Enabling Ontology-based Access to Streaming Data Sources

Jean-Paul Calbimonte[1], Oscar Corcho[1], and Alasdair J G Gray[2]

[1]Ontology Engineering Group, Departamento de Inteligencia Artificial,
Facultad de Informática, Universidad Politécnica de Madrid,
Campus de Montegancedo s/n 28660, Boadilla del Monte, Spain
jp.calbimonte@upm.es,ocorcho@fi.upm.es
[2]School of Computer Science, The University of Manchester,
Oxford Road, Manchester M13 9PL, United Kingdom
a.gray@cs.man.ac.uk

Abstract. The availability of streaming data sources is progressively increasing thanks to the development of ubiquitous data capturing technologies such as sensor networks. The heterogeneity of these sources introduces the requirement of providing data access in a unified and coherent manner, whilst allowing the user to express their needs at an ontological level. In this paper we describe an ontology-based streaming data access service. Sources link their data content to ontologies through S_2O mappings. Users can query the ontology using $SPARQL_{Stream}$, an extension of SPARQL for streaming data. A preliminary implementation of the approach is also presented. With this proposal we expect to set the basis for future efforts in ontology-based streaming data integration.

1 Introduction

Recent advances in wireless communications and sensor technologies have opened the way for deploying networks of interconnected sensing devices capable of ubiquitous data capture, processing and delivery. Sensor network deployments are expected to increase significantly in the upcoming years because of their advantages and unique features. Tiny sensors can be installed virtually anywhere and still be reachable thanks to wireless communications. Moreover, these devices are inexpensive and can be used for a wide variety of applications including security surveillance, healthcare provision, and environmental monitoring.

As an example, consider a web application which aids an emergency planner to detect and co-ordinate the response to a forest fire in Spain. This involves retrieving relevant data from multiple sources, e.g. weather data from AEMET (Agencia Española de Meteorología)[1], sensor data from sensor networks deployed in the region, and any other relevant sources of data such as the ESA satellite imagery providing fire risks[2]. Typically sources are managed autonomously and

[1] http://www.aemet.es accessed 15 September 2010.
[2] http://dup.esrin.esa.int/ionia/wfa/index.asp accessed 15 September 2010.

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 fire can be detected, and presented to the user in terms of their domain, e.g. fire risk assessment. We propose that ontologies can be used as such a common model. For the scenario presented here, we use an ontology that extends ontologies from SWEET[3] and the W3C incubator group's semantic sensor network ontology[4].

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 [1] and D2RQ [2]. These systems provide semantic access to traditional (stored) data sources by providing mappings between the elements in the relational and ontological models [3]. 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 [4] and Borealis [5] have focused on query evaluation and optimisation over streams with high, variable, data rates. Other systems such as SNEE [6] and TinyDB [7], 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 [8,9]. 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, and 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.

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

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

2 Background

This section describes the existing work upon which our approach for enabling ontology-based access to streaming data sources is based, *viz.* R$_2$O which provides ontology-based access to stored relational data, and SNEE, a query processing engine over relational data streams. A full discussion of related work can be found in Section 7.

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 [3]. The objective of these systems is to allow users to construct queries over an ontology (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. ODEMAPSTER is one such system which uses R$_2$O (Relational-to-Ontology) to express the mappings between the relational data source and the ontology [1].

The mapping definition language R$_2$O defines relationships between a set of ontologies and relational schemas [1]. The mappings are expressed in terms of selections and transformations over database relations following a Global-as-View (GAV) approach [10], and can be created either manually or with the help of a mapping tool. The resulting mappings are saved as XML which enables them to be independent of any specific DBMS or ontology language.

Mapping relations to ontologies often requires performing operations on the relational sources. Several cases are handled by R$_2$O 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$_2$O 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 Querying Relational Data Streams

A relational data stream is an append only, potentially infinite, sequence of timestamped tuples [11], 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:

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 [12,13] and data stream management systems (DSMS) [4,5,6,7] 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 [12]. SNEEql can be viewed as extending SQL for processing data streams. The additional constructs are explained below.

