first extends SPARQL 1.0 but switched to support SPARQL 1.1 in their latest version to benefit from new functions like aggregation ones. The following C-SPARQL query example returns every 10 minutes, the average pressure values captured in the last hour.

```
REGISTER QUERY AverageWaterPressure AS
SELECT  ?sensorID  (AVG(?WaterPressure) as ?AvgWaterPressure)
FROM STREAM <http://waterdist.org/sens> [RANGE 1h STEP 10m]
WHERE {?sensorID ex:hasPressure ?WaterPressure . }
GROUP BY  ?sensorID
```

Like C-SPARQL, **CQELS** [Le-Phuoc et al., 2011] language adds its own window operators analogous to CQL and allows possibilities to combine static and streaming data processing. Unlike C-SPARQL, CQELS applies an eager execution strategy which means that query evaluation is triggered by the arrival of new triples. The system does not delegate stream evaluation to an external component (DSMS) but defines its own native processing model, which is implemented internally in the query engine. Author's strategies allow full control over the execution plan consequently, enabling query optimisations. The following CQELS query retrieves every 5 minutes, the minimal value of pressure captured in the latest hour.

```
SELECT  ?sensorID  (MIN(?WaterPressure) as ?MinWaterPressure)
FROM STREAM <http://waterdist.org/sens> [RANGE 1h SLIDE 5m]
WHERE {?sensorID ex:hasPressure ?WaterPressure . }
GROUP BY  ?sensorID
```

SPARQL$_{\textbf{stream}}$'s authors [Calbimonte et al., 2010] were at base inspired by C-SPARQL systems. Similarly to it, SPARQL$_{stream}$ delegates the processing to external engines. SPARQL$_{stream}$ also proposes its own stream and window operators and aims more at enabling ontology-based access for semantic stream processing. The proposed system (Morph-Stream) is based on the concept of virtual RDF streams where each of them is identified by an IRI. SPARQL$_{stream}$ only considers time-based window. SPARQL$_{stream}$ is the only proposed extension which supports all Relation-to-Stream operators (R$_{stream}$, I$_{stream}$ and D$_{stream}$) inspired from CQL. The following **SPARQL$_{\textbf{stream}}$** query shows sensors list having observed, in the last hour, a temperature value above 60°C.

```
SELECT  ?sensorID
FROM NAMED STREAM <http://waterdist.org/sens> [NOW-1 HOURS]
WHERE {?sensorID ex:hasTemperature ?temperature .
                    FILTER (?temperature > 60)}
```

EP-SPARQL [Anicic et al., 2011a] delegates the processing to an event processing engine. Unlike the previous extensions which focus on windows and stream registering, EP-SPARQL proposes an unified language for event processing. The language follows the concept of Complex Event Processing (CEP) systems. For expressing complex event of input data, EP-SPARQL mainly includes logical and

temporal operators expressed in terms of sequence and simultaneity. SEQ operator specifies that two graphs patterns are joined if one occurs after the other. EQUALS operator placed between two patterns, indicates that they are joined if they occur at the same time. OPTIONALSEQ and EQUALSOPTIONAL represent respectively optional version of SEQ and EQUALS operators. EP-SPARQL simulates window operators by using filter operators to isolate portions of the input data streams. While other systems use a single point in time approach to represent each pattern in the stream, EP-SPARQL adopts an interval approach which represents the lower and upper bound of the occurring interval. The system ETALIS [Anicic et al., 2011b] is a complex event processing framework based on Prolog language. The following EP-SPARQL query gives all pressure sensors whose measured values have dropped by more than half and then have dropped by over a quarter in a period of 24 hours.

```
SELECT    ?sensorID
WHERE { ?sensorID ex:hasPressure ?waterPressure1}
      SEQ {?sensorID ex:hasPressure ?waterPressure2 }
      SEQ {?company ex:hasPressure ?waterPressure3 }
   FILTER (?waterPressure2 < ?waterPressure1/2 && ?waterPressure3 <
     ?waterPressure1/4 && getDURATION() < "P24H"8sd:duration)
```

Sparkwave [Komazec et al., 2012] is a complex system which adopts graph pattern detection on RDF data streams and supports temporal nature of RDF streams. Sparkwave implements windowing mechanism based on Rete algorithm [Rete, 1982] (with ε-network Pre-processing). Sparkwave supports time-based windows but does not support temporal operators, arithmetic operators and logical operators (disjunctions and negations). In fact, as mentioned above, Sparkwave only supports graph pattern detection which constitutes a limit of the system.

