Streaming SPARQL - Extending SPARQL to Process Data Streams

Andre Bolles, Marco Grawunder, and Jonas Jacobi

University of Oldenburg, Germany,
Department for Computer Science, Information Systems and Databases
{andre.bolles,marco.grawunder,jonas.jacobi}@uni-oldenburg.de

Abstract. A lot of work has been done in the area of data stream processing. Most of the previous approaches regard only relational or XML based streams but do not cover semantically richer RDF based stream elements. In our work, we extend SPARQL, the W3C recommendation for an RDF query language, to process RDF data streams. To describe the semantics of our enhancement, we extended the logical SPARQL algebra for stream processing on the foundation of a temporal relational algebra based on multi-sets and provide an algorithm to transform SPARQL queries to the new extended algebra. For each logical algebra operator, we define executable physical counterparts. To show the feasibility of our approach, we implemented it within our Odysseus framework in the context of wind power plant monitoring.

1 Introduction

Wind power plants are one promising way to reduce CO_2 emissions and by this the greenhouse effect. Large off-shore wind parks are capable of delivering electric power in a constant and reliable way. These parks need high initial investments, so keeping maintenance costs low is indispensable. Monitoring and early problem detection is essential. The current approach to monitor wind parks are proprietary SCADA (*S*upervisory *C*ontrol *a*nd *D*ata *A*cquisition) systems. This leads to the problem of missing interoperability between different wind power plant vendors and makes it difficult or even impossible for customers to extend the system. In analogy to the application of database management systems for many standard applications like personnel or warehouse management, we believe that data stream management systems (DSMS) [1] can help in reducing initial wind parks costs and providing higher interoperability. The International Electrotechnical Commission has proposed a standard to monitor and communicate with wind parks [2]. IEC 61400-25 contains a standard data model and a common information model. The standard determines no communication format. A common approach could be the usage of XML. In our work we analyze the use of RDF [3] as an alternative, because RDF can be used to model IEC 61850 family and substandards like 61400-25 in future. Furthermore a triple contains all information needed to understand the content therefore RDF is an ideal format for serialization and missing triples (e.g. because of network failure) do not necessary corrupt the whole stream.

In DSMS continuous queries are used to determine stream information. SPARQL is the W3C Recommendation for a query language over RDF. Unfortunately, this approach is not applicable directly to RDF data streams. A sensor produces typically unlimited datasets (e.g. an element every second). Some operators in SPARQL need the whole dataset to process, like distinct, join or sort. A common approach for this problem is the use of window queries. Instead of regarding the whole stream, only subparts, so called windows, are treated. Because recent data have typically higher relevance (e.g. average wind speed in the last 10 minutes) this approach is applicable. We regard different kinds of windows. Time based windows define the window by a unit of time (e.g. the last 10 minutes). These windows are called sliding windows, if they are moved for each new point in time. If the windows are moved only after a distinct period, these windows are called sliding δ windows (e.g. move the window every 5 minutes over the stream with the window size 10 minutes) [4]. Element based windows are defined by the number of elements in the window (e.g. the last 100 statements). We model these windows by defining temporal element validity, i.e. if an element is valid it is part of the window.

In query processing we follow the usual approach of translating a descriptive query into an internal representation (the logical plan) that can more easily be optimized and transformed. This logical plan will then be translated into a set of physical plans, containing executable physical algebra operators. Finally, one of these plans will be selected, e.g. by means of a cost model, and executed.

The contributions of this paper are:

- the extension of SPARQL to handle RDF based data streams,
- the precise definition of the semantics of this extension by extending the existing SPARQL algebra,
- a transformation algorithm to map the language extension to the extended SPARQL algebra and
- a set of physical algebra operators to execute the queries.

The remainder of this paper is as follows. The next section describes how we extended the SPARQL grammar to allow referencing data streams and defining windows over these streams. After that we give a clear formal definition of what our data streams are (Sec. 3) and, based on this definition, we show how we extended the SPARQL algebra (Sec. 4) and how SPARQL queries can be translated into that algebra (Sec. 5). Because algebra operators are not executable we defined physical counterparts of each logical operator (Sec. 6). Finally, we discuss related work (Sec. 7) and give an outlook on future work.

2 Extending the SPARQL Grammar

SPARQL as defined in [5] is not sufficient to define queries over data streams. One of our main goals is to preserve the known syntax and semantics of SPARQL as much as possible. We extend SPARQL with the capability to explicitly state data streams and to define windows over them. Windows can be defined in the FROM part of a SPARQL query to define a common window for the whole data stream. To allow a finer granularity we also allow windows in graph patterns.

Table 1 shows the extended grammar production rules in EBNF. First are listed those rules that are affected by our extensions[1]. Then new rules are listed. Extensions and new rules are shown in bold. The other rules from [5] do not need to be modified for processing data streams[2]. All nonterminals (underlined) that are not explained further are identical to [5].

