Enabling Query Technologies for the Semantic Sensor Web

*Jean-Paul Calbimonte *, Universidad Politécnica de Madrid, Spain*
Hoyoung Jeung, SAP Research, Australia
Oscar Corcho, Universidad Politécnica de Madrid, Spain
Karl Aberer, Ecole Polytechnique Fédérale de Lausanne, Switzerland

ABSTRACT

Sensor networks are increasingly being deployed in our environment for many different purposes. The observations that they produce are made available with heterogeneous schemas, vocabularies and data formats, making it difficult to share and reuse this data, for other purposes than those for which they were originally set up. We propose an ontology-based approach for providing data access and query capabilities to streaming data sources, allowing users to express their needs at a conceptual level, independent of implementation and language-specific details. In this article we describe the theoretical foundations and technologies that enable exposing semantically enriched sensor metadata, and querying sensor observations through SPARQL extensions, using query rewriting and data translation techniques according to mapping languages, and managing both pull and push delivery modes.

Keywords: Semantic Web, Sensor Technologies, Sensor Networks, Semantic Sensor Web, Query Processing.

INTRODUCTION

Every second, massive amounts of data are being produced by sensors all around the world. From environmental measurement devices to smartphones, the sources of sensor data continue to proliferate, increasing the possibility of blending the diverse sources to collaboratively detect and identify a multitude of observations, from simple phenomena to complex events and situations. As these sensors become more accessible, due to lower costs and simpler configuration and maintenance, they can be deployed not only by companies and government institutions, but also by enthusiasts and citizen scientists. Therefore the volume of data produced is extremely large and highly heterogeneous, making it complex to discover and use.

The heterogeneity of data as well as sensing environments is a key obstacle for realizing a *connected* sensor world. Different sensor network deployments usually represent the information that they capture in different ways. The data models and schemas are different, the data types and structures are not always compatible, and even the data values often use different representations. For example, consider multiple sensor networks measuring the same type of physical phenomenon. Each sensor deployment may have its own way to represent semantically identical information, e.g., “wind speed” vs. “average wind speed”, or “temperature” vs.

"thermometer". If a user wants to obtain the latest wind speed or temperature data values over the region where all the sensor networks are deployed, the user must employ a mechanism for letting the system understand the semantically equivalent but different representations of data, in order to fully answer the query.

One of the solutions to deal with heterogeneity is through the semantic annotation of sensor data (Sheth et al., 2008), and the provision of ontology-based access to it (Calbimonte et al., 2010; Taylor & Leidinger, 2011). However, there is a lack of evidence of how this approach scales, especially with high data rates, and in push-based delivery of streaming data.

In this article we focus on two problems in this context: (i) how to find relevant heterogeneous sensor data sources based on their metadata, and (ii) how to query streaming sensor data from these sources. We summarize our contributions as follows:

- Our main contribution to the first problem is the use of the SSN ontology (Compton et al., 2012), along with domain-specific vocabularies, for modeling sensor metadata and observations, augmented with mappings to the original sensor schemas. To this end, we use R2RML[1] (RDB-to-RDF mapping language) for mapping relational streams -instead of tables- to ontologies. Thus we use ontologies as a common model for representing sensor data and metadata, to make it possible to search for data sources and to access them through ontological schemas.

- For the second problem, we propose a query rewriting and data translation approach that allows querying virtual RDF[2] streams using the SPARQL[3] language with streaming extensions. This approach exploits the R2RML mappings to provide access to the sensor streaming data, not only the metadata. Furthermore, we show that our query rewriting and execution mechanisms are applicable for both pull and push delivery modes, and also for various state-of-the-art stream processing engines, such as SNEE (Galpin et al., 2009), GSN (Aberer et al., 2006), Pachube[4], and Esper[5]. We provide empirical evidence of performance with respect to sampling rates and delivery latency in both pull and push-based modes.

As an illustrative example, we show how the GSN implementation of this approach was used in a federated sensor network in the Swiss-Experiment[6] project, a collaborative platform for sharing real-time sensor data across various institutions to improve environmental hazard forecasting and warning.

The paper is organized as follows: first we introduce the fundamental concepts of ontology-based access to streams. then, we discuss the modeling of sensor data, metadata and mappings. Afterwards we provide the theoretical foundations of query translation and how it can be implemented. An experimental evaluation is also presented, before our conclusions.

PRELIMINARIES

Our approach is based on the fundamental concepts detailed below: streaming data querying, and semantic data access using ontology-based queries and mappings.

Queries over Streaming Data

A data stream consists of an unbounded sequence of values continuously appended, each of which carries a timestamp that typically indicates when it has been produced. Examples of data streams include stock market tickers, heart rate measurements or wave height observations from a coastal sensor network. Normally, recent data stream values are more useful and valuable than old ones, since applications are interested in processing the current observations, while historical data is often summarized, aggregated and stored for later analysis.

While traditional one-off (SQL-like) queries are suitable for stored data, streams require continuous long-lived queries that process and yield results as tuples arrive. To cope with these requirements, several continuous query languages have been designed, as well as data stream management systems (DSMSs).

Continuous Query Languages

Continuous query languages, such as CQL (Arasu et al., 2006), SNEEql (Brenninkmeijer et al., 2008) and Esper EPL[5], to name but a few, extend relational query languages with the ability to deal with temporal constraints and the unlimited nature of streams. Most of these languages introduce the concept of a window. A window transforms the unbounded sequence of tuples into a bounded bag of tuples, so that conventional relational operators can be applied. Windows can be specified in terms of rows or time. A time window can be used, for instance, to query only the values registered in the last 10 minutes. For illustrative purposes, we show in Listing 1 how to represent the same query in SNEEql and EPL, that retrieves the wind speed values measured in the latest 10 minutes by a sensor named `windsensor`:

```
SELECT wind_speed FROM windsensor [FROM NOW-10 MINUTES TO NOW]    (1)
SELECT wind_speed FROM wind_sensor.win:time(10 min)               (2)
```

Listing 1. Sample SNEEql(1) and EPL(2) queries.

The window operator is applied to the stream of values coming from the sensor, and the time boundaries are not absolute, but relative to the current time (i.e. `"NOW"` in SNEEql). In this way, the processor is able to execute this query continuously, and to produce resulting tuples as the time passes. The same notion of window is implemented in EPL although with a slightly different syntax.

As aforementioned, most continuous query languages are based on SQL, and differ on their support of different stream operators, their syntax, semantics and the optimizations that they implement for some operators. For instance, they compute windows at different times, evaluate joins differently w.r.t. timestamps, etc. Although important, we will not focus on these differences in this paper.

Data Stream Management Systems

DSMSs have been developed both in research, e.g. Aurora/Borealis (Abadi et al., 2005), STREAM (Arasu et al., 2007), TelegraphCQ (Chandrasekaran et al., 2003), TinyDB (Madden et al., 2005), SNEE (Galpin et al., 2009) and Cougar (Yao & Gehrke, 2002), and also in the

industry (Oracle CEP[7], Sybase CEP[8], StreamBase[9]). Most of them exploit the power of continuous queries like those aforementioned, and they generally provide two types of data access mechanisms for streams: pull and push-based. In pull-based access the client periodically retrieves the data from the DSMS, while in push-based access, it subscribes to a data resource (typically the continuous results of a query) and receives notifications as data becomes available.

Some DSMSs do not use continuous query languages but offer ad-hoc data access APIs. For instance Global Sensor Networks (GSN) (Aberer et al., 2006) provides a Web Service interface and a REST API to pose queries to the sensor data streams. An example of a URL query that requests wind speed values for the latest 10 minutes is given in Listing 2 (assuming the current time is 15/09/2011-15:00:00).

```
http://montblanc.slf.ch:22001/multidata?vs[0]=wind_sensor&field[0]=wind_speed&
from=15/09/2011+05:00:00&to=15/09/2011+15:00:00
```

Listing 2. Sample URL query for GSN.

In this example the GSN server (`montblanc.slf.ch:22001`) exposes a virtual sensor named `wind_sensor` whose values are fed to the GSN system through some data adapter. This URL query is equivalent to the previous ones except that the time window has to be set explicitly. It is also possible to define a query as continuous and set sliding windows, but only through configuration. More complex queries can be built using these URLs, including selection constraints or joins with other sensors and stored tables.

