this title must be changed

Jean-Paul, Oscar

Ontology Engineering Group, Departamento de Inteligencia Artificial,
Facultad de Informática,Universidad Politécnica de Madrid, Spain
jpcalbimonte@fi.upm.es,ocorcho@fi.upm.es

Abstract. Las pipa apsdadjfjadlfjlksdjfsdf dsfsdfdsfdsf dsf dfdsfdsfdsfsdfdsf dfsdfdsfdsf ds fds fsdafsddddddddddfa asdf dsfdsfdsfsd blablablabal blablablabalblablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabalblablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabalblablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabalblablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabalblablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal blablablabal

Keywords: nice keywords, keywords

1 Introduction

Latest advances in wireless communications and sensor technologies have opened the way for deploying networks of interconnected sensing devices capable of ubiquitous data capture, collection and processing. Sensor networks 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.

Streaming data sources coming form sensor networks is typically accessed through query processors that use declarative continuous query languages. Managing streaming data differs significantly from classical static data. Streaming data is potentially infinite and transient, with tuples being constantly added. Therefore queries on this data are also continuous, i.e. their results are updated regularly as time passes [TGN+92].

Sensor networks characteristics also raise several challenges for the research community, such as the integration of the data collected by these sensors. Client applications such as mashups require a suitable platform to access and combine heterogeneous data coming from different sensor networks and other data sources such as databases. Furthermore, these applications require accessing this

data in terms of a uniform schema that hides the diverse and internal data representations of each source.
Semantic technologies and ontologies have been successfully used in the latest years for data integration solutions, particularly in the area of databases. The problem of integrating these sources involves not only granting access to information but also providing mechanisms for interpreting and processing the data consistently, overcoming syntactical and semantic heterogeneity.
Although proposals for streaming data access, ontology-based integration and query languages for RDF streams have been proposed and implemented, to the best of our knowledge there is no framework yet that integrates these different pieces into a coherent solution for integration of distributed heterogeneous streaming and static data sources through ontological model. In this paper we focus on the first step to achieve this level of integration, that is providing ontology-based access to continuous data sources, including sensor networks. It is organised as follows: in Section **??** we introduce the previous works and foundations of our approach. In Section **??** we present the syntactic extensions for RDF streams SPARQL operators and streams-to-ontology mappings. The semantics of these extensions are detailed in Section **??** and a first implementation of the execution of the streaming data access approach is explained in Section **??**. Then we present the future works in Section **??** before concluding.

2 Previous Works

The following sections describe the state of the art in streaming data access, ontology-based data access and integration, and query languages for RDF streams. We focus on the works and technologies that we used, extended and integrated for our approach: R_2O, ODEMapster and SNEEql.

2.1 Streaming Data Access

Managing streaming data differs significantly from classical static data. Streaming data is potentially infinite and its items are constantly added, therefore queries on them are also continuous, i.e. their results are updated regularly as time passes [TGN+92]. On the other side classical database systems deal with mostly static data, with lower insert rates and queries that retrieve the state of the data at the current time. While in database systems datasets are queried several times, streaming data is transient and usually the latest registered information is used for queries, and the old items are less relevant. These items are either discarded or archived. Sliding windows provide a suitable way of selecting the stream data that a continuous query will handle [BBD+02]. For example, a sliding window selecting tuples received in the last 30 minutes can be used to process only a desired subset of the incoming data. Otherwise it would be impossible to compute aggregates such as averages, maximums and minimums, as streaming data is potentially infinite.

Several Data Stream Management Systems (DSMS) or systems managing continuous data, have been designed and built in the past years, such as Tapestry [TGN+92], Tribeca [Sul96], TelegraphCQ [CCD+03], Aurora [ACC+03, CCC+02], STREAM [ABB+04], Borealis [AAB+05], NiagaraCQ [CDT+00], Gigascope [CJS+02], CAPE [RDS+04], TinyDB [MFH+05], Cougar [YG02] and SNEE [GBJ+08].

Two main types of streaming data sources can be identified: event streams and acquisitional streams [Galpin]. In event streams, the system does not have control over the data arrival rate, which is often high and potentially unknown. For acquisitional streams, it is possible to control when data is obtained from the source, generally a sensor network. Several restrictions must be considered in this case, such as the low energy resources, limited computing power and storage capabilities. In order address these issues, research has produced Sensor Networks Query Processing engines such as TinyDB [MFH+05], Cougar [YG02] and SNEE [GBJ+08]. 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. These engines must also consider several optimisation techniques in order to efficiently gather the information from the sensor nodes. Architectures for query optimisation in these constrained scenarios have surfaced [GBJ+09, MFH+05], showing that even with such limitations it is still possible to use rich and expressive declarative query languages.

The SNEEql [BGF+08] language for querying streaming data sources is based on the widely known CQL [ABW06], although it provides greater expressiveness in queries, including streams and relations, time and tuple windows, blocking and non-blocking operators, and pull and push based streams. In the SNEEql data model the basic compound tuple type is defined as a set of typed attributes. A tagged tuple in a stream is a tuple that includes a named attributes called tick. The tick attribute is a timestamp that indicates when the tuple entered the stream. This attribute is essential to define the semantics of stream operators in these languages: two tuples having the same tick are considered to have entered the stream at the same instant. SNEEql queries over streams are of the form: SELECT a1 . . . an FROM s WHERE p where a1 . . . an is a project list, s denotes a stream of tagged tuples, and p is a predicate [BGF+08]. The result of the execution of a stream query is a stream of tagged tuples. Stream systems require operators to limit streams to finite structures in order to process only a smaller subset of data. Windows are widely used to perform such transformations from streams to bounded extents such as relations[BGF+08]. The most common stream-to-relation operators are time windows. A time window usually takes a time interval as parameter and returns a bag of tuples of the stream whose timestamp falls in that specified time interval. It can be said that a time window returns a finite snapshot of the stream in the window interval. SNEEql defines a window as a pair consisting of a bag of tuples and a tick that indicates when the window was created. A stream is a potentially infinite sequence of tagged tuples

