Querying the Real World through Stream-to-Ontology Mappings

Jean-Paul Calbimonte[a,*], Oscar Corcho[a], Alasdair J. G. Gray[b]

[a] *Ontology Engineering Group, Departamento de Inteligencia Artificial, Facultad de Informática, Universidad Politécnica de Madrid, Campus de Montegancedo s/n 28660, Boadilla del Monte, Spain*
[b] *School of Computer Science, The University of Manchester, Oxford Road, Manchester M13 9PL, United Kingdom*

Abstract

Ubiquitous and inexpensive networks of data capturing devices are being developed and deployed nowadays, increasing the availability of streaming data sources for a number of applications and domains. The heterogeneity of these sources introduces the requirement of providing data access and query capabilities in a unified and coherent manner, whilst allowing users to express their needs at a conceptual level, independent of implementation and language-specific details. In this paper we describe the theoretical foundations and the implementation of an ontology-based streaming data access approach for overcoming this problem. First, sources link their data content to existing ontologies through S_2O declarative mappings. Then, users can query over these ontologies using $\text{SPARQL}_{\text{Stream}}$, an extension of SPARQL for streaming data, and the translation algorithms generate the necessary queries for the underlying streaming data source.

Keywords: Streaming query processing, Ontology-based data access, Semantic query translation

1. Introduction

In recent years, advances in wireless sensor technologies and communications have opened the possibility of deploying large-scale and ubiquitous sensor networks capable of live data capture, processing and delivery.

Deployments of sensor networks are expected to increase significantly in the upcoming years because of their advantages and unique features. Tiny and inexpensive devices can be installed virtually anywhere, execute elementary computations and still be reachable thanks to wireless communications. Therefore these devices can be used for a wide variety of applications including security surveillance, healthcare provision, traffic control and environmental monitoring. In this context several challenges are raised for the research community, and the one we are concerned with is about accessing and querying the enormous amount of loosely structured data collected by these sensors. Client applications require a suitable platform to query data from sensors, streaming query engines or databases, in terms of a uniform schema that hides the diverse and internal data representations of each source.

As an example, consider a web application which aids an emergency planner to detect and co-ordinate the response to flood risk alerts in the coast of South East England. This involves retrieving relevant data from multiple sources, e.g. meteorological forecasts from the Met Office (UK's National Weather Service)[1], real-time wave and tide data from sensor networks deployed in the region by the CCO (Channel Coastal

*Corresponding author

Email addresses: jp.calbimonte@upm.es (Jean-Paul Calbimonte), ocorcho@fi.upm.es (Oscar Corcho), a.gray@cs.man.ac.uk (Alasdair J. G. Gray)

[1] http://www.metoffice.gov.uk accessed 15 September 2010.

Observatory)[2], and any other relevant sources of data such as the NFCDD (Environment Agency's[3] National Flood and Coastal Defence Database) providing data about coastal defences. Typically sources are managed autonomously and model their data according to the needs of the deployment. To integrate the data requires linking the sources to a common data model so that conditions that are likely to cause a flood can be detected using prediction models, and presented to the user in terms of their domain, e.g. flood risk assessment.

We propose that ontologies can be used as such a common model as it has been done in the past for other types of data sources (e.g. relational [1]) . For the scenario presented here, we use a network of ontologies that includes and extends ontologies from SWEET[4] and the W3C incubator group's semantic sensor network ontology. [5]

The work presented in this paper considers advances done by the semantic web and database communities over the last decade. On the one hand, the semantic web research has produced mapping languages and software for enabling ontology-based access to stored data sources, e.g. R_2O [2] and D2RQ [3]. These systems provide semantic access to traditional (stored) data sources by providing mappings between the elements in the relational and ontological models [4]. However, similar solutions for streaming data mapping and querying using ontology-based approaches have not been explored yet.

On the other hand, the database research community have investigated data stream processing where the data is viewed as an append-only sequence of tuples. Systems such as STREAM [5] and Borealis [6] have focused on query evaluation and optimization over streams with high, variable, data rates. Other systems such as SNEE [7] and TinyDB [8], have focused on data generated by sensor networks, which tends to be at a lower rate, and query processing in the sensor network where resources are more constrained and energy efficiency is the primary concern. There have also been proposals for query processing over streaming RDF data [9, 10]. However there is still no bridging solution that connects these technologies coherently in order to answer the requirements of

- i) establishing mappings between ontological models and streaming data source schemas.
- ii) accessing streaming data sources through queries over ontology models.

In this paper we focus on providing ontology-based access to streaming data sources, including sensor networks, through declarative continuous queries. We build on the existing work of R_2O for enabling ontology-based access to relational data sources, and SNEE for query evaluation over streaming and stored data sources. This constitutes a first step towards a framework for the integration of distributed heterogeneous streaming and stored data sources through ontological models. In Section 2 we provide more detailed descriptions of R_2O and stream query processing in order to present the foundations of our approach in Section 3. In Section 4 we present the syntactic extensions for SPARQL to enable queries over RDF streams, and present S_2O for stream-to-ontology mappings. The semantics of these extensions are detailed in Section 5 and a first implementation of the execution of the streaming data access approach is explained in Section 6. Related work is discussed in Section 7 and our conclusions in Section 8.

2. Preliminaries

This section describes the existing work upon which our approach is based, i.e. ontology-based data access and querying through mappings, and data access for streaming data. A full discussion of related work can be found in Section 7.

[2] `http://www.channelcoast.org/` accessed 15 September 2010.

[3] `http://www.environment-agency.gov.uk/` accessed 15 September 2010.

[4] `http://sweet.jpl.nasa.gov/` accessed 15 September 2010.

[5] `http://www.w3.org/2005/Incubator/ssn/wiki/Semantic_Sensor_Network_Ontology` accessed 15 September 2010.

2.1. Ontology-based Access to Stored Relational Data

The goal of ontology-based data access is to generate semantic web content from existing relational data sources available on the web [4]. The objective of systems following this approach is to allow users to construct queries over an ontology (e.g. using a query language as SPARQL), which are then rewritten into a set of queries expressed in the query language of the data source (typically SQL), according to the specified mappings. The query results are then converted back from the relational format into RDF, which is returned to the user. ODEMAPSTER is one such system which uses the R_2O (Relational-to-Ontology) language to express the mappings between the relational data source and the ontology [2].

The mapping definition language R_2O defines relationships between a set of ontologies and relational schemas [2]. R_2O specifically considers classes, object and datatype properties in an ontology. They are described in terms of selections and transformations over database relations following a Global-as-View (GAV) approach [11], and can be created either manually or with the help of a mapping tool[6]. R_2O covers a wide set of mapping cases common in relational database to ontology situations:

- A database table maps to one class in the ontology. Then the table columns map to attributes or relations of the concept. For each row in the table a corresponding instance in the ontology will be generated, with its attribute values filled with the columns data.
- A single database table is mapped to more than one class in the ontology, and for each row a single instance of each class is generated.
- A single database table is mapped to more than one class in the ontology, and multiple instances can be generated for each class. It is a more general case than the previous one; multiple instances of the same ontology concept can be generated from a single database record.

Mapping relations to ontologies often requires performing operations on the relational sources. Several cases are handled by R_2O and detailed below.

Direct Mapping. A single relation maps to an ontology class and the attributes of the relation are used to fill the property values of the ontology instances. Each row in the relation will generate a class instance in the ontology.

Join/Union. A single relation does not correspond alone to a class, but it has to be combined with other relations. The result of the join or union of the relations will generate the corresponding ontology instances.

Projection. Not all the attributes of a relation are always required for the mapping. The unnecessary attributes can simply be ignored. In order to do so, a projection on the needed attributes can be performed.

Selection. Not all rows of a relation correspond to instances of the mapped ontology class. A subset of the rows must be extracted. To do so, selection conditions can be applied to choose the desired subset for the mapping.