Ontology-based Data Access and Mappings

One of the goals of ontologies is to provide vocabularies and a conceptual model to represent coherently a domain of knowledge. Formal, usable and extensible ontological models with rich semantic descriptions can be used for searching and reasoning with sensor data (Corcho & García-Castro, 2010). Several ontologies have been proposed in the past to model sensor observations and metadata (Compton et al., 2009). However, many of the early approaches focused only on sensor meta-information, overlooking observation descriptions, or did not adhere to standard ontology engineering practices of alignment and reuse.

The emergence of the OGC (Open Geospatial Consortium)[10] standards for sensor and observation descriptions has been crucial for interoperability for sensor services, but as opposed to ontologies, lack explicit semantics. Although some ontology proposals align with the OGC standards (Witt et al., 2008; Russomanno et al., 2005), they were often too domain-specific, difficult to reuse, or only partially covered the spectra of sensor-related information. Other works focused on providing semantic annotations for OGC-compliant sensor services, as in (Babitski et al., 2009) and (Henson et al., 2009).

The Semantic Sensor Network (SSN) ontology (Compton et al., 2012), proposed by the W3C SSN-XG group, is a domain independent model that captures information about sensors and the observations they produce. The SSN ontology is compatible with the OGC standards at both the sensor level (SensorML[11]) and observation level (O&M[12]).

Once the ontology model to be used is selected, we need a mechanism to transform sensor data sources to this ontology-based representation. A body of work has been done in the area of ontology-based data access, aiming at generating semantic web content from existing relational databases (Sahoo et al., 2009). Declarative mapping languages that relate the ontology concepts to the relational schema terms have been designed for this purpose, such as R2O (Barrasa et al., 2004), D2R (Bizer & Cyganiak, 2006) or Virtuoso Meta Schema (Erling & Mikhailov, 2007). The W3C RDB2RDF Group has recently proposed the R2RML[1] mapping language, which can be used either for generating RDF content with massive dumps, or on-demand, for instance by translating SPARQL queries to SQL at execution time.

Some of these mapping languages have been used and extended to support data access not only to static relational databases but also to data streams, as in the case of S2O (Calbimonte et al., 2010). This work continued with the use of the R2RML language instead of S2O, to define mapping relations between ontologies and sensor streams (Calbimonte et al., 2011).

MODELING SENSOR DATA AND METADATA

Our proposal for ontology-based data access to sensor data relies on the use of ontologies, to model sensors and observations, and R2RML mappings, to translate between the raw sensor schemas and those shared ontologies. Sensor networks produce raw and often unstructured data streams, as the example in Listing 3. It shows two sensors (`wan7` and `imis_wfbe`) that have different schemas, although they both measure wind speed.

```
wan7:      {wind_speed_scalar_av FLOAT,timed DATETIME}
imis_wbfe: {vw FLOAT,timed DATETIME}
```

Listing 3. Heterogeneous sensor stream schemas

There is no commonly-agreed metadata that indicates that these sensors measure wind speed, so that if we want to query these observations, we need to know the names of the sensors and the names of all the attributes that correspond to the semantic concept of wind speed. This is an error-prone task that turns unfeasible when the number of sensors is large. Although in previous works (Patni et al., 2010; Barnaghi et al., 2010) sensor observations have been published as RDF and linked data, they do not provide the means to query live sensor data in terms of an ontological model.

Going beyond these approaches, we propose using the SSN ontology to model sensors and observations, combined with mappings to the raw sensor schemas, using the R2RML language. Thus we can query the metadata and also the observations, as we will see in the next sections.

An SSN-based Ontology Network

The SSN ontology (Compton et al., 2012) is a domain independent model, that needs to be extended with specialized domain ontologies, e.g. alpine, coastal or forest environments, defense, health, etc, hence building an ontology network. In Figure 1, we can see the main concepts of the SSN ontology, connected with other domain-specific ontologies that specify types of properties, features, notions of time and geo-location.

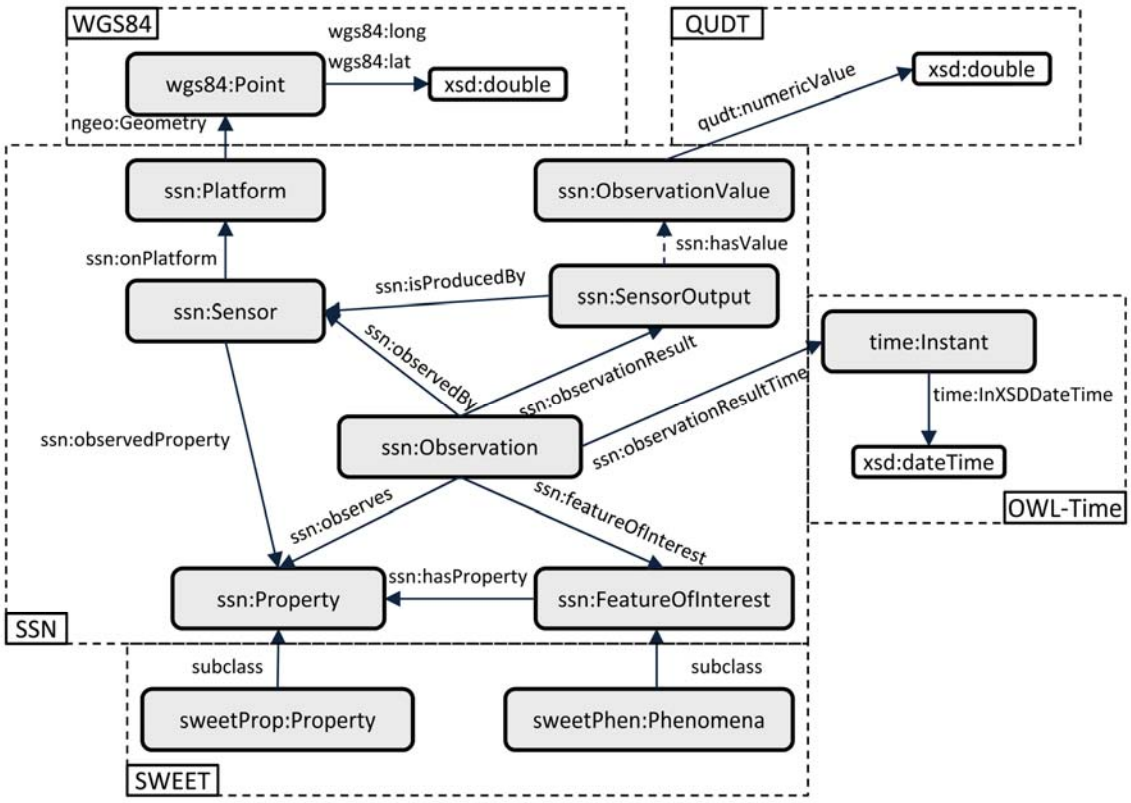

Figure 1. Core concepts of the SSN ontology, combined with other domain ontologies.

For instance, consider a weather station in a mountain site in the Alps that hosts various sensors, including a wind speed monitor and an air temperature sensor. Both of them are sensor instances, in terms of the SSN ontology, but each one observes a different property (speed or temperature) of a different feature of interest (wind or air). In Listing 4 we show an RDF representation of these sensors using the ontology network.

```
swissex:Sensor1
  rdf:type ssn:Sensor;
  ssn:onPlatform swissex:Station1;
  ssn:observes   [rdf:type sweetSpeed:WindSpeed].
swissex:Sensor2
  rdf:type ssn:Sensor;
  ssn:onPlatform swissex:Station1;
  ssn:observes   [rdf:type sweetTemp:Temperature].
swissex:Station1
  ngeo:Geometry [rdf:type wgs84:Point;
                         wgs84:lat "46.8037166";
                         wgs84:long "9.7780305"].
```

Listing 4. SSN ontology sensor descriptions

Each sensor is identified with a URI (e.g. `swissex:Sensor1`, `swissex:Sensor2`) and both of them are attached to the weather station `swissex:Station1`. Location coordinates for the station are also specified, using the WGS84 vocabulary[13]. Each sensor instance is linked through the `ssn:observes` predicate to the property that the device is capable of sensing. Since the SSN model

is intended to be generic, it does not define all possible types of observed properties, but these can be taken from a specialized vocabulary such as the NASA SWEET[14] ontology (e.g. `sweetSpeed:WindSpeed, sweetTemp:Temperature`).