Table 1: Comparison of SPARQL extensions for RDF stream processing

System	DIF[a]	TiW[b]	TrW[c]	U, J, O, F[d]	TF[e]	A[f]	S/S[g]	CQ[h]
Streaming SPARQL	RS[i]	✓	✓	✓	×	×	×	✓ (P[l])
C-SPARQL	RS &S[j]	✓	✓	✓	✓	✓	×	✓ (P[l])
CQELS	RS &S[j]	✓	✓	✓	×	✓	×	✓ (T[m])
SPARQL$_{stream}$	(V)RS[k]	✓	×	✓	×	✓	×	✓ (P[l])
EP-SPARQL	RS&S[j]	×	×	✓	✓	✓	✓	✓ (P[l])
Sparkwave	RS&S[j]	✓	×	✓	×	×	×	✓ (T[m])

[a]Data Input Format [b]Time Window [c]Triple Window
[d]UNION,JOIN,OPTIONAL,FILTER [e]Temporal Function [f]Aggregate
[g]Sequence/Simultaneity [h]Continuous Query [i]RDF streams
[j]RDF stream & Statics [k]Virtual RDF Stream [l]Periodic
[m]Trigger

Table 1 briefly compares features and functionalities of SPARQL extensions for stream processing. In general, they all have similar approaches for supporting temporal and window process over streams. They also all extend SPARQL but have small or important differences depending on the field of application.

Streaming SPARQL seems to be the most limited system in terms of operators. It lacks of group by and aggregation operators and does not support SPARQL 1.1. Moreover, the proposed system is still theoretical. **C-SPARQL**, **CQELS** and **SPARQL$_{\textbf{stream}}$** come with a SPARQL language extension and a query processing engine. They are also based on SPARQL 1.1 and all support sliding windows. Among these three engines, only SPARQL$_{stream}$ lacks of window based elements. **EP-SPARQL** is different from the other proposed extensions as they adopt sequence (SEQ and OPTIONALSEQ) and simultaneity (EQUALS and EQUALSOPTIONAL) operators following Complex Event Processing (CEP) paradigm. Like EP-SPARQL, **Sparkwave** focused on event processing. However, Sparkwave has no SEQ or EQUALS operators but is only based on Rete algorithm for pattern matching. Unlike the other systems, which only deal with algebraic optimisation, CQELS and Sparkwave use native approach and then can bring adaptive optimisations. In C-SPARQL and SPARQL$_{stream}$, the data is stored in relational tables and relational streams before any further processing. In our opinion, among all, C-SPARQL is the only system that provides at the same time Union, Join, Optional and Filter operators, logical and physical window, aggregation, continuous processing, multiple streams sources, combining static and RDF stream evaluation, temporal function, and is built on a modular architecture.

4 Sampling algorithms

RDF stream processing systems require rapid, continuous and intelligent data processing. Given the volume and speed of data generation, it is necessary to extract data samples from input streams. Several sampling techniques exist such as random sampling, reservoir sampling, deterministic sampling and chain sampling.

4.1 Uniform random sampling without replacement

The simple random sampling [Cochran, 2007] may be with or without replacement. It selects without replacement and with the same probability p a random sample of size n from a set of indexes within a window W. The index of an element in W can be selected only once.

This method is very basic and has the advantage of being simple and easy to implement. However, this technique gives to all elements the same chance of being

included in the sample. This can be seen as a disadvantage because in a streaming context we are often interested in recent data. In the sample constitution, we should give a more chance for relatively recent data.

