A Compressed, Inference-Enabled Encoding Scheme for RDF Stream Processing

Jérémy Lhez[1(✉)], Xiangnan Ren[1,2,3], Badre Belabbess[1,2], and Olivier Curé[1]

[1] LIGM (UMR 8049), CNRS, ENPC, ESIEE, UPEM, 77454 Marne-la-vallée, France
{jeremy.lhez,xiangnan.ren,badre.belabbess,olivier.cure}@u-pem.fr
[2] Atos, Bezons, France
{xiangnan.ren,badre.belabbess}@atos.fr
[3] ISEP - LISITE, 75006 Paris, France

Abstract. The number of sensors producing data streams at a high velocity keeps increasing. This paper describes an attempt to design an inference-enabled, distributed, fault-tolerant framework targeting RDF streams in the context of an industrial project. Our solution gives a special attention to the latency issue, an important feature in the context of providing reasoning services. Low latency is attained by compressing the scheme and data of processed streams with a dedicated semantic-aware encoding solution. After providing an overview of our architecture, we detail our encoding approach which supports a trade-off between two common inference methods, *i.e.*, materialization and query reformulation. The analysis of results of our prototype emphasize the relevance of our design choices.

1 Introduction

Semantic information of the Web of data, generally represented with the Resource Description Framework (RDF)[1] data model, is now being considered for real time analysis. This is the case in the Waves project[2], where we provide real-time analytics of RDF data streams for an international company leading innovation technologies for smart water network management. In particular, we are analyzing data captured from potable water networks in major cities in the world, e.g., studying pressure, flow, turbidity, pH, chlore and other chemical measures, in almost real-time. Some of the key goals of this project are to identify malfunctions in these water networks, *e.g.*, water leaks by analyzing flow and pressure measures, to explain their origins leveraging knowledge base enrichment and to predict potential issues within the pipeline system. With more relevant and faster agent interventions on the network, such research and development can have a substantial impact at both the environmental (to limit potable water loss) and economic (to reduce financial costs) levels. In fact, one must bear in mind that worldwide water leaks peaked to 32 billion m^3/year within last years,

[1] http://www.w3.org/TR/rdf-mt/.
[2] http://www.waves-rsp.org/.

E. Blomqvist et al. (Eds.): ESWC 2017, Part II, LNCS 10250, pp. 79–93, 2017.
DOI: 10.1007/978-3-319-58451-5_6

90% of them being invisible due to the underground nature of the network, which makes it a burning issue for the 21st Century.

Detecting water leakage could be performed using quantitative data without exploiting the possibilities of semantic web technologies. However, since we aim to explain discovered leaks, taking advantage of RDF technologies (*e.g.*, RDFS, OWL and SPARQL) and functionalities (*e.g.*, data and knowledge integration, reasoning) becomes a necessity. Such scenarios imply the association of expressive schemata, denoted as ontologies, and explicit measured data. Therefore, an intelligent knowledge management system should enable to infer valuable information that can help in providing sound and complete answers to a continuous query processing component or to help in the design of efficient data analytics.

The integration of a reasoning component in Event Stream Processing (henceforth ESP) is a complex task due to the general cost, in terms of computing resources and time, of inferring data using expressive ontologies. In order to address these requirements, we have designed a prototype system based on the following contributions: (i) we present a generic distributed streaming architecture that addresses materialization and query reformulation reasoning services (Sect. 2), (ii) we propose an encoding approach that minimizes system latency and supports inferences (Sects. 4 and 5), (iii) we highlight the efficiency of our compressing approach with results of an experimentation (Sect. 6).

2 Architecture

In Fig. 1, we present an overview of our architecture. Due to the usage of the Apache Kafka [14] and Apache Storm [12] components, we have designed a system capable of ensuring scalability, fault-tolerance, high throughput and low latency properties.

One characteristic of our project is its capacity to handle both static and dynamic data and knowledge. By static, we mean data and knowledge that are rarely updated while the dynamic aspect relates to the notion of streams arriving at a fast pace, potentially thousands of them per second.

The static aspect of our system consists in encoding a set of ontologies and knowledge bases that are specific to the application domain. In the case of the Waves project, the ontologies are addressing the following topics: sensors, *e.g.*, SSN[3] (Semantic Sensor Network), hydrology, *e.g.*, CUAHSI[4] and modeling physical quantities, units of measure, and their dimensions, *e.g.*, QUDT[5]. The system also integrates additional knowledge bases to represent water network geographical aspects. This is supported by the Geonames[6] and DBpedia[7] ontologies. These knowledge bases are stored in our external knowledge base component which is currently handled by the Virtuoso RDF store[8].

[3] https://www.w3.org/2005/Incubator/ssn/ssnx/ssn.
[4] http://his.cuahsi.org/ontologyfiles.html.
[5] http://linkedmodel.org/catalog/qudt/1.1/.
[6] http://www.geonames.org/.
[7] http://wiki.dbpedia.org/.
[8] http://virtuoso.openlinksw.com/.