The previous ontology network can also be used to represent actual data measurements, i.e. instances of the Observation class as shown in Listing 5.

```
swissex:WindSpeedObservation1
  rdf:type ssn:Observation;
  ssn:featureOfInterest [rdf:type sweetAtmoWind:Wind];
  ssn:observedProperty  [rdf:type sweetSpeed:WindSpeed];
  ssn:observationResult [rdf:type ssn:SensorOutput;
                         ssn:hasValue [qudt:numericValue "6.245"^^xsd:double]];
  ssn:observationResultTime [time:inXSDDatetime "2001-10-26T21:32:52"^^xsd:dateTime];
  ssn:observedBy swissex:Sensor1 ;
```

Listing 5. SSN ontology observations

Each observation (e.g. `swissex:WindSpeedObservation1`) is linked to a certain feature of interest, e.g. the wind phenomenon at that location (an instance of `sweetAtmoWind:Wind`). Similarly, the `ssn:observedProperty` indicates the property of the feature of interest that has been observed. The observation values captured by the sensor are represented as instances linked to a `ssn:SensorOutput` through the `ssn:hasValue` property. The data values, instances of some data type (e.g. `xsd:double`), can be linked to the observation values using specialized properties from vocabularies that represent quantities (e.g. `qudt:numericValue` from the QUDT[15] ontology).

Mappings to Streams

We can map both the metadata and observations from the original sensor data schemas, to an RDF representation according to the ontology network described above, using the R2RML language. As an example, the following mappings indicate how to generate SSN `Observation` values from the sensor schemas in Listing 3. For every tuple in the `wan7` sensor, an instance of the `ObservationValue` class is created according to the R2RML definition in Figure 2 (see Listing 6, the mappings are expressed themselves in RDF).

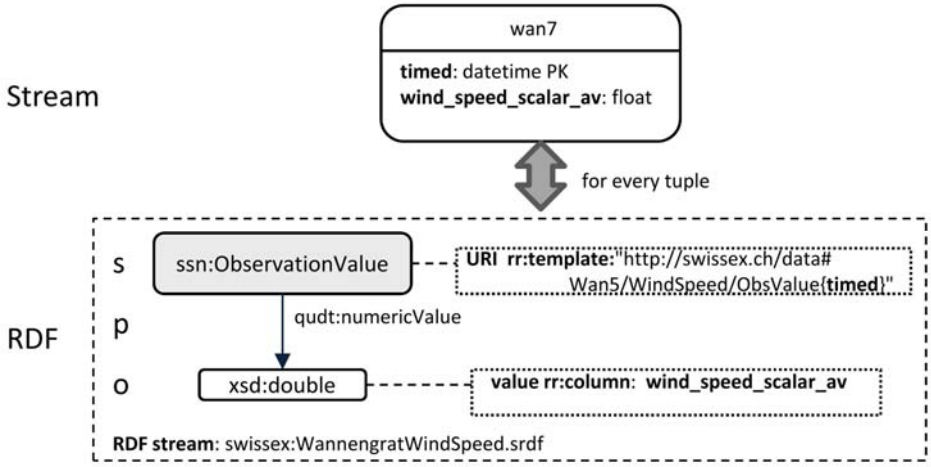

Figure 2. Mapping from the wan7 sensor to a SSN ObservationValue

The mapping definition indicates first from which sensor it will get the data, in this case `wan7`, with the `rr:tableName` property. The triples, each one with a subject, predicate and object, will be generated as follows:

- The subject of all triples will be created according to the `rr:subjectMap` specification. The URI is built using a template (`rr:template` rule), which concatenates a prefix with the value of the `timed` column.
- The subject will be an instance of `ssn:ObservationValue`.
- The triples will belong to the virtual RDF stream `swissex:WannengratWindSpeed.srdf`.
- The predicate of each triple is fixed, in this case `qudt:numericValue`.
- Finally the object will be a `xsd:double`, whose value will be retrieved from the `wind_speed_scalar_av` attribute from the `wan7` stream.

```
:Wan7WindMap a rr:TriplesMapClass;
  rr:tableName "wan7";
  rr:subjectMap [rr:template "http://swissex.ch/data#Wan5/WindSpeed/ObsValue{timed}";
                 rr:class ssn:ObservationValue;
                 rr:graph swissex:WannengratWindSpeed.srdf];
  rr: predicateObjectMap
                 [rr:predicateMap [rr:predicate qudt:numericValue];
                  rr:objectMap    [rr:column "wind_speed_scalar_av"]];.
```

Listing 6. Mapping a sensor to a SSN ObservationValue in R2RML

More triple mappings could be specified in a more complex definition, for example including several properties and object instances, not only data values.

Querying Metadata: a Use Case for the Swiss-Experiment

As we evidenced in the example of Listing 3, sensor streams have their own structure, and may use different vocabularies, making it difficult to consistently find all sources that match the query parameters of a user. With the mapping information as part of the sensor metadata, and using the SSN-based ontology network described previously, we have built a sensor data search prototype (Calbimonte et al., 2011) for the Swiss-Experiment project. The web-based user interface is designed to help users narrow the number of sensors to be queried (Figure 3) through filtering criteria: sensor location, observed properties, temporal extent, etc, enhancing the existing metadata infrastructure (Jeung et al., 2010).

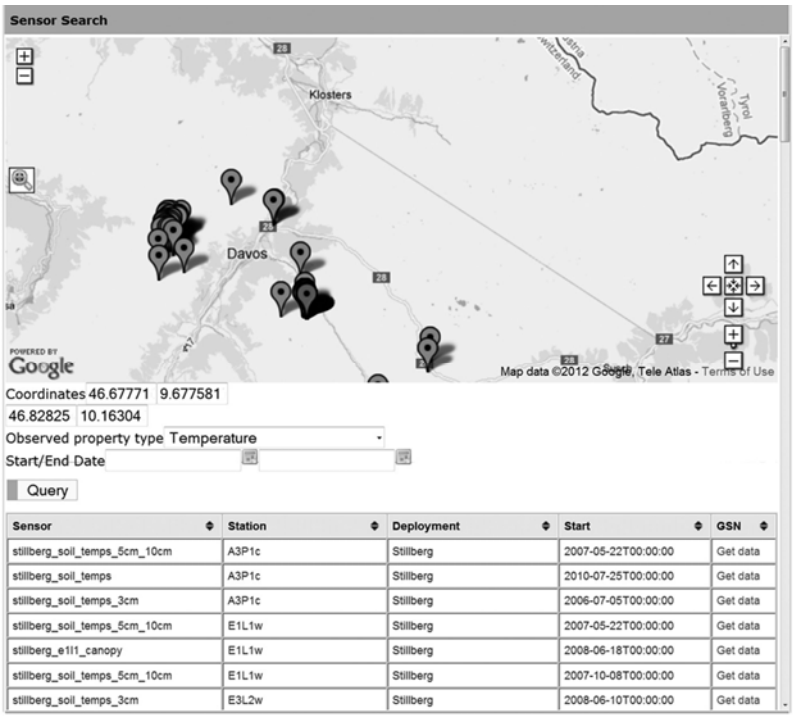

Figure 3. Sensor data search user interface.