It is possible to combine joins, unions, projections and selections for more complex mapping definitions. R_2O also enables the application of functions, e.g. concatenation, sub-string, or arithmetic functions, to transform the relational data into the appropriate form for the ontology.

2.2. Relational Data Streams

A relational data stream is an append only, potentially infinite, sequence of timestamped tuples [12], examples of which include stock market tickers, heart rate monitors, and sensor networks deployed to monitor the environment. Data streams can be classified into two categories:

[6] `http://www.neon-toolkit.org/wiki/ODEMapster` accessed 29 September 2010.

Event-streams. A tuple is generated each time an event occurs, e.g. the sale of shares, and can have variable, potentially very high, data rates.

Acquisitional-streams. A tuple is measured at a predefined regular interval, e.g. the readings made by a sensor network.

Users are typically interested in being informed continuously about the most recent stream values, with older tuples being less relevant. Classical database query processing is not adequate since data must first be stored and then queried with one-off evaluation. Hence, query languages [13, 14] and data stream management systems (DSMS) [5, 6, 7, 8] have been developed to process continuous long-lived queries over data streams as tuples arrive.

One existing approach is SNEEql, which has a well defined, unified semantics for declarative expressions of data needs over event-streams, acquisitional-streams, and stored data [13]. SNEEql can be viewed as extending SQL for processing data streams. The main additional constructs of relational streaming query languages are explained below and we present them using the SNEEql syntax.

Window. A window over a data stream transforms the infinite sequence of tuples into a bounded bag of tuples over which traditional relational operators can be applied. A window is specified as:

FROM *start* TO *end* [SLIDE *int unit*]

Where *start* and *end* are of the form '$\mathsf{NOW}-$*literal*' and define the range of the window with respect to the evaluation time. The optional SLIDE parameter specifies how often windows are evaluated.

Window-to-Stream. Window-to-stream operators are used to convert a stream of windows into a stream of tuples. SNEEql supports three such operators: RSTREAM for all tuples appearing in the window, ISTREAM for tuples that have been added since the last window evaluation, and DSTREAM for tuples that have been deleted since the last window evaluation.

3. Ontology-based Streaming Data Access

Our approach to enable ontology-based access to streaming data is depicted in Fig 1. The service receives queries specified in terms of the classes and properties[7] of the ontology using $\text{SPARQL}_{\text{Stream}}$, an extension of SPARQL that supports operators over RDF streams (see Section 4.1). In order to transform the $\text{SPARQL}_{\text{Stream}}$ query, expressed in terms of the ontology, into queries in terms of the data sources, a set of mappings must be specified. These mappings are expressed in S_2O, an extension of the R_2O mapping language, which supports streaming queries and data, most notably window and stream operators (see Section 4.2). This transformation process is called *query translation*, and the target is the continuous query language SNEEql, which is expressive enough to deal with both streaming and stored sources.

After the continuous query has been generated, the query processing phase starts, and the evaluator uses distributed query processing techniques [15] to extract the relevant data from the sources and perform the required query processing, e.g. selection, projection, and joins. Note that query execution in sources such as sensor networks may include in-network query processing, pull or push based delivery of data between sources, and other data source specific settings. The result of the query processing is a set of tuples that the *data translation* process transforms into ontology instances.

This approach requires several contributions and extensions to the existing technologies for continuous data querying, ontology-based data access, and SPARQL query processing. This paper focuses on a first stage that includes the process of transforming the $\text{SPARQL}_{\text{Stream}}$ queries into queries over the streaming data sources using SNEEql as the target language. The following

[7] We use the OWL nomenclature of classes, and object and datatype properties for naming ontology elements.

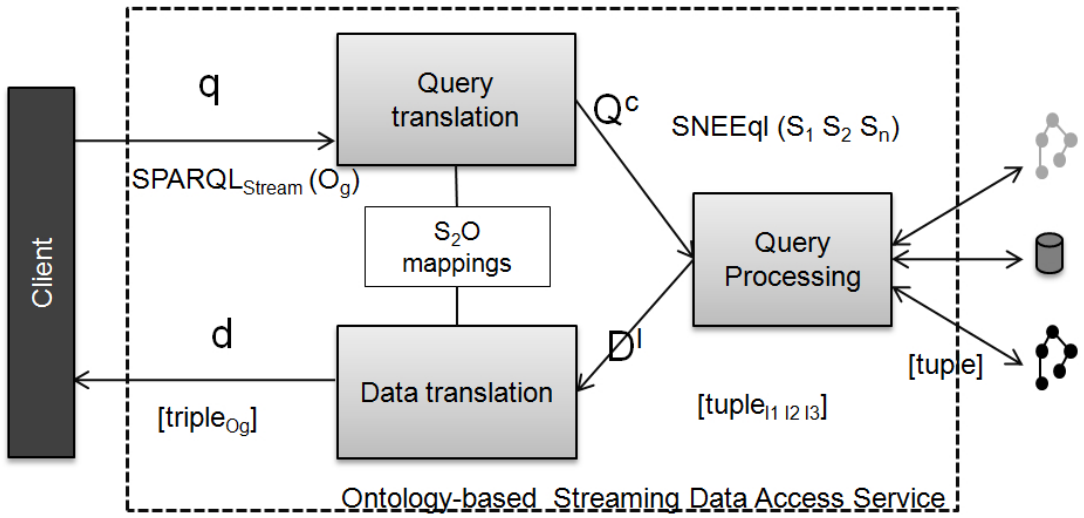

Figure 1: Ontology-based streaming data access service

sections provide the syntax and semantics for the querying of streaming RDF data and the mappings between streaming sources and an ontology. We will then provide details of an actual implementation of this approach.

4. Query and Mapping Syntax

In this section we introduce the $\text{SPARQL}_{\text{Stream}}$ query language, an extension to SPARQL for streaming RDF data, which has been inspired by previous proposals such as C-SPARQL [10] and SNEEql [13]. However, significant improvements have been made that correct the types supported and the semantics of windowing operations, which can be summarized as: (i) we only support windows defined in time, (ii) the result of a window operation is a window of triples, not a stream, over which traditional operators can be applied, as such we have added window-to-stream operators, and (iii) we have adopted the SPARQL 1.1 definition for aggregates. We also present S_2O for the definition of stream-to-ontology mappings.

4.1. $SPARQL_{Stream}$

Just as in C-SPARQL we define an RDF *stream* as a sequence of pairs (T_i, τ_i) where T_i is an RDF triple $\langle s_i, p_i, o_i \rangle$ and τ_i is a timestamp which comes from a monotonically non-decreasing sequence. An RDF stream is identified by an IRI, which provides the location of the data source[8].

Window definitions are of the form '`FROM` *Start* `TO` *End* [`STEP`] [*Literal*]', where the *Start* and *End* are of the form `NOW` or `NOW` – *Literal*, and *Literal* represents some number of time unit (`DAYS`, `HOURS`, `MINUTES`, or `SECONDS`)[9]. The optional `STEP` indicates the gap between each successive window evaluation. Note, if the size of the step is smaller than the range of the window, then the windows will overlap, if it coincides with the size of the window then every triple will appear in one and only one window, and if the step is larger than the range then the windows *sample* the stream. Also note that the definition of a window can be completely in the past. This is useful for correlating current values on a stream with values that have previously occurred.

The result of applying a window over a stream is a timestamped bag of triples over which conjunctions between triple patterns, and other "classical" operators can be evaluated. Windows can be converted back into a stream of triples by applying one of the window-to-stream operators in the `SELECT` clause: `ISTREAM` for returning all

[8]Note in our work the IRI's identify virtual RDF streams since they are derived from the streaming data sources.