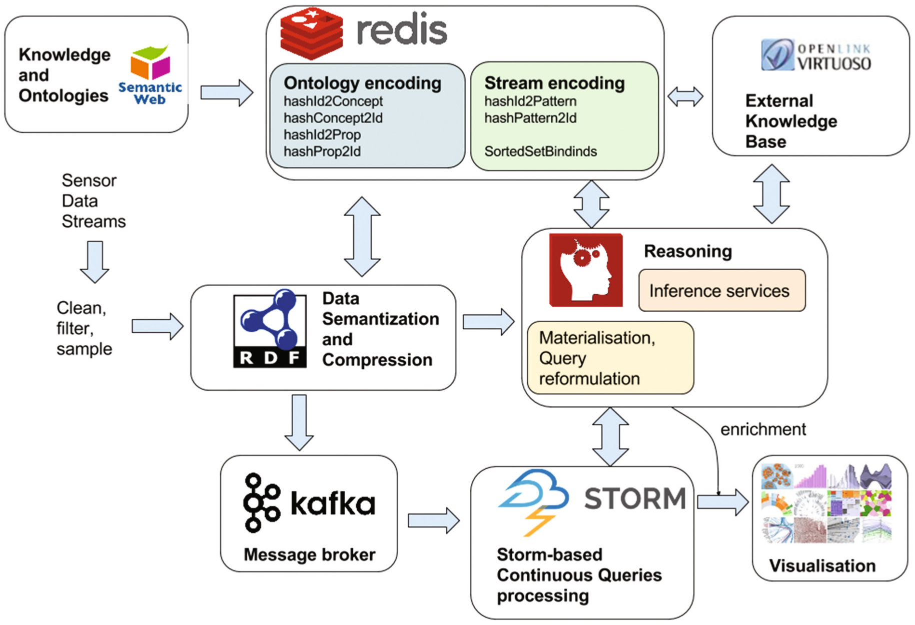


Fig. 1. Architecture overview

The remaining of the architecture is concerned with dynamic, event-based data which are handled by distributed components: Kafka as a distributed, partitioned, replicated commit log service, Storm as the distributed stream processing engine and Redis[9] as a key-value memory store.

A typical scenario in our system is as follows. First, measures are captured from a given sensor network. These streams are cleaned, filtered and possibly sampled before being serialized in a compact RDF format. These data are persisted on-demand to a Redis key-value store and sent to the Apache Kafka message broker. The Kafka component is becoming a standard in streaming processing engines and can be connected to most open source streaming engines (*e.g.*, Storm). Data are fetched from Kafka by a set of distributed nodes which implement the so-called Storm topology, *i.e.*, a network of so-called spouts and bolts. A spout is the source of data streams and can read data from an external framework like Kafka. A bolt is a processing logic unit that performs any kind of processing such as filtering, aggregating, joining, interacting with data stores. Each spout or bolt executes as many tasks across a Storm cluster, and each task corresponds to one thread of execution. Topologies execute across one or more worker processes. Each worker process is a physical Java Virtual Machine (JVM) and executes a subset of all the tasks for the topology. In a Waves topology, each spout subscribes to one stream represented by a Kafka topic, and each bolt

[9] http://redis.io/.

decompresses data and performs a continuous SPARQL query, whose language is inspired by C-SPARQL [2].

The streaming engine is connected to a visualization module whose only goal is to ease the interpretation of analyzed streams through different forms of graphics. Some of these visualizations may require some data enrichment supported by the set of reasoning services.

In the remaining of this paper, we focus on reasoning aspects which is tightly coupled with the RDF serialization solution. Due to space limitations, we do not address other components of the architecture.

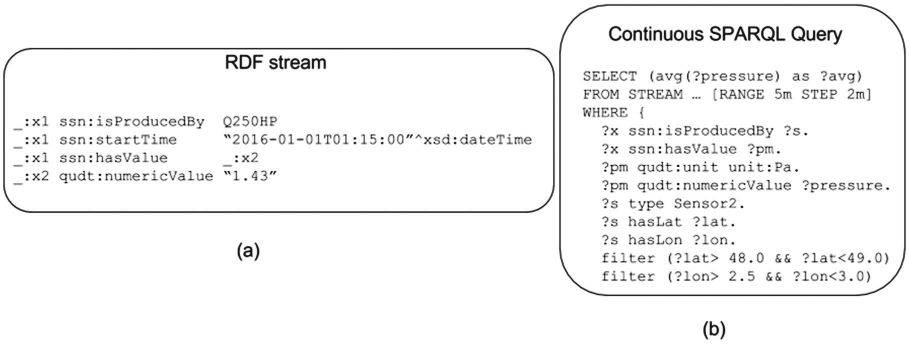
3 Running Scenario

In this section, we present a practical use case of the Waves project in which a set of sensors is generating simple RDF streams corresponding to some physical measures. In Fig. 2(a), we present a simple, raw stream, denoted S, providing a pressure measure from a sensor characterized with identifier "Q250HP". In order to detect and predict interesting situations in real-time, end-users of our platform can define continuous queries to the system. Figure 2(b) proposes such a query expressed in C-SPARQL [2], henceforth denoted Q. Intuitively, the query computes the pressure average, expressed in the Pascal unit, measured in fixed windows lasting 5 min and sliding every 2 min. Moreover, these averages are only computed for sensors situated in a certain location (a bounding box is specified from ranges of latitude and longitude values) and for a certain sensor type (namely Sensor2). Clearly, this raw stream S does not satisfy the `WHERE` clause of the C-SPARQL query Q: neither the type of the sensor, the unit of its measure and location are specified in the raw stream. Hence, the result set of Q over S would be empty. We consider that given the messages sent by real-world sensors, such situations are bound to occur frequently.