4.2 Reservoir sampling

The main idea of any type of reservoir sampling algorithm [Vitter, 1985] is maintaining a random sample with a fixed size n into a "reservoir". After each windowing process over streams, a random sample of size n can be extracted. Initially, we put the first n received items into the "reservoir" R. Then, each new item in the window has the probability $\frac{n}{i}$ to replace the item of index i in the reservoir R. This method clearly favors old items ($\lim_{i\to\infty} \frac{n}{i} = 0$). Therefore, oldest items are more likely to be included in the sample.

4.3 Chain sampling

[Babcock et al., 2002] described that the chain sampling technique consists in building a sample of size n over a sliding windows of size $\omega > n$. For sliding windows, chain sampling algorithm randomly generates replacements among expired items, and stores the replacement. As shown in chain sampling algorithm, in the first window, we add i indexes in the sample with the probability $p = \frac{min(\omega,i)}{\omega}$. The successor index r_i of index i is randomly chosen from the interval $[i+1, i+\omega]$ and replaces it in the sample after its expiration (i out of the window). The successor of r is randomly selected in the same manner in the interval $[r+1, r+\omega]$. This process is thus repeated independently. This method is particularly suitable for sliding windows

Algorithm Chain sampling algorithm

```
1: function CHAINSAMP(ω, p)
2: R_epl ← ∅
3: S← put indexes (i) from the first window (with size ω) with probability Min(ω,i)/ω
4:     for each index i in S do
5:         Select a random successor r_i with probability p between i+1 and j+ω
6:         R_epl ← r_i
7:     end for
8:     while each new index is added do
9:         i ← i+1 //move window index by on step
10:        Replace each expired index j in S by its successor r_j in R without redondancy
11:        Choose a random successor for r_j between r_j+1 and r_j+ω without redondancy
12:    end while
13: end function
14: Return S
```

but has a significant memory usage due to the non-redundant selection criterion for successors.

5 Our C-SPARQL extension

C-SPARQL system [Barbieri et al., 2010] provides a modular architecture for processing C-SPARQL queries over RDF streams. This architecture is composed of two parts: a data stream management system like STREAM or Esper and a SPARQL engine module like Jena or Sesame.

This architecture uses STREAM or Esper as RDF streams manager module and Jena or Sesame as SPARQL engine module.

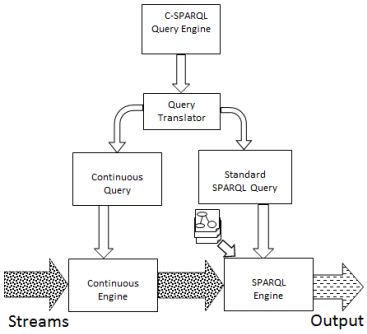

Fig. 2: C-SPARQL architecture

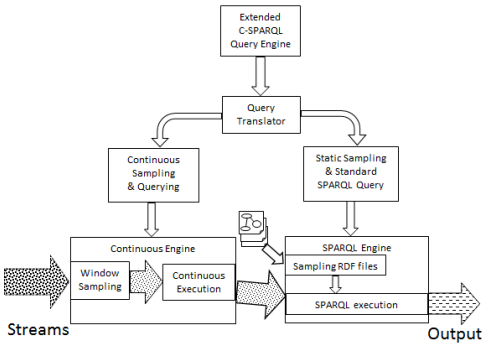

Fig. 3: C-SPARQL extended architecture

Fig. 2 presents the C-SPARQL architecture. The *Query Translator* module configures, initializes and dispatches continuous and static parts of a C-SPARQL query. This dispatching task processes a correct C-SPARQL query and creates two (2) instances:

1. ***ContinuousEngine*** consists of a DSMS (Esper in the last version) that processes on the fly RDF triples and applies the concept of window over them through a CQL query. This module outputs a set of quadruplets ($subject$, $predicate$, $object$, $timestamp$) intended to the SPARQL engine SparqlEngine.
2. ***SparqlEngine*** is the "semantic" component in this all architecture. Its main task is running SPARQL deduced from the C-SPARQL one, each time the ContinuousEngine produces quadruples.