or windows. Window queries are of the form: SELECT a1 . . .an FROM w1 . . . wm WHERE p where a1 . . . an is a projection list of attributes of the stream, w1 . . .wm is a list of window definitions, and p is a predicate. [BGF+08] The result of the execution of a window query is a stream of windows. A window on as stream can be specified as follows: s [FROM t1 TO t2 SLIDE int unit]

where FROM t1 TO t2 indicates a time interval. The slide parameter indicates the frequency of the window creation in time units or rows. A window on a table can be specified as follows: t [SCAN int timeUnit] where t is a table. This parameter indicates the frequency of the table scan in time units. Window-to-stream operators are also required for stream query languages. Based on the CQL [ABW06] operators, SNEEql provides the ISTREAM, DSTREAM and RSTREAM operators. The ISTREAM operator appends tuples that were not on the previous window, to the output stream. DSTREAM appends tuples that were deleted form the previous window. RSTREAM appends all tuples from the previous window to the output stream. Queries over streams usually require aggregation functions. Querying for sums, averages, maximums and minimums is quite common. Streaming query languages such as CQL and SNEEql provide support for this kind of operators, for either streams or relations. Another feature of some query languages is the possibility of specifying quality of service requirements. SNEEql allows for instance specifying the desired acquisition rate and delivery time: ACQUISITION RATE = 3s ; DELIVERY TIME = 5s These parameters can be used by the query engine to optimise the query plans and these can be extremely useful in the context of sensor networks because of the limited processing and power resources.

2.2 Ontology-based Data Access

The realisation 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 is currently preserved in relational databases and therefore it has become a need to generate Semantic Web content from them [SHH+08]. 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 the domain of interest [PLC+08]. This is the goal of Ontology-based data access (OBDA). Most of the approaches attempt to provide some kind of mapping from a relational entity to a concept in an ontological model. Many problems arise then, as they have to deal with model mismatch issues, query interpretation, semantic reasoning, etc. Some of the proposed systems use their own languages [BCG05, BC07] to define the mappings between ontology concepts and roles and their corresponding tables in the relational models. Others rely on SPARQL [PS08] extensions and SQL [SQL92] expressions to define these mappings [PLC+08, SSW07, EM07, PC06]. OBDA systems use these kinds of mappings between the global ontology and the different data sources. These mappings allow constructing queries over the ontology (e.g. in SPARQL) which are then re-written to the data sources query language (typically SQL). Then the results are converted to RDF which

is the result that is returned to the user [SHH+08]. One of the common approaches is to first generate a syntactical translation of the database schema to an ontological representation. Although the resulting ontology has no real semantics behind, it may be argued that it is a first step through an ontology model and that ontology alignment could be used later to map it to a real domain ontology [LT09, MBR01]. Variations of ontology generation and syntactic mappings have been presented in [SSW07,Cer08,PC06,Pru07]. 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. The Virtuoso [EM07] 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 [BC07], 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 [PLC+08] is a DL reasoner for ontologies whose data is accessed through mappings in an external source, i.e. an ODBA-enabled reasoner. It works over the DL-LiteA 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).

2.3 R_2O and ODEMapster

R2O (Relational to Ontology) [BCG05] is a mapping definition language that defines a relationship between an ontology and local relational schemas. In R_2O each element in the global ontology is characterised in terms of a query over the relational sources. The R2O language is XML-based, independent of any specific DBMS and allows complex mapping expressions between ontology and relational elements. R2O mapping assertions can be created either manually or with the help of a mapping tool. The mapping definition could also be used to perform validation of the database integrity by checking it against the ontology restrictions and axioms. R2O specifically considers Concepts, Attributes and Relations in an ontology. They are described in terms of selections and transformations over database tables and columns. In order to be able to describe these mappings, R2O defines the ontology elements and also the database elements using its own XML notation. R2O covers a wide set of mapping cases common in relational to ontology situations. Depending on the scenario, finding correspondences between ontological elements and relational elements can be straightforward or complex. R2O is designed to cope with the following cases: a) A database table maps one concept 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. b) A single database table is mapped to more than one concept in the ontology, and for each row a single instance of each concept is generated. In this

case some columns will be mapped to a concept attributes while other columns will be mapped to other concept attributes. c) A single database table is mapped to more than one concept in the ontology, and multiple instances can be generated for each concept. 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 tables and columns to concepts and attributes often requires performing some operations on the relational sources. Several cases are handled by R2O and detailed below.

- Direct Mapping. When the relational table maps an ontology concept and the column values are used to fill the attribute values of the ontology instances. Each table record will generate a concept instance in the ontology.
- Join/Union. In some occasions a single table does not correspond alone to a concept, but it has to be combined with other table or tables to match the ontology concept. 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 (e.g. a SELECT).
- Selection. In some situations not all the records of a table correspond to instances of the mapped ontology concept. Then a subset of the records must be extracted. To do so, selection conditions can be applied to choose the right subset for the mapping.

It is of course possible to combine joins, unions, projections and selections for more complex mapping definitions.Values from the database records can be copied as-is to the attributes of instances in the ontology. However in many situations it is necessary to perform some transformations on the values using some function. R2O allows the use of defined functions for this purpose, e.g. concatenation, sub-strings, arithmetic functions, etc.