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 '`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.

Queries expressed in the SNEEql language are optimized for evaluation within a sensor network over acquisitional-streams by the SNEE compiler [6]. SNEE has recently been extended to enable query evaluation over event-streams either within the sensor network (in-network query processing) or on computational hardware outside of the sensor network.

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[5] 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

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

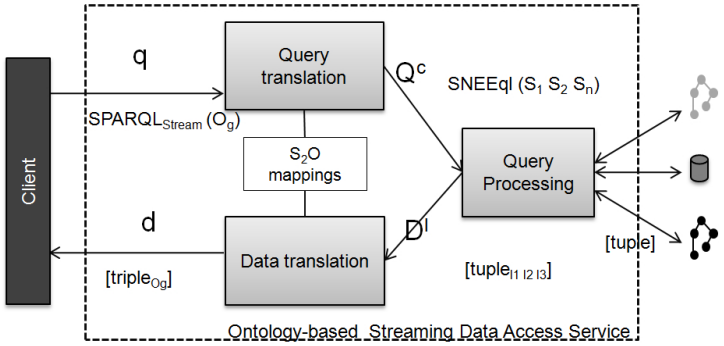

Fig. 1. Ontology-based streaming data access service

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$_2$O, an extension of the R$_2$O 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 [14] 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 SPARQL$_{\text{Stream}}$ queries into queries over the streaming data sources using SNEEql as the target language. The following 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 SPARQL$_{\text{Stream}}$ query language, an extension to SPARQL for streaming RDF data, which has been inspired by previous proposals such as C-SPARQL [9] and SNEEql [12]. However, significant improvements

have been made that correct the types supported and the semantics of windowing operations, which can be summarised 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$_{\text{Stream}}$

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[6].

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`)[7]. 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 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 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, SPARQL$_{\text{Stream}}$ only supports time-based windows. C-SPARQL also has the notion of a triple-based window. However, such windows are problematic 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, and hence it would not be possible to compute the query answer.

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

[7] 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:WindSpeedMeasurement;
            fire:hasSpeed ?speed; }
    } GROUP BY ?WindSpeed
  }
  {
    SELECT AVG(?archivedSpeed) AS ?WindSpeedHistoryAvg
    WHERE
    {
      GRAPH <www.semsorgrid4env.eu/SensorArchiveReadings.srdf>  {
        ?ArchWindSpeed a fire:WindSpeedMeasurement;
        fire:hasSpeed ?archivedSpeed;  }
    } GROUP BY ?ArchWindSpeed
  }
  FILTER (?WindSpeedAvg > ?WindSpeedHistoryAvg)
}
```

Listing 1. An example 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.

4.2 S$_2$O: Expressing Stream-to-Ontology Mappings

The mapping document that describes how to transform the data source elements to ontology elements is written in the S$_2$O mapping language, an extended version of R$_2$O [1]. An R$_2$O mapping document includes a section that describes the database relations, dbscehma-desc. In order to support data streams, R$_2$O 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 the stream is similar to a relation. An additional attribute streamType has been added, it denotes the kind of stream in terms of data acquisition, i.e. event or acquistional. 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$_2$O extends R$_2$O, relations can also be specified using the existing R$_2$O mechanism. For the class and property mappings, the existing R$_2$O definitions can be used for stream schemas just as it was for relational schemas. This is specified in the conceptmap-def element 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 SPARQL$_{\text{Stream}}$ streaming query requires an RDF stream to have an IRI identifier. S$_2$O creates a *virtual* RDF stream

and its IRI is specified in the S2O mapping using the virtualStream element. It can be specified at the conceptmap-def level or at the attributemap-def level.