A sample query to this repository is shown in Listing 7. It requests sensors active since 2009, for the region specified by a bounding-box, and only for those sensors that measure motion properties. The geo-location query boundaries are specified using the `omgeo:within` function, and can be computed by RDF stores such as OWLIM[16], which uses spatial indexes. Considering that the `MotionProperty` is defined in the SWEET ontology as a superclass of all motion-related classes such as Wind Speed, Acceleration or Velocity, all sensors that capture these properties can also be considered in the query.

```
SELECT DISTINCT ?etime ?stime ?lat ?long ?platformName ?deploymentName
WHERE {
  ?sensor ssn:observes [a sweetProp:MotionProperty];
          ssn:startTime ?stime.
          ssn:endTime ?etime.
  FILTER (xsd:dateTime(?stime)>=xsd:dateTime("2009-01-01T00:00:00")).
  FILTER (xsd:dateTime(?etime)<=xsd:dateTime("9999-01-01T00:00:00")).
  ?system ssn:hasSubSystem ?sensor;
          ssn:onPlatform ?platform;
          ssn:hasDeployment ?deployment.
  ?deployment foaf:name ?deploymentName.
  ?platform dul:hasLocation [ngeo:Geometry ?link];
            foaf:name ?platformName.
  ?link omgeo:within(46.3 8.7 47.2 9.8);
        geo-pos:lat ?lat;
        geo-pos:long ?long. }
```

Listing 7. Querying sensor metadata

ONTOLOGY-BASED QUERY TRANSLATION

Having defined an ontological representation for sensor observations, we will normally be interested in querying not only the metadata (as in the previous use case), but also the data values (the actual measurements), using the mappings described in the previous section. Since data values are accessible through DSMSs, we require *query rewriting* mechanisms to transform our semantic queries (e.g. in SPARQL) to queries in terms of the underlying data models. After executing the queries, a *data translation* process is performed to transform the incoming tuples into triples that will become the final result set.

Our ontology-based sensor query service (Figure 4) receives queries specified in terms of an ontology (e.g. SSN) using $SPARQL_{stream}$ (Calbimonte et al., 2010), an extension of SPARQL that supports operators over RDF streams such as time windows. Since the $SPARQL_{stream}$ query is expressed in terms of the ontology, it has to be transformed into queries in terms of the data sources, using the mappings written in R2RML.

As a result we expose *virtual RDF streams* that can be queried with $SPARQL_{stream}$. The results are triples that are generated from the original data stream, through a translation process. Query rewriting uses the R2RML mappings to produce streaming query expressions over the sensor streams. These are represented as algebra expressions extended with time window constructs, so that logical optimizations (including pushing down projections, selections, and join and union distribution) can be performed over them and can be easily translated to a target language or stream request, such as a REST API, as we will see later.

Query processing is delegated to the DSMS, which ingests the translated query or request built from the algebra expression, and can be performed by explicitly requesting the incoming data from a query (pull) or by subscribing to the new events (triples) that are produced by a continuous query (push). The final step of data translation takes the pulled or pushed tuples from the DSMS and translates them into triples (or tuples, depending on whether it is a CONSTRUCT or SELECT query respectively), which are the final result.

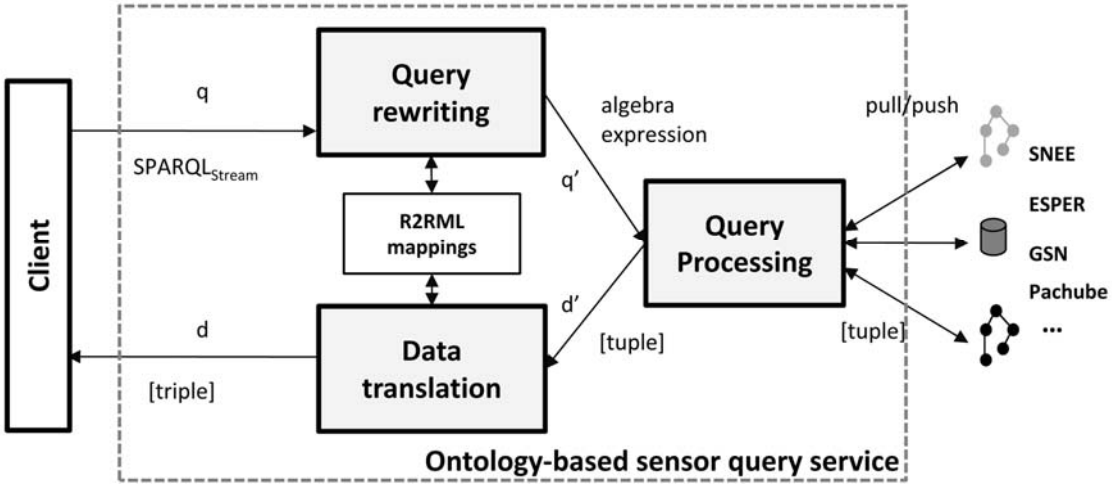

Figure 4. Ontology-based sensor query rewriting, processing and data translation

$SPARQL_{stream}$

SPARQL extensions for streaming data and continuous queries have emerged in proposals such as C-SPARQL (Barbieri et al., 2010) and SPARQL$_{stream}$. Both are based on the idea of using RDF streams, i.e. RDF triples annotated with timestamps. In SPARQL$_{stream}$ these streams are virtual, relying on the original data streams for generating the query results, while C-SPARQL natively manages the RDF stream triples in its data model. Both include time windows for transforming infinite streams of data into bounded sequences to which other standard operators can be applied. Moreover, SPARQL$_{stream}$ considers time windows in the past (upper bound different to the current time) and adheres to the SPARQL 1.1 definition for aggregates.

In SPARQL$_{stream}$ each virtual RDF stream is identified by an IRI that can later be used in a query. Window definitions are of the form `'[Start TO End SLIDE Literal]'`, where the Start and End are of the form `NOW` or `NOW - Literal Unit`, and `Literal` represents the time offset and `Unit` its time unit. The optional `SLIDE` indicates the gap between each successive window evaluation. Windows are applied to named RDF stream graphs with the `STREAM` keyword. Only the triples in that graph are affected by the window operator.

The result of applying a window over a stream is a time-stamped bag of triples over which conjunctions between triple patterns, and other classical operators can be evaluated. Listing 8 shows a SPARQL$_{stream}$ query which returns the last 10 minutes of wind and tide speed measurements, if the wind speed is higher than the value of the tide speed.

```
SELECT ?windspeed ?tidespeed
FROM NAMED STREAM <http://swiss-experiment.ch/data#WannengratSensors.srdf>
[NOW-10 MINUTES TO NOW-0 MINUTES]
WHERE {
  ?WaveObs a ssn:Observation;
           ssn:observationResult ?windspeed;
           ssn:observedProperty sweetSpeed:WindSpeed.
  ?TideObs a ssn:Observation;
           ssn:observationResult ?tidespeed;
           ssn:observedProperty sweetSpeed:TideSpeed.
  FILTER (?tidespeed<?windspeed)}
```

Listing 8. A sample SPARQL$_{stream}$ query

SPARQL$_{stream}$ Semantics

The SPARQL extensions presented here are based on the formalization of (Pérez et al., 2009). An RDF stream S is defined as a sequence of pairs (T,τ) where T is a triple $\langle s_i, p_i, o_i \rangle$ and τ is a timestamp in the infinite set of timestamps $\mathbb{T}$. More formally,

$$S = \left\{ \left(\langle s, p, o \rangle, \tau \right) | \langle s, p, o \rangle \in \left(\left(I \cup B \right) \times I \times \left(I \cup B \cup L \right) \right), \tau \in \mathbb{T} \right\}$$

where I, B and L are sets of IRIs, blank nodes and literals respectively. Each of these pairs can be called a tagged triple. We define a stream of windows as a sequence of pairs (ω,τ) where ω is a set of triples, each of the form $\langle s,p,o \rangle$, and τ is a timestamp that represents when the

window was evaluated. More formally, we define the triples that are contained in a time-based window evaluated at time $\tau \in \mathbb{T}$, denoted ω^{τ}, as

$$\omega^{\tau}_{t_s,t_e,\delta}(S) = \left\{ \langle s,p,o \rangle \mid \left(\langle s,p,o \rangle, \tau_i \right) \in S, t_s \le \tau_i \le t_e \right\}$$

where t_s, t_e define the start and end of the window time range respectively, and may be relative to the evaluation time τ. The rate of window evaluation is controlled by the δ slide.

We have provided a brief explanation of the semantics of $\text{SPARQL}_{\text{stream}}$. However, as the actual data source is not an RDF stream but a sensor network-based or an event-based stream, we also need to transform the $\text{SPARQL}_{\text{stream}}$ queries into requests that can be processed by the corresponding DSMS.

Semantics of the Query Rewriting

In this section we show how we can use the mapping definitions to transform a set of queries over an ontological schema, into expressions that can be serialized as streaming query languages used to access the data sources. This work is based on the formalization work of (Calvanese et al., 2005).