The ODEMapster [BG06] system is an ontology-based data processor that provides access to data in a relational database through declarative queries over an ontology. The ODEMapster System uses the R2O mapping language to define how the ontology elements must be accessed in terms of the relational tables. The query language used by ODEMapster over the ontology is ODEMQL, an XML-based language that is at least as expressive as SPARQL for conjunctive queries [Fus08].

2.4 Ontology-based data integration

Data integration can be defined as the process of providing unified and transparent data access to multiple and heterogeneous data sources [Len02]. The problem of integrating heterogeneous data sources involves not only providing access to information but also providing means of interpreting and processing the data in a consistent manner. When dealing with different sources it is often complicated to establish the meaning of the incoming data. And if the meaning cannot

be clearly identified, then it is impossible to pair information for the different sources [RAY+00]. Problems of this kind are mainly due to confusion in term meanings and naming conflicts. Added to syntactical conflicts, the semantic integration challenge is hard to tackle. We can identify the main issues of data integration as [WVV+01]:

- Incompatibility in communication and protocols, across different data management systems
- Syntactical heterogeneity: differences in data model representation and structure, incompatibility of data values.
- Semantic heterogeneity: differences in naming and abstraction level, homonyms and synonyms in schemas.

. Ontologies provide a means of explicitly specifying the meaning of the information that will be interchanged between systems. In that way it can help achieve semantic interoperability: regardless of the source's data syntax or semantics, we can map it to a known ontology and access the data in terms of the mediated ontological view [WVV+01]. As a consequence semantic declarative queries can be written in terms of the mediated schema. It has been acknowledged that ontologies may be useful to solve this semantic heterogeneity problem as it is described in [WVV+01, DH05]. We can mention several systems and prototypes based on ontologies for data integration support: OBSERVER [MIK+00], SIMS [ACH+93], Carnot [HJK+93], DWQ [CDL+98], PICSEL [GLR00], MOMIS [BBV+07]. The main role and purpose of ontologies in an integration system is to explicitly state the semantics of the data sources so that it can be possible to identify and establish semantic relationships between these sources. For instance, if a data source stores person information and another source stores student information, the ontology may represent both concepts, with person subsuming student. This information can later be exploited by query integrators to coherently perform joins or unions over these sources. Data sources can be relational databases, data files, xml files or any other storing media. Most of the work on data integration has been centred on relational databases, as it is the most widespread way of storing large volumes of information. In any case the ultimate goal is to allow querying the different sources through a uniform interface, hiding the underlying heterogeneous schemas. Therefore the applications querying the integrated system must only focus on the information they wish to obtain, leaving to the integration system the work of searching for the information in the different sources, clearly simplifying the development of upper layer applications on top of the integration infrastructure. Research on the subject has produced many approaches to building such integration systems. We mention two main alternatives: the *virtualisation* and *materialisation* approaches [IKK05]. In the virtualisation approach queries are posed over the mediated schema and a mediator component identifies the sources that will be needed to produce the answer. Then a series of appropriate sub-queries are automatically written for these sources and executed. The results of each source are retrieved, post-processed and transformed to the mediated schema and returned to the caller application. This alternative allows answering queries on demand, even if

the update rate of data is high and the queries are potentially arbitrary. As the queries are rewritten dynamically for each of the sources, it is possible to retrieve any portion of data exposed through the mediated schema. On the other hand the transformations on the queries and the processing of the data from the sources to the mediated schema may be complex and incur in performance issues. Another potential problem is data source availability, as the queries are performed live; if the source is unreachable then the queries may not be able to be completed. Even if the source is available, latency caused by networking or connection problems may increase the query response time. The materialisation approach differs from the virtualised one in that it extracts and transforms the data from the sources periodically, instead of performing on-the-fly conversions each time a query is being executed. Instead, relevant data is first identified, and extracted from the multiple data sources in a batch process operation. The extracted data is then stored in a data warehouse where it can be accessed by the external applications. The data warehouse can then be updated periodically depending on the requirements. Under this approach, the time consumed in retrieving the data from the sources is moved to the off-line phase of updating the centralised data warehouse. Therefore when the external applications query the integration system, results can be accessed almost immediately. However the data may be stale in occasions, and it may not be possible to access all possible pieces of data, as the warehouse only stores selected materialised views of the original sources. In the case of the virtualised approach, a mediator component is introduced as a major feature of the integration architecture. This mediator is in charge of transforming the original query into sub-queries for the sources and then transforming the results back to the mediated schema. This process of transformation is performed thanks to mapping definitions that establish relationships between the mediated and source schemas.

There are three main alternatives for defining these mappings [Len02, IKK05]: *Global-as-view (GAV)*, *Local-as-view (LAV)* 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. If our system $\mathcal{I}$ is represented as: $\mathcal{I} =< \mathcal{G}, \mathcal{S}, \mathcal{M} >$, where $\mathcal{G}$ is the global schema, $\mathcal{S}$ is a source schema and $\mathcal{M}$ is the mapping between $\mathcal{G}$ and $\mathcal{S}$, then the mapping assertions in $\mathcal{M}$ will be a set of elements of the form: $s \rightsquigarrow q_{\mathcal{G}}$ Where s is an element of $\mathcal{S}$ and $q_{\mathcal{G}}$ is a query over the global schema $mathcalG$. This approach is useful if the global schema is well established and if the sources suffer modifications constantly. Notice that changes in the sources do not affect the global schema. However, the query processing is not obvious, as it is not explicitly stated how to obtain the data in the mapping definition. Query rewriting techniques such as query answering using views can be used in this approach [Hal01]. Examples of LAV based works are DWQ [CDL+98], InfoMaster [GKD97], Information Manifold [Lev98], PICSEL [GLR00]. The other approach, GAV, defines the mappings in the opposite way. The global schema elements are represented as views over the source schemas. The mapping definition explicitly defines how to query the sources to obtain the desired information in terms of the global schema. Following the system rep-

