Enabling Ontology-based Access to Streaming Data Sources

First Author Name (Blank for Blind Review)
Affiliation (Blank for Blind Review)
Address (Blank for Blind Review)
e-mail address (Blank for Blind Review)

Second Author Name (Blank for Blind Review)
Affiliation (Blank for Blind Review)
Address (Blank for Blind Review)
e-mail address (Blank for Blind Review)

Third Author Name (Blank for Blind Review)
Affiliation (Blank for Blind Review)
Address (Blank for Blind Review)
e-mail address (Blank for Blind Review)

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. In this paper we describe an ontology-based streaming data access service, based on extensions to the R_2O mapping language and its query processor ODEMapster, and to the C-SPARQL RDF stream query language. A preliminary implementation of the approach is also presented. With this proposal we expect to set the basis for future efforts in ontology-based integration of sensor networks streaming data sources.

Author Keywords

Streaming data access, Ontology-based data access, Sensor networks Querying.

ACM Classification Keywords

H.3.3 Information Search and Retrieval: Miscellaneous; H.2.3 Languages: Query languages.

General Terms

Algorithms, Languages, Theory.

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 range of applications such as security surveillance, traffic control, environmental monitoring, healthcare provision, industrial monitoring, etc.

One of the means to access streaming data sources coming from sensor networks is through query processors [16, 2, 11] that handle streaming data (which differs significantly from classical stored data, as it is potentially infinite and transient, with tuples being constantly added) and support declarative continuous query languages (for which query results are updated regularly as time passes [24]).

In the context of the Semantic Web vision, several initiatives that aim at providing semantic access to traditional (stored) data sources have been launched in the past years. Most of the existing approaches attempt to provide mappings between the elements in the relational and ontological models [22], as we will describe in the Background section.However, to the best of our knowledge, similar solutions for streaming data mapping and querying using ontology-based approaches have not been explored yet in depth.

In this paper we focus on providing ontology-based access to streaming data sources, including sensor networks, through declarative continuous queries. This constitutes a first step towards a framework for the integration of distributed heterogeneous streaming and stored data sources through ontological models and to the provision of Linked Data for streams [13, 17, 23]. The paper is organised as follows: in the Background section we introduce previous work. The foundations of our approach are explained in the Ontology-based Streaming Data Access section. In the Extensions Syntax section we present the syntactic extensions for RDF stream SPARQL operators, and R_2O stream-to-ontology mappings. The semantics of these extensions are detailed in the Streaming Extensions Semantics section and a first implementation of the execution of the streaming data access approach is explained in the Implementation and Walkthroug section. Finally we present the conclusions and future work.

BACKGROUND

The following sections describe the state of the art in streaming data access and continuous queries (Streaming Data Access section), and query languages for RDF streams (Con-

UbiComp '10, Sep 26-Sep 29, 2010, Copenhagen, Denmark.

tinuous Queries for RDF Streams section).

Streaming Data Access

Streaming data is characterised by the fact that it is normally transient and potentially infinite, with new data items being regularly added and where old items are usually less relevant than newer ones. Hence Data Stream Management Systems (DSMS) are quite different from classical database systems, which deal mostly with static data, with lower insert rates and queries that retrieve the state of the data at the current time. Stream systems require additional operators in their query languages, such as time-based windows to limit streams to finite bounded structures in order to process only a smaller subset of data [3, 7].

Several DSMS have been built in the past years and can be grouped in two main areas: event stream systems (e.g. Aurora/Borealis [1], STREAM [2], TelegraphCQ [9]) and acquisitional stream systems (e.g. TinyDB [16], SNEE [11], Cougar [25]). For the first, the stream system does not have control over the data arrival rate, which is often potentially high and usually unknown. For acquisitional streams, it is possible to control when data is obtained from the source. Some restrictions must be considered in the case of sensor networks streams, 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 such as the abovementioned TinyDB, Cougar and SNEE. 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. Therefore the server engine produces optimised query plans that are locally executed by the sensor network nodes in a distributed in-network scheme. These engines must also consider several optimisation techniques in order to efficiently gather the information from the sensor nodes. This approach has been proven to be efficient especially in terms of energy consumption [16]. Architectures for query optimisation in these constrained scenarios have surfaced [11, 16], showing that even with such limitations it is still possible to use rich and expressive declarative query languages.