A conjunctive query *q* over an ontology *O* can be expressed as:

$$q\left(\vec{x}\right) \leftarrow \varphi\left(\vec{x}, \vec{y}\right)$$

where $\varphi\left(\vec{x}, \vec{y}\right)$ is a conjunction of atomic classes $C(x)$ and properties $R(x,y)$ in O; with x,y being variables in either $\vec{x}, \vec{y}$ or constants; $\vec{x}$ is a tuple of distinguished variables, and $\vec{y}$ is a tuple of non-distinguished existentially quantified variables. The answer to this query consists of instances of the distinguished variables. For example, query q_1 requests all wind speed observations *x*, captured by wind sensors.:

$$q_1(x) \leftarrow \text{swissex:WindSpeedObservation}(x) \wedge \text{ssn:observedBy}(x, y) \wedge \text{swissex:WindSensor}(y)$$

Now we can define a mapping assertion μ as:

$$C\left(f^{s}_{sub}\right) \wedge \ldots \wedge R_i\left(f^{s}_{sub}, f^{s}_{i}\right) \wedge \ldots \wedge R_j\left(f^{s}_{sub}, f^{\mu'}_{sub}\right) \wedge \ldots \rightarrow s_{\vec{a}}$$

where the left-hand expression is composed of terms of the form $C(x)$, $R(x,y)$, with *C* being a class and *R* a property in *O*. *s* is a logical stream with attributes $\vec{a} = a_1, a_2, \ldots, a_m$, each of them with a certain data type. Notice that this logical stream can be a *view* over several real streams, however in the rest of this discussion we will not cover this case and assume that the logical stream has a number of attributes, and one of them is the timestamp.

The mapping μ explains how to construct an instance of the class *C*, with properties R_i and R_j. The instance is computed with the f^{s}_{sub} function, that generates a URI, taking as a parameter the attributes $\vec{a}$ of the logical stream *s*. The object of the R_i properties is computed with the f^{s}_{i} function, that generates either a URI or a literal value for the property using the attributes of *s* as

input. Alternatively, the objects of a property R_j can be the subject of another mapping μ', denoted with $f_{sub}^{\mu'}$.

A time-windowed query over the ontology O, $q_O[t_s, t_e, \delta]$ (where t_s and t_e are the window boundaries and δ is the slide), is a conjunction of atoms $C(x)$, $R(x,y)$ as described before, but with a window that limits the elements to those contained within its boundaries. To translate this query in terms of the streams s_i, we find the terms of the query in each of the mapping definitions μ, and then build an algebra expression in terms of the streams that are defined in each of those mappings. For example given the following query:

$$q_O[t_s, t_e, \delta] = (C_1(x) \wedge R_1(x, y) \wedge C_2(y))[t_s, t_e, \delta]$$

And mappings:

$$\mu_1 : C_1\left(f_{sub}^{s1}\right) \wedge R_1\left(f_{sub}^{s1}, f_{sub}^{\mu_2}\right) \rightarrow s_1^{\vec{a}}$$

$$\mu_2 : C_2\left(f_{sub}^{s2}\right) \wedge R_2\left(f_{sub}^{s2}, f_{R_2}^{s2}\right) \rightarrow s_2^{\vec{b}}$$

The goal is to find all terms of the query in the mappings and build an expression in terms of $s_1^{\vec{a}}$ and $s_2^{\vec{b}}$. First we create projections for the attributes of the streams that are in the mappings that match a term. These are partial matches as they cover only a part of the query. If the query had a term that cannot be found in the mappings, then there is no possible translation and thus no results are produced. If more than one mapping matches a term, all the corresponding projections are merged in a union (if only one stream matches there is no need for a union). Then all these partial matches are merged in a join to complete all terms of the query expression.

For this example the algebra expression can be constructed as follows (Figure 5): the first two terms of the query are matched to μ_1 and the third is matched to μ_2. Therefore we project the attributes of the streams of μ_1 and μ_2: s_1 and s_2, previously applying the window (ω) operator to both streams. These partial matches are finally merged in a join. In this case no union is necessary since there are no overlaps between mappings and query terms.

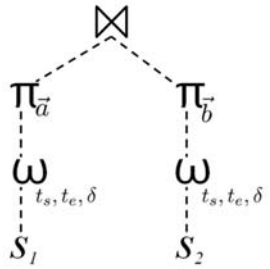

Figure 5. Constructed algebra expression

Notice that in this case the window is applied to the whole query, but it could be restricted to only some terms of it instead. Projection of only the necessary attributes is also applied during the construction of the algebra expression. The addition of filters, which are translated into

selection operators, is not explicitly discussed in this paper, although its inclusion is trivial and was implemented in the prototypes.

In the presence of other mapping definitions, the algebra expression will be modified. For example if we add a new mapping $\mu_3 : C_1\left(f_{sub}^{s3}\right) \wedge R_1\left(f_{sub}^{s3}, f_{sub}^{\mu_2}\right) \rightarrow s_3^{\vec{c}}$, it also matches the first two terms of the query so it is merged in union with the stream of the mapping μ_1. The algebra expression is modified as follows (Figure 6):

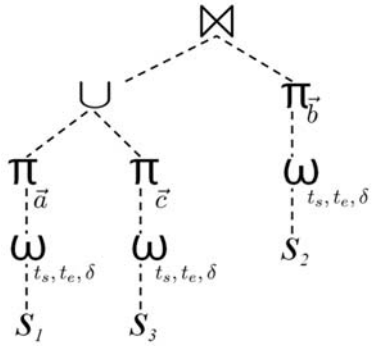

Figure 6. Modified algebra expression

Notice that standard query optimization techniques can be used to transform the query expression to an equivalent one, which can be executed more efficiently later when the query is delegated to the stream processing engine. For instance the join in the example can be distributed to the terms of the union.

Query Processing

In the previous section we described the query rewriting process, which generates algebra expressions that are generic and can be serialized as query languages. While in previous works the processing was limited to specific platforms (Calbimonte et al., 2010), we have now implemented adapters for systems with different characteristics, namely SNEE, GSN, Pachube and Esper, showing the generality of our approach.

As an example, we will consider the $\text{SPARQL}_{\text{stream}}$ query in Listing 8, which incorporates time windows, filters and two different observations. Suppose that we have mappings that relate the SSN-based ontology network concepts to the streams `wan7` and `wan6` of Listing 9.

```
wan7: {wind_speed_scalar_av FLOAT,timed DATETIME}
wan6: {tide_speed FLOAT,timed DATETIME}
```

Listing 9. Wind and tide speed sensor stream schemas

We show a sample mapping for the `wan7` sensor in Figure 7. This mapping generates not only the `ObservationValue` instance but also a `SensorOutput` and an `Observation` for each record of the sensor `wan7`. Notice that each of these instances constructs its URI with a different template rule and the `Observation` has a `observedProperty` property to the `sweetSpeed:WindSpeed` property. Suppose a similar mapping for `wan6`, only that has `sweetSpeed:TideSpeed` as observed property.