resentation $\mathcal{I} =< \mathcal{G}, \mathcal{S}, \mathcal{M} >$, in the GAV approach the mapping assertions are elements of the form: $g \rightsquigarrow q_{\mathcal{S}}$ Where g is an element of the global schema $\mathcal{G}$ and is expressed as a view $q_{\mathcal{S}}$, a query on the source schema. The advantage is that the mapping itself already indicates how to query the sources to obtain the data, so the processor can directly use this information to perform the query rewriting. The main disadvantage is that in case of changes or additions on the sources -e.g. a new source is added- then the global schema may suffer changes and this may affect other mapping definitions. Examples of GAV based works are SIMS [ACH+93], TSIMMIS [GHI+97], Carnot [HJK+93], Gestalt [RS98], MOMIS [BBV+07], IBIS [CCDG+03], DIS@DIS [CDL+03]. For the sake of completeness we can mention the GLAV (Global-Local-as-view) approach, which is a generalisation of the previous two. This formalism allows more expressive mappings than LAV and GAV combined, and it has been shown to reach the limits of tractability of these description languages [FLM99].

2.5 Continuous Queries for RDF Streams

As it is described in Section **??** query languages for data streams have been proposed and implemented in the last years. These languages borrow much of the relational query languages such as SQL. In ontology-based integration solutions, 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 [DCB+09, GGK+07]. SPARQL [PS08] is the W3C Recommendation for a query language over RDF. SPARQL has been used to query RDF triples annotated with time constructs such as timestamps [BBC+09,BGJ08] can be used to represent data coming from streaming sources. However SPARQL currently lacks the necessary operators to effectively query streaming data. In [BGJ08] extensions for SPARQL are provided, so that the resulting language is able to handle RDF based data streams. The semantics of this extension is also provided as well as the algorithm to map the language extension to the extended algebra. They call this extended language Streaming SPARQL. The extended grammar of Streaming SPARQL basically consists in adding the capability of defining windows over streams which are explicitly stated using the STREAM keyword. The windows are defined in the FROM section, using the WINDOW keyword. Both time-based and tuple based windows are supported. For time-based windows the RANGE keyword allows specifying the window size in time units. A SLIDE parameter can be specified to indicate the frequency of the window creation. For tuple-base windows the ELEMS keyword is used instead of RANGE. Here's an example of a Streaming SPARQL query, it 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
WHERE { ?sensor fire:hasTempMeasurement ?temperature .}
```

Another extension to SPARQL for continuous queries is C-SPARQL (Continuous SPARQL) [BBC+09]. C-SPARQL also works over RDF Streams, sequences

of triples annotated with non-decreasing timestamps. As in Streaming SPARQL, it defines sliding windows, time or tuple-based. C-SPARQL offers aggregates, such as COUNT, SUM, AVG, MIN and MAX. It also allows combining static and streaming knowledge and also combining multiple streams. Here's an example of a C-SPARQL query, it obtains the temperature average of the values sensed in the last 10 minutes:

```
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 neither language supports time windows with other upper bound different than the current timestamp (now). Relation-to-stream operators are also missing in these specifications. Only Streaming SPARQL provides an extended algebra for these streaming features. C-SPARQL still lacks any formal description of the semantics of the language.

3 Approach

As it has been seen, accessing streaming data and integrating data sources through ontologies 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 to the following requirements:

- establishing mappings between global ontological models and streaming data source schemas.
- accessing streaming data sources through queries over ontology global models.
- integrating streaming and stored data sources through an ontological unified view.
- combining data from event-based streams and/or sensor networks acquisitional streams considering time and tuple windows.
- considering quality-of-service requirements for query optimisation and source selection during the integration.

Our approach consists in creating a Semantic Integrator service that is in charge of receiving requests over a global ontological view that can be transformed into queries to multiple, distributed and heterogeneous acquisitional or event-based stream sources or stored sources. The results of these queries can be integrated following a distributed query plan and returned as triples in terms of the global ontology. The approach is depicted in Fig **??**.

The queries are expected to be specified in terms of the concepts and roles of the ontology using a declarative language (i.e. SPARQL) with extensions that support operators over RDF streams and windows. Then each source may have its own ontological representation, generated automatically from the original