In fact, sensor "Q250HP" is providing measures in the Pascal unit, is of type Sensor3 and is situated in the bounding box expressed in Q. But these information are only stored in some external knowledge base.

This knowledge base contains two components. An ontology stating that Sensor3 and Sensor4 are sub classes of Sensor2, expressed in a Description Logic [1] formalism as $Sensor3 \sqsubseteq Sensor2$ and $Sensor4 \sqsubseteq Sensor2$. And a set of facts stating that sensor Q250HP provides pressure values expressed in the Pascal unit and is located at latitude 48.59 and longitude 2.75. Thus the data stream, if properly enriched, can satisfy the continuous query Q.

Instead of performing joins at run-time for each incoming events, we prefer to materialize these events with the information that may satisfy a continuous query. Intuitively, the continuous queries are retrieving events from a given set of Kafka topics. Thus it is possible to define possible materialization when a query is associated to a topic. The problem then amounts to define a compact and efficient serialization for the RDF graphs corresponding to the events.


RDF stream
(a)
Continuous SPARQL Query
(b)

```
_:x1 ssn:isProducedBy  Q250HP
_:x1 ssn:startTime     “2016-01-01T01:15:00”^xsd:dateTime
_:x1 ssn:hasValue      _:x2
_:x2 qudt:numericValue “1.43”
```

```
SELECT (avg(?pressure) as ?avg)
FROM STREAM … [RANGE 5m STEP 2m]
WHERE {
  ?x ssn:isProducedBy ?s.
  ?x ssn:hasValue ?pm.
  ?pm qudt:unit unit:Pa.
  ?pm qudt:numericValue ?pressure.
  ?s type Sensor2.
  ?s hasLat ?lat.
  ?s hasLon ?lon.
  filter (?lat> 48.0 && ?lat<49.0)
  filter (?lon> 2.5 && ?lon<3.0)
```

Fig. 2. (a) RDF stream S and (b) continuous query Q

4 Compression Approaches

4.1 Knowledge Base Encoding

With our knowledge base encoding approach, we provide an efficient encoding scheme and data structures to support the reasoning services associated to the terminological and assertional boxes (resp. Tbox and Abox). The input ontology is considered to be the union of all ontologies necessary to operate over one's application domain (*e.g.*, SSN, CUAHSI, QUDT, Geonames and DBpedia). In the current version of our work, we address the ρdf [10] subset of RDFS, meaning that we are only interested in the `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain` and `rdfs:range` constructors. Our Tbox encoding scheme uses our LiteMat system (full details in [5]). Intuitively, it provides a unique, semantic-based identifier to each entry of the Tbox (*i.e.*, class and properties). That is, the identifier of each Tbox element is prefixed by the binary identifier of its super element. This approach enables to represent the class and property hierarchies in a compact way since the identifier of a given class (resp. property) provides all its direct and indirect super classes (resp. properties). Moreover, to capture all inferences related to the class hierarchy, the encoding relies on a classification performed by an OWL reasoner. To support `rdfs:domain` and `rdfs:range` inferences, the property dictionary is extended with the class identifiers that respectively correspond to their domain and range. A final dictionary is generated over instances of the Abox. Once all these dictionaries have been computed and stored in a Redis key/value store, the system can encode the whole Abox, which is then constituted of integer value triples and stored in a Virtuoso instance.

4.2 RDF Distributed Stream Compression

A distributed architecture integrating reasoning services requires low latency to cope with massive real-time streams. However, frequent data transfers between several components (*e.g.*, messaging middleware, data stores, etc.) produce significant network overhead. There are quite various methods to deal with this complex issue and the one we focus on is compression. RDF data are particularly adapted to efficient compression.

Like RDSZ [8] and ERI [6], our approach assumes that events in a given stream share structural similarities, *i.e.*, the RDF graph shapes are similar. We can leverage on this aspect to limit the stream memory footprints. As the compression exploits structural similarities, a new graph (*e.g.*, set of RDF triplets) can be represented on the basis of the previous graph. Our approach breaks up each graph into two parts, namely the graph Pattern and value/variable Bindings that are associated to a graph pattern, hence the PatBin denotation.