```
streamschema-desc
  name MeteoSensors
  has-stream SensorWind
    streamType pushed
    documentation "Wind measurements"
    keycol-desc measurementId
      columnType integer
    timestamp-desc  measureTime
      columnType datetime
    nonkeycol-desc  measureSpeed
      columnType float
    nonkeycol-desc measureDirection
      columnType float
...
conceptmap-def Wind
  virtualStream <http://semsorgrid4env.eu/SensorReadings.srdf>
  uri-as
    concat(SensorWind.measurementID)
  applies-if
    <cond-expr>
  described-by
    attributemap-def hasSpeed
      virtualStream http://semsorgrid4env.eu/SensorReadings.srdf>
      operation constant
        has-column SensorWind.measureSpeed
```

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$_{\text{Stream}}$ Semantics

The SPARQL extensions presented here are based on the formalisation of Pérez *et al.* [15]. 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 S$_2$O mappings.

5.2 S$_2$O Semantics

In this section we will present how we can use the S$_2$O 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 [1] and the formalisation work of Calvanese *et al.* [16] and Poggi *et al.* [17].

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

$$\begin{aligned}
q(\boldsymbol{x}) &\leftarrow \varphi(\boldsymbol{x}, \boldsymbol{y}) \\
\varphi(\boldsymbol{x}, \boldsymbol{y}) : \bigwedge_{i=1\ldots k} P_i, \text{ with } P_i &\begin{cases} C_i(x), C \text{ is an atomic class.} \\ R_i(x,y), R \text{ is an atomic property.} \\ x = y \end{cases} \\
& x, y \text{ are variables either in } \boldsymbol{x}, \boldsymbol{y} \text{ or constants.}
\end{aligned}$$

where $\boldsymbol{x}$ is a tuple of distinct distinguished variables, and $\boldsymbol{y}$ a tuple of non-distinguished existentially quantified variables. The answer to this query consists in the instantiation of the distinguished variables [16]. For instance consider:

$$q_1(x) \leftarrow WindSpeedMeasurement(x) \wedge measuredBy(x,y) \wedge WindSensor(y)$$

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: $WindSpeedMeasurement(x)$, $measuredBy(x, y)$, 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 $q(\boldsymbol{x}) \leftarrow \varphi(\boldsymbol{x}, \boldsymbol{y})$ the answer to q is the set $q^{\mathcal{I}}_{\boldsymbol{x}}$ of tuples $\boldsymbol{c} \in \Delta^{\mathcal{I}} \times \cdots \times \Delta^{\mathcal{I}}$ that substituted to $\boldsymbol{x}$, make the formula $\exists \boldsymbol{y}.\varphi(\boldsymbol{x}, \boldsymbol{y})$ true in $\mathcal{I}$ [16,17,18]. 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 $C(f^{Id}_{C}(\boldsymbol{x})) \rightsquigarrow \Phi_{S_1,\ldots,S_n}(\boldsymbol{x})$ 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 $\boldsymbol{x}$ 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:

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

where $\boldsymbol{v}$ is a tuple of variables in either $\boldsymbol{x}, \boldsymbol{y}$. 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 $\boldsymbol{v}$ in the streams S_i (e.g. conditions using $<, \leq, \geq, >, =$ operators).

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

$$\Phi_{S_1,\ldots,S_n}(\boldsymbol{x_1}, \boldsymbol{x_2}) = \exists \boldsymbol{y}.\Phi_{S_1,\ldots,S_k}(\boldsymbol{x_1}) \wedge \Phi_{S_{k+1},\ldots,S_n}(\boldsymbol{x_2}) \wedge p^{Join}_{S_1,\ldots,S_n}(\boldsymbol{v})$$

where $\Phi_{S_1,\ldots,S_k}, \Phi_{S_{k+1},\ldots,S_n}$ 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 $A(f^{Id}_{C}(\boldsymbol{x}), f^{Trf}_{A}(\boldsymbol{z})) \rightsquigarrow \Phi_{S_1,\ldots,S_n}(\boldsymbol{x}, \boldsymbol{z})$ 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 $\boldsymbol{z}$ to obtain the property value (e.g. arithmetic operations, or string operations). The declarative form of Φ in this case is:

$$\Phi_{S_1,\ldots,S_n}(\boldsymbol{x}, \boldsymbol{z}) = \exists \boldsymbol{y}.\Phi_{S_1,\ldots,S_k}(\boldsymbol{x}) \wedge \Phi_{S_{k+1},\ldots,S_n}(\boldsymbol{z}) \wedge p^{Join}_{S_1,\ldots,S_n}(\boldsymbol{v})$$

The definition follows the same idea as the previous one. The variables of $\boldsymbol{z}$ 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 [1] (that is based on the PerfectRef algorithm of [16]) 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 $Q_{\mathcal{O}}(\boldsymbol{x})[t_s, t_e, \delta]$ 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:

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

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

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

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 Walkthrough

The presented approach of providing ontology-based access to streaming data has been implemented as an extension to the ODEMAPSTER processor [1]. This implementation generates SNEEql queries that can be executed by the streaming query processor.