All these systems have their own continuous query language, generally based on SQL, although most of them share the same features. In order to exemplify these query language features we will use the syntax of one of them: SNEEql [7] (which is based on CQL).

The first concept to be considered in stream data models is that of a *tagged tuple*, which is a tuple that includes a named *timestamp* attribute. This special attribute indicates when the tuple entered the stream, and is essential to define the semantics of stream operators in these languages: two tuples having the same timestamp are considered to have entered the stream at the same time instant. A stream is a potentially infinite sequence of tagged tuples.

Next we can move into *queries*. Queries over streams are of the form:

$$\begin{aligned}&\mathsf{SELECT}\ \langle{*}\mathsf{STREAM}\rangle\ a_1,\ldots,a_n\\&\mathsf{FROM}\ w_1\langle\mathsf{window}\rangle,\ldots,w_m\langle\mathsf{window}\rangle\ \mathsf{WHERE}\ p\end{aligned}$$

where $a1,\ldots,a_n$ is a project list, $w_1,\ldots,w_m$ is a list of streams of tagged tuples with optional window definitions, and p is a predicate [3, 7]. The result of the execution of a stream query is a stream of tagged tuples or a stream of windows.

In queries, a *time window* operator produces bounded sequences of tagged tuples whose timestamp falls in the specified interval. A window can be specified as follows: s [FROM t_1 TO t_2 SLIDE int unit] where FROM t_1 TO t_2 indicates a time interval. The slide parameter indicates the frequency of the window creation in time units or rows. Notice that windows are not only time dependant, but may also be tuple (row) dependant.

Other important and useful features of continuous query languages are *aggregation functions*, *window-to-stream operators* such as ISTREAM, DSTREAM and RSTREAM [3], and *quality of service requirements* [11] (acquisition rate, delivery time, network lifetime, etc.).

Ontology-based Data Access

The goal of Ontology-based Data Access (OBDA) is to generate semantic web content from existing relational data sources available in the web [22]. As mentioned above, most of the existing approaches are based on the exploitation of mappings between the relational (rows and columns) and the ontological (concepts and roles) 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.

There are two main alternative approaches for defining these mappings [14], *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.

We will now describe in detail one of these ODBA approaches (R_2O and ODEMapster), which is the one that we will extend in this paper.

R_2O and ODEMapster

R_2O(Relational-to-Ontology)[5] is a GaV mapping definition language that defines relationships between a set of ontologies and relational schemas. The R_2O language is XML-based, independent of any specific DBMS and allows complex mapping expressions between ontology and relational elements, described in terms of selections and transformations over database tables and columns. R_2O covers a wide set of mapping cases common in relational to ontology situations. R_2O is designed to cope with the following mapping cases:

- A database table maps to one class in the ontology.
- 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.

Mapping tables and columns to concepts and attributes often requires performing some operations on the relational sources. Several cases are handled by R_2O and detailed below.

- *Direct Mapping*. When the relational table maps an ontology class and the column values are used to fill the property values of the ontology instances. Each table record will generate a class instance in the ontology.
- *Join/Union*. In some occasions a single table does not correspond alone to a class, but it has to be combined with other tables. The result of the join or union of the tables will generate the corresponding ontology instances.
- *Projection*. Sometimes not all the columns are required for the mapping. The unnecessary columns can simply be ignored. In order to do so, a projection on the needed columns can be performed.
- *Selection*. In some situations not all the records of a table correspond to instances of the mapped ontology class. Then a subset of the records must be extracted. To do so, selection conditions can be applied to choose the desired subset for the mapping.

It is off course possible to combine joins, unions, projections and selections for more complex mapping definitions. Values from the database can be copied as-is to the properties of instances in the ontology. However in many situations it is necessary to perform some transformations on the values using some function. R_2O allows the use of defined functions for this purpose, e.g. concatenation, sub-strings, arithmetic functions, etc.

The ODEMapster [5] system is the processor that exploits R_2O mappings, offering a query language that is a subset of SPARQL for conjunctive queries.