In Fig. 3, we present the different steps necessary to generate a pattern signature. This deterministic approach will serve to compare stream graph signature with continuous query signatures in an efficient manner. Considering an arriving graph event corresponding to data stream S of our running example, Fig. 2(a), we use our previously computed property dictionary, Fig. 3(a), to replace property IRIs with integer values. We thus obtain a more compact set of triples (Fig. 3(b)). The compactness of this signature takes benefits from the facts that all events we have encountered correspond to trees. Starting from the root node of our tree, we then sort the graph, in a level-wise manner, according to the property integer values. This order is then used to define a pattern signature (Fig. 3(c)). Intuitively, a signature is composed of property identifiers separated by ':' symbols to delimit properties occurring at the same tree level, '(','')' symbols to describe sub-trees. Note that subjects and objects are not necessary in these signatures since our signature language enables to easily reconstruct the original shape of the tree, *i.e.*, by abstracting subjects and objects with variables.

Correspondences between implicit graph pattern variables and their values (*i.e.*, on triple subjects and objects) are represented by bindings and are sent to Kafka. For a graph N, the bindings are compressed using a differential approach

(a) Property Dictionary extract

```
http://purl.oclc.org/NET/ssnx/ssn#isProducedBy : 144
http://purl.oclc.org/NET/ssnx/ssn#hasValue : 140
http://purl.oclc.org/NET/ssnx/ssn#startTime : 80
http://data.nasa.gov/qudt/owl/qudt#numericValue : 86
```

(b) Intermediary (sorted) pattern

```
_:x1 140 "2016-01-01T01:15:00"^^xsd:dateTime
_:x1 144 Q250HP
_:x1 80  _:x2
_:x2 86  1.43
```

(c) Output pattern signature

```
140:144:80:(86)
```

Fig. 3. Pattern signature process

based on the bindings of the previous graph $N-1$. If the graph N shares some bindings with the $N-1$ graph, then they are replaced by blanks.

Moreover, the mechanism is still not adapted for distributed computing, since encoding the current N graph is based on the bindings of the previously processed $N-1$ graph. However, this implies data exchange between distributed machines if these graphs are processed in different nodes, leading to a network overhead. To solve this, we propose to encode the current N graph based on the bindings of the initially processed graph from which the pattern has been extracted and stored. Hence, we create a context in which we put the pattern, the bindings of the graph from which the pattern has been extracted and the occurring namespaces. All the incoming graphs are encoded based on the context with which they share the same pattern. To guarantee the access to contexts in the distributed infrastructure, we need to store them in a centralized system. Again, Redis has been chosen for storage due to its convenient features (e.g. key-value in-memory store, fast read/write, etc.). Each context created is automatically stored in Redis. The contexts being stored in a centralized system, all the machines have access to compress and decompress operations of RDF graphs. In addition, each machine benefits from its local cache LRU (Least Recently Used) mechanism. That is each machine contains the latest recently used patterns processed by this machine and serves to speed up the contexts read access.

5 Inference Solution

In this section, we present our reasoning approach which is based on a trade-off between materialization and query reformulation. Although these inference solutions can be used independently, *i.e.*, materialization or query reformulation alone, we highlight that the full potential of the approach is to combine both of them.

5.1 Materialization

The goal of the materialization step is to enrich raw RDF streams in such a way that they can potentially satisfy some given continuous queries. By potentially satisfying a query, we mean that there is a graph homomorphism between a stream and a continuous query graph pattern. It does not necessarily means that a materialized streaming graph pattern actually satisfies the query since some values may not satisfy certain conditions, *e.g.*, filters, of the query. This enrichment is based on retrieving some additional data from external knowledge bases which are stored and possibly encoded in an RDF repository (*e.g.*, the Virtuoso RDF store).

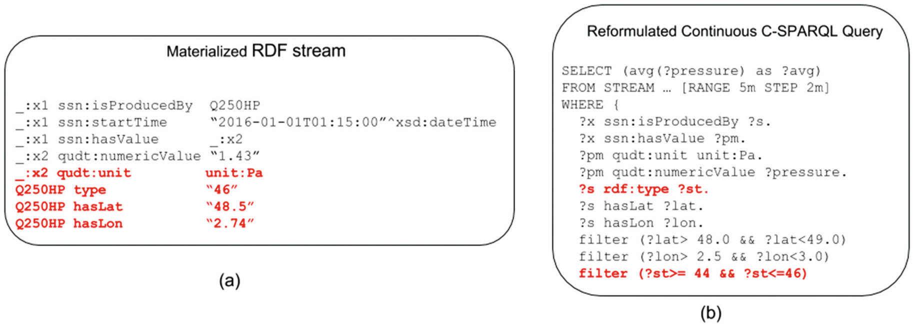
Of course, the task of discovering to what extent a materialization can transform an unsatisfiable raw stream into a potentially satisfiable one, must be performed automatically by the system. That is, the system has to find out a set of sound transformations according to a set of continuous queries and knowledge base axioms. We compute such discoveries using graph matching operations over the graph patterns of RDF streams and continuous queries. This approach is valid since the vocabularies used in these two components correspond to our predefined set of encoded ontologies (Sect. 4.2).