Table 1. Extended SPARQL Grammar (extract)

SelectQuery	::=	'SELECT' ('DISTINCT' | 'REDUCED')? (Var | '*') (DatasetClause* | **DatastreamClause***) WhereClause SolutionModifier
NamedGraphClause	::=	'NAMED' SourceSelector
DatastreamClause	**::=**	**'FROM' (DefaultStreamClause | NamedStreamClause)**
DefaultStreamClause	**::=**	**'STREAM' SourceSelector Window**
NamedStreamClause	**::=**	**'NAMED' 'STREAM' SourceSelector Window**
SourceSelector	::=	IRIref
GroupGraphPattern	::=	'{' TriplesBlock? ((GraphPatternNotTriples | Filter) '.'? TriplesBlock?)* **(Window)?** '}'
Window	**::=**	**(SlidingDeltaWindow | SlidingTupelWindow | FixedWindow)**
SlidingDeltaWindow	**::=**	**'WINDOW' 'RANGE' ValSpec 'SLIDE' (ValSpec)?**
FixedWindow	**::=**	**'WINDOW' 'RANGE' ValSpec 'FIXED'**
SlidingTupelWindow	**::=**	**'WINDOW' 'ELEMS' INTEGER**
ValSpec	**::=**	**INTEGER Timeunit?**
INTEGER	::=	[0-9]+
Timeunit	**::=**	**('MS' | 'S' | 'MINUTE' | 'HOUR' | 'DAY' | 'WEEK')**

To state an input as a data stream the keyword STREAM followed by an IRI has to be used. We allow different window types, which require specific language extensions. If a window is defined in a query in the FROM and in the graph pattern parts, only the more special window of the graph pattern is evaluated. We provide time based windows and element based windows. Sliding δ-windows allow the definition of the window size with the RANGE keyword whereas we assume milliseconds to be the default timeunit. SLIDE defines the delay after which the window is moved. The value after SLIDE can be omitted. In that case the size is 1 of the unit defined in the RANGE-part. If SLIDE and RANGE contain the same values, we call this a fixed (or tumbling) window. As syntactic sugar we also allow the definition of this window using FIXED. Finally, ELEMS defines element based windows.

Listing 1.1 below gives an example of a query containing time and element based windows in the FROM and in the optional graph pattern part. The idea of the query is to return the number of starting a wind turbine in the last 30 minutes and, if available, the number of stopping a wind turbine that has to be reported in the last 1500 elements of the data stream.

[1] For space reasons we omitted Construct, Describe and Ask queries which are defined similar to Select.

[2] Of course there are possibly different semantics when defining queries over data streams.

```
PREFIX wtur: <http://iec.org/61400-25/root/ln/classes/WTUR#>
SELECT ?x ?y ?z
FROM STREAM <http://iec.org/61400-25/root/td.rdf>
     WINDOW RANGE 30 MINUTE SLIDE
WHERE { ?x wtur:StrCnt ?y .
         OPTIONAL { ?x wtur:StopCnt ?z .
                    WINDOW ELEMS 1500 }}
```

Listing 1.1. SPARQL query over an RDF stream with windows

3 Defining Data Streams

Before we present our extended algebra we define the logical base of our algebra, the data stream. Like [6] we distinguish between a raw data stream, representing the data received by the DSMS, a physical data stream which can be processed by the operators of the DSMS and finally a logical data stream over which the algebra and therefore the semantics of the extensions can be defined. Many of the following definitions are inspired by [6].

A raw data stream represents simply the data arriving at the DSMS. The data format could be in different formats like XML/RDF or Turtle representation. A special access operator can transform the statements to a unified representation. Because some sources in data stream applications typically produce timestamped information (e.g. a sensor for wind speed also transmits a timestamp for the measured value), we defined an RDF raw data stream with timestamps, too.

The physical RDF data stream contains prepared elements that can be used as input for physical data stream operators. This means especially that all elements are timestamped, defined either by the data source (raw data stream with timestamps) or by the DSMS (raw data streams without timestamps). Let $\mathbb{T} = (T; \leq)$ be a discrete time domain as proposed by [7][3]. Let $\mathbb{I} := \{[t_s, t_e) \in T \times T | t_s < t_e\}$ be the set of right open time intervals:

Definition 1 (Physical RDF data stream). *Let $\mathbb{S}^P_{RDF}$ be the set of all physical RDF data streams. A physical RDF data stream $S^P_{RDF} \in \mathbb{S}^P_{RDF}$ is defined as a pair $S^P_{RDF} = (M^P_{RDF}, \leq^P_t)$ where $M^P_{RDF} = [((s,p,o),[t_S,t_E))|(s,p,o) \in \Omega, [t_S,t_E) \in \mathbb{I}]$ is an ordered possibly unlimited sequence of pairs, consisting of an RDF statement and a right open time interval. The order is defined by $\leq^P_t$:*

$$\forall x_i, x_j \in \Omega \times \mathbb{I}, i, j \in \mathbb{N}, i < j : x_i \leq^P_t x_j \Leftrightarrow x_i.t_S \leq x_j.t_S$$

In SPARQL only the first operators typically handle RDF based data. E.g. a basic graph pattern transforms RDF statements to SPARQL solutions μ which are the input to the following operators. To represent this in our algebra we define a set $\mathbb{S}^P_\mu$ of physical μ data streams analogous to $\mathbb{S}^P_{RDF}$.

[3] Without reducing generality, we allow only natural numbers for $\mathbb{T}$.

Logical algebra operators consume logical RDF streams. These logical streams are not defined as sequences but as multi-sets. Because no elements are processed on the logical level, it is only used for transformation and optimization; the order in the streams is not relevant. The great advantage of defining logical streams on multi-set semantics is to make it possible to base our algebra on the extended relational algebra [8]. Just like there, duplicates are allowed and expressed by the number of occurrences.