```
windsamples: (sensorId INT PK, ts DATETIME PK, speed FLOAT, direction FLOAT)
sensors: (sensorId INT PK, sensorName CHAR(45), lat FLOAT, long FLOAT)
```

Listing 3. Relational schema of the data source.

Consider the motivating example where a sensor network generates a stream `windsamples` of wind sensor measurements. The associated stored information about the sensors, e.g. location and type, are stored in a relation `sensors`. The schemas are presented in Listing 3. Also consider the following ontological view:

```
conceptmap-def WindSpeedMeasurement
  virtualStream <http://semsorgrid4env.eu/SensorReadings.srdf>
  uri-as
    concat('http://semsorgrid4env.eu/WindSpeedMeasurement_',windsamples.
        sensorId,windsamples.ts)
  described-by
    attributemap-def hasSpeed
      operation constant
        has-column windsamples.speed
    dbrelationmap-def isProducedBy
      toConcept Sensor
      joins-via
        condition equals
          has-column sensors.sensorId
          has-column windsamples.sensorId

conceptmap-def Sensor
  uri-as
    concat('http://semsorgrid4env.eu/Sensor_',sensors.sensorId)
  described-by
    attributemap-def hasSensorId
      operation constant
        has-column sensors.sensorId
```

Listing 4. S_2O mapping from the data stream windsamples to the ontology concepts $WindSpeedMeasurement$.

$$
\begin{aligned}
SpeedMeasurement &\sqsubseteq Measurement \\
WindSpeedMeasurement &\sqsubseteq SpeedMeasurement \\
WindDirectionMeasurement &\sqsubseteq Measurement \\
SpeedMeasurement &\sqsubseteq \exists hasSpeed \\
Measurement &\sqsubseteq \exists isProducedBy.Sensor \\
Sensor &\sqsubseteq \exists hasName
\end{aligned}
$$

We can define an S_2O mapping that splits the `windsamples` stream tuples into instances of two different concepts $WindSpeedMeasurement$ and $WindDirection\text{-}Measurement$. Listing 4 is an extract of the S_2O mapping document concerning the $WindSpeedMeasurement$. The mapping extract defines how to construct the $WindSpeedMeasurement$ and $Sensor$ class instances from the `windsamples` stream and the `sensors` table: $\Psi_{WindSpeedMeasurement} \rightsquigarrow \Phi_{\texttt{windsamples}}$ and $\Psi_{Sensor} \rightsquigarrow \Phi_{\texttt{sensors}}$. In the case of the $WindSpeedMeasurement$ the function $f^{Id}_{WindSpeedMeasurement}$ produces the URI's of the instances by concatenating the `sensorId` and `ts` attributes. Now we can pose a query over the ontology using $\text{SPARQL}_{\text{Stream}}$, for example to obtain the wind speed measurements taken in the last 10 minutes (See the query in Listing 5).

A class query atom $WindSpeedMeasurement(x)$ and a datatype property atom $hasSpeed(x, z)$ can be extracted from the $\text{SPARQL}_{\text{Stream}}$ query. The window specification $[t_s = \mathsf{NOW} - 10, t_e = \mathsf{NOW}, \delta = 1]$ is also obtained[8]. The S_2O map-

[8] 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.