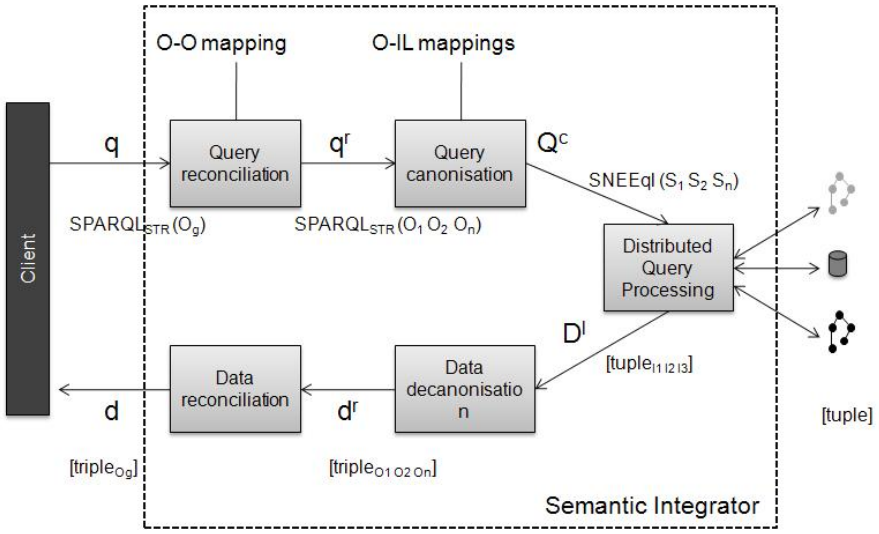

Fig. 1. Semantic Integrator service

schema or not. In that case an ontology-to-ontology mapping should be performed at this point, a process that we call query reconciliation. This step simply transforms the query over the global ontology to a query over the set of local ontologies using ontology mapping assertions. Then in order to transform the query in terms of the local ontologies to queries in terms of the sources, another set of mappings must be specified beforehand. These mappings are based on the R2O mapping language but additionally include support for streaming queries and data, most notably windows and stream operators. This transformation process is called query canonisation, and the target query language is SNEEql, as it is expressible enough to deal with both streaming and stored sources.

After the SNEEql query as bee generated, the distributed query processing phase starts, creating a distributed query plan that indicates how the different sources will be accessed and how the data will be joined and combined using the available operators. These operators will include wrappers over the different possible storage resources (e.g. sensor network, database, etc). The result of the distributed query processing will be a set of tuples that will be passed to a data decanonisation process that will transform these tuples to instances of the local ontologies. These in turn will be transformed to instances in terms of the global ontology by the subsequent data reconciliation stage.

As it can be seen, this approach requires several contributions and extensions to the existent technologies for continuous data querying, ontology-based integration and distributed query processing. This paper 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 a target. In the next sections a description of the query and mapping extensions syntax and

semantics will be detailed and afterwards we will provide preliminary results of an implementation of this approach.

4 Extensions Syntax

In this section we introduce the extensions to SPARQL for RDF Streams and then the extensions for R2O for stream mappings.

4.1 Streaming Extensions to SPARQL

As exposed in Section **??** C-SPARQL introduces extensions for RDF streams support, mainly time and tuple based windows and registration of queries and streams. We further extend and modify C-SPARQL as detailed below in the syntax specification. The extensions borrow syntax from both CQL and SNEEql languages.

Just as in [csparql] 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. A Stream is specified using the STREAM clause in the FORM part of the query. The stream itself is identified with an *IRI* (Internationalized Resource Identifier) that uniquely identifies it. Windows are specified using square brackets [] and the RANGE TO keywords for specifying the time based window width.For instance the expression [RANGE 10 h] applied to a stream will take the triples registered in the last hour. The NOW keyword can be used to specify the current timestamp. The NOW keyword can be used to specify intervals in the past, in conjunction with the TO keyword. 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 TimeUnit, where t is a number and TimeUnit some time unit of measurement. For example the window [RANGE NOW-2 d TO NOW-1 d] will take all triples registered between one and two days ago. The higher boundary can be ignored if we want it to be the current timestamp (i.e. TO NOW-0). A slide parameter can be specified using the STEP keyword and the time interval for the sliding window creation, e.g. [RANGE 30 m STEP 5m] will take triples registered 30 minutes ago, every 5 minutes. Tuple based windows are of the form [ROWS N] where N is the number of triples to be taken.

4.2 Streaming extensions to R2O

The Semantic Integrator requires a mapping document that describes how to transform the data source elements to ontology elements. The mapping document is an extended version of R2O. As it is explained in [**?**], R2O includes a component in the mapping document that describes the database tables and columns, dbscehma-desc. In order to support streams, R2O 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 name SensorWaves
  streamType pushed
  documentation "Wave measurements"
  keycol-desc
    name measurementID
    columnType integer
  timestamp-desc
    name time
    columnType timestamp
  nonkeycol-desc
    name mesheight
    columnType float
  nonkeycol-desc
    name mestemperature
    columnType float
```

The description of the stream is similar to a table's. 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 variable potentially unknown rate. Relations can also be specified just like tables in R2O. Just 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 concept and attribute mapping, the R2O 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
  name Wave
  identified-by SensorWaves.measurementID
  uri-as
    <transformation>
  applies-if
    <cond-expr>
  described-by
    attributemap-def
      name height
      selector
      aftertransform
      constant
        arg-restriction const-val
          has-column SensorWaves.mesheight
```

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). As it has been seen, R2O requires some changes to support creating mapping documents for stream schemas. It is expected that common and simple mappings in the R2O language can be defined using the ODEMapster tool, enhanced with streaming data support.

5 Semantics of the Streaming Extensions

Now that the syntactic extensions to SPARQL for streams have been presented, we introduce their semantics. We also provide the semantics of SNEEql and finally the semantics of the mapping expressions of the R2O extensions.

5.1 SPARQL Streamings Extensions Semantics

The SPARQL extensions presented here are based on the formalisation of C-SPARQL [c-sparql] which are in turn based on the work of [Gutierrez]. RDF triples $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$, where I, B and L are sets of IRIs, blank nodes and literals. In a triple, s, p and o are the subject, predicate and object respectively. SPARQL graph pattern expressions semantics require the introduction of the concept of a mapping function $\mu : V \rightarrow T$. V is a set of variables and T is defined as $I \cup B \cup L$. The function $\mu(t)$, for a triple pattern t, denotes the triple obtained by replacing all the variables in t as defined by μ. Then the domain of μ is $dom(\mu)$,the subset of V where μ is defined.Two mappings μ' and μ'' are said to be compatible if for all $x \in dom(\mu') \cap dom(\mu'')$, then $\mu'(x) = \mu''(x)$. Basic operators for composition of mappings are join, union and difference (Ω_1 and Ω_2 are sets of mappings):