Definition 2 (Logical RDF data stream). *Let $\mathbb{S}^L_{RDF}$ be the set of all logical RDF data streams. A logical RDF data stream is a possibly unlimited multi-set of triples: $S^L_{RDF} = \{((s,p,o),t,n) | (s,p,o) \in \Omega, t \in \mathbb{T}, n \in \mathbb{N}\backslash\{0\}\}$. The triple $((s,p,o),t,n)$ expresses: The RDF statement (s,p,o) is valid at time t and occurs n-times at this point in time.* $\forall((s,p,o),t,n) \in S^L_{RDF} : \nexists((\hat{s},\hat{p},\hat{o}),\hat{t},\hat{n}) \in S^L_{RDF} : (s,p,o) \equiv_{RDF} (\hat{s},\hat{p},\hat{o}) \wedge t = \hat{t}$. $\equiv_{RDF}$ *means that subject, predicate and object of two statements are pairwise equal.*

Analogously, we define the set of logical μ data streams $\mathbb{S}^L_{\mu}$, because not all logical algebra operators can consume logical RDF data streams. We also define sequential logical RDF and sequential logical μ data streams to support sort operations. We will use $\mathbb{S}^L_x$ if no distinction between RDF and μ is necessary.

To relate physical and logical algebra operators, and thus defining the semantics of the physical operators, we need to define transformations between the different stream types. Thereby it is possible to assign plan transformations and optimizations on the logical query level also to the physical level. Due to space limitations we need to omit these transformations here.

4 Extending the SPARQL Algebra

Given the base definitions, we can now present some of the formal definitions of our new or extended SPARQL algebra operators.

The first operator we introduce is needed to define windows over the logical stream $\mathbb{S}^L_x$. It takes parameters $w \in \mathbb{T}$ defining the width of the window and $\delta \leq w$ defining the delay of window moving:

Definition 3 (Sliding δ-window). *Let $w \in \mathbb{T}$ be the width of a sliding window and $\delta \leq w \in \mathbb{T}$ be a time span, by which the window is moved. Let $\hat{t} \in \mathbb{T}$ be a point in time. A sliding δ-window is a function $\omega^{slide}_{w,\delta} : \mathbb{S}^L_x \times \mathbb{T} \times \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{S}^L_x$, with:*

$$\begin{aligned}\omega^{slide}_{w,\delta}(S^L_x, \hat{t}) := \{(z,\hat{t},\hat{n}) | &\exists i \in \mathbb{N} : (i \cdot \delta) - w \leq \hat{t} < i \cdot \delta \wedge \\ &\exists Y \subseteq S^L_x : Y \neq \emptyset \wedge \\ &Y = \{(z,t,n) | \max\{(i \cdot \delta) - w, 0\} \leq t \leq \hat{t}\} \wedge \\ &\hat{n} = \textstyle\sum_{\{(z,t,n) \in Y\}} n\}\end{aligned}$$

The special case $\delta = 1$ defines a sliding window, the case $\delta = w$ defines a tumbling window. Defining $\delta \leq w$ ensures not to miss elements by moving the window. The second kind of window, called a sliding tuple window, is defined over the element count in a stream. To define this operator we need to define a count function on logical data streams, that gives the number of elements in a logical RDF or μ stream (in the following represented by x) until a point in time t [6]:

Definition 4 (Count function of logical x data streams). *Let $t \in \mathbb{T}$ be a point in time. The function $m : \mathbb{S}_x^L \times \mathbb{T} \rightarrow \mathbb{N}$ calculates the number of all elements in the data stream until t:*

$$m(t, S_x^L) := \left|\{(z, \hat{t}, n\}|\hat{t} \leq t\}\right|$$

This definition is only valid for $n \leq 1$, i.e. at every point in time only one element is added to the stream. With this function a sliding tuple window can be defined as follows:

Definition 5 (Sliding tuple window). *Let $c \in \mathbb{N}$ be the maximum size of a sliding tuple window and $\hat{t} \in \mathbb{T}$ a point in time. A sliding tuple window is a function $\omega_c^{count} : \mathbb{S}_x^L \times \mathbb{N} \times \mathbb{T} \rightarrow \mathbb{S}_x^L$ with:*

$$\begin{aligned}\omega_c^{count}(S_x^L, \hat{t}) := \{(z, \hat{t}, \hat{n})|\exists Y \subseteq S_x^L : Y \neq \emptyset \wedge \\ Y = \{(z, t, n) \in S_x^L | max\{m(\hat{t}, S_x^L) - c, 1\} \leq m(t, S_x^L) \leq m(\hat{t})\} \\ \wedge\, \hat{n} = \textstyle\sum_{(z,t,n)\in Y} n\}\end{aligned}$$

Triple Pattern Matching. In SPARQL [5] no operator for the transformation from RDF statements to SPARQL is defined. To be more general, we introduce the operator υ that filters statements from a (logical) RDF stream and transforms them to SPARQL solutions. To cope with the problem of blank nodes we need to extend the definition for an RDF instance mapping from [9]:

Definition 6 (Extended RDF instance mapping). *Let $\mathbb{E}$ be the set of all extended RDF instance mappings. An extended RDF instance mapping $\sigma_{ext} \in \mathbb{E}$ is a function $\sigma_{ext} : RDF\text{-}T \cup V \rightarrow RDF\text{-}T \cup V$ using a function $f : RDF\text{-}B \rightarrow RDF\text{-}T$:*

$$\sigma_{ext}(x) := \begin{cases} f(x) \in RDF\text{-}T, & \text{if } x \in RDF\text{-}B \\ x, & \text{if } x \in I \cup RDF\text{-}L \cup V \end{cases}$$

We can then define the triple matching operator:

Definition 7 (Triple pattern matching). *Let $p = (x, y, z) \in \Lambda$ be a triple pattern in the set of all triple patterns, $S_{RDF}^L \in \mathbb{S}_{RDF}^L$ a logical data stream and $\sigma_{ext} : RDF\text{-}T \cup V \rightarrow RDF\text{-}T \cup V$ an extended RDF instance mapping. The triple pattern matching $\upsilon : \mathbb{S}_{RDF}^L \times \Lambda \rightarrow \mathbb{S}_\mu^L$ is defined as follows:*

$$\begin{aligned}\upsilon_p(S_{RDF}^L) := \{(\mu, t, n)|\exists((s, p, o), t, n) \in S_{RDF}^L : \exists\sigma_{ext} \in \mathbb{E} : \\ \sigma_{ext}(x) \in RDF\text{-}T \Rightarrow \sigma_{ext}(x) = s \ \wedge \sigma_{ext}(y) \in RDF\text{-}T \Rightarrow \sigma_{ext}(y) = p \wedge \\ \sigma_{ext}(z) \in RDF\text{-}T \Rightarrow \sigma_{ext}(z) = o \ \wedge \sigma_{ext}(x) \in V \Rightarrow \mu(\sigma_{ext}(x)) = s \wedge \\ \sigma_{ext}(y) \in V \Rightarrow \mu(\sigma_{ext}(y)) = p \ \wedge \sigma_{ext}(z) \in V \Rightarrow \mu(\sigma_{ext}(z)) = o\}\end{aligned}$$

The triple pattern matching operator transforms a logical RDF stream into a logical μ data stream, i.e. it changes the schema of the contained elements. All the following described operators consume logical μ streams. Therefore, the triple pattern matching operator must be placed in a plan before any of the following operators.

Filter. The filter operator evaluates the FILTER term of a SPARQL query. It selects from a logical μ stream those elements that satisfy a predicate $p(\mu, t) \in \mathbb{P}_s$ from the set of all SPARQL predicates.[4]

Definition 8 (Filter). *Let $p : \Phi \times \mathbb{T} \rightarrow \{true, false\} \in \mathbb{P}_s$ be a SPARQL predicate. A filter operator f_p is a function $f : \mathbb{S}^L_\mu \times \mathbb{P} \rightarrow \mathbb{S}^L_\mu$, defined as follows:*

$$f_p(S^L_\mu) := \{(\mu, t, n) | (\mu, t, n) \in S^L_\mu \ \wedge \ p(\mu, t)\}$$

Union. The union of two logical μ streams evaluates the UNION term of a SPARQL query.

Definition 9 (Union of logical μ data streams). *The union $\cup_+$ of two logical μ data streams is a function $\cup_+ : \mathbb{S}^L_\mu \times \mathbb{S}^L_\mu \rightarrow \mathbb{S}^L_\mu$, defined as follows:*

$$\begin{aligned}\cup_+(S^L_{\mu,1}, S^L_{\mu,2}) := \{&(\mu, t, n_1 + n_2) | \\ &(\exists(\mu, t, n_1) \in S^L_{\mu,1} \ \vee \ n_1 = 0) \ \wedge \ (\exists(\mu, t, n_2) \in S^L_{\mu,2} \ \vee \ n_2 = 0) \wedge \\ &n_1 + n_2 > 0\}\end{aligned}$$

The operator adds an element to the result stream if it occurs in one of the two streams. Because $\cup_+$ is a multi-set operator we need to sum the multiplicities [8].

Join. A join operator combines two compatible elements from two μ data streams. Two solutions are compatible [5] if:

$$\mu_1 \asymp \mu_2 : \forall v_1 \in dom(\mu_1), v_2 \in dom(\mu_2) : v_1 = v_2 \Rightarrow \mu_1(v_1) = \mu_2(v_2)$$

$\mu_1(v_1) = \mu_2(v_2)$ does also apply in this work, if v_1 in μ_1 and v_2 in μ_2 are unbound. Further let $merge : \Phi \times \Phi \rightarrow \Phi$ be a mapping corresponding to the merge function from [5] defined as follows:

$$\begin{aligned}merge(\mu_1, \mu_2) = \mu_{merge} :\Leftrightarrow\ &\mu_1 \asymp \mu_2, \\ &\text{with } dom(\mu_{merge}) = dom(\mu_1) \cup dom(\mu_2) \ \wedge \\ &(\forall v \in dom(\mu_1) \backslash dom(\mu_2) : \mu_{merge}(v) = \mu_1(v)) \ \wedge \\ &(\forall v \in dom(\mu_2) \backslash dom(\mu_1) : \mu_{merge}(v) = \mu_2(v)) \ \wedge \\ &(\forall v \in dom(\mu_1) \cap dom(\mu_2) : \mu_{merge}(v) = \mu_1(v) = \mu_2(v))\end{aligned}$$

We can then define the join as follows.