```
PREFIX fire: <http://www.ssg4env.eu#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT RSTREAM ?speed
FROM STREAM <www.ssg4env.eu/SensorReadings.srdf> [FROM NOW - 10 MINUTES TO
    NOW STEP 1 MINUTE]
WHERE {
  ?WindSpeed a fire:WindSpeedMeasurement;
    fire:hasSpeed ?speed;
}
```

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

```
SELECT RSTREAM concat('http://ssg4env.eu#WindSpeedMeasurement',windsamples.
    sensorId,windamples.ts) AS id, windsamples.speed AS speed
FROM windsamples[FROM NOW - 10 MINUTES TO NOW SLIDE 1 MINUTE];
```

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

ping defines that $WindSpeedMeasurment$ instances are generated based on the `sensorId` and `ts` attributes of the `windsamples` stream, using a concatenation function to generate each instance URI. Similarly the S$_2$O mapping defines that *hasSpeed* properties are generated from the values of the speed attribute of the `windsamples` stream. The processor will evaluate this as:

$$\lambda(\Phi_{\texttt{windsamples}}(x_{\texttt{sensorId}}, x_{\texttt{ts}}, z_{\texttt{speed}})[now-10, now, 1]) = \\ \pi_{\texttt{sensorId,ts,speed}}(\omega_{now-10,now,1}(\texttt{windsamples}))$$

In this case no joins and other selection conditions are needed, and only one stream has to be queried to produce the results. The query generated in the SNEEql language is shown in Listing 6[9]. 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$_2$O mappings. This results in a stream of tagged triples which are instances of the class $WindSpeedMeasurement$.

7 Related Work

Several systems exist to provide ontology-based access to stored data, mainly in the form of relational databases, as described in [3].

A simple 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 could later be mapped to a real domain ontology [18]. Virtuoso [19] and D2RQ [2], like R$_2$O, use mappings between the source relational schema to RDF ontologies enabling users to issues queries over a semantically rich domain

[9] Although the current available implementation of the SNEE processor lacks the `concat` operator, we include the sample query in its complete form here.

ontology. The expressiveness of the queries supported by these systems is limited to conjunctive queries, and none of the approaches takes into account streaming data and continuous queries.

Several stream processing and querying engines have been built in the last decade and can be grouped in two main areas: event stream systems (e.g. Aurora/Borealis [5], STREAM [4], TelegraphCQ [20]), and acquisitional stream systems (e.g. TinyDB [7], SNEE [6], Cougar [21]). 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 optimisation goal is to maximize network lifetime. All these systems have their own continuous query language, generally based on SQL, although most of them share the same features. CQL (Continuous Query Language) [13] is the best known of these languages, but there is still no common standard language for stream queries. The SNEEql [12] language for querying streaming data sources 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.

Finally, there are two existing proposals for extending SPARQL with stream-based operators: StreamingSPARQL [8] and C-SPARQL [9]. 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 $\text{SPARQL}_{\text{Stream}}$ 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. The aggregate semantics introduced in C-SPARQL follow an approach of extending the data, which differs from standard aggregation semantics of summarising the data. We have opted to support the aggregation semantics being defined for SPARQL 1.1 [22], which summarise the data.

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 for RDF streams, and S_2O, an extension to R_2O 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 summarised in Table 1.