Continuous Queries for RDF Streams

SPARQL [21] is the W3C Recommendation for a query language over RDF. Even though SPARQL has been used to query RDF triples annotated with time constructs [4, 6] 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: Streaming SPARQL and C-SPARQL.

Streaming SPARQL

In [6] extensions for SPARQL are provided, so that the resulting language 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 Streaming SPARQL basically consists in adding the capability of defining time and tuple-based windows over streams which are defined in the FROM clause. The RDF stream is identified by a unique IRI after the STREAM keyword. This proposal also allows specifying windows on graph patterns, which complicates its evaluation semantics. Here is an example of a Streaming SPARQL query, that obtains the sensor temperature values sensed in the latest 30 minutes, every minute:

```
PREFIX fire:<http://www.ssg4env.eu/fire#>
SELECT ?sensor ?temperature
FROM STREAM <www.ssg4env/Temperature.srdf>
WINDOW RANGE 30 MINUTE SLIDE 1 MINUTE
WHERE { ?sensor fire:hasTempMeasurement ?temperature .}
```

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, Streaming SPARQL 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, Streaming SPARQL allows windows in group graph patterns and redefines the language semantics due to the introduction of timestamps.

C-SPARQL

C-SPARQL (Continuous SPARQL) [4] also works over RDF streams, sequences of triples annotated with non-decreasing timestamps. As in Streaming SPARQL, it defines both time or tuple-based sliding windows. C-SPARQL offers aggregates, such as COUNT, SUM, AVG, MIN and MAX. It also allows combining stored and streaming knowledge and also combining multiple streams. Here is an example of a C-SPARQL query, it obtains the temperature average of the values sensed in the last 10 minutes, every minute:

```
REGISTER QUERY AvergaeTemperature AS
PREFIX fire: <http://www.ssg4env.eu/fire#>
SELECT DISTINCT ?sensor ?average
FROM STREAM <www.ssg4env.eu/fire.srdf>
      [RANGE 10 MIN STEP 1 MIN]
WHERE { ?sensor fire:hasTempMeasurement ?temperature .}
AGGREGATE {(?average, AVG, {?temperature})}
```

Notice that both languages lack support for time windows with upper bounds different to now. Window-to-stream operators are also missing in these specifications. Streaming SPARQL provides an extended algebra for these features, but it is the C-SPARQL approach the one that allows clearly separating the stream management and query evaluation concerns, hence it will be the one that we will consider in our approach.

ONTOLOGY-BASED STREAMING DATA ACCESS

Querying streaming data and ontology-based access to stored data sources have already been studied by the research community and concrete proposals and software have been produced to deal with them. However there is still no bridging solution that allows connecting 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.

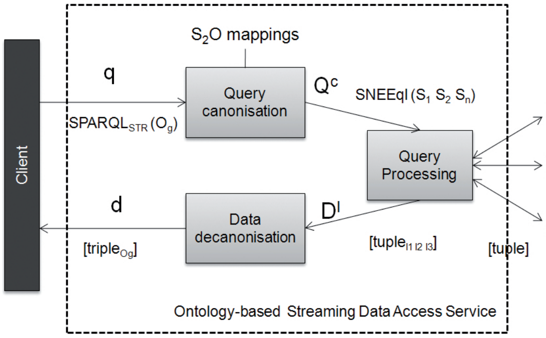

Figure 1. Ontology-based Streaming Data Access service

Our approach consists in creating an Ontology-based Streaming Data Access service, depicted in Fig 1. The service receives queries specified in terms of the classes and properties[1] of the ontology using extensions of SPARQL that support operators over RDF streams and windows ($\text{SPARQL}_{\text{STR}}$, see the Streaming Extensions to SPARQL section). Then in order to transform the query in terms of the ontology into queries in terms of the sources, a set of mappings must be specified. These mappings are based on the R_2O mapping language, which has been extended to support streaming queries and data, most notably window and stream operators (see the Streaming Extensions to R_2O section).This transformation process is called *query canonisation*, and the target is a continuous query language (e.g. SNEEql), that is expressive enough to deal with both streaming and stored sources, and to apply window, aggregates and window-to-stream operations.