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible}\}$$
$$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}$$
$$\Omega_1 \backslash \Omega_2 = \{\mu \in \Omega_1 \mid \forall \mu' \in \Omega_2, \mu \text{ and } \mu' \text{ are not compatible}\}$$

The left outer-join operator can be derived from above:

$$\Omega_1 ⟕ \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$$

With the operators and data types above, the semantics of graph pattern expressions is defined as a function $[[\cdot]]_D$ that takes a pattern expressions and returns a set of mappings, for and RDF dataset D. The evaluation of $[[\cdot]]_D$ is defined recursively below. Let t be a triple pattern, $var(t)$ the set of variables occurring in t, P_1, P_2 graph patterns and D an RDF dataset over T:

(1) $[[t]]_D = \mu \mid dom(\mu) = var(t) \wedge \mu(t) \in D$
(2) $[[(P_1 \texttt{AND} P_2)]]_D = [[P_1]]_D \bowtie [[P_2]]_D$
(3) $[[(P_1 \texttt{OPT} P_2)]]_D = [[P_1]]_D ⟕ [[P_2]]_D$
(4) $[[(P_1 \texttt{UNION} P_2)]]_D = [[P_1]]_D \cup [[P_2]]_D$

For the FILTER operator, we need to introduce the notion of condition satisfiability. A mapping μ satisfies the condition R, denoted as $\mu \vDash R$, if:

(1) R is bound(X) and $X \in$ dom(μ);
(2) R is $?X = c, ?X \in$ dom(μ) and $mu(?X) = c$;
(3) R is $?X = ?Y, ?X \in$ dom(μ), $?Y \in$ dom(μ) and $\mu(?X) = \mu(?Y)$;
(4) R is $(\neg R_1), R_1$ is a builtin condition, and it is not the case that $\mu \vDash R_1$;
(5) R is$(R_1 \vee R_2), R_1$ and R_2 are built-in conditions, and $\mu \vDash R_1$ or $\mu \vDash R_2$;
(6) R is$(R_1 \wedge R_2), R_1$ and R_2 are built-in conditions, $\mu \vDash R_1$ and $\mu \vDash R_2$.

The the evaluation of the expression (P `FILTER` R) is given by:

$$[[(P \texttt{ FILTER } R)]]_D = \{\mu \in [[P]]_D \mid \mu \vDash R\}.$$

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 $\mathbb{T}$:

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

Each of these pairs can be called a tagged triple.We can now define a time-based window as:

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

For the tuple-based window, we need to define first the c function 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 tuple-based window 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 the SPARQL language extensions for streams and windows. This is particularly useful in the sense that the user may know what to expect when he issues 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 or even a database, then we need to transform the SPARQL queries into a language able to deal with those sources, such as SNEEql. Therefore describe the basic semantics of SNEEql in the next section.

5.2 SNEEql Semantics

The operational semantics of the SNEEql query language are detailed in [SNEEql] using an approach that consists on translating the logical algebraic operators into the Haskell functional programming language. The main data types are first defined(all complete definitions are available in [Brennik]):

```
data Data
 =      Tuple ([Attribute],[RawData])
 |      TaggedTuple Tick Tuple
 | IndexedTuple Index Tick Tuple
 |      BagOf [Tuple]
 |      StreamOf [Data]
 |      Window Tick BagOfTuples
```

As we can see we have streams of data defined as lists of `Data` elements, which can in turn be of type `TaggedTuple` or even `Window`. As Haskell is a lazy evaluation language, we can use lists for representing unbounded streams. As it can be seen windows are composed on a `Tick` and a bag of `Tuple`. Notice that a relation can be seen as a list of `Tuple`. Creating a window in SNEEql involves two steps, first determine when to create the window taking into account the slide parameter. Then the set of tuples to be included in the window must be determined using the time range of the window. Time-based window creation is performed by the `createTimeWindow` function that takes as parameters the time range `WindowScopeDef` the slide, a tick representing the current timestamp (`NOW`) and a list of tagged tuples. A `Window` tagged with `NOW` tick is created, taking all tuples tagged before or at most at the current tick. Then the function is called recursively for `NOW + timeSlide`:

```
createTimeWindow :: WindowScopeDef -> Tick -> Tick -> [IndexedTuple] -> [Window]
createTimeWindow windowScopeDef timeSlide now indexedTuples
 =      let input = takeWhile (lessEqualsTick now) indexedTuples
 in [Window now (getWindowTuples windowScopeDef now input)]
        ++ createTimeWindow windowScopeDef timeSlide (now+timeSlide) taggedTuples
```

The `getWindowTuples` function filters the stream data according to the time range specified in the `TimeScopeDef`:

```
getWindowTuples windowScope@(TimeScopeDef from to) now input
 =      let passedFrom = dropWhile (lessThanTick (now - from)) input
 in let window = filter (lessEqualsTick (now - to)) passedFrom
 in BagOf (map stripTags window)
```

These functions are similarly defined for the case of tuple-based windows. In order to transform streams of windows to streams of tagged tuples, the `doRStream` function is defined:

```
doRStream :: StreamOfWindows -> StreamOfTaggedTuples
doRStream (StreamOf windows)
 =      let taggedTuples = toRStream windows
 in (StreamOf taggedTuples)

toRStream :: [Window] -> [TaggedTuple]
toRStream [] = []
toRStream ((Window tick (BagOf tuples)):moreWindows)
 =      let taggedTuples = map (tagTuple tick) tuples
 in taggedTuples ++ (toRStream moreWindows)
```

In the previous definition `tagTuple` simply tags a `Tuple` element with a tick. Projection, selection, cross product and other operators are also defined in this way using operational semantics.