[9]Note that the parser will also accept the non-plural form of the time units and is not case sensitive.

```
PREFIX fire: <http://www.semsorgrid4env.eu#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-
    syntax-ns#>
SELECT RSTREAM ?WindSpeedAvg
FROM STREAM <www.semsorgrid4env.eu/SensorReadings
    .srdf> [FROM NOW - 10 MINUTES TO NOW STEP 1
    MINUTE]
FROM STREAM <www.semsorgrid4env.eu/
    SensorArchiveReadings.srdf> [FROM NOW - 3
    HOURS TO NOW -2 HOURS STEP 1 MINUTE]
WHERE {
  {
    SELECT AVG(?speed) AS ?WindSpeedAvg
    WHERE
    {
        GRAPH <www.semsorgrid4env.eu/
            SensorReadings.srdf> {
        ?WindSpeed a fire:WindSpeedObservation;
            fire:hasSpeed ?speed; }
    } GROUP BY ?WindSpeed
  }
  {
    SELECT AVG(?archivedSpeed) AS
       ?WindSpeedHistoryAvg
    WHERE
    {
      GRAPH <www.semsorgrid4env.eu/
          SensorArchiveReadings.srdf> {
        ?ArchWindSpeed a fire:
            WindSpeedObservation;
        fire:hasSpeed ?archivedSpeed; }
    } GROUP BY ?ArchWindSpeed
  }
  FILTER (?WindSpeedAvg > ?WindSpeedHistoryAvg)
}
```

Listing 1: An example $\text{SPARQL}_{\text{Stream}}$ query which every minute computes the average wind speed measurement for each sensor over the last 10 minutes if it is higher than the average of the last 2 to 3 hours.

newly inserted answers since the last window, DSTREAM for returning all deleted answers since the last window, and RSTREAM for returning all answers in the window. Listing 1 shows a complete $\text{SPARQL}_{\text{Stream}}$ query which, every minute, returns the average of the last 10 minutes of wind speed measurements for each sensor, if it is higher than the average speed from 2 to 3 hours ago.

Note, $\text{SPARQL}_{\text{Stream}}$ only supports time-based windows. Other similar languages, such as C-SPARQL also have the notion of triple-based windows. However, such windows are problematic to define in an approach where RDF triples are generated on the fly, since the number of triples required to generate an answer may be greater than the size of the triple window. For example, consider a window size of 1 triple and the graph pattern from the example query in Listing 1. Only one of the triples that form the graph pattern would be kept by the window, which provides insufficient information to compute the expected query answer.

4.2. S_2O*: Defining Stream-to-Ontology Mappings*

The mapping document that describes how to transform the data source elements to ontology elements is written in the S_2O mapping language, an extended version of R_2O [2]. An R_2O mapping document includes a section that describes the database relations, dbscehma-desc. In order to support data streams, R_2O has been extended to also describe the data stream schema. A new component called streamschema-desc has been created, as shown in the top part of Listing 2.

The description of a relational stream is similar to a relation in terms of the R_2O language. An additional attribute streamType has been added, it denotes the kind of stream in terms of data acquisition, i.e. event or acquisitional, as this will have an impact on the underlying implementation. In the same way as key and non-key attributes are defined, a new timestamp-desc element has been added to provide support for declaring the stream timestamp attribute. Since S_2O extends R_2O, relations can also be specified using the existing R_2O mechanism. For the class and property mappings, the existing R_2O definitions can be used for stream schemas just as it was for relational schemas. This is specified in the conceptmap-def element and the attributemap-def elements, as shown in the bottom part of Listing 2.

In addition, although they are not explicitly mapped, the timestamp attribute of stream tuples could be used in some of the mapping definitions, for instance in the URI construction (uri-as element). Finally, a $\text{SPARQL}_{\text{Stream}}$ streaming query requires an RDF stream to have an IRI identifier. For this, S_2O creates a *virtual* RDF stream and its IRI is specified in the S_2O mapping using the virtualStream element. It can be specified at the conceptmap-def level or at the attributemap-def level.

The S_2O grammar can be found in Appendix A.

```
streamschema-desc
  name MeteoSensors
  has-stream SensorWind
    streamType push
    documentation "Wind measurements"
    keycol-desc sensorId
      columnType integer
    timestamp-desc  timestamp
      columnType datetime
    nonkeycol-desc  speed
      columnType float
    nonkeycol-desc direction
      columnType float

conceptmap-def WindSpeedObservation
  virtualStream <http://semsorgrid4env.eu/
      SensorReadings.srdf>
  uri-as
    concat(SensorWind.sensorId)
  described-by
    attributemap-def observationResult
      virtualStream http://semsorgrid4env.eu/
          SensorReadings.srdf>
      operation constant
        has-column SensorWind.speed
    attributemap-def observationResultTime
      virtualStream http://semsorgrid4env.eu/
          SensorReadings.srdf>
      operation constant
        has-column SensorWind.timestamp
```

Listing 2: An example S2O declaration of a data stream schema and mapping from a stream schema to an ontology concept.

5. Semantics of the Streaming Extensions

Now that the syntax of SPARQL$_{\text{Stream}}$ and S2O have been presented, we define their semantics.

5.1. SPARQL$_{Stream}$ Semantics

The SPARQL extensions presented here are based on the formalization of Pérez *et al.* [16]. An RDF stream S is defined as a sequence of pairs (T, τ) where T is a triple $\langle s, p, o\rangle$ and τ is a timestamp in the infinite set of timestamps $\mathbb{T}$. More formally,

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

where I, B and L are sets of IRIs, blank nodes and literals. 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 in the infinite set of timestamps $\mathbb{T}$, and 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

$$\omega^\tau_{t_s,t_e,\delta}(S) = \{\langle s, p, o\rangle \mid (\langle s, p, o\rangle, \tau_i) \in S, t_s \leq \tau_i \leq t_e\}$$

where t_s, t_e define the start and end of the window time range respectively, and may be defined relative to the evaluation time τ. Note that the rate at which windows get evaluated is controlled by the STEP defined in the query, which is denoted by δ.

We define the three window-to-stream operators as

$$\begin{aligned}
\mathsf{RStream}_{(\omega^\tau,\tau)} &= \{(\langle s, p, o\rangle, \tau) \mid \langle s, p, o\rangle \in \omega^\tau\} \\
\mathsf{IStream}_{(\omega^\tau,\tau),(\omega^{\tau-\delta},\tau-\delta)} &= \{(\langle s, p, o\rangle, \tau) \mid \langle s, p, o\rangle \in \omega^\tau, \langle s, p, o\rangle \notin \omega^{\tau-\delta}\} \\
\mathsf{DStream}_{(\omega^\tau,\tau),(\omega^{\tau-\delta},\tau-\delta)} &= \{(\langle s, p, o\rangle, \tau) \mid \langle s, p, o\rangle \notin \omega^\tau, \langle s, p, o\rangle \in \omega^{\tau-\delta}\}
\end{aligned}$$

where δ is the time interval between window evaluations. Note that RStream does not depend on the previous window evaluation, whereas both IStream and DStream depend on the contents of the previous window.

We have provided a brief explanation of the semantics of SPARQL$_{\text{Stream}}$. This is particularly useful in the sense that users may know what to expect when they issue a query using these new operators. However, as the actual data source is not an RDF stream but a sensor network or an event-based stream, e.g. exposed as a SNEEql endpoint, we need to transform the SPARQL$_{\text{Stream}}$ queries into SNEEql queries. The next section describes the semantics of the transformation from SPARQL$_{\text{Stream}}$ to SNEEql using the S2O mappings.

5.2. S2O Semantics

In this section we will present how we can use the S2O mapping definitions to transform a set of conjunctive queries over an ontological schema, into the streaming query language SNEEql that is used to access the sources. This work is based on extensions to the ODEMAPSTER processor [2] and

the formalization work of Calvanese *et al.* [17] and Poggi *et al.* [18].