After the continuous query has been generated, the query processing phase starts, and the processor will deploy distributed query processing techniques [12] to extract the relevant data from the sources and perform the required joins, etc. Note that the execution in sources such as sensor networks may include in-network query processing, pull or push based data delivery and other data source specific settings. The result of the query processing will be a set of tuples that will be passed to a *data decanonisation* process, which will transform these tuples to ontology instances.

As it can be seen, this approach requires several contributions and extensions to the existing technologies for continuous data querying, ontology-based data access and SPARQL query processing. This work focuses on a first stage that includes the process of transforming the SPARQL extended queries into queries over the streaming data sources using a language such as SNEEql as the target. In the next sections a description of the query and mapping extensions syntax and semantics will be detailed, and afterwards we will provide details of an implementation of this approach.

EXTENSIONS SYNTAX

In this section we introduce the language extensions to SPARQL, for RDF stream management, and to R_2O for the definition of stream-to-ontology mappings.

Streaming Extensions to SPARQL

As shown previously , C-SPARQL introduces extensions for the support of RDF streams. The language is expressive enough to support most of the constructs we require, including time and tuple windows, aggregates, query and stream registration, and joins between streaming and stored data. Moreover, as in C-SPARQL, we follow the approach of applying windows to streaming data and afterwards applying standard operators over the resulting non-streaming output [10]. This slightly extended C-SPARQL variation is named $\text{SPARQL}_{\text{STR}}$ in the rest of the paper.

Just as in [4] 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. In addition to the time width specified with the RANGE keyword, we introduce the possibility of specifying initial and final time boundaries for windows, using the TO keyword. We also add the NOW keyword, used to denote the current timestamp. Using these additions it is possible to specify more complex time ranges such as intervals in the past. The general form of the window range is [RANGE t_i TO t_f], where t_i and t_f are the lower and higher time boundaries respectively. Both boundaries are of the form NOW-t, where t is a number of time units. For example the window [RANGE NOW-2 d TO NOW-1 d] will take all triples registered between one and two days ago. A slide parameter can be specified using the STEP keyword and the time interval for the sliding window creation. Triple based windows are of the form [ROWS N] where N is the number of triples to be taken.

Streaming Extensions to R_2O

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. As it is explained in [5], R_2O includes a section in the mapping document that describes the database tables and columns,

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

dbscehma-desc. In order to support streams, R_2O has been extended to also describe the data stream schema. A new component called streamschema-desc has been created, as in the following example:

```
streamschema−desc
    name CoastalSensors
    has−stream SensorWaves
        streamType pushed
        documentation ”Wave measurements”
        keycol−desc measurementid
            columnType integer
        timestamp−desc  measuretime
            columnType datetime
        nonkeycol−desc  measureheight
            columnType float
        nonkeycol−desc measuretemperature
            columnType float
```

The description of the stream is similar to a table. An additional attribute streamType has been added, it denotes the kind of stream in terms of data acquisition. It can be a sensed stream, i.e. pull based arriving at some acquisition rate. Or it can be pushed, arriving at some potentially variable and/or unknown rate. Relations can also be specified just like tables in S_2O. 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. For the class and property mappings, the R_2O existent definitions can be used for stream schemas just as it was for relational schemas. This is specified in the conceptmap-def element:

```
conceptmap−def Wave
    virtualStream <http://virtualStreamIRI>
    uri−as
        concat(SensorWaves.measurementID)
    applies−if
       <cond−expr>
    described−by
        attributemap−def hasHeight
            virtualStream <http://virtualStreamIRI>
            operation constant
                has−column SensorWaves.measureheight
```

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, we have seen that at the moment of generating a SPARQL streaming query, an RDF Stream IRI is expected along with the window parameters. In this case the RDF Stream is virtual and its IRI can be 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.

SEMANTICS OF THE STREAMING EXTENSIONS

Now that the syntactic streaming extensions to SPARQL ($SPARQL_{STR}$) and R_2O(S_2O) have been presented, we introduce their semantics.

$SPARQL_{STR}$Semantics

The SPARQL extensions presented here are based on the formalisation of C-SPARQL [4], which are in turn based on the work described in [18].