5.3 Extended R2O Semantics

Both the semantics of SPARQL streaming extensions and SNEEql have been described in the previous sections so that we have a formalisation of what to expect from queries written in those languages. However in the context of our work we are particularly interested in answering union of conjunctive queries UCQ) over the global ontological schema and access the underlying data sources

through mappings. In this section we will present how we can use the mapping definitions to transform the set of CQs into queries into the internal query language SNEEql that is used to access the sources. This work is based on extensions to the ODEMapster processor [Barrasa] query rewriting algorithm and the formalisation work of [Calvanese, Calvanese Poggi, etc].

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

$$q(\mathcal{X}) \leftarrow \varphi(\mathcal{X}, \mathcal{Y})$$

$$\varphi(\mathcal{X}, \mathcal{Y}): \quad \bigwedge_{i=1\ldots k} P_i(\mathcal{X}, \mathcal{Y}), \text{ with } P_i \begin{cases} A_i(x), A \text{ is an atomic concept.} \\ R_i(x, y), R \text{ is an atomic role.} \\ x = y \end{cases}$$

Where X is a tuple of distinguished variables, Y a tuple of non-distinguished variables. P_i are atoms of the form $A_i(x)$, $R_i(x, y)$ and $x = y$, with x and y being variables in either X or Y, or constants such as concrete values or individuals. The answer to such query consists in the instantiation of the distinguished variables. For instance consider the following conjunctive query q_1:

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

It requires all instances x that are wind speed measurements taken by sea sensors. Concerning the formal definition of the query answering, let $\mathcal{I} = (\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ be an interpretation, where $\Delta^{\mathcal{I}}$ is the interpretation domain and $.^{\mathcal{I}}$ the interpretation function that assigns an element of $\Delta^{\mathcal{I}}$ to each constant, a subset of $\Delta^{\mathcal{I}}$ to each concept and a subset of $\Delta \times \Delta$ to each role of the ontology. A query $q(\mathcal{X}) \leftarrow \varphi(\mathcal{X}, \mathcal{Y})$ is interpreted as the set of tuples $o_{\mathcal{X}} \in \Delta^{\mathcal{I}} \times \cdots \times \Delta^{\mathcal{I}}$ such that there exists $o_{\mathcal{Y}} \in \Delta^{\mathcal{I}} \times \cdots \times \Delta_{\mathcal{I}}$ such that if we assign to the tuple of variables $mathcalX, Y$ the tuple $(o_{\mathcal{X}}, o_{\mathcal{Y}})$, then φ is true in I [Calvanese, Poggi,etc]. The tuples $q^{\mathcal{I}}$ of constants that substitute the $\mathcal{X}$ variables such that $\exists \mathcal{Y} \varphi(\mathcal{X}, \mathcal{Y})$ is true in $\mathcal{I}$. [Calvanese, Lubyte]

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 $C(x), R(x, y), A(x, y)$, with C, R and A being concepts, roles and attributes respectively in $\mathcal{O}$; and x, y are tuples of variables. ϕ is set of expressions $\mathcal{E}_C, \mathcal{E}_R, \mathcal{E}_A$ that can be translated to queries in the SNEEql language over the sources. A $\mathcal{E}_C$ expression describes how to construct the concept C from the source relations and/or streams, and has a declarative representation of the form:

$$\mathcal{E}_C = \{f_C^{Id}(x.A_1, \ldots, x.A_n, y.B_1, \ldots, y.B_m) \mid S_1(x) \wedge S_2(y) \wedge e_C^{Reun}(x, y) \wedge e_C^{Cond}(x) \wedge e_C^{Cond}(y)\}$$

Where S_1, S_2 are relations or streams, $A_1 \ldots A_n, B_1 \ldots B_m$ are the keys of S_1 and S_2. $e_C^{Reun}(x, y)$ represents a join and $e_c^{Cond}(x)$ represents a selection of a

subset of x. A function f_C^{Id} is also defined and it is used to produce a URI for each instance of the produced concept, based on the key attributes of the original relation or stream. A $\mathcal{E}_R$ expression describes how to construct the relationship R from the source streams and relations, and its declarative form is:

$$\begin{aligned}\mathcal{E}_R = \{(x.A_1, \ldots, x.A_n, y.B_1, \ldots, y.B_m) \mid & \mathcal{E}_{C_1}(x) \wedge \mathcal{E}_{C_2}(y) \wedge \\ & \exists z_1, \ldots, z_p (AUX_1(z_1) \wedge \cdots \wedge AUX_p(z_p) \\ & \wedge e_R^{Reun}(x, y, z_1, \ldots, z_p) \\ & \wedge e_R^{Cond}(x, y, z_1, \ldots, z_p))\}\end{aligned}$$

Where $AUX_1 \ldots AUX_p$ are auxiliary relations or streams on the sources. A $\mathcal{E}_A$ expression describes how to construct the attribute relationship A from the source streams and relations, and its declarative form is:

$$\mathcal{E}_A = \{f_C^{Id} f_A^{Trf}(x.A_1, \ldots, x.A_n, x.b_1, \ldots, x.b_m) \mid \mathcal{E}_C(x) \wedge e_A^{Cond}(x)\}$$