The main contribution of our work is the extension of the C- SPARQL syntax and architecture level for continuous sampling of RDF graphs. To reduce the load of data streams to process and store only a sample for potential analysis or reasoning, our implementation includes three (3) main steps:

1. Considering input graphs data streams instead of an ordered sequence of RDF triples for semantic preservation
2. Adding three (3) sampling operators within C-SPARQL query syntax
3. Implementing continuous sampling methods within Esper.

5.1 Sampling operators

As shown below, we extend the C-SPARQL query syntax by adding new operators in **FROM STREAM** (stream sources) and **FROM** (static sources) clauses.

1. *PREFIX $prefixName : < IRI >$
2. SELECT $'?variables'$
3. *FROM STREAM $< StreamIRI >$ $[Window]$ **[SAMPLING Token]**
4. *FROM $< StaticIRI >$ **[SAMPLING Token]**
5. WHERE { $'Mapping\ variables'$;|.
6. $^*FILTER\ ('condition')\}$
7. GROUP BY $'?variables'|expression$
8. HAVING $'aggregation\ condition'$
9. ORDER BY $'?variables'$

SAMPLING $\rightarrow UNIFORM \mid RESERVOIR \mid CHAIN$
Token $\rightarrow [Window]$ $\%P|Size$
$[Window] \rightarrow 'chain\ window\ size'$
$\%P \rightarrow 'sampling\ percentage'$
$Size \rightarrow 'reservoir\ size'$

The extended syntax contains new operators for sampling methods continuously applied over input RDF graph streams and statics (RDF repositories). The following query randomly samples input graphs and provides every minute, all sensors whose sensed pressure value exceeds 27 (unit of measure) in the last 10 minutes.

```
REGISTER QUERY exceedsPressures AS
SELECT ?sensorID ?WaterPressure
FROM STREAM <http://waterdist.org/sens> [RANGE 10m STEP 1m]
                                                    [UNIFORM %60]
FROM <http://waterdistrib.org/staticdata.rdf>
WHERE {?sensorID ex:hasPressure ?WaterPressure .
    FILTER (?WaterPressure > 27)
          }
```

5.2 Architecture extension

Our approach is based on the implementation of sampling methods within Esper. Fig. 3 shows our proposed extension of C-SPARQL architecture where its traditional modules remain independent plugins. We extend the three (3) modules *Query Translator*, *ContinuousEngine* and *SparqlEngine*.

1. In ***QueryTranslator***, we parse the received request by first checking sampling operators contained in *FROM* and *FROM STREAM* lines. If the request does not include a sampling operator, input RDF streams are processed continuously without any sampling phase. If not, after validation we create two (2) instances both *ContinuousSampQuery* and *SparqlSampQuery* that will be processed respectively by *ContinuousEngine* and *SparqlEngine*.
2. ***ContinuousEngine*** module receives a continuous query with associated sampling operators. Each sampling method is continuously applied on a window of input graph streams (*WindowSampling*). We then process the CQL part of C-SPARQL query on graphs within our buffered sample according to Esper syntax. Results are transmitted to the third module *SPARQLEngine*.
3. ***SPARQLEngine*** module also receives graphs from sampled static RDF repository (*SampStaticRDF*). It runs, in last step, the SPARQL query over continuous and static graphs already sampled.

6 Evaluation

This section evaluates the quality and relevance of our extension. To do this, we focus on performance achieved in terms of execution time and preserving data semantics. As a case study, we consider a set of data from sensors deployed in a large water distribution network. Data observed can be pressure, flow, chlorine or temperature. The management of water distribution network needs capabilities for real time processing and reasoning over streams in order to rapidly detect anomalies like water leaks. We consider the treatment of 80000 graphs, where each graph contains 10 triples. This set of data is sent in form of graphs and triples streams respectively at rates of 500 graphs/second and 5000 triples/second. Experiments in this section are performed on a computer with a processor at 2.66GHz, Core 2 Duo and 4 GB of RAM.