RDF streams can be defined as sequences 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 $\mathbb{T}$:

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

where I, B and L are sets of IRIs, blank nodes and literals. Each of these pairs can be called a tagged triple. We can now define a time-based window as:

$$\omega_{time}(R, t_i, t_f, \delta) = \{(\langle s,p,o\rangle, \tau) \in R \mid \\ \delta\cdot k + t_i < \tau \leq \delta\cdot k + t_f, k \in \mathbb{N}\}$$

where t_i, t_f defines the window time range and δ is the time slide parameter. A window ω_{time} as defined above, is a set of tagged triples whose timestamp is between the initial and final time boundaries. Notice that a new window (i.e. bounded subset of triples) is created every δ time units.

For the triple-based window, we need to define first the function c that counts the items in R in a certain time range (t_i, t_f):

$$c(R, t_i, t_f) = |\{(\langle s,p,o\rangle, \tau) \in R \mid t_i < \tau \leq t_f\}|$$

A triple-based window, with n being the ROWS parameter, can be defined as:

$$\omega_{tuple}(R, n) = \{(\langle s,p,o\rangle, \tau) \in \omega_{time}(R, t_i, t_f) \mid \\ c(R, t_i, t_f) = n\}$$

We have provided a brief explanation of the semantics of $SPARQL_{STR}$. 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, exposed as a SNEEql endpoint, we need to transform the $SPARQL_{STR}$ queries into SNEEql queries. The formal semantics of SNEEql can be found in [7]. The next section describes the mapping from $SPARQL_{STR}$ to SNEEql.

Extended R_2O(S_2O) Semantics

We are particularly interested in answering unions of conjunctive queries (a subset of $SPARQL_{STR}$) over an ontological schema, and accessing the underlying data sources through mappings. In this section we will present how we can use the mapping definitions to transform the set of conjunctive queries into the internal query language SNEEql that is used to access the sources. This work is based on extensions to the ODEMapster processor [5] and the formalisation work of [8, 20].

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

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

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

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

where $\vec{x}$ is a tuple of distinct distinguished variables, $\vec{y}$ a tuple of non distinguished existentially quantified variables. The answer to this query consists in the instantiation of the distinguished variables [8]. For instance consider the following conjunctive query q_1:

$$\begin{aligned} q_1(x) \leftarrow & WindSpeedMeasurement(x) \\ & \wedge measuredBy(x,y) \wedge SeaSensor(y) \end{aligned}$$

It requires all instances x that are wind speed measurements captured by sea 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 $SeaSensor(y)$.

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 of tuples $q^{\mathcal{I}}_{\vec{x}} \in \Delta^{\mathcal{I}} \times \cdots \times \Delta^{\mathcal{I}}$ that substituted to $\vec{x}$, make the formula $\exists \vec{y}.\varphi(\vec{x}, \vec{y})$ true in $\mathcal{I}$ [20, 15]. 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 $C(f^{Id}_C(\vec{x})) \rightsquigarrow \Phi_{S_1,\ldots,S_n}(\vec{x})$ mapping assertion describes how to construct the concept C from the source streams (or relations) $S_1, \ldots, S_n$. The f^{Id}_C function creates an instance of the class C, given the tuple $\vec{x}$ of variables returned by the Φ expression. In concrete this function will construct the instance identifier (URI) from a set of attributes from the streams and relations. In this case the Φ expression has a declarative representation of the form:

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

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

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

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

$\Phi_{S_1,\ldots,S_k}, \Phi_{S_{k+1},\ldots,S_n}$ 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 p^{Join} term is the set of predicates that denotes the join between the streams and relations $S_1, \ldots, S_n$.

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

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

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^{Trf}_A.