A conjunctive query q over an ontology $\mathcal{O}$ can be expressed as:

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

$$\varphi(\vec{x}, \vec{y}) : \bigwedge_{i=1...k} P_i, \text{ with } P_i \begin{cases} C_i(x), C_i \text{ is an atomic class.} \\ R_i(x,y), R_i \text{ is an atomic property.} \\ x=y \end{cases}$$

x, y are variables in $\vec{x}, \vec{y}$ or constants.

where $\vec{x}$ is a tuple of distinct distinguished variables, and $\vec{y}$ a tuple of non-distinguished existentially quantified variables. The answer to this query consists in the instantiation of the distinguished variables [17]. For instance consider:

$$\begin{aligned} q_1(x) \leftarrow & WindSpeedObservation(x) \wedge \\ & isProducedBy(x, y) \wedge \\ & WindSensor(y) \end{aligned}$$

It requires all instances x that are wind speed measurements captured by wind sensors. In this example x is a distinguished variable and y a non-distinguished one. The query has three atoms: $WindSpeedObservation(x)$, $isProducedBy(x, y)$, and $WindSensor(y)$.

Concerning the formal definition of the query answering, let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be an interpretation, where $\Delta^{\mathcal{I}}$ is the interpretation domain and $\cdot^{\mathcal{I}}$ the interpretation function that assigns an element of $\Delta^{\mathcal{I}}$ to each constant, a subset of $\Delta^{\mathcal{I}}$ to each class and a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ to each property of the ontology. Given a query $q(\vec{x}) \leftarrow \varphi(\vec{x}, \vec{y})$ the answer to q is the set $q^{\mathcal{I}}_{\vec{x}}$ of tuples $\vec{c} \in \Delta^{\mathcal{I}} \times \cdots \times \Delta^{\mathcal{I}}$ that substituted to $\vec{x}$, make the formula $\exists \vec{y}.\varphi(\vec{x}, \vec{y})$ true in $\mathcal{I}$ [17, 18, 19]. Now we can introduce the definition of the mappings. Let $\mathcal{M}$ be a set of mapping assertions of the form:

$$\Psi \rightsquigarrow \Phi$$

where Ψ is a conjunctive query over the global ontology $\mathcal{O}$, formed by terms of the form $C(x), R(x, y), A(x, z)$, with C, R, and A being classes, object properties and datatype properties respectively in $\mathcal{O}$; x, y being object instance variables, and z being a datatype variable. Φ is a set of expressions that can be translated to queries in the target continuous language (e.g. SNEEql) over the sources.

A mapping assertion $C(f^{Id}_C(\vec{x})) \rightsquigarrow \Phi_{S_1,...,S_n}(\vec{x})$ describes how to construct the concept C from the source streams (or relations) $S_1, \ldots, S_n$. The function f^{Id}_C creates an instance of the class C, given the tuple $\vec{x}$ of variables returned by the Φ expression. More specifically this function will construct the instance identifier (URI) from a set of attributes from the streams and relations. In this case the expression Φ has a declarative representation of the form:

$$\Phi_{S_1,...,S_n}(\vec{x}) = \exists \vec{y}.p^{Proj}_{S_1,...,S_n}(\vec{x}) \wedge p^{Join}_{S_1,...,S_n}(\vec{v}) \wedge p^{Sel}_{S_1,...,S_n}(\vec{v})$$

where $\vec{v}$ is a tuple of variables in either $\vec{x}, \vec{y}$. The term p^{Join} denotes a set of join conditions over the streams and relations S_i. Similarly the term p^{Sel} represents a set of condition predicates over the variables $\vec{v}$ in the streams S_i (e.g. conditions using $<, \leq, \geq, >, =$ operators).

A mapping assertion $R(f^{Id}_{C_1}(\vec{x_1}), f^{Id}_{C_2}(\vec{x_2})) \rightsquigarrow \Phi_{S_1,...,S_n}(\vec{x_1}, \vec{x_2})$ describes how to construct instances of the object property R from the source streams and relations S_i. The declarative form of Φ is:

$$\begin{aligned} \Phi_{S_1,...,S_n}(\vec{x_1}, \vec{x_2}) = & \exists \vec{y}.\Phi_{S_1,...,S_k}(\vec{x_1}) \\ & \wedge \Phi_{S_{k+1},...,S_n}(\vec{x_2}) \wedge p^{Join}_{S_1,...,S_n}(\vec{v}) \end{aligned}$$

where $\Phi_{S_1,...,S_k}, \Phi_{S_{k+1},...,S_n}$ describe how to extract instances of C_1 and C_2 from the streams $S_1, \ldots, S_k$ and $S_{k+1}, \ldots, S_n$ respectively. The term p^{Join} is the set of predicates that denotes the join between the streams and relations $S_1, \ldots, S_n$.

Finally an expression $A(f^{Id}_C(\vec{x}), f^{Trf}_A(\vec{z})) \rightsquigarrow \Phi_{S_1,...,S_n}(\vec{x}, \vec{z})$ describes how to construct instances of the datatype property A from the source streams and relations $S_1, \ldots, S_n$. The function f^{Trf}_A executes any transformation over the tuple of variables $\vec{z}$ to obtain the property value (e.g. arithmetic operations, or string operations). The declarative form of Φ in this case is:

$$\begin{aligned} \Phi_{S_1,...,S_n}(\vec{x}, \vec{z}) = & \exists \vec{y}.\Phi_{S_1,...,S_k}(\vec{x}) \\ & \wedge \Phi_{S_{k+1},...,S_n}(\vec{z}) \wedge p^{Join}_{S_1,...,S_n}(\vec{v}) \end{aligned}$$

The definition follows the same idea as the previous one. The variables of $\vec{z}$ will contain the actual values that will be used to construct the datatype property value using the function f_A^{Trf}.

When a conjunctive query is issued against the global ontology, the processor first parses it and transforms it into an abstract syntax tree and then uses the expansion algorithm described in [2] (that is based on the PerfectRef algorithm of [17]) to produce an expanded conjunctive query based on the TBox of the ontology. Afterwards the rewritten query can be translated to an extended relational algebra.

A query $Q_{\mathcal{O}}(\vec{x})[t_s, t_e, \delta]$ is a conjunctive query with a window operator (where t_s, t_e are the start and end points of the window range and δ is the slide) in order to narrow the data set according to a given criteria. For a query:

$$Q_{\mathcal{O}}(\vec{x})[t_s, t_e, \delta] = (C_1(x) \wedge R(x, y) \wedge A(x, z))[t_s, t_e, \delta]$$

the translation is given by $\lambda(\Phi)$, following the mapping definition:

$$\lambda(\Phi_{S_1,\ldots,S_n}(\vec{x})_{[t_s,t_e,\delta]}) = \pi_{p^{Proj}}(\bowtie_{p^{Join}} (\sigma_{p^{Sel}}(\omega_{t_s,t_e,\delta}S_1), \ldots , \sigma_{p^{Sel}}(\omega_{t_s,t_e,\delta}S_n)))$$

The expression denotes first a window operation $\omega_{t_s,t_e,\delta}$ over the relations or streams $S_1, \ldots, S_n$, with t_s, t_e, and δ being the time range and slide. A selection $\sigma_{p^{Sel}}$ is applied over the result, according to the conditions defined in the mapping. A multi-way join $\bowtie_{p^{Join}}$ is then applied to the selection, also based on the corresponding mapping definition. Finally a projection $\pi_{p^{Proj}}$ is applied over the results. For any conjunctive query with more atoms, the construction of the algebra expression will follow the same direct translation using the GAV approach.

6. Implementation and Execution: Walkthrough

The presented approach of providing ontology-based access to streaming data has been implemented as an extension to the ODEMAPSTER processor [2]. This implementation generates SNEEql queries that can be executed by the streaming query processor.

Consider the motivating example where a sensor network generates two streams `WindFolkestone` and `WindHernebay` of wind sensor measurements. The associated stored information about the sensors, e.g. location and type, are stored in a relation `Sensors`.

```
WindFolkestone: (sensorId INT PK,
        timestamp DATETIME PK,
        speed FLOAT,
        direction FLOAT)
WindHernebay: (sensorId INT PK,
        timestamp DATETIME PK,
        speed FLOAT,
        direction FLOAT)
Sensors: (sensorId INT PK,
        sensorName CHAR(45),
        lat FLOAT,
        long FLOAT)
```

Listing 3: Relational schema of the stream data source.