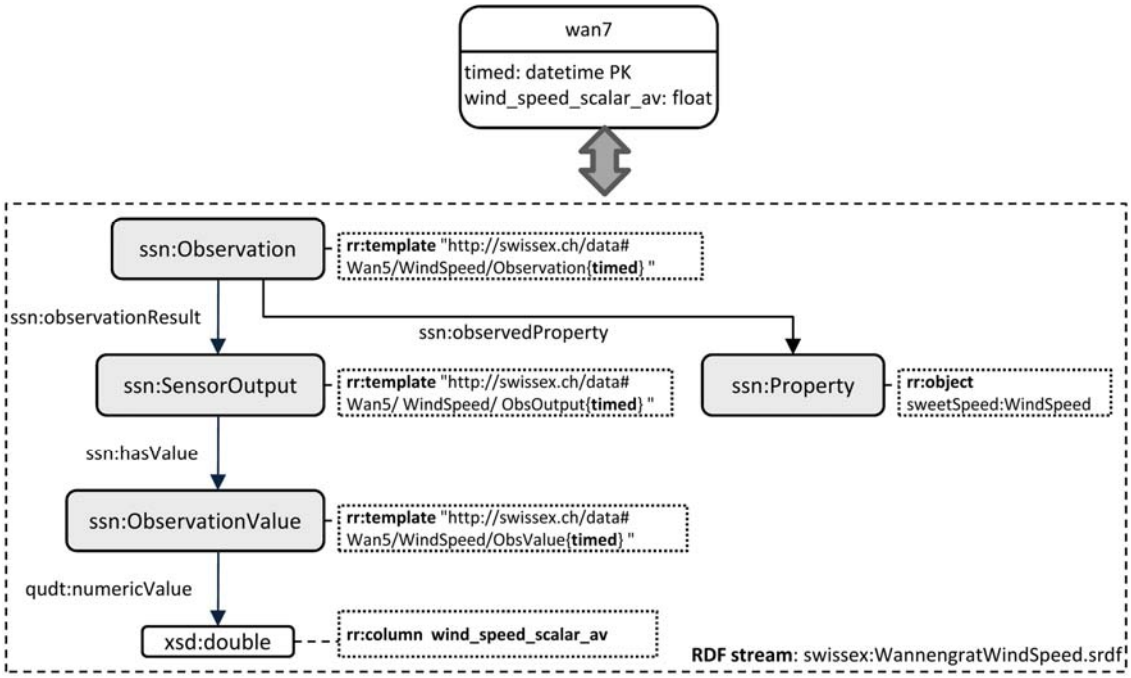

Figure 7. R2RML mapping for the wan7 sensor.

The query translation process will use these mappings to generate an algebra representation. As the query terms are matched against the mapping definitions, both sensors are included in the expression, first applying the window to both sensors, and projecting the required fields to build the URIs and values. Then both are merged in a join, as they are both part of the query, but the join includes the condition that compares the values of the wave and tide speed. The result is depicted in Figure 8. This can be later serialized into a query and executed by a query engine.

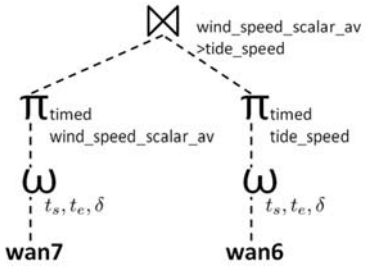

Figure 8. Algebra expression after query rewriting

The query engines may accept query languages or requests through APIs, and in both cases are straightforward to represent as the expressions discussed above. In the following subsections we discuss four implementations of this approach: the DSMS SNEE, the sensor web middleware platforms GSN and Pachube, and the complex-event processor Esper.

SNEE

SNEE (Galpin et al., 2009) is a streaming data query engine able to integrate stored relational sources and capable of in-network query evaluation, delegating parts of the query plan to the

sensor nodes. It uses the SNEEql language, which has a well defined semantics for queries over event streams, acquisitional streams and stored data.

Constructing SNEEql sentences from the algebra expressions we showed in the previous sections is straightforward. For example, the query in Listing 10 is produced for the expression in Figure 8:

```
SELECT wan7.wind_speed_scalar_av AS windspeed, wan7.timed AS windts,
       wan6.tide_speed AS tidespeed,wan6.timed as tidets
FROM wan7[FROM NOW-10 MINUTES TO NOW], wan6[FROM NOW-10 MINUTES TO NOW]
WHERE wan7.wind_speed_scalar_av>wan6.tide_speed
```

Listing 10. SNEEql translated query

Although SNEE is interesting from the point of view of its ability to perform in-network query processing, some features like union of windows or joins between streams are only partially supported in the current prototype.

Global Sensor Networks (GSN)

GSN is a sensor data middleware that supports flexible integration of sensor networks and can be deployed in a federated environment. The GSN server instances can be queried through web-services or RESTful URL interfaces. The ontology-based sensor query processor can generate GSN API[17] URLs from the algebra expressions, which are executed by the GSN server.

Back to our example in Figure 8, the GSN URL API does not support joins between streams, unless there is a virtual sensor that already joins them (complex queries can be defined with virtual sensors in GSN through configuration). Therefore the query is not translatable, but it can be split into two simpler queries and then join the results. We show one of this simpler $SPARQL_{stream}$ queries (Listing 11) and its translation to a GSN URL (

Listing 12). We used this implementation in the Swiss-Experiment use case described previously.

```
SELECT ?windspeed
FROM NAMED STREAM <http://swiss-experiment.ch/data#WannengratSensors.srdf>
[NOW-10 MINUTE TO NOW-0 MINUTE]
WHERE {
  ?WaveObs a ssn:Observation;
           ssn:observationResult ?windspeed;
           ssn:observedProperty sweetSpeed:WindSpeed.
}
```

Listing 11. Simplified SPARQLstream query

```
http://montblanc.slf.ch:22001/multidata?vs[0]=wan7&
field[0]=wind_speed_scalar_av&
from=15/05/2011+05:00:00&to=15/05/2011+15:00:00
```

Listing 12 Generation of a GSN API URL

Pachube

While GSN is used in several projects and research initiatives, other wide-open sensor data systems are emerging, such as Pachube, which offers data management of real-time data from sensors. The data hosted in Pachube is organized as tagged environments or feeds, each one having one or more datastreams, which represent an individual measuring device, and the actual values, called datapoints.

Pachube data can be queried through a RESTful API, although the complexity of these queries is low, compared to those in GSN. For instance, it is not possible to perform joins nor selections or aggregates, but it remains an interesting data source for open and large-scale use. The API allows retrieving the latest datapoints of a certain datastream, or the datapoints in a time interval specified as part of the request. For example the following request: `http://api.pachube.com/v2/feeds/14321`, returns data from the environment with id=14321, including the list of its data streams. A time-based query can be specified for a particular datastream, like in the following example:

```
http://api.pachube.com/v2/feeds/14321/datastreams/4?start=2011-09-02T14:01:46Z&end=2011-09-02T17:01:46Z
```

In this case the datastream has an id=4, and the time boundaries are given by the start and end parameters. To query these Pachube streams with our approach, we specify the environment id as the stream name in the R2RML mapping, and the datastream id as an attribute name.

Esper

Esper is a commercial event processing engine that supports streaming data and continuous queries. It provides a rich declarative query language, EPL, with support for a number of streaming data operators, including time windows. Although the query syntax is slightly different from SNEEql or CQL, the ideas are similar. For instance, Listing 13 is the translated EPL query for the Listing 11 expression.

```
SELECT wind_speed_scalar_av, timed FROM wan7.win:time(10 min)
```

Listing 13. EPL translated query

One of the features of Esper is that it supports both pull and push based delivery of query results. While all the previous implementations we explored dealt only with pull mechanisms, we implemented a push adapter for Esper. In fact the query rewriting phase as such does not change at all (only the serialization to the EPL syntax has to be handled, as mentioned above), but it is mainly the data translation phase that changes. Each time that Esper notifies about a new event, this data is translated to tuples (or triples) and a new event is raised to the subscriber, containing the new tuples (or triples) as argument.

EXPERIMENTATION

We have provided implementations for four different systems with diverse characteristics and designed for different purposes. For instance, SNEE is capable of executing joins with stored data, but does not support push delivery. Pachube has a wide range of available data streams,

although is limited in query expressivity. GSN offers more query operators, but is also limited, for instance in the case of joins between streams. Esper offers event pattern matching, but does not allow union operators and joins in pull mode.

It is not our goal in this paper to evaluate our approach with all these systems and to compare them exhaustively, given their large heterogeneity. Instead we will focus on the evaluation of the main characteristic of our system: the query rewriting and data translation steps that we add to the processing stack.

Therefore, we analyze the overhead added by these steps, so as to assess their potential impact in real-world scenarios. In order to do that we have evaluated both the pull and push based delivery mechanisms, with Esper as DSMS, since it is mature and stable, and provides both delivery modes. Since our framework delegates the query processing to the DSMS, we did not cover query complexity in the evaluation, hence we limited the tests to simple queries. All these tests have been performed on an Intel Core i7 1.60 GHz, 6 GB.