Given the potential high volume of different data stream types, *e.g.*, in our use case, measures such as pressure, flow, chlorine, turbidity, etc., and the number of continuous queries, it is important to propose an efficient discovery approach. Our method considers that streams are submitted to Kafka topics and that these topics are processed to retrieve stream graph patterns. Moreover, the continuous queries (implemented as Storm bolts) are connected to Storm Spouts which are themselves related to Kafka topics. Hence, it is possible to reduce the space search by matching pairs of stream and continuous query graph patterns connected to the same Kafka topics.

Given a Kafka topic T, the graph matching discovery problem amounts to finding if a Stream Graph Pattern SGP is a sub graph of a given continuous query graph pattern CQ, *i.e.*, excluding FILTER, GROUP BY and OPTIONAL clauses and considering group graph patterns related by UNION clauses as individual queries. This search for a sub graph relationship is semantic-aware, meaning that class and property subsumption relationships are taken into account. For instance, with our previously defined ontology, the following situation: $SGP = \_ : x1\ type\ Sensor3$, $CQ = \_ : x4\ type\ Sensor2$ would correspond to a sub graph relationship due to the $Sensor3 \sqsubseteq Sensor2$ axiom. Note that this is not the case for this other example: $SGP = \_ : x1\ type\ Sensor2$, $CQ = \_ : x4\ type\ Sensor3$.

If SGP is not a sub graph of CQ then we consider that this sort of data streams can not be enriched to satisfy the continuous query. In the case SGP is equal to CQ then no materialization is required since SGP can potentially satisfy CQ out-of-the-box. Finally, if SGP is a sub graph of CQ then the triple-based difference between CQ and SGP is computed to identify the set of triples that are missing in SGP to potentially satisfy CQ. Based on mapping assertions between subject and object identifiers of SGP and CQ, we can instantiate a computed triple set from external knowledge bases. In our running example, this amounts to generating the bold lines of Fig. 4(a). Basically, the unit, location and type of sensor "Q250HP" triples are added to the streams. Note that the sensor type is expressed with the integer value corresponding to its binary encoding: the binary identifiers of $Sensor2$, $Sensor3$ and $Sensor4$ are respectively 101100, 101110 and 101101 which respectively correspond to the 44, 45 and 46 integer values.

The discovery of a graph match is fast due to our compact, deterministic graph signature representation. Nevertheless, it may become a performance bottleneck due to high velocity stream production. To prevent this from happening, the system stores discovered graph pattern correspondences and only searches for new ones when novel stream patterns are recorded in the system and/or when continuous queries are updated or inserted. A discovered graph pattern exactly matches the graph associated to a materialized stream and is expressed as the original graph patterns, *i.e.*, as defined in Sect. 4.2.


Materialized RDF stream

```
_:x1 ssn:isProducedBy  Q250HP
_:x1 ssn:startTime     "2016-01-01T01:15:00"^xsd:dateTime
_:x1 ssn:hasValue      _:x2
_:x2 qudt:numericValue "1.43"
_:x2 qudt:unit         unit:Pa
Q250HP type            "46"
Q250HP hasLat          "48.5"
Q250HP hasLon          "2.74"
```

(a)

Reformulated Continuous C-SPARQL Query

```
SELECT (avg(?pressure) as ?avg)
FROM STREAM … [RANGE 5m STEP 2m]
WHERE {
  ?x ssn:isProducedBy ?s.
  ?x ssn:hasValue ?pm.
  ?pm qudt:unit unit:Pa.
  ?pm qudt:numericValue ?pressure.
  ?s rdf:type ?st.
  ?s hasLat ?lat.
  ?s hasLon ?lon.
  filter (?lat> 48.0 && ?lat<49.0)
  filter (?lon> 2.5 && ?lon<3.0)
  filter (?st>= 44 && ?st<=46)
```

(b)

Fig. 4. Materialized RDF stream and reformulated continuous query

5.2 Query Reformulation

The goal of the query reformulation component is to modify the original continuous query such that subsumption relationships are properly addressed. A special attention is given to classes specified in `rdf:type` triples. If any of these classes are at some point a super class in our Tbox then some reformulation is necessary. The system proceeds as follows: in each triple pattern with a `rdf:type` property, replace the class C (object position) with a non previously used variable (denoted V_i). In Fig. 2(b), the `?s type Sensor2` triple is replaced by the triple `?s type ?st` in Fig. 4(b). Then a SPARQL `FILTER` clause is introduced in the reformulated query on that variable V_i. The goal is to cover all possible sub classes of the original class C. The specification of these classes are performed at the encoding level and hence benefits from the nice properties of our ontology encoding. Due to our encoding approach, we know that sub classes of $Sensor2$ are necessarily included in the "101100" and "101111" identifier range which correspond to respectively to the 44 and 46 values. These lower and upper bound values are easily computed (using two bit shift operations) from the binary version of C's identifier. With this approach, we cover all sub classes of a given class with a single `FILTER` query line, independently of the length of this class subsumption relationships. The last bold line of Fig. 4(b) represents this filter clause for our running example.