The schemas are presented in Listing 7. Also consider the following ontological view:

$$\begin{aligned}
SpeedObservation &\sqsubseteq Observation \\
WindSpeedObservation &\sqsubseteq SpeedObservation \\
WindDirectionObservation &\sqsubseteq Observation \\
SpeedObservation &\sqsubseteq \exists observationResult \\
SpeedObservation &\sqsubseteq \exists observationResultTime \\
Observation &\sqsubseteq \exists isProducedBy.Sensor \\
Sensor &\sqsubseteq \exists hasLatitude \\
Sensor &\sqsubseteq \exists hasLongitude
\end{aligned}$$

We can define an S$_2$O mapping that unifies the `WindFolkestone` and `WindHernebay` stream tuples into instances of a *WindSpeedObservation* concept. Listing 4 is an extract of the S$_2$O mapping document concerning the *WindSpeedObservation*. The mapping extract defines how to construct the *WindSpeedObservation* and *Sensor* class instances from the streams `WindFolkestone` and `WindHernebay` and the `Sensors` table:

$$\Psi_{WindSpeedObservation} \rightsquigarrow \Phi_{\substack{\texttt{WindFolkestone} \\ \texttt{WindHernebay}}}$$

$$\Psi_{Sensor} \rightsquigarrow \Phi_{\texttt{Sensors}}$$

It needs to perform a union operation over both streams, as specified in the mapping

```
conceptmap-def WindSpeedObservation
  virtualStream <http://semsorgrid4env.eu/
      SensorReadings.srdf>
  uri-as
      concat('http://semsorgrid4env.eu/
          WindSpeedObservation_', SensorWind.
          sensorId,   SensorWind.timestamp)
  union
    name SensorWind
    has-stream WindFolkestone
    has-stream WindHernebay
  described-by
    attributemap-def observationResult
      operation constant
        has-column SensorWind.speed
    attributemap-def observationResultTime
      operation constant
        has-column SensorWind.timestamp
    dbrelationmap-def isProducedBy
      toConcept Sensor
      joins-via
        condition equals
          has-column Sensors.sensorId
          has-column SensorWind.sensorId

conceptmap-def Sensor
  uri-as
    concat('http://semsorgrid4env.eu/Sensor_',
        Sensors.sensorId)
  described-by
    attributemap-def hasLatitude
      operation constant
        has-column Sensors.lat
    attributemap-def hasLongitude
      operation constant
        has-column Sensors.lon
```

Listing 4: S_2O mapping from the data streams `WindFolkestone` and `WindHernebay` to the ontology concepts *WindSpeedObservation*.

document, and assign an alias name to this union (`SensorWind`). In the case of the *WindSpeedObservation* the function $f^{Id}_{WindSpeedObservation}$ produces the URI's of the instances by concatenating the `sensorId` and `timestamp` attributes. Now we can pose a query over the ontology using SPARQL$_{\text{Stream}}$, for example to obtain the wind speed measurements taken in the last 10 minutes (See the query in Listing 5).

A class query atom $WindSpeedObservation(x)$ and the $observationResultTime(x,t)$ and $observationResult(x,z)$ datatype property atoms can be extracted from the SPARQL$_{\text{Stream}}$ query.
The window specification:

$$[t_s = \mathsf{NOW} - 10, t_e = \mathsf{NOW}, \delta = 1]$$

```
PREFIX fire: <http://www.ssg4env.eu/
    fireObservation#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-
    syntax-ns#>
SELECT RSTREAM ?speed ?time ?lat ?lon
FROM STREAM <www.ssg4env.eu/SensorReadings.srdf>
    [FROM NOW - 10 MINUTES TO NOW STEP 1 MINUTE]
WHERE {
    ?WindSpeed a fire:WindSpeedObservation;
        fire:observationResult ?speed;
        fire:observationResultTime ?time;
        fire:isProducedBy ?Sensor.
    ?Sensor a fire:Sensor;
        fire:hasLatitute ?lat;
        fire:hasLonitude ?lon.
}
```

Listing 5: SPARQL$_{\text{Stream}}$ query which every minute returns the wind speed for the last ten minutes.

is also obtained[10]. The S_2O mapping defines that *WindSpeedObservation* instances are generated based on the `sensorId` and `timestamp` attributes of the `WindFolkestone` and `WindHernebay` streams, using a concatenation function to generate each instance URI. Similarly the S_2O mapping defines that *observationResult* properties are generated from the values of the `speed` attribute of the streams.
The processor will evaluate this as:

$$\lambda(\Phi_{\substack{\texttt{WindFolkestone}\\ \texttt{WindHernebay}}} \begin{pmatrix} x_{\texttt{sensorId}} \\ x_{\texttt{timestamp}} \\ z_{\texttt{speed}} \end{pmatrix} [now{-}10,now,1]) =$$
$$\pi_{\substack{\texttt{sensorId}\\ \texttt{timestamp}\\ \texttt{speed}}} \left(\cup \begin{pmatrix} \omega_{now-10,now,1}(\texttt{WindFolkestone}) \\ \omega_{now-10,now,1}(\texttt{WindHernebay}) \end{pmatrix} \right)$$

Similarly the evaluator will compute the *Sensor* instances needed for the join that is produced in order to generate the *isProducedBy* object property. The query generated in the SNEEql language is shown in Listing 6. The relational answer stream that results from evaluating the query in Listing 6 are transformed by the *Data Transformation* module depicted in Figure 1 according to the S_2O mappings. This results in a stream of tagged triples which are instances of the class *WindSpeedObservation*. According to the select statement of the SPARQL$_{\text{Stream}}$ query, the result in this case will be a set of bound variables.

[10]For the simplicity of presentation, we assume that the system rewrites all time specifications to minutes. The implemented system uses milliseconds as the common time unit.

```
SELECT RSTREAM concat('http://ssg4env.eu#
    WindSpeedObservation_',
        WindSensor.sensorId,WindSensor.timestamp)
             AS id,
        WindSensor.speed AS speed, Sensors.lat,
            Sensors.lon
FROM (
        SELECT sensorId, timestamp
        FROM WindFolkestone [FROM NOW - 10
            MINUTES TO NOW SLIDE 1 MINUTE]
        UNION
        SELECT sensorId, timestamp
        FROM WindFolkestone [FROM NOW - 10
            MINUTES TO NOW SLIDE 1 MINUTE]
) AS WindSensor, Sensors
WHERE WindSensor.sensorId = Sensors.sensorId;
```

Listing 6: The SNEEql query that is generated for the input query in Listing 5.

7. Related Work

In this section we provide details about related work on the areas of relational-to-RDF data access, streaming data querying and RDF streams.

*7.1. Relational-To-*RDF *Data Access*

The realization of the Semantic Web vision, where data is available, understandable and processable by computers, has launched several initiatives that aim at providing semantic access to traditional data sources. Most stored data in the web is currently preserved in relational databases and it has therefore become a need to generate Semantic Web content from them [4]. In this context there is a considerably large amount of research in the community, with the goal of exposing data in terms of ontologies that formally express a domain of interest [18]. This is the goal of Ontology-based data access (OBDA). As aforementioned, most of the existing approaches are based on the exploitation of mappings between the relational (rows and columns) and the ontological (classes and properties) models. Some of them use their own languages to define these mappings, while others use SPARQL extensions or SQL expressions. In all cases, the objective of these systems is to allow constructing ontology-based queries (e.g. in SPARQL), which are then rewritten into a set of queries expressed in the query language of the data source (typically SQL), according to the specified mappings. The query results are then converted back from the relational format into RDF, which is returned to the user.