For performance evaluation of query execution time, we consider the simple query given below. The aim of the query is to return the average pressure values captured by each sensor. The query is performed over 1000 sampled graphs with respectively UNIFORM, RESERVOIR and CHAIN operators.

```
REGISTER QUERY AvgWaterPressure AS
PREFIX ex: <http://waterdist.org/>
SELECT    ?sensorID    (AVG(?pressureValue)  AS  ?AvgPressure)
FROM STREAM <http://waterdist.org/stream> [RANGE TRIPLES 1000]
                                  [SAMPLING [window] percent|size]
WHERE {?sensorID ex:hasPressure ?pressureValue . }
GROUP BY     ?sensorID
```

For sampling operators (*UNIFORM*) and (*RESERVOIR*) we observe in Fig. 4 and 5, the evolution of the query processing time by respectively varying the sampling percentage (*percent*) and the reservoir size (*size*). Finally, with CHAIN operator we

evaluate in Fig. 6, the query execution time under two conditions: varying sampling percentage and Chain window size (*Window*).

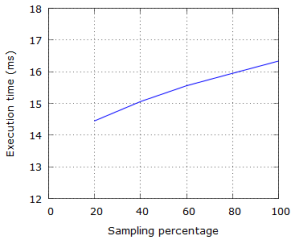

Fig. 4: Uniform.

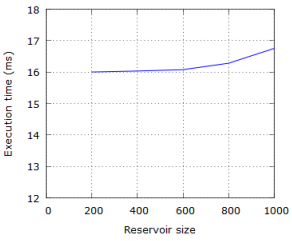

Fig. 5: Reservoir.

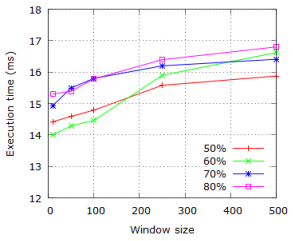

Fig. 6: Chain.

With uniform random sampling without replacement (Fig. 4), the execution time increases depending on the sampling percentage. We also note a similar trend with reservoir sampling (Fig. 5) depending on reservoir size kept fixed in memory. The evolution of the execution time with chain sampling (Fig. 6) depends on the window size and the ratio. Whatever the sampling percentage, query execution time follows a growing trend. This can be explained by the selection of a random successor items without redundancy. Thus, observations confirm performances gained in reducing on the fly the load from input streams.

For the evaluation of the semantic links preservation between data in sample, we consider the following query:

```
REGISTER QUERY sourceSensorID AS
PREFIX ex: <http://waterdist.org/>
SELECT    ?sensorID    ?pressureValue
FROM STREAM <http://waterdist.org/stream> [RANGE TRIPLES 10]
                                  [SAMPlING [window] percent|size]
WHERE { ?sensorID ex:hasPressure ?pressureMnemonic .
        OPTIONAL{ ?pressureMnemonic ex:value ?pressureValue .} }
ORDER BY    ?sensorID
```

We first execute this query continuously, in base scenario (without any sampling operation), and then with triple-oriented sampling and finally with graph-oriented sampling. The request selects for the last 10 graphs or triples observed in sample, the sensor ID and its corresponding pressure value. In the graph, the sensor identifier is connected to its captured pressure value through an another node. We compute for each sampling method (triple-based and graph-based), the number of correct and complete results (e.g sensor ID and corresponding pressure value). We subsequently calculate the loss rate in both cases (triple-oriented and graph-oriented) using the following formula:

$$\textbf{Loss rate (\%)} = \frac{\textbf{NbrNS - NbrWS}}{\textbf{NbrNS}} * \textbf{100}, \text{ where}$$

NbrNS = Number of correct and complete results in base scenario
NbrWS = Number of correct and complete results with sampling.

Table 2 shows the loss rates calculated by varying sampling parameters of uniform and reservoir methods. The number of correct and complete results in base scenario is **NbrNS=10625**. With graph-oriented sampling, we observe a lower than

Table 2: Loss rate between triple-oriented and graph-oriented sampling

Operator	**Triple oriented sampling**		**Graph oriented sampling**	
	Correct and complete results	**Loss rate (%)**	Correct and complete results	**Loss rate (%)**
UNIFORM				
$P = 20\%$	19	**99.82**	2108	**80.16**
$P = 40\%$	73	**99.31**	3614	**65.98**
$P = 80\%$	181	**98.2**	7137	**32.82**
RESERVOIR				
$Size = 2$	24	**99.77**	2117	**80.07**
$Size = 4$	144	**98.64**	4210	**60.37**
$Size = 8$	522	**95.08**	7240	**31.85**

loss rate with triple-oriented. Indeed, this can be explained by random sampling in the case of the triple-oriented where the triple containing a sensor identifier and the one containing its corresponding pressure value might not be both in the sample. In comparison, graph-oriented sampling maintains semantic links ensuring that sensors are always associated with their corresponding pressure values in the sample.