A similar approach is perform for the property hierarchy. It consists of analyzing whether any of the non `rdf:type` properties is at some point a super property. Then the system operates in an identical manner: it replaces the property with a new variable and inserts a `FILTER` line that restricts the range of accepted property values for that variable.

Note that this approach is particularly efficient when several reformulation (*e.g.*, on classes and properties) are needed in a single query. With our filter approach, a reformulated query grows linearly and not exponentially as is generally the case for standard query reformulation approaches.

6 Evaluation

In this evaluation section, we provide results of experiments about the ontology and the stream compression components. The evaluation has been conducted on real dataset describing some characteristics of Waves's water network for a large city in the Paris area (France). In the following, we present the computational environment, the dataset and the results obtained. Due to space limitations, this experimentation focuses on the PatBin and reasoning aspects.

6.1 Computational Environment

Throughout this experimentation section, we are using two different computational settings. The evaluation concerning the compression have been realized on a laptop with a Windows 8 operating system, equipped with an Intel Core i7 processor (2.90 GHz), 16 GB of RAM, running JDK/JRE 1.8. The ontology encoding evaluation has been performed on a Linux Ubuntu 14.04 distribution with 16 GB of RAM, Intel Core I5 quad-core processor and running a JDK 1.8. We used the HermiT version 1.3.8 as an external reasoner and programmed the encoding solution with Apache Jena 3.0.0. Finally, we are using Apache Spark [15] version 1.5.2 for the encoding of the ABox. The Spark cluster consists of 3 Dell PowerEdge machines equipped with 64 GB of RAM.

6.2 Datasets

For experimentation, we use a real world dataset describing different water measurements captured by sensors. Values of flow, pressure and chlorine are examples of these measurements. These data are provided in CSV format and need to be represented in a semantic model. For this, we are annotating values using three popular ontologies: SSN, CUAHSI-HIS and QUDT. Each sensor observes at least one physical phenomenon or a chemical property, and hence produces timestamped streams containing an observation.

6.3 Results

Knowledge Base Encoding Evaluation. We ran our ontology compression Java program a total of five times and obtained an average of 18.8 s for the merged ontology presented in Table 1. With respect to the low numbers of classes and properties, this duration can be considered rather long. In fact, this can be justified by the rather high expressivity of the resulting ontology which happens to correspond to $\mathcal{SROIQ}$(D) Description Logics. This expressiveness matches the OWL2 DL ontology language which is known to be the OWL fragment with the highest computational complexity for standard inference services (apart from OWL Full which is undecidable).

Comparatively, The same algorithm is able to encode the DBPedia OWL ontology, which contains over 800 classes and 3000 properties, in less than

Table 1. Compressed ontology in terms of number of classes, object and data type properties

Ontology	#classes	#object pr	#data pr	Duration (sec.)
SSN	117	142	6	–
QUDT	229	69	29	–
CUAHSI extract	103	0	0	–
Merged ontology	449	174	35	18.8
DBPedia	814	3,035	1,310	4.1
Wikidata	213,958	255	98	118

4 s for an expressivity corresponding to $\mathcal{ALCHF}$(D) DL. The encoding of the Wikidata ontology takes approximatively 2 min. This is mainly due to the large class hierarchy (over 200,000 classes) and not to its expressiveness which corresponds to the $\mathcal{AL}$ DL.

Finally, we provide an evaluation of a data instance encoding which is needed for static knowledge bases. This processing is distributed over our Spark cluster and the measures are presented in Table 2. These measures are about 70% faster than state of the art compression approaches defined over Apache Hadoop [13].

Table 2. Duration and throughput of data instances, triples in $*10^6$, duration in seconds and throughput in triples/sec.

Dataset	#Triples	Duration	Throughput
DBPedia	79.1	282.2	280 943
Wikidata	242.1	1 334.8	181 394

Signature Generation Performance. We used RDSZ results to check the algorithm's compression performance. A specific Java class stores the algorithm statistics in terms of performance and compression rate, thus we made some similar measures for PatBin to ensure a fair comparison. The system time was measured once the input file was parsed as a Java String containing all triples, and a second time right after the compression step. The subtraction of those values gives the compression performance. RDSZ's statistics also provide information about the compression rate, by giving the size of the compressed output (in UTF-8 bytes); therefore we used this method for our algorithm again. Both those measures are presented in Table 3; we performed a series of verification for different input file sizes (using the turtle serialization). We used the basic configuration of RDSZ algorithm, with no specific argument. As we can see, the compression performance is much faster for PatBin; this is mostly due to the fact that we only have to deal with predicates. Indeed, RDSZ must initialize its binding table with both subject and predicates, and verify for each class if it is

Table 3. Signature generation performance for RDSZ and PatBin, time of compression in microseconds, size of he signature in bytes

	RDSZ time	PatBin time	RDSZ size	PatBin size
5 triples	383	1	312	13
10 triples	387	2	370	29
25 triples	394	3	425	89
50 triples	397	6	523	184
100 triples	401	10	750	382

not already present in the table. We have twice less work to do with only the properties. PatBin also has better results in terms of compression rate, which tends to decrease for big input files: this is mostly due to the fact that we used examples files that are represented as big forests, thus having a long signature on several lines.