One common approach is to first generate a syntactical translation of the database schema to an ontological representation. Although the resulting ontology has no real semantics, it may be argued that this is a first step through an ontology model and that ontology alignment techniques could be used later to map it to a real domain ontology [19]. Variations of ontology generation and syntactic mappings have been presented in previous works [20, 21, 22, 23]. In SquirrelRDF [24] they take this simple approach. A mapping generated from the relational database schema is built in RDF. There is no mapping to a mediated ontology. SPARQL queries can be executed and results are return in RDF.

An additional step is taken in RDBToOnto [21] , where the ontology generation process does not only take the database schema into account, but also the data. For instance it is able to discover subsumption relationships by finding categorization columns in the database tables. Even after this resulting ontology is produced, RDBToOnto allows users to create custom constraints. However this work focuses on the ontology generation and does not provide a querying mechanism to the database data. Relational.OWL [22] builds an ontology based on the relational schema and then maps it to the mediated ontology through a RDF query language. The first phase -transforming the database schema to the Relational.OWL ontology- produces a syntactical representation without real semantics. It is thus necessary to proceed with the second step, mapping the Relational.OWL representation to the domain ontology. This mapping can be done with an RDF query language like SPARQL and it is therefore not necessary to create a new mapping language. The SPASQL implementation [23] is an extension for MySQL to support SPARQL queries. It is thus technologically restricted to MySQL as it uses its query engine, although similar extensions could be built for other DBMSs. This extension is able to parse SPARQL queries and compile them and directly execute them in the MySQL engine. The mapping is limited (e.g. no multi-field keys al-

lowed) and it is not formalized.

Although automatic generation of ontologies and mappings can be useful in simple scenarios, for complex ones it is a limited approach. User expert knowledge may become necessary for complex mapping definitions, but it is also necessary to provide well defined languages that express those mappings. A number of OBDA systems use these mapping languages to access the underlying relational data sources [4].

The Virtuoso [25] declarative meta-schema language allows mapping relational schemas to RDF ontologies and is based on Quad Map Patterns that define the transformations. In the D2RQ platform [3], the D2RQ language is introduced and formally defined by an RDF schema and its engine is implemented as a Jena graph. It provides a Jena or Sesame API plug-in for querying and reasoning that rewrites the API calls to SQL queries which are executed in the database server, using the mappings. MASTRO [26] is a OBDA system that works with ontologies whose data is accessed through mappings in an external source, i.e. using an OBDA-enabled reasoner. It works over the DL-Lite$_A$ language, a fragment of OWL-DL. The mappings are specified through assertions that include SQL queries over the database. The expressiveness of the queries is limited to conjunctive queries (CQ).

There are two main alternative approaches for defining these kinds of mappings [11], *Local-as-view (LaV)* and *Global-as-view (GaV)*, and a combination of both, *GLAV*. In the LaV approach, each of the source schemas is represented as a view in terms of the global schema. This approach is useful if the global schema is well established or if the set of sources or their schemas may constantly change. However, query processing in this approach is not obvious, as it is not explicitly stated in the mapping definition how to obtain the data from the global view. In the GaV approach, the global schema elements are represented as views over the source schemas and it is explicitly defined how to query the sources. The advantage is that the processor can directly use this information to perform the query rewriting. The main disadvantage is that mapping definitions are affected in case of changes in the set of sources or in any of their schemas.

7.1.1. Streaming Data Querying

Several Stream processing and query engines have been built in the past years and can be grouped in two main areas: event stream systems (e.g. Aurora/Borealis [6], STREAM [5], TelegraphCQ [27]) and acquisitional stream systems (e.g. TinyDB [8], SNEE [7], Cougar [28]). For the first, the stream system does not have control over the data arrival rate, which is often potentially high and usually unknown and the query optimization goal is to minimize latency. For acquisitional streams, it is possible to control when data is obtained from the source, typically a sensor network, and the query optimization goal is to maximize network lifetime.

A number restrictions must be considered in the case of acquisitional streams from sensor networks, namely the usually low energy resources, limited computing power and storage capabilities of sensors. In order to address these issues, research has produced Sensor Networks Query Processing engines.These processors use declarative query languages for continuous data which describe logically the set of information that is to be collected but leaves to the engine to determine the algorithms and plans that are needed to get the data form the different nodes of the sensor network. Hence the server engine produces optimized query plans that are locally executed by the sensor network nodes in a distributed in-network scheme. These engines must also consider several optimization techniques in order to efficiently gather the information from the sensor nodes. This approach has been proved to be efficient especially in terms of energy consumption [8]. Architectures for query optimization in these constrained scenarios have surfaced [7, 8], showing that even with such limitations it is still possible to use rich and expressive declarative query languages.

Some of these systems have their own stream query language generally based on SQL. Although CQL (Continuous Query Language) [14] is the best

known of these languages there is still no common language for stream queries. The SNEEql [13] language for querying streaming data sources, which is used throughout the examples of this paper, is inspired by CQL, but it provides greater expressiveness in queries, including both event and acquisitional streams, and stored extents. Our work does not aim to improve on relational stream query processing, but to enable these systems to be accessible via ontology-based querying.

7.1.2. RDF *Streams*

As it is described in Section 7.1.1 query languages for relational data streams have been proposed and implemented in recent years. These languages borrow much of relational query languages such as SQL. In ontology-based data access solutions, continuous queries are expected to be posed in terms of an ontological view. In order to do so, it is necessary to have a stream query language that natively supports semantic models [29, 30].

SPARQL [31] is the W3C Recommendation for a query language over RDF. Even though SPARQL has been used to query RDF triples annotated with time constructs and can be used to represent data coming from streaming sources, it currently lacks the necessary operators to effectively query streaming data. There are two main approaches in the literature for extending SPARQL with stream-based operators: StreamingSPARQL and C-SPARQL. Both languages introduce extensions for the support of RDF streams, and both define time-based and triple-based window operators where the upper bound is fixed to the current evaluation time.

The StreamingSPARQL language presented in [9] is able to handle RDF based data streams. The semantics of these extensions are also provided as well as the algorithm to map the language additions to the extended algebra. The grammar of StreamingSPARQL basically consists in adding the capability of defining time and tuple-based windows over streams which are defined in the FROM clause. This proposal also allows specifying windows on graph patterns, which complicates its evaluation semantics.

The operators correspond to a large extent to the operators seen in DSMS query languages. Instead of tagged tuples we have tagged triples and the time and tuple based attributes are similar as well in syntax and semantics. However, StreamingSPARQL still lacks support for many features such as windows with higher boundaries different to the current timestamp *now*, aggregates, projection functions, and acquisitional parameters, among others. In addition, StreamingSPARQL allows windows in group graph patterns and redefines the language semantics due to the introduction of timestamps.

C-SPARQL (Continuous SPARQL) [10] also operates over RDF streams, sequences of triples annotated with non-decreasing timestamps. It defines, as in StreamingSPARQL, both time or tuple-based sliding windows. It allows combining stored and streaming knowledge and also combining multiple streams. C-SPARQL offers aggregates, such as COUNT, SUM, AVG, MIN and MAX. The aggregate semantics introduced in C-SPARQL follow an approach of extending the data, which differs from standard aggregation semantics of summarizing the data. We have opted to support the aggregation semantics being defined for SPARQL 1.1 [32], which summarize the data.

Unlike both previous approaches, the $\text{SPARQL}_{\text{Stream}}$ windowing operator enables windows to be defined in the past so as to support correlation with historic data. We have not included triple-based windows in $\text{SPARQL}_{\text{Stream}}$ due to the problems with their semantics, discussed in Section 4.1. Window-to-stream operators are also missing in both existing approaches, which provides ambiguous semantics for the language. In $\text{SPARQL}_{\text{Stream}}$ the result of a window operator is a bag to triples over which traditional operators can be applied. We have introduced three window-to-stream operators inspired by SNEEql and CQL.