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 [5] (that is based on the PerfectRef algorithm of [8]) 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 $Q_{\mathcal{O}}(\vec{x})[t_i, t_f, \delta]$ is a conjunctive query with a window operator (where t_i, t_f is the time range and δ is the slide) in order to narrow the data set according to a given criteria. For a query $Q_{\mathcal{O}}$ of the form:

$$C_1(x) \wedge R(x,y) \wedge A(x,z)[t_i, t_f, \delta]$$

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

$$\begin{aligned} \lambda(\Phi_{S_1,\ldots,S_n}(\vec{x})[t_i, t_f, \delta]) = \pi_{p^{Proj}}(\bowtie_{p^{Join}} & (\sigma_{p^{Sel}}(\omega_{t_i,t_f,\delta}S_1) \\ & , \ldots, \sigma_{p^{Sel}}(\omega_{t_i,t_f,\delta}S_n))) \end{aligned}$$

The expression denotes first a window operation $\omega_{t_i,t_f,\delta}$ over the relations or streams $S_1, \ldots, S_n$, with t_i, t_f and δ being the range and slide. A selection $\sigma_{p^{Sel}}$ is applied over the result, according the conditions defined in the mapping. A multiple 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.

IMPLEMENTATION AND WALKTHROUGH

The presented approach of providing ontology-based access to streaming data has been implemented as an extension to the ODEMapster processor [5]. This implementation generates queries that can be executed by the SNEE in-network or out-of-network streaming query processor, whose SNEEql query language is presented in [7].

Consider the following example, a stream `windsamples` of wind sensor measurements and a table `sensors`:

```
windsamples: (sensorid INT PK,ts DATETIME PK,speed FLOAT,
              direction FLOAT)
sensors: (sensorid INT PK,sensorname CHAR(45))
```

And consider the following ontological view:

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

Then we can define an S_2O mapping that splits the `windsamples` stream tuples into instances of two different concepts *WindSpeedMeasurement* and *WindDirectionMeasurement*. Here is an extract of the S_2O mapping concerning the *WindSpeedMeasurement*.

```
conceptmap-def WindSpeedMeasurement
 virtualStream <http://ssg4env.eu/SensorReadings.srdf>
 uri-as
   concat('ssg4env:WindSM_',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('ssg4env:Sensor_',sensors.sensorid)
 described-by
  attributemap-def hasSensorid
   operation constant
     has-column sensors.sensorid
```

The mapping extract here defines how to construct the *WindSpeedMeasurement* (*WindSM*) and *Sensor* class instances from the `windsamples` stream and the `sensors` table: $\Psi_{WindSM} \rightsquigarrow \Phi_{\texttt{windsamples}}$ and $\Psi_{Sensor} \rightsquigarrow \Phi_{\texttt{sensors}}$. In the case of the *WindSpeedMeasurement* the function f^{Id}_{WindSM} produces the URI's of the instances by concatenating the `sensorid` and `ts` attributes. Now we can pose a query over the ontology using SPARQL$_{\text{STR}}$ for example to obtain the wind speed measurements taken in the last 10 minutes.

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

A class query atom *WindSpeedMeasurement(x)* and a datatype property atom *hasSpeed(x,z)* can be extracted from the SPARQL$_{\text{STR}}$ query. The window specification $[t_i = now - 10, t_f = now, \delta = 1, unit = minutes]$ is also obtained. As it is defined in the S_2O mapping the *WindSpeedMeasurement* 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_2O 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 the following[2]:

```
SELECT RSTREAM concat('http://ssg4env.eu#WindSM',
              windsensor.id,windsensor.ts )
as  id ,( windsamples.speed ) as  speed
FROM windsamples[FROM NOW - 10 MINUTE]
```

The results will be transformed into tagged triples, instances of the class *WindSpeedMeasurement*.

CONCLUSIONS AND FUTURE WORK

We have presented an approach for providing access to streaming data based on ontologies, by extending the R_2O mapping definition language, the ODEMapster processor and the C-SPARQL language. We have presented the SPARQL$_{\text{STR}}$ extensions to C-SPARQL for RDF streams and the S_2O extensions to R_2O for stream mappings. Then 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. Finally the prototype implementation 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 latest 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, combination of time and tuple windows and joins between streams. We also plan to adapt our query rewriting approach to more recent and promising works such as [19]. We are also aware of the need of optimising the generated queries using techniques from sensor networks and continuous data approaches [1, 2, 11]. It is also our goal to provide a characterisation 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

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

Basis	Extension	Syntax	Semantics
C-SPARQL	Window variable upper boundary	`RANGE` t_i `TO` t_f	$k \,.\, \delta + t_i < \tau \leq k \,.\, \delta + t_f$
	Syntax for current timestamp	`NOW`	$\tau = now$