Rewriting and Translation Overhead in Pull delivery

In this experiment our objective is to assess the overhead caused by the query rewriting and data translation steps, during pull-based queries. We evaluated the response time to pull requests every 100 milliseconds, using a simple $\text{SPARQL}_{\text{stream}}$ query equivalent to the one in Listing 11. The system was loaded with 30 pre-configured synthetic streams, fed at the specified rate by a tuple generator (1 to 1000 tuples/s). We compared these results to pull requests to the equivalent EPL query, using Esper's API directly, without query rewriting nor data translation steps. Since the query is translated only once, the real interest of this experiment is to verify the overhead of the data translation process. We experimented with three different time windows (1, 10 and 30 seconds), because the time boundaries significantly affect the number of tuples that are retrieved and translated to SPARQL results. As we can see in Figure 9, the executions with data translation have a significant overhead which is more noticeable as the tuple rate increases. As expected, the translation overhead depends directly of the number of tuples that the query is handling, and this may depend either on the tuple rate or the time window. Nevertheless, even for the relatively high rates we obtained acceptable response times.

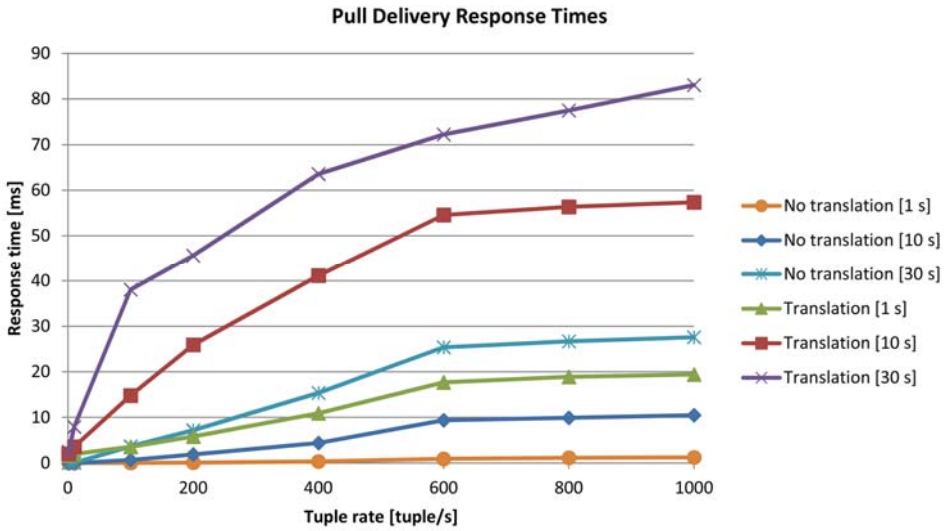

Figure 9. Pull response times for different tuple rates and windows

Rewriting and Translation Overhead in Push delivery

Esper provides its own benchmark for performance evaluation in push delivery, which is free to be used and modified. We focused on the end-to-end latency of the generated tuples, which is featured in this benchmark, comparing the results of executing Esper EPL queries without our framework and then with the query and data translation mechanism in place. For both cases we experimented with 100 to 8000 data values per second, and we plotted the results in Figure 10 and Figure 11 respectively, grouping the messages by latency ranges (as indicated in the Esper benchmark). As expected, we see higher end-to-end latency for executions with our translation mechanism. For instance for 600 tuples/s, the original version has almost all messages under the line of 1 ms of latency. On the contrary, our implementation has most results between 1 and 5 ms. Considering that query translation is performed only once, most of this penalty comes from the data translation process, which could be optimized for push delivery, for instance by processing data in batches. Even with these limitations, for low and medium throughput requirements (e.g. 1-100 tuples/s), such as the case of the Swiss-Experiment environmental sensors, this component is comparable to the version without translation.

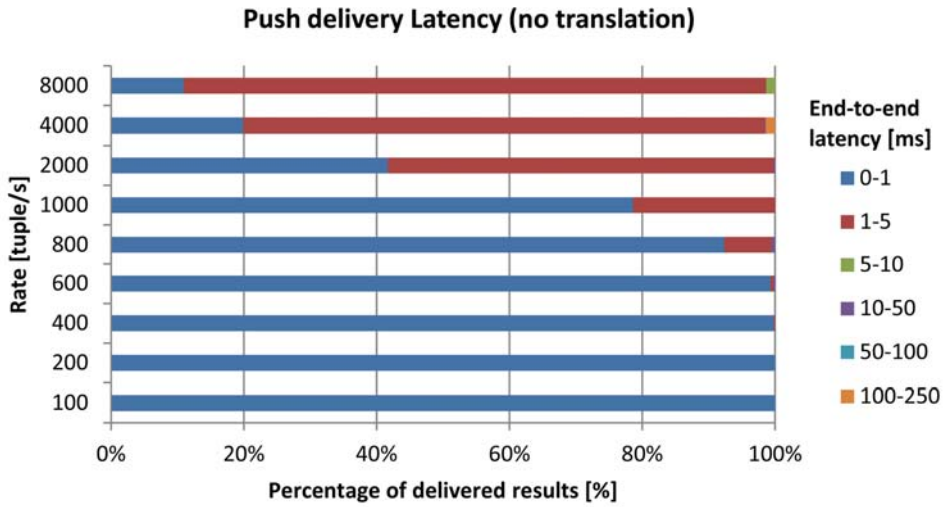

Figure 10. Push-delivery latency, without query/data translation

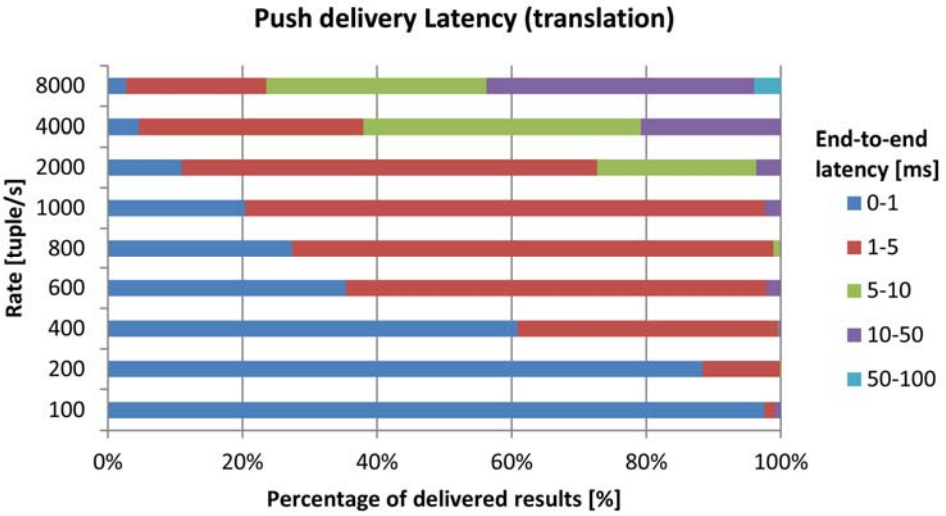

Figure 11. Push-delivery latency, with query/data translation.

CONCLUSIONS

In this paper, we have presented our approach for representing and querying sensor data through ontological models, while internally managing the data with streaming or event processing engines using push and pull-based models, depending on the characteristics of the underlying implementation. Our objective with this approach is to hide the heterogeneity of data schemas of different sensor deployments by using a common model (e.g. an ontology network with the SSN ontology at the core). The translation of queries and data results from ontologies to algebra expressions has been formalized and implemented using the R2RML mapping language, extending its use from relational databases to data streams (this is the first implementation in this direction).

Our approach has been implemented using four stream and sensor management systems, each one with different goals and querying capabilities, showing that these principles can be applied to a potentially wide range of situations. Moreover, we have implemented both pull and push based delivery modes, the latter being an addition to previous efforts. Finally, we have provided experimental evidence of feasibility and reasonable performance, for sensors with medium-low rates, which are common in real environmental deployments such as the Swiss Experiment.

We are planning to analyze and add more expressivity to the $SPARQL_{stream}$ queries that can be rewritten, for instance, to support linking the results to existing static RDF graphs hosted in a remote endpoint and providing optimizations for these join operations. Following this direction we can contribute to the combination of sensors and Linked Data, in the lines of (Wei & Barnaghi, 2009) and (Le-Phuoc et al., 2010). We also aim at optimizing the data translation process, which incurs in an important overhead in some scenarios, especially in those cases where queries are changing more dynamically. Finally, we will consider the case of data integration of different sensor data sources, where ontology-based queries can help solving semantic heterogeneity, also from the point of view of data quality.