Definition 10 (Join). *The join of two logical μ data streams is a function $\bowtie: \mathbb{S}^L_\mu \times \mathbb{S}^L_\mu \rightarrow \mathbb{S}^L_\mu$ with:*

$$\begin{aligned}\bowtie (S^L_{\mu,1}, S^L_{\mu,2}) := \{&(merge(\mu_1, \mu_2), t, n_1 \cdot n_2) | \exists(\mu_1, t, n_1) \in S^L_{\mu,1} \ \wedge \\ &\exists(\mu_2, t, n_2) \in S^L_{\mu,2} \ \wedge \ \mu_1 \asymp \mu_2\}\end{aligned}$$

Only elements that are valid at the same point in time can be joined to a new element.

[4] A SPARQL predicate is a boolean expression, consisting of operators from section 11.3 in [5] with boolean return values whose operands are variables or RDF terms.

Basic Graph Pattern Matching. This operator is defined in the SPARQL algebra, too, although it is a join over two or more triple pattern matchings. To assure that blank nodes are handled correctly, the included triple pattern matchings must use the same RDF instance mapping.

Definition 11 (Basic Graph Pattern matching). *Let $p_1, p_2 \in \Lambda$ be two triple patterns and $S^L_{RDF,1}$, $S^L_{RDF,2} \in \mathbb{S}^L_{RDF}$ be two logical RDF data streams. A basic graph pattern matching is the mapping $\chi : \mathbb{S}^L_{RDF} \times \mathbb{S}^L_{RDF} \to \mathbb{S}^L_\mu$, defined as follows:*

$$\chi(S^L_{RDF,1}, S^L_{RDF,2}) := \ \bowtie (\upsilon_{p_1}(S^L_{RDF,1}), \upsilon_{p_2}(S^L_{RDF,2})),$$

with $\sigma_{ext,1} = \sigma_{ext,2}$.

Left Join. In [5] a left join is used to evaluate an OPTIONAL term of a SPARQL query. We combine the left join directly with a filter predicate to preserve the correct OPTIONAL semantics. Unfortunately, the formal definition and its full description in [5] are inconsistent. According to the full description the left join does not deliver a result if there is no right element to potentially join with, so we use the following definition of a left join as opposed to the definition in [5]:

Definition 12 (W3C LeftJoin). *Let $p \in \mathbb{P}_s$ be a SPARQL predicate. A left join $\ltimes_{w3c}$ is a mapping $\ltimes_{w3c}$: $\Omega \times \Omega \to \Omega$, evaluated as follows:*

$$\begin{aligned}\ltimes_{w3c}(\Omega_1, \Omega_2) := \{&merge(\mu_1, \mu_2) | \mu_1 \in \Omega_2 \ \wedge \ \mu_2 \in \Omega_2 \ \wedge \ \mu_1 \simeq \mu_2 \ \wedge \ p(merge(\mu_1, \mu_2))\} \\ \cup \{&\mu_1 | \mu_1 \in \Omega_1 \ \wedge \ \nexists \mu_2 \in \Omega_2 : \mu_1 \simeq \mu_2 \ \wedge \ p(merge(\mu_1, \mu_2))\}\end{aligned}$$

Based on this modified left join we give our definition of the stream based left join:

Definition 13 (LeftJoin). *Let $p \in \mathbb{P}_s$ be a SPARQL predicate. A left join $\ltimes$ is a mapping $\ltimes$: $\mathbb{S}^L_\mu \times \mathbb{S}^L_\mu \times \mathbb{P}_s \to \mathbb{S}^L_\mu$, defined as follows:*

$$\begin{aligned}\ltimes (S^L_{\mu,1}, S^L_{\mu,2}) := \{&(merge(\mu_1, \mu_2), t, n_1 \cdot n_2) | \exists(\mu_1, t, n_1) \in S^L_{\mu,1} \ \wedge \\ &\exists(\mu_1, t, n_2) \in S^L_{\mu,2} : \mu_1 \simeq \mu_2 \ \wedge \ p(merge(\mu_1, \mu_2))\} \cup_+ \\ \{&(\mu_1, t, n_1) | \exists(\mu_1, t, n_1) \in S^L_{\mu,1} \ \wedge \ \nexists(\mu_2, t, n_2) \in S^L_{\mu,2} : \mu_1 \simeq \mu_2 \\ &\wedge \ p(merge(\mu_1, \mu_2))\}\end{aligned}$$

Other Operators. In addition to the described operators we also defined tolist, order by, duplicate elimination, duplicate reduction, slice, projection, construct, describe and ask.