R$_2$O	Stream definitions in mappings	`streamschema-desc`
		`has-stream`
		`timestamp-desc`
			Streaming data types:
			`StreamOf [Data]`
			`TaggedTuple,`
			`Window.`

	Virtual RDF Stream IRIs	`virtualStream <IRI>`	`extentName`
ODEMapster	Window translation in the processor, classes, object and dataype attributes	-	$\lambda(\Phi_{S_i}(\vec{x})[t_i, t_f, \delta])$ $\omega_{t_i, t_f, \delta} S_i$

Table 1. Extensions and additions to R$_2$O, ODEMapster and C-SPARQL

streams and/or sensor networks acquisitional streams considering time and triple windows; iii) considering quality-of-service requirements for query optimisation and source selection during the integration.

The present work can be just considered as a first step to our goal of providing an ontology-based integration platform for continuous heterogeneous data sources. Therefore we will address the problems of heterogeneity, distributed query processing and integration as part of this research track.

ACKNOWLEDGEMENTS

This work has been supported by (Change for Blind Review). We also thank (Change for Blind Review), for their valuable ideas and suggestions.

REFERENCES

1. 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, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR 2005*, 2005.

2. A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. In M. Garofalakis, J. Gehrke, and R. Rastogi, editors, *Data Stream Management*. 2006.

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

4. D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-sparql: A continuous query language for rdf data streams (to appear). In *(IJSC)*, 2010.

5. J. Barrasa, Óscar Corcho, and A. Gómez-Pérez. R2O, an extensible and semantically based database-to-ontology mapping language. In *SWDB2004*, pages 1069–1070, 2004.

6. A. Bolles, M. Grawunder, and J. Jacobi. Streaming SPARQL - extending SPARQL to process data streams. In *ESWC 08*, pages 448–462, 2008.

7. C. Y. Brenninkmeijer, I. Galpin, A. A. Fernandes, and N. W. Paton. A semantics for a query language over sensors, streams and relations. In *BNCOD '08*, pages 87–99, 2008.

8. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. DL-Lite: Tractable description logics for ontologies. In *AAAI 2005*, pages 602–607, 2005.

9. S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: continuous dataflow processing. In *SIGMOD '03*, pages 668–668, 2003.

10. E. Della Valle, S. Ceri, D. Braga, I. Celino, D. Fensel, F. van Harmelen, and G. Unel. Research chapters in the area of stream reasoning. In *SR2009*, pages 1–9, 2009.

11. I. Galpin, C. Y. Brenninkmeijer, F. Jabeen, A. A. Fernandes, and N. W. Paton. Comprehensive optimization of declarative sensor network queries. In *SSDBM 2009*, pages 339–360, 2009.

12. D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

13. D. Le-Phuoc and M. Hauswirth. Linked open data in sensor data mashups. In *SSN09*, pages 1–16, 2009.

14. M. Lenzerini. Data integration: a theoretical perspective. In *PODS '02*, pages 233–246, 2002.

15. L. Lubyte and S. Tessaris. Supporting the development of data wrapping ontologies. In *4th Asian Semantic Web Conference*, December 2009.

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

17. K. Page, D. D. Roure, K. Martinez, J. Sadler, and O. Kit. Linked sensor data: RESTfully serving RDF and GML. In *SSN09*, pages 49–63, 2009.

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

19. H. Pérez-Urbina, I. Horrocks, and B. Motik. Efficient query answering for owl 2. In *ISWC 2009*, pages 489–504, 2009.

20. A. Poggi, D. Lembo, D. Calvanese, G. D. Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. *J. Data Semantics*, 10:133–173, 2008.

21. E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF, W3C recommendation. Technical report, World Wide Web Consortium, January 2008.

22. S. S. Sahoo, W. Halb, S. Hellmann, K. Idehen, T. T. Jr, S. Auer, J. Sequeda, and A. Ezzat. A survey of current approaches for mapping of relational databases to RDF. W3C, January 2009.

23. J. Sequeda and O. Corcho. Linked stream data: A position paper. In *SSN09*, pages 148–157, 2009.

24. D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *SIGMOD '92*, pages 321–330. ACM, 1992.

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