Acknowledgements

This work is supported by the myBigData project (TIN2010-17060) funded by MICINN, and the European project PlanetData (FP7-257641).

REFERENCES

Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A. S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., & Zdonik, S. (2005). The Design of the Borealis Stream Processing Engine. In *Proc. 2nd Conference on Innovative Database Research* (pp. 277–289).

Aberer, K., Hauswirth, M., & Salehi, A. (2006). A middleware for fast and flexible sensor network deployment. In *Proc. 32nd International Conference on Very Large Databases* (pp. 1199–1202).

Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., & Widom, J. (2007). STREAM: The Stanford data stream management system. In M. Garofalakis, J. Gehrke, & R. Rastogi (Eds.), *Data Stream Management*. Springer.

Arasu, A., Babu, S., & Widom, J. (2006). The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2), 121–142.

Babitski, G., Bergweiler, S., Hoffmann, J., Schön, D., Stasch, C., & Walkowski, A. (2009). Ontology-based integration of sensor web services in disaster management. In *Proc. 3rd International Conference on GeoSpatial Semantics* (pp. 103–121).

Barbieri, D. F., Braga, D., Ceri, S., Della Valle, E., & Grossniklaus, M. (2010). C-SPARQL: A continuous query language for RDF data streams. *International Journal of Semantic Computing*, 4(1), 3–25.

Barnaghi, P., Presser, M., & Moessner, K. (2010). Publishing linked sensor data. In *Proc. 3nd International Workshop on Semantic Sensor Networks*.

Barrasa, J., Corcho, O., & Gómez-Pérez, A. (2004). R2O, an extensible and semantically based database-to-ontology mapping language. In *Proc. 2nd Workshop on Semantic Web and Databases*.

Bizer, C. & Cyganiak, R. (2006). D2R server - publishing relational databases on the semantic web. In *Poster Proc. 5th International Semantic Web Conference*.

Brenninkmeijer, C. Y., Galpin, I., Fernandes, A. A., & Paton, N. W. (2008). A semantics for a query language over sensors, streams and relations. In *Proc. 25th British National Conference on Databases* (pp. 87–99).

Calbimonte, J.-P., Corcho, O., & Gray, A. J. G. (2010). Enabling ontology-based access to streaming data sources. In *Proc. 9th International Semantic Web Conference* (pp. 96–111).

Calbimonte, J.-P., Jeung, H., Corcho, O., & Aberer, K. (2011). Semantic sensor data search in a large-scale federated sensor network. In *Proc. 4th International Workshop on Semantic Sensor Networks* (pp. 14–29).

Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., & Rosati, R. (2005). DL-Lite: Tractable description logics for ontologies. In *Proc. 20th National Conference on Artificial Intelligence* (pp. 602–607).

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S. R., Reiss, F., & Shah, M. A. (2003). TelegraphCQ: continuous dataflow processing. In *Proc. 22th ACM SIGMOD International Conference on Management of Data* (pp. 668–668).

Compton, M., Barnaghi, P., Bermudez, L., García-Castro, R., Corcho, O., Cox, S., Graybeal, J., Hauswirth, M., Henson, C., Herzog, A., Huang, V., Janowicz, K., Kelsey, W. D., Phuoc, D. L., Lefort, L., Leggieri, M., Neuhaus, H., Nikolov, A., Page, K., Passant, A., Sheth, A., & Taylor, K. (2012). The SSN ontology of the W3C semantic sensor network incubator group. *Journal of Web Semantics*, (In press).

Compton, M., Henson, C., Lefort, L., Neuhaus, H., & Sheth, A. (2009). A survey of the semantic specification of sensors. In *Proc. 2nd International Workshop on Semantic Sensor Networks* (pp.17).

Corcho, O. & García-Castro, R. (2010). Five challenges for the Semantic Sensor Web. *Semantic Web*, 1(1), 121–125.

Erling, O. & Mikhailov, I. (2007). RDF support in the Virtuoso DBMS. In *Proc. 1st Conference on Social Semantic Web*, volume 113 of *LNI* (pp. 59–68).: GI.

Galpin, I., Brenninkmeijer, C. Y., Jabeen, F., Fernandes, A. A., & Paton, N. W. (2009). Comprehensive optimization of declarative sensor network queries. In *Proc. 21st International Conference on Scientific and Statistical Database Management* (pp. 339–360).

Henson, C., Pschorr, J., Sheth, A., & Thirunarayan, K. (2009). SemSOS: Semantic Sensor Observation Service. In *Proc. 2009 International Symposium on Collaborative Technologies and Systems* (pp. 44–53).

Jeung, H., Sarni, S., Paparrizos, I., Sathe, S., Aberer, K., Dawes, N., Papaioannou, T., & Lehning, M. (2010). Effective Metadata Management in Federated Sensor Networks. In *Proc.*

3rd International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (pp. 107–114).

Le-Phuoc, D., Parreira, J., Hausenblas, M., Han, Y., & Hauswirth, M. (2010). Live linked open sensor database. In *Proc. 6th International Conference on Semantic Systems* (pp. 1–4).

Madden, S. R., Franklin, M. J., Hellerstein, J. M., & Hong, W. (2005). TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1), 122–173.

Patni, H., Henson, C., & Sheth, A. (2010). Linked sensor data. In *Proc. 2010 International Symposium on Collaborative Technologies and Systems* (pp. 362–370).: IEEE.

Pérez, J., Arenas, M., & Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 1–45.

Russomanno, D., Kothari, C., & Thomas, O. (2005). Sensor ontologies: from shallow to deep models. In *Proc. 37th Southeastern Symposium on System Theory* (pp. 107–112).

Sahoo, S. S., Halb, W., Hellmann, S., Idehen, K., Jr, T. T., Auer, S., Sequeda, J., & Ezzat, A. (2009). *A Survey of Current Approaches for Mapping of Relational Databases to RDF.* Technical report, W3C RDB2RDF Incubator Group.

Sheth, A., Henson, C., & Sahoo, S. (2008). Semantic sensor web. *IEEE Internet Computing*, 12(4), 78–83.

Taylor, K. & Leidinger, L. (2011). Ontology-driven complex event processing in heterogeneous sensor networks. In *Proc. 8th Extended Semantic Web Conference* (pp. 285–299).

Wei, W. & Barnaghi, P. (2009). Semantic annotation and reasoning for sensor data. In *Proc. 4th European Conference on Smart Sensing and Context* (pp. 66–76).

Witt, K., Stanley, J., Smithbauer, D., Mandl, D., Ly, V., Underbrink, A., & Metheny, M. (2008). Enabling sensor webs by utilizing SWAMO for autonomous operations. In *Proc. 8th Eighth Annual NASA Earth Science Technology Conference.*

Yao, Y. & Gehrke, J. (2002). The Cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3), 9–18.

[1] R2RML mapping language, http://www.w3.org/2001/sw/rdb2rdf/r2rml/
[2] RDF Resource Description Framework, http://www.w3.org/RDF/
[3] SPARQL Protocol and RDF Query Language, www.w3.org/TR/rdf-sparql-query
[4] Pachube, https://pachube.com/
[5] Esper and Event Processing Language EPL, http://esper.codehaus.org/
[6] Swiss-Experiment, http://www.swiss-experiment.ch/
[7] Oracle Complex Event Processing, http://www.oracle.com/technetwork/middleware/complex-event-processing/
[8] Sybase Complex Event Processing, http://www.sybase.com/products/financialservicessolutions/complex-event-processing
[9] StreamBase, http://www.streambase.com/
[10] Open Geospatial Consortium, http://www.opengeospatial.org/
[11] OGC SensorML, http://www.opengeospatial.org/standards/sensorml
[12] Observations & Measurements, http://www.opengeospatial.org/standards/om

[13] Basic Geo-WGS84 Vocabulary, http://www.w3.org/2003/01/geo/
[14] NASA SWEET Ontology, http://sweet.jpl.nasa.gov/
[15] Quantities, Units, Dimensions and Data Types ontologies, http://www.qudt.org/
[16] OWLIM, http://www.ontotext.com/owlim
[17] GSN Web URL-API, http://sourceforge.net/apps/trac/gsn/wiki/web-interfacev1-server