5 Query Translation

We are now able to show how we can translate SPARQL queries over data streams into logical algebra plans. Most transformation rules from [5] can be applied directly, using our new algebra operators. Only the definition of windows over the data streams and the new triple pattern operator need special rules. The triple pattern matching operator transforms RDF statements into SPARQL solutions. Listing1.2 shows a simple SPARQL query without data streams. The transformation of this query can be found in Figure 1.

```
SELECT ?w ?x ?y ?z
FROM <http://src.net/graph.rdf>
WHERE {?w my:name ?x } UNION { ?y my:power ?z }
```

Listing 1.2. Query 1

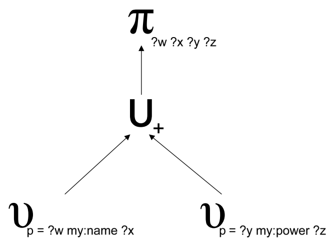

Fig. 1. Query plan for Query 1

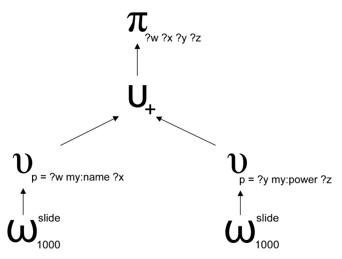

Fig. 2. Query plan for Query 2

As you can see, the triple patterns `{?w my:name ?x}` and `{?y my:power ?z}` are both transformed to one triple pattern matching operator. By this, the RDF statements are filtered from the source and transformed into SPARQL solutions. Afterwards, the union and project operators can process the triple pattern matching results.

Window definitions in the FROM parts of a query are placed directly before the corresponding triple pattern matching operator as demonstrated in Query 2 in Listing 1.3 and Figure 2.

```
SELECT ?w ?x ?y ?z
FROM STREAM <http://src.net/graph.rdf> WINDOW RANGE 1000 SLIDE
WHERE {?w my:name ?x } UNION { ?y my:power ?z }
```

Listing 1.3. Query 2

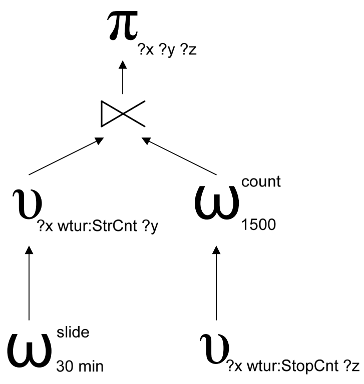

Fig. 3. Query plan for query 1.1

Algorithm 1. Method AlgebraOp createPlan(clause, actStream, namedStreams, win, graphVar)

```
AlgebraOp retVal = null;
List subplans = new List();
if clause instanceof GroupGraphPattern then
    win = clause.getWindow();
    foreach Subclause sc: clause.getSubclauses() do
        subplans.add(createPlan(sc, actStream, namedStreams, win, graphVar));
    retVal = createJoinHierarchyOverAll(subplans);
    setLeftJoinInHierarchyIfRightInputIsOptional(retVal);
else if clause instanceof TriplesBlock then
    foreach Subclause sc: clause.getSubclauses() do
        subplans.add(createPlan(sc, actStream, namedStreams, win, graphVar));
    retVal = createJoinHierarchyOverAll(subplans);
else if clause instanceof GroupOrUnionGraphPattern then
    foreach Subclause sc: clause.getSubclauses() do
        subplans.add(createPlan(sc, actStream, namedStreams, win, graphVar));
    retVal = createUnionHierarchyOverAll(subplans);
else if clause instanceof Filter then
    retVal = new Filter(clause.getExpression());
else if clause instanceof GraphGraphPattern then
    if (uri = clause.getGraphClause().getURI()) ≠ null then
        retVal = createPlan(clause.getSubclause(), namedStreams.get(uri),
        namedStreams, win, null);
    else
        foreach s : namedStreams do
            subplans.add(createPlan(clause.getSubclause(), s, namedStreams, win,
            clause.getGraphVar()));
        retVal = createUnionHierarchyOverAll(subplans);
else if clause instanceof Triple then
    TriplePatternMatching tpm = new TriplePatternMatching(clause, graphVar);
    if win == null then
        win = actStream.getWindow();
        retVal = tpm.withInput(win.withInput(createAccessOp(actStream.getURI())));
    else
        retVal = win.withInput(tpm.withInput(createAccessOp(actStream.getURI())));
return retVal;
```

Our complete transformation algorithm can be found in Algorithm 1. The basic idea of this algorithm is to recursively run through the query starting at the WHERE-clause and create operators for the serveral query parts (e. g. a left join operator for an optional graph pattern). With the parameter `win` it is possible to determine whether to put a window operator of a FROM-part or of a basic graph pattern part of a query into the queryplan.

Finally, we present the translation result of the query in Listing 1.1 in Figure 3 in our logical algebra.

6 Physical SPARQL Data Stream Algebra

The logical algebra is used to define the operator semantics and allow some static plan optimizations. A logical algebra cannot be used to execute queries. For this a physical algebra with executable operators is needed. In line with [6] we apply an interval based approach for the physical algebra.

The set of operators is divided into two groups. Stateless operators do not need any information about the stream history or stream future. Each stream element can be processed directly. Operators of this group are triple pattern matching, filter, union, project, construct and our time based window operators. The other group contains stateful operators. These operators need further information to process an element, e.g. a sort operator needs to know all elements before it can sort the set. Join, left join, basic graph pattern matching, duplicate elimination/reduction, orderBy, slice, ask and element based windows are stateful operators. They need a special data structure to process their input. As in [6] a so called sweep area is used as this data structure. It is an abstract datatype with methods to insert, replace or query elements. We will not further describe this data structure but refer to [6].

We will present our implementation of the sliding δ-window (Algorithm 2) first: A sliding δ window has a width $w \in \mathbb{T}$ and a step size $\delta \in \mathbb{T}$ to move the window over the data stream. In the following, x is used to either describe the set of RDF statements or the set of SPARQL solutions; z is an instance of this set.