Graph Matching Performance. The graph matching performance has been performed by checking the equality between a newly compressed string, and an array of stored compressed strings, acting as a cache. We made our evaluation on several sizes of cache, to vary the number of comparison made; we also tested different sizes of files (different numbers of triples) in order to have an output longer or shorter. For each individual evaluation, we took files with the same number of triples, and we also made sure that the input file was not in the cache; this ensured each value in the cache would be verified, and thus the test would not be biased by ending the checking too soon. Both the results in cache and to be checked were (different) compressed results obtained from a C-SPARQL query. The results are displayed in the Fig. 5; since the results have a very high variance, we had to do an average of different results to have valid results. The measures concern only the matching: the signature generation for the file to be matched is not taken in account. Each measure has been identified by a point on its curve, for better visualization. The three measures for PatBin appear mingled with the lower (X) axis, because the computation time is much shorter than RDSZ. In both cases, the matching time increases when we the cache size and/or the triple number. For PatBin, the results are much better: with 25 triples and a cache size of 100 compressed strings, the checking time is only about 19 µs, *i.e.*, two orders of magnitude lower than RDSZ. This proportion cannot be established precisely because of the variance, however the computation times remain much better for PatBin. This is due to the fact that the signature obtained after compression is much more compact that the one of RDSZ, since PatBin does not retain the triples in its signature. We also checked hot and cold performances for cache searching: in both cases, we filled a cache with 1000 random patterns, and checked if a new entry was present in the cache. We verified that the randomness ensures the caches are completely verified in both cases. The cold performances give an execution time of 105 µs for PatBin, and 156 for

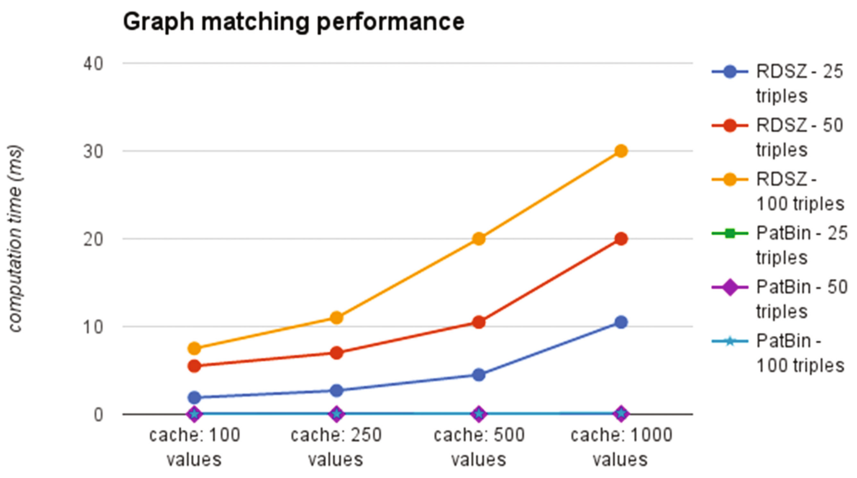


Fig. 5. Materialized RDF stream and reformulated continuous query

RDSZ. For the hot performances, we computed the average of five executions: PatBin is still more efficient, with 95 µs of matching time against 160 for RDSZ.

7 Related Work

We consider two systems that integrate reasoning within a RDF streaming context. IMaRS [3] incrementally maintains a materialization of ontology entailments in a timely manner. The system extends the DRed [9] approach with the use of the window operators and the introduction of an expiration time for each triple. The system does not interact with a query reformulation component, is not distributed and it is recognized that automatically defining efficient expiration time is difficult in a streaming context. Finally, StreamQR [4] proposes a query reformulation solution which is based on the kyrie rewriter. The architecture of the system does not support scalability and interactions with a materialization component have not been considered.

Several systems consider RDF stream compression. The Zstreamy [7] system is presented as a scalable platform for publishing semantic streams on the Web. The compression approach is simply based on a standard Zlib compression. CQELS Cloud [11] addresses the problem of scalable stream processing and proposes a simple dictionary encoding approach reminiscent of RDF stores. RDSZ [8] (RDF Differential Stream compressor based on Zlib) and ERI [6] (Efficient RDF Interchange format) correspond to lossless RDF stream compression approaches. Both take advantage of structural similarities of RDF graph events. ERI proposes a more fine-grained approach to pattern and pattern binding representations. Moreover, ERI does make an extended usage of differential compression as RDSZ does. In general, the compression approaches of the two systems are comparable with RDSZ being slightly more efficient for randomly distributed data and streams using a small set of predicates. In terms of processing performance, ERI is more efficient than RDSZ for the compression phase while the RDSZ is faster than ERI for the decompression operation. Concerning compression, RDSZ pays the cost of the differential computing while for decompression, ERI is slower due to the possibly large numbers of sequence of RDF molecules. These systems are not benefiting from a compact, semantic-aware KB encoding, do not propose a graph pattern signature nor interact with materialization/query reformulation components.