7 Conclusion and future works

Massive data stream management is a permanent industrial concern and a scientific challenge. The application of semantic web technologies for data stream remains troubled by the current volumes and rapid generation of streams. In this paper, we took advantages of existing sampling techniques and proposed an extension of the C-SPARQL system for real-time sampling of RDF graph streams. The oriented-graph approach allowed us to preserve semantic links within samples and thereby improved its representativeness. We implemented three sampling operators with different performances observed in terms of execution time.

We are interested in our future work into sampling on the fly over RDF streams using methods based on biased algorithm by associating different weights to the semantic stream items. We will also take into account specificities and background (context-based) to produce suitable samples, in a distributed environment.

Acknowledgment

This work was performed under the FUI Waves project. This project aims to design and develop a distributed processing platform of massive data streams. The case study concerns the real-time monitoring of a drinking water distribution network.

References

[Anicic et al., 2011a] Anicic, D., Fodor, P., Rudolph, S., and Stojanovic, N. (2011a). Ep-sparql: a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*, pages 635–644. ACM.

[Anicic et al., 2011b] Anicic, D., Fodor, P., Rudolph, S., Stuhmer, R., Stojanovic, N., and Studer, R. (2011b). Etalis: Rule-based reasoning in event processing. *Reasoning in Event-Based Distributed Systems*, 347:99.

[Arasu et al., 2004a] Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., and Widom, J. (2004a). Stream: The stanford data stream management system. *Book chapter*.

[Arasu et al., 2004b] Arasu, A., Babu, S., and Widom, J. (2004b). Cql: A language for continuous queries over streams and relations. In *Database Programming Languages*, pages 1–19. Springer.

[Babcock et al., 2002] Babcock, B., Datar, M., and Motwani, R. (2002). Sampling from a moving window over streaming data. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 633–634. Society for Industrial and Applied Mathematics.

[Barbieri et al., 2010] Barbieri, D. F., Braga, D., Ceri, S., and Grossniklaus, M. (2010). An execution environment for c-sparql queries. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 441–452. ACM.

[Bolles et al., 2008] Bolles, A., Grawunder, M., and Jacobi, J. (2008). Streaming sparql-extending sparql to process data streams. *The Semantic Web: Research and Applications*, pages 448–462.

[Calbimonte et al., 2010] Calbimonte, J.-P., Corcho, O., and Gray, A. J. (2010). Enabling ontology-based access to streaming data sources. In *The Semantic Web–ISWC 2010*, pages 96–111. Springer.

[Cao et al., 2012] Cao, J., Zhang, W., and Tan, W. (2012). Dynamic control of data streaming and processing in a virtualized environment. *IEEE Transactions on Automation Science and Engineering*, 9(2):365–376.

[Cochran, 2007] Cochran, W. G. (2007). *Sampling techniques*. John Wiley & Sons.

[Komazec et al., 2012] Komazec, S., Cerri, D., and Fensel, D. (2012). Sparkwave: continuous schema-enhanced pattern matching over rdf data streams. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 58–68.

[Le-Phuoc et al., 2011] Le-Phuoc, D., Dao-Tran, M., Parreira, J. X., and Hauswirth, M. (2011). A native and adaptive approach for unified processing of linked streams and linked data. In *The Semantic Web–ISWC 2011*, pages 370–388. Springer.

[Rete, 1982] Rete, C. (1982). A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence*, 19:17–37.

[Vijayakumar et al., 2010] Vijayakumar, S., Zhu, Q., and Agrawal, G. (2010). Dynamic resource provisioning for data streaming applications in a cloud environment. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 441–448. IEEE.

[Vitter, 1985] Vitter, J. S. (1985). Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57.