Algorithm 2. Sliding δ window

```
Input: $S^P_{x,in} \in \mathbb{S}^P_x, w, \delta \in \mathbb{T}, \delta \leq w$
Output: $S^P_{x,out} \in \mathbb{S}^P_x$
$S^P_{x,out} \leftarrow \emptyset$;
foreach $e{:=}(z, [t_S, t_E)) \hookleftarrow S_{RDF,in}$ do
    $e.t_E := \left(\left\lfloor \frac{t_S}{\delta} \right\rfloor \cdot \delta\right) + w$;
    $e \hookrightarrow S^P_{RDF,out}$;
```

If a new element occurs in the stream, it will be read out and its end timestamp will be set according to $\left(\left\lfloor \frac{t_S}{\delta} \right\rfloor \cdot \delta\right) + w$. Afterwards, the element will be written into the output stream of the operator.

As stated above, a triple pattern matching transforms filtered statements of an RDF input stream to SPARQL solutions. This operator is stateless and can be implemented as in Algorithm 3.

The algorithm gets a triple pattern and an extended RDF instance mapping σ_{ext} as input. If the subject, predicate and object of the triple pattern match the subject, predicate and object of an RDF stream element, a SPARQL solution is generated, where variables of the triple pattern are mapped to the corresponding terms in the RDF stream statement. The SPARQL solution is written to the μ output stream.

We present our solution for the LeftJoin operator in Algorithm 4 and 5.

Algorithm 3. Triple Pattern Matching

```
Input: $S^P_{RDF,in} \in \mathbb{S}^P_\mu, p = (x, y, z) \in \Lambda, \sigma_{ext} \in \mathbb{E}$
Output: $S^P_{\mu,out} \in \mathbb{S}^P_\mu$
$S^P_{\mu,out} \leftarrow \emptyset$;
SPARQL Solution $\mu$ := new SPARQL Solution();
foreach $e$:=$((s, p, o), [t_S, t_E)) \hookleftarrow S_{RDF,in}$ do
    if $(x \in V \lor \sigma_{ext}(x) == e.s) \land (y \in V \lor \sigma_{ext}(y) == e.p) \land (z \in V \lor \sigma_{ext}(z) == e.o)$
    then
        if $x \in V$ then
            $\mu$.add(x,s);
        if $y \in V$ then
            $\mu$.add(y,p);
        if $z \in V$ then
            $\mu$.add(z,o);
        $\mu \hookrightarrow S^P_{\mu,out}$;
```

Algorithm 4. LeftJoin

```
Input: $S^P_{\mu,in,1}, S^P_{\mu,in,2} \in \mathbb{S}^P_\mu$, SweepAreas $SA_1, SA_2$, Min-Heap $H$, $p \in \mathbb{P}_s$
Output: $S^P_{\mu,out} \in \mathbb{S}^P_\mu$
Let $S^P_{\mu,in,1}$ be the left and $S^P_{\mu,in,2}$ the right input stream;
foreach $e := (\mu, [t_S, t_E)) \hookleftarrow S^P_{\mu,in,j}, j \in \{1, 2\}$ do
    if $j == 1$ then
        /*Input from left input stream                                   */
        doLeft(this);
    else
        /*Input from right input stream                                  */
        doRight(this);
    $min_{t_S} \leftarrow min(\{t_S | (\mu, [t_S, t_E)) \in SA_1\})$;
    if $min_{t_S} \neq null$ then
        while $H \neq \emptyset$ do
            $top := (\tilde{\mu}, [\tilde{t}_S, \tilde{t}_E)) \leftarrow$ top element of $H$;
            if $top.\tilde{t}_S < min_{t_S}$ then
                $top \hookrightarrow S^P_{\mu,out}$;
                Remove $top$ from $H$;
            else
                break;
```

7 Related Work

This work is based on the W3C Recommendation SPARQL [5], especially on the defined algebra and grammar. Also, possibilities for the serialization of RDF statements in data streams have been described. Our work is based on [6]. In that work a schema independent algebra for stream processing has been introduced. Because of some differences between the relational model and the RDF graph model, this algebra could not completely be reused for RDF stream processing. But some ideas like the time instant based approach for logical algebra operators and the interval based approach for physical algebra operators offer advantages for the definition of algebra operators in this work.