Although we have shown initial results querying the underlying SNEE engine with basic queries, we expect to consider in the near future more complex query expressions including aggregates, and joins involving both streaming and stored data sources. Another important strand of future work is the optimization of distributed query processing [14] and the streaming queries [5,6]. 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 and acquisition-based streams, and stored data sources; iii) considering quality-of-service requirements for query optimization and source selection during the integration.

Extension	Base Approach	Summary
$\text{SPARQL}_{\text{Stream}}$	SPARQL 1.1	Window definitions with variable upper boundary Window-to-stream operators
S_2O	R_2O	Stream definitions in mapping Streaming data types Virtual RDF stream IRIs
	ODEMAPSTER	Translation of $\text{SPARQL}_{\text{Stream}}$ queries into SNEEql

Table 1. Summary of key contributions.

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. Barrasa, J., Corcho, O., Gómez-Pérez, A.: R2O, an extensible and semantically based database-to-ontology mapping language. In: SWDB2004. (2004) 1069–1070
2. Bizer, C., Cyganiak, R.: D2RQ . Lessons Learned. W3C Workshop on RDF Access to Relational Databases (October 2007)
3. Sahoo, S.S., Halb, W., Hellmann, S., Idehen, K., Jr, T.T., Auer, S., Sequeda, J., Ezzat, A.: A survey of current approaches for mapping of relational databases to RDF. W3C (January 2009)
4. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J.: Stream: The stanford data stream management system. In Garofalakis, M., Gehrke, J., Rastogi, R., eds.: Data Stream Management. (2006)

5. 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.: The Design of the Borealis Stream Processing Engine. In: CIDR 2005. (2005)
6. Galpin, I., Brenninkmeijer, C.Y., Jabeen, F., Fernandes, A.A., Paton, N.W.: Comprehensive optimization of declarative sensor network queries. In: SSDBM 2009. (2009) 339–360
7. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: TinyDB: an acquisitional query processing system for sensor networks. ACM Trans. Database Syst. **30**(1) (2005) 122–173
8. Bolles, A., Grawunder, M., Jacobi, J.: Streaming SPARQL - extending SPARQL to process data streams. In: ESWC 08. (2008) 448–462
9. Barbieri, D.F., Braga, D., Ceri, S., Grossniklaus, M.: An execution environment for C-SPARQL queries. In: EDBT 2010, Lausanne, Switzerland (March 2010) 441–452
10. Lenzerini, M.: Data integration: a theoretical perspective. In: PODS '02. (2002) 233–246
11. Golab, L., Özsu, M.T.: Issues in data stream management. SIGMOD Record **32**(2) (June 2003) 5–14
12. Brenninkmeijer, C.Y., Galpin, I., Fernandes, A.A., Paton, N.W.: A semantics for a query language over sensors, streams and relations. In: BNCOD '08. (2008) 87–99
13. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. The VLDB Journal **15**(2) (June 2006) 121–142
14. Kossmann, D.: The state of the art in distributed query processing. ACM Comput. Surv. **32**(4) (2000) 422–469
15. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. **34**(3) (2009) 1–45
16. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: DL-Lite: Tractable description logics for ontologies. In: AAAI 2005. (2005) 602–607
17. Poggi, A., Lembo, D., Calvanese, D., Giacomo, G.D., Lenzerini, M., Rosati, R.: Linking data to ontologies. J. Data Semantics **10** (2008) 133–173
18. Lubyte, L., Tessaris, S.: Supporting the development of data wrapping ontologies. In: 4th Asian Semantic Web Conference. (December 2009)
19. Erling, O., Mikhailov, I.: RDF support in the Virtuoso DBMS. In: Conference on Social Semantic Web. Volume 113 of LNI., GI (2007) 59–68
20. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Reiss, F., Shah, M.A.: TelegraphCQ: continuous dataflow processing. In: SIGMOD '03. (2003) 668–668
21. Yao, Y., Gehrke, J.: The Cougar approach to in-network query processing in sensor networks. SIGMOD Rec. **31**(3) (2002) 9–18
22. Harris, S., Seaborne (eds), A.: SPARQL 1.1 query language. Working draft, W3C (2010)