8. Conclusions and Future Work

We have presented an approach for providing ontology-based access to streaming data, which is based on $\text{SPARQL}_{\text{Stream}}$, a SPARQL extension

Extension	Base Approach	Summary
SPARQL$_{\text{Stream}}$	SPARQL 1.1	Window definitions with variable upper boundary Window-to-stream operators
S$_2$O	R$_2$O	Stream definitions in mapping Streaming data types Virtual RDF stream IRIs
	ODEMAPSTER	Translation of SPARQL$_{\text{Stream}}$ queries into SNEEql

Table 1: Summary of key contributions.

for RDF streams, and S$_2$O, an extension to R$_2$O for expressing mappings from streaming sources to ontologies. We have shown the semantics of the proposed extensions and the mechanism to generate data source queries from the original ontological queries using the mappings. The case presented here generated SNEEql queries but the techniques are independent of the target stream query language, although issues of stream data model and language evaluation semantics would need to be considered for each case. Finally the prototype implementation, which extends ODEMAPSTER, has shown the feasibility of the approach. This work constitutes a first effort towards ontology-based streaming data integration, relevant for supporting the increasing number of sensor network applications being developed and deployed in the recent years. The extensions presented in this paper can be summarized in Table 1.

This approach is being used in the context of two applications that are being developed in the SemSorGrid4Env[11] (Semantic Sensor Grids for Rapid Application Development for Environmental Management) project. The first is the Flood Warning Use Case[33] which incorporates a network of heterogeneous sensors run by the Channel Coastal Observatory which are already in operation, and datasets collected by this project from data already in existence. These sensors are deployed in buoys at 36 locations of the South East England coast, measuring wave height, direction spread, temperature, etc. One of the aims of the Use Case is to utilize the CCO data, in order to produce real-time flood risk predictions for The Solent, Southern UK as well as possibly supporting longer term coastal planning through the development of scenarios. The outcomes of the project will also enable users to combine datasets from different sources, by creating mashups. We can see a screenshot of the application in Figure 2. The second application is the Fire Monitoring Use Case[34] which is focused on creating a fire monitoring and warning system in a forest region near Cercedilla in Spain. The system aims to prevent and detect fire combining two real-world real-time data sources: satellite data and measurements taken from a deployed Wireless Sensor Network in the area.

We are also preparing, with those data sources, a benchmark that will allow us to evaluate our approach in both query language expressiveness and performance. We will use the Linear Road Benchmark[35] as a basis but we will need extensions because it does not fit our evaluation focus. As our system delegates the query execution to the underlying stream engine, the focus of the evaluation is on the query and data translation.

Although we have shown initial results querying the underlying SNEE engine with basic queries, we expect to consider in the near future joins involving both streaming and stored data sources. Another important strand of future work is the optimization of distributed query processing [15] and the streaming queries [6, 7]. It is also our goal to provide a characterization of our algorithms. In the scope of a larger streaming and sensor networks integration framework, we intend to achieve the following goals: i) integrating streaming and stored data sources through an ontological unified view; ii) combining data from event-based

[11] `http://www.semsorgrid4env.eu` accessed 29 September 2010.

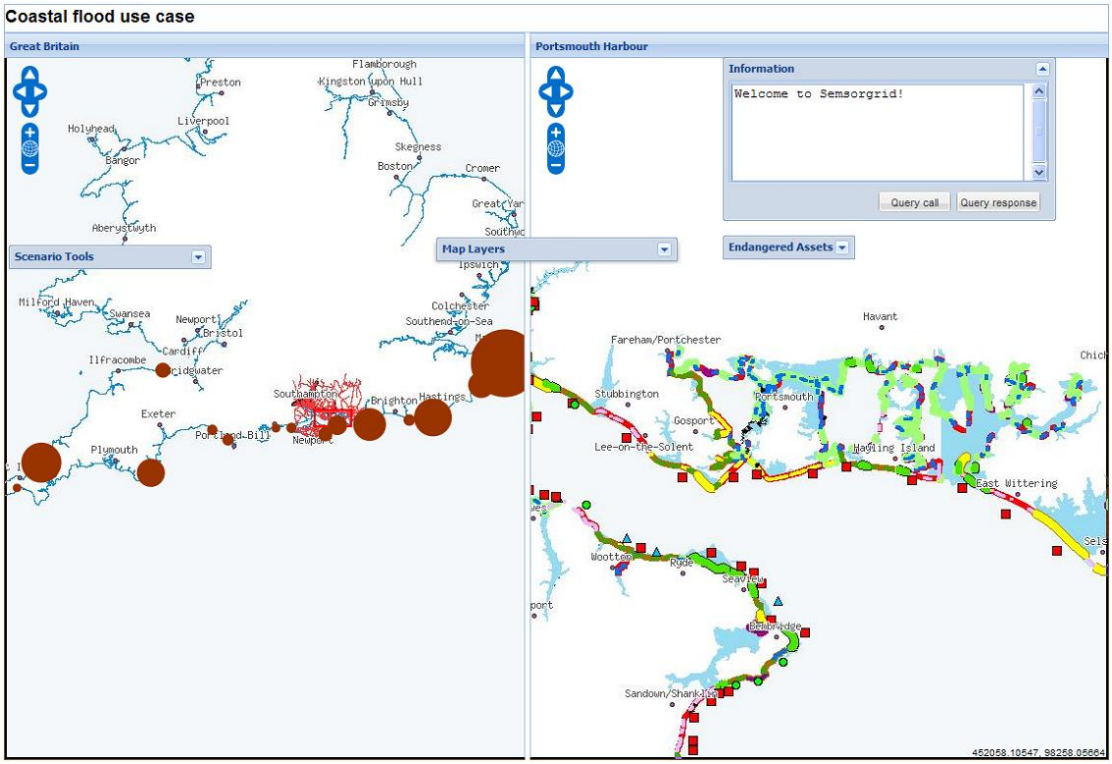

Figure 2: Flood Warning Use Case application. Red circles on the left map represent wave heights captured by the CCO sensors in the coast of South East England, queried and retrieved using the Ontology-based streaming data access service. On the right map, flood defences of the area are depicted.

and acquisition-based streams, and stored data sources; iii) considering quality-of-service requirements for query optimization and source selection during the integration.

9. Acknowledgments

This work has been supported by the European Commission project SemSorGrid4Env (FP7-223913). We also thank Alvaro A. A. Fernandes, Ixent Galpin, and Norman W. Paton, from the University of Manchester, for their valuable ideas and suggestions.

References

[1] H. Wache, T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, S. Hübner, Ontology-based integration of information - a survey of existing approaches, 2001, pp. 108–117.

[2] J. Barrasa, O. Corcho, A. Gómez-Pérez, R2O, an extensible and semantically based database-to-ontology mapping language, in: SWDB2004, 2004, pp. 1069–1070.

[3] C. Bizer, R. Cyganiak, D2RQ . Lessons Learned. W3C Workshop on RDF Access to Relational Databases (October 2007).

[4] S. S. Sahoo, W. Halb, S. Hellmann, K. Idehen, T. T. Jr, S. Auer, J. Sequeda, A. Ezzat, A survey of current approaches for mapping of relational databases to RDF, W3C (January 2009).
URL `http://www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF_SurveyReport.pdf`

[5] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, J. Widom, Stream: The stanford data stream management system, in: M. Garofalakis, J. Gehrke, R. Rastogi (Eds.), Data Stream Management, 2006.

[6] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. Zdonik, The Design of the Borealis Stream Processing Engine, in: CIDR 2005, 2005.

[7] I. Galpin, C. Y. Brenninkmeijer, F. Jabeen, A. A. Fernandes, N. W. Paton, Comprehensive optimization of declarative sensor network queries, in: SSDBM 2009, 2009, pp. 339–360.

