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

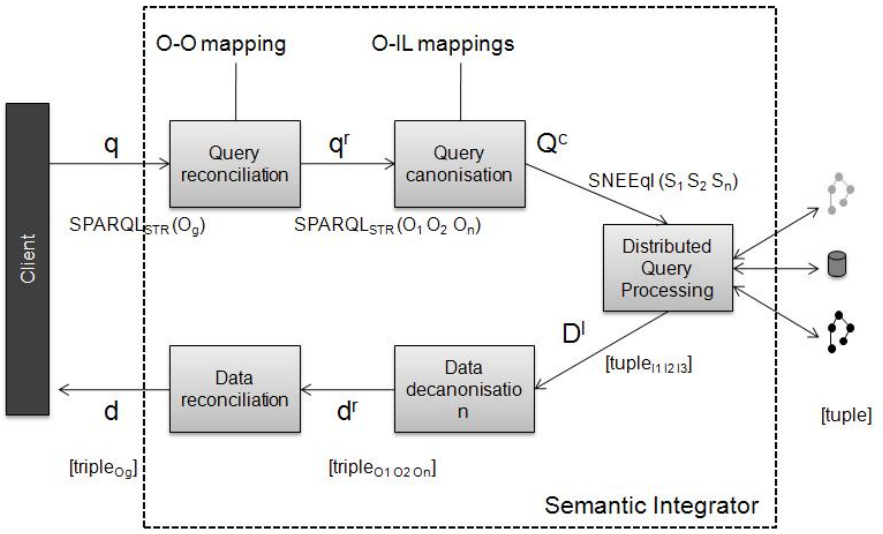

Fig. 1. Semantic Integrator service

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):

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

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:

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

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 $X \in \text{dom}(\mu)$;

(2) R is $?X = c, ?X \in \text{dom}(\mu)$ and $mu(?X) = c$;

(3) R is $?X =?Y, ?X \in \text{dom}(\mu), ?Y \in \text{dom}(\mu)$ 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:

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

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

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 $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ 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 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:

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

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
```