Where $b_1, \ldots, b_m$ are attributes in the stream or relation specified by $\mathcal{E}_C(x)$. A function f_A^{Trf} is also defined and it is used to apply any transformation over the b_i attributes. For instance unit transformation, concatenation, etc. The extended R2O document represents all these mappings in a XML serialised format. When a conjunctive query is issued against the global ontology, the processor first parses it and transforms it into an abstract tree and then uses the expansion algorithm described in [barrasa], that is based on the PerfectRef algorithm of [Calvanese]. This algorithm Afterwards the rewritten query can be translated to an extended relational algebra. A query Q_O is a conjunctive query that may contain window operators in order to narrow the data set according to a given criteria. For a query Q_O of the form $C(x)$ the translation is given by $\lambda(\mathcal{E}_C)$. :

$$\lambda(\mathcal{E}_C(Q_O)) = \pi_{f_C^{Id}}(\bowtie_{e_C^{Reun}} (\sigma_{e_C^{Cond}}(\omega_{\rho,\delta} S_1), \ldots, \sigma_{e_C^{Cond}}(\omega_{\rho,\delta} S_n)))$$

The expression denotes first a window operation $\omega_{\rho,\delta}$ over the relations or streams $S_1, \ldots, S_n$, with ρ and δ being the range and slide. A selection $\sigma_{e_C^{Cond}}$ is applied over the result, with the conditions defined in the mapping. A multiple join $\bowtie_{e_C^{Reun}}$ is then applied to the selection, also based on the corresponding mapping definition. Finally a projection is applied over the result of the function f_C^{Id} that calculates the instance identifier for C. For a query Q_O of the form $C(x) \wedge A(x, y)$ the translation is given by:

$$\begin{aligned}\lambda(\mathcal{E}_A(Q_O)) = \pi_{f_C^{Id} f_A^{Trf}}(\bowtie_{e_C^{Reun}} (\sigma_{e_C^{Cond}, e_A^{Cond}}(\omega_{\rho,\delta} S_1), \ldots \\ , \sigma_{e_C^{Cond}, e_A^{Cond}}(\omega_{\rho,\delta} S_n)))\end{aligned}$$

Similarly to the previous case, the window, selection, join and projection operations are applied to the source relations and streams according to the mapping definition. Additionally, the f_A^{Trf} is applied over the resulting attributes in case of any necessary transformations. For a query Q_O of the form $C_1(x) \wedge R(x, y) \wedge C_2(y)$

the translation is given by:

$$\lambda(\mathcal{E}_R(Q_O)) = \pi_{f_{C_1^{Id}} f_{C_2^{Id}}}(\bowtie_{e_{C_1^{Reun}}, e_{C_2^{Reun}}, e_{C_{AUX_i}}^{Reun}} (\sigma_{e_{C_1}^{Cond}, e_{C_2}^{Cond}, e_{C_{AUX_i}}^{Cond}} (\omega_{\rho,\delta} S_1), \ldots, \sigma_{e_{C_1}^{Cond}, e_{C_2}^{Cond}, e_{C_{AUX_i}}^{Cond}} (\omega_{\rho,\delta} S_n)))$$

As in the previous cases, the logical operators are applied following the mapping definition $\mathcal{M}$. The join in this case may potentially involve any auxiliary relations AUX_i.

6 Execution

The presented approach of providing ontology-based access to streaming data has been implemented as an extension to the ODEMapster processor [bar,corcho]. This implementation generates queries that can be executed by the SNEE in-network streaming query processor. Consider the following example, a stream of wind sensor measurements `windsamples` 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:

$$\begin{aligned} SpeedMeasurement &\sqsubseteq Measurement \\ WinddSpeedMeasurement &\sqsubseteq SpeedMeasurement \\ WindDirectionMeasurement &\sqsubseteq Measurement \\ SpeedMeasurement &\sqsubseteq \exists hasSpeed \\ Measurement &\sqsubseteq \exists isProducedBy \\ \exists isProducedBy^{-} &\sqsubseteq Sensor \\ Sensor &\sqsubseteq \exists hasSpeed \end{aligned}$$

Then we can define an R2O mapping that splits the windsamples stream tuples into instances of two different concepts WindSpeedMeasurement and WindDirectionMeasurement. Here's an extract of the R2O mapping concerning the WindSpeedMeasurement.

```
conceptmap-def WindSpeedMeasurement
 uri-as
  concat('ssg4env: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('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 and Sensor concept instances from the windsamples stream and the sensors table ($\mathcal{E}_{WindSpeedMeasurement}$ and $\mathcal{E}_{Sensor}$). 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, 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;
}
```

After applying complete here the text.

```
SELECT DISTINCT concat('http://www.semsorgrid4env.eu#WindSpeedMeasurement' ,
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 WindSpeedMeasurement concept.

7 Future Work

Although we have shown initial results querying the underlying SNEE engine with basic queries, we expect to consider 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 works such as [Perez-urbina]. We are also aware of the need of optimising the generated queries using techniques from sensor networks and continuous data approaches [stream, snee, aurora, borealis]. It is also our goal to provide characterisation of our algorithms in future deliveries. The present work is considered as just 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.

8 Conclusions

We have presented an approach for providing access to streaming data based on ontologies, by extending the R2O mapping definitions and the ODEMapster

processor. We have provided details about the underlying technologies that we use as a basis: SPARQL, SNEEql, R2O and ODEMapster. We have then presented the set of extensions to SPARQL for RDF streams and the extensions to R2O for stream mappings. Then we have shown the semantics of the proposed extensions and the mechanism to generate SNEEql queries from the original ontological queries using the extended R2O mappings. 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 the increasing number of sensor network applications being developed and deployed in the latest years.

Acknowledgments This work has been partially supported by the European Commission project SemSorGrid4Env (FP7-223913).

References

1. J. Barrasa, O. Corcho, and A. Gómez-Pérez. R2O, an Extensible and Semantically Based Database-to-Ontology Mapping Language. In *Second Workshop on Semantic Web and Databases (SWDB2004)*, 2004.