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

[9] A. Bolles, M. Grawunder, J. Jacobi, Streaming SPARQL - extending SPARQL to process data streams, in: ESWC 08, 2008, pp. 448–462.
[10] D. F. Barbieri, D. Braga, S. Ceri, M. Grossniklaus, An execution environment for C-SPARQL queries, in: EDBT 2010, Lausanne, Switzerland, 2010, pp. 441–452.
[11] M. Lenzerini, Data integration: a theoretical perspective, in: PODS '02, 2002, pp. 233–246.
[12] L. Golab, M. T. Özsu, Issues in data stream management, SIGMOD Record 32 (2) (2003) 5–14.
[13] C. Y. Brenninkmeijer, I. Galpin, A. A. Fernandes, N. W. Paton, A semantics for a query language over sensors, streams and relations, in: BNCOD '08, 2008, pp. 87–99.
[14] A. Arasu, S. Babu, J. Widom, The CQL continuous query language: semantic foundations and query execution, The VLDB Journal 15 (2) (2006) 121–142.
[15] D. Kossmann, The state of the art in distributed query processing, ACM Comput. Surv. 32 (4) (2000) 422–469.
[16] J. Pérez, M. Arenas, C. Gutierrez, Semantics and complexity of SPARQL, ACM Trans. Database Syst. 34 (3) (2009) 1–45.
[17] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, R. Rosati, DL-Lite: Tractable description logics for ontologies, in: AAAI 2005, 2005, pp. 602–607.
[18] A. Poggi, D. Lembo, D. Calvanese, G. D. Giacomo, M. Lenzerini, R. Rosati, Linking data to ontologies, J. Data Semantics 10 (2008) 133–173.
[19] L. Lubyte, S. Tessaris, Supporting the development of data wrapping ontologies, in: 4th Asian Semantic Web Conference, 2009.
[20] A. Seaborne, D. Steer, S. Williams, Sql-rdf (October 2007).
[21] F. Cerbah, Learning highly structured semantic repositories from relational databases the rdbtoonto tool, in: Proceedings of the 5th European Semantic Web Conference (ESWC 2008), Springer, 2008, p. 777.
[22] C. P. de Laborda, S. Conrad, Database to semantic web mapping using rdf query languages, in: D. W. Embley, A. Olivé, S. Ram (Eds.), ER, Vol. 4215 of Lecture Notes in Computer Science, Springer, 2006, pp. 241–254.
[23] E. Prud'hommeaux, Spasql, http://www.w3.org/2005/05/22-SPARQL-MySQL/XTech (2007).
[24] D. Steer, SquirrellRDF (2006).
URL http://jena.sourceforge.net/SquirrelRDF
[25] O. Erling, I. Mikhailov, RDF support in the Virtuoso DBMS., in: Conference on Social Semantic Web, Vol. 113 of LNI, GI, 2007, pp. 59–68.
[26] A. Poggi, M. Ruzzi, Ontology-based data access with MASTRO, in: OWLED, Vol. 258 of CEUR Workshop Proceedings, CEUR-WS.org, 2007.
[27] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, M. A. Shah, TelegraphCQ: continuous dataflow processing, in: SIGMOD '03, 2003, pp. 668–668.
[28] Y. Yao, J. Gehrke, The Cougar approach to in-network query processing in sensor networks, SIGMOD Rec. 31 (3) (2002) 9–18.
[29] E. Della Valle, S. Ceri, D. Braga, I. Celino, D. Fensel, F. van Harmelen, G. Unel, Research chapters in the area of stream reasoning, in: SR2009, 2009, pp. 1–9.
[30] S. Groppe, J. Groppe, D. Kukulenz, V. Linnemann, A sparql engine for streaming rdf data, in: SITIS '07: Proceedings of the 2007 Third International IEEE Conference on Signal-Image Technologies and Internet-Based System, IEEE Computer Society, Washington, DC, USA, 2007, pp. 167–174.
[31] E. Prud'hommeaux, A. Seaborne, SPARQL query language for RDF, W3C recommendation, Tech. rep., World Wide Web Consortium (January 2008).
URL http://www.w3.org/TR/rdf-sparql-query/
[32] S. Harris, A. Seaborne (eds), SPARQL 1.1 query language, Working draft, W3C (2010).
[33] C. Hutton, J. Sadler, K. Page, M. Clark, R. Newman, S. Roe, Wt 7.1 flood user requirements specification. deliverable d7.1v2, semsorgrid4env (September 2010).
[34] D. Guillén, I. Castanedo, I. Liébana, A. Izquierdo, Wp 6.1 requirements specification, version 2. deliverable d6.1v2, semsorgrid4env (September 2010).
[35] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, R. Tibbetts, Linear Road: A Stream Data Management Benchmark, in: VLDB '04: Proceedings of the 30th international conference on Very large data bases, VLDB Endowment, 2004, pp. 480–491.

Appendix A. S_2O Grammar

We present below the grammar of the S_2O mapping language.

```
BNF for S2O mapping descriptions

s2o::= import? dschema-description+
       stschema-description+
       conceptmapping-definition+ ontology+
import::= import literal
ontology::= ontology literal
literal::= '<string literal>'

BNF for schema descriptions

dschema-description::= dbschema-desc name
                       documentation? (has-table
                           table-desc)+
stschema-description::= streamschema-desc name
                       documentation?
                       (has-table table-desc)+
                       (has-stream stream-desc)+
name::= name literal
```

```
documentation::= documentation literal
table-desc::= name tabletype? documentation?
              (column-description)+
tabletype::= tableType literal
stream-desc::= name streamtpe? documentation?
               (column-description)+
streamtype::= streamType literal
column-description::= (keycol-desc |
                       forkeycol-desc |
                       nonkeycol-desc |
                       timestamp-desc) name
                       columnType documentation?
                       col-reference?
                       implicit-col-reference?
columnType::= columnType datatype
col-reference::= refers-to literal
implicit-col-reference::= implicitlyrefers-to
                          literal
datatype::=string | boolean | decimal | float |
           double | date | integer
           ... (XML Schema Datatypes)

BNF for concept mapping definitions

conceptmapping-definition::= conceptmap-def name
                     documentation identified-by+
                     (uri-as transformation)?
                     (described-by propertymap-
                         def)*
                     (applies-if cond-expr)?
                     (joins-via join-list)?
                     (union union-def)?
identified-by::= identified-by literal
join-list::= documentation? (hasjoin joindesc)+
                            (overwrites literal)?
joindesc::=(hasCol literal)+

BNF for condition expressions

cond-expr::= orcond-expr |
             AND andcond-expr orcond-expr
orcond-expr::= notcond-expr |
               OR orcond-expr notcond-expr
notcond-expr::= condition | NOT condition
condition::= primitive-condition
             (arg-restriction arg-restriction)*
primitive-condition::=lo_than | loorequal_than |
              lo_than_str | loorequal_than_str |
              hi_than | hiorequal_than |
              hi_than_str | hiorequal_than_str |
              equals | equals_str | in_keyword |
              in_set | in_set_str | between |
              between_str | date_before |
              date_after | date_equal
arg-restriction::= parameter-selector restriction
parameter-selector::= on-param literal
restriction::= has-value constant-value |
               has-column literal |
               has-transform transformation
constant-value::=datatype literal

BNF for transformations

transformation::= primitive-transformation
                  (arg-restriction arg-
                      restriction)*
primitive-transformation::= get_nth_char | get
    _delimited |
                    get_substring | concat |
                        add_type |
                        Subtract_type |
                            Multiply_type |
                        divide_type | constant

BNF for property mappings

propertymap-def::=attributemap-def | relfromatt-
    def |
                  relationmap-def
attributemap-def::= attributemap-def name use-
    dbcol*
                    selector* documentation
relfromatt-def::=relfromatt-def name use-dbcol*
                 selector* newobj-type?
                    documentation?
relationmap-def::=relationmap-def to-concept
use-dbcol::=use-dbcol literal
selector::=selector (applies-if cond-expr)?
          (aftertransform transformation)?
newobj-type::=newobject-type literal
to-concept::=to-concept literal}
union-def::=name (has-stream literal)+
```

Listing 7: Grammar of the S_2O mapping language.