8 Conclusion and Lessons Learned

In the context of the Waves project, we were confronted to a real-world use case that is principally ingesting numerical measures from a set of sensors. At first sight, such a scenario does not seem like the ideal playground for semantic technologies. Nevertheless, due to the integration of external (*e.g.*, Geonames, DBpedia) and domain specific (*e.g.*, SSN, CUAHSI) knowledge bases, as well as RDF related technologies (*e.g.*, SPARQL, RDFS, OWL), we were able to highlight the added value of a semantic approach. The main impact was the ability to explain some network malfunctions via the execution of inference-enabled continuous SPARQL queries. Of course, one of the key learned lesson concerns the impact of reducing latency when reasoning over large event streams. We found out that finding a trade-off between materialization and query reformulation was an important factor in reducing processing latency. But this approach is reaching its full potential with the kind of semantic-aware encoding and compression presented in this work.

As future work, we aim to test Waves's system on diverse IoT contexts and thus emphasize that our approach can be generalized to different use cases. Moreover, we are currently implementing an adaptive query processing engine to guarantee the execution of optimized continuous SPARQL queries. Finally, we will extend LiteMat's inference capabilities with support for RDFS++ (an ontology language supported by the Allegrograph RDF Store), *i.e.*, supporting RDFS as well as `owl:sameAs`, `owl:transitiveProperty` and `owl:inverseOf` ontology constructs.

Acknowledgment. This work has been supported by the Waves project which is partially supported by the French FUI (Fonds Unique Interministériel) call #17.

References

1. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, Cambridge (2003)
2. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: C-SPARQL: SPARQL for continuous querying. In: Proceedings of the 18th International Conference on World Wide Web, pp. 1061–1062 (2009)

3. Barbieri, D.F., Braga, D., Ceri, S., Valle, E., Grossniklaus, M.: Incremental reasoning on streams and rich background knowledge. In: Aroyo, L., Antoniou, G., Hyvönen, E., Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) ESWC 2010. LNCS, vol. 6088, pp. 1–15. Springer, Heidelberg (2010). doi:10.1007/978-3-642-13486-9_1
4. Calbimonte, J.-P., Mora, J., Corcho, O.: Query rewriting in RDF stream processing. In: Sack, H., Blomqvist, E., d'Aquin, M., Ghidini, C., Ponzetto, S.P., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9678, pp. 486–502. Springer, Cham (2016). doi:10.1007/978-3-319-34129-3_30
5. Curé, O., Naacke, H., Randriamalala, T., Amann, B.: LiteMat: a scalable, cost-efficient inference encoding scheme for large RDF graphs. In: 2015 IEEE International Conference on Big Data, Big Data 2015, pp. 1823–1830 (2015)
6. Fernández, J.D., Llaves, A., Corcho, O.: Efficient RDF interchange (ERI) format for RDF data streams. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8797, pp. 244–259. Springer, Cham (2014). doi:10.1007/978-3-319-11915-1_16
7. Fisteus, J.A., Garcia, N.F., Fernandez, L.S., Fuentes-Lorenzo, D.: Ztreamy: a middleware for publishing semantic streams on the web. Web Semant. Sci. Serv. Agents World Wide Web **25**, 16–23 (2014)
8. Fernández, N., Arias, J., Sánchez, L., Fuentes-Lorenzo, D., Corcho, Ó.: RDSZ: an approach for lossless RDF stream compression. In: Presutti, V., d'Amato, C., Gandon, F., d'Aquin, M., Staab, S., Tordai, A. (eds.) ESWC 2014. LNCS, vol. 8465, pp. 52–67. Springer, Cham (2014). doi:10.1007/978-3-319-07443-6_5
9. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. SIGMOD Rec. **22**(2), 157–166 (1993)
10. Muñoz, S., Pérez, J., Gutierrez, C.: Simple and efficient minimal RDFS. J. Web Sem. **7**(3), 220–234 (2009)
11. Le-Phuoc, D., Nguyen Mau Quoc, H., Le Van, C., Hauswirth, M.: Elastic and scalable processing of linked stream data in the cloud. In: Alani, H., et al. (eds.) ISWC 2013. LNCS, vol. 8218, pp. 280–297. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41335-3_18
12. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., Ryaboy, D.: Storm@twitter. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD 2014, pp. 147–156 (2014)
13. Urbani, J., Maassen, J., Bal, H.E.: Massive semantic web data compression with mapreduce. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC 2010, pp. 795–802 (2010)
14. Wang, G., Koshy, J., Subramanian, S., Paramasivam, K., Zadeh, M., Narkhede, N., Rao, J., Kreps, J., Stein, J.: Building a replicated logging system with apache Kafka. PVLDB **8**(12), 1654–1665 (2015)
15. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: 2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2010 (2010)