Algorithm 5. doLeft(LeftJoin op) and doRight(LeftJoin op) of a LeftJoin

```
**doLeft**(LeftJoin op);
$SA_2$.purgeElements(e);
$e.\tilde{t}_E = e.t_S$;
Iterator qualifies ← $SA_2$.query(e);
**while** *qualifies.hasNext()* **do**
    $\hat{e} := (\hat{\mu}, [\hat{t}_S, \hat{t}_E)) \leftarrow$ qualifies.next();
    **if** $e \risingdotseq \hat{e} \wedge p(merge(\mu, \hat{\mu}))$ **then**
        $e_{merge} := (merge(\mu, \hat{\mu}), intersection([t_S, t_E), [\hat{t}_S, \hat{t}_E))$;
        **if** $e_{merge}.t_S > e.\tilde{t}_E$ **then**
            Insert $(\mu, [e.\tilde{t}_E, e_{merge}.t_S))$ into H;
        Insert $e_{merge}$ into H;
        $e.\tilde{t}_E = e_{merge}.t_E$;
    **else**
        intersect := intersection($[t_S, t_E), [\hat{t}_S, \hat{t}_E)$);
        **if** $intersect.t_S > e.\tilde{t}_E$ **then**
            Insert $(\mu, [e.\tilde{t}_E, intersect.t_S))$ into H;
            $e.\tilde{t}_E$ = intersect.$t_S$;
$SA_1$.insert(e);
**doRight**(LeftJoin op);
Iterator invalids = $SA_1$.extractElements(e);
**while** *invalids.hasNext()* **do**
    $\check{e}$ := invalids.next();
    **if** $\check{e}.\tilde{t}_E < \check{e}.t_E$ **then**
        Insert $(\check{e}.\mu, [\check{e}.\tilde{t}_E, \check{e}.t_E))$ into H;
Iterator qualifies ← $SA_1$.query(e);
**while** *qualifies.hasNext()* **do**
    $\hat{e} := (\hat{\mu}, [\hat{t}_S, \hat{t}_E)) \leftarrow$ qualifies.next();
    **if** $e \risingdotseq \hat{e} \wedge p(merge(\mu, \hat{\mu}))$ **then**
        $e_{merge} := (merge(\mu, \hat{\mu}), intersection([t_S, t_E), [\hat{t}_S, \hat{t}_E))$;
        **if** $e_{merge}.t_S > \hat{e}.\tilde{t}_E$ **then**
            Insert $(\hat{\mu}, [\hat{e}.\tilde{t}_E, e_{merge}.t_S))$ into H;
        Insert $e_{merge}$ into H;
        $\hat{e}.\tilde{t}_E = e_{merge}.t_E$;
    **else**
        intersect = intersection($[t_S, t_E), [\hat{t}_S, \hat{t}_E)$);
        **if** $intersect.t_S > \hat{e}.\tilde{t}_E$ **then**
            Insert $(\hat{\mu}, [\hat{e}.\tilde{t}_E, intersect.t_S))$ into H;
            $\hat{e}.\tilde{t}_E := intersect.t_S$;
$SA_2$.insert(e);
```

Another approach for the expression of the validity of data stream elements is the positive-negative tuple approach introduced in [10]. This approach allows for using so called negative tuples, that mark the end of their positive counterparts. But this approach has the disadvantage that at least twice the elements have to be processed in comparison to the interval based approach of [6] and that all algebra operators have to distinguish between different types of elements.

There are other approaches like [11,12] which also introduce stream processing languages. But the continuous query language CQL [11] uses the relational model in defining operators that transform a stream into a relation and vice versa. This is not useful for SPARQL stream processing because of the differences between the RDF and the relational models. It is the same with the extended XQuery language [12]. This language provides many constructs for handling XML data. Indeed RDF can be serialized as XML, but the extended XQuery constructs do not handle the RDF semantics correctly. So a real RDF query language has to be extended for stream processing.

Extending SPARQL for stream processing window operators had to be integrated into the SPARQL language. These operators have been introduced earlier (see [11, 12, 13, 14, 15]). Ideas for windows have been taken from these works, but also in this case the interval based approach in [6] offer advantages in implementing the corresponding physical algebra operators. So time based windows can be realized by calculating the end of a validity interval and tuple based windows can be realized by using a sweep area and setting the end of an earlier element to the start of a later element (see [6]).

8 Conclusions and Outlook

In this work we presented an extension to SPARQL to cope with window queries over data streams. We extended the SPARQL language to allow the definition of time and count based windows over data streams. We implemented the extended SPARQL processing in our Odysseus system, which is an enhancement of our DynaQuest [16] framework with the ability to process data streams. For the translation of SPARQL we used the ARQ project[5] and slightly extended their query translation process. We added to this base our own query translation and execution framework and extended it with the described logical and physical algebra operators. We showed that query processing over RDF streams is possible. We now need to determine if this format is applicable for wind park monitoring. We will do this in the context of the Alpha Ventus[6] project, which creates an offshore test platform in the North Sea. Another application for our approach might be in the context of mobile information systems as they are developed in the C3World[7] research group.

There are multiple possible extensions for this work. Predicate based windows [17] define the boundaries using predicates over the stream content. We have already developed initial approaches but further research is necessary. Additionally, we are currently extending SPARQL to support group by clauses and aggregation, which are necessary in monitoring applications. Also alternatives to using windows like the positive-negative tuple approach [10] will be evaluated in future work.

References

1. Babu, S., Widom, J.: Streamon: An adaptive engine for stream query processing. In: Demo Track Session of ACM SIGMOD Conference (2004)
2. International Electrotechnical Commission Technical Commitee 88: 61400-25 communications for monitoring and control of wind power plants(version 61400-25-1_r0-4draftfdis_2006-05-31). Technical report, International Electrotechnical Commission (2006)
3. Manola, F., Miller, E., McBride, B.: RDF Primer. W3C Recommendation. In: World Wide Web Consortium (2004)
4. Tatbul, N., Zdonik, S.: Window-aware load shedding for aggregation queries over data streams. In: VLDB 2006: Proceedings of the 32nd international conference on Very largedata bases, VLDB Endowment, pp. 799–810 (2006)

[5] http://jena.sourceforge.net/ARQ/
[6] http://www.alpha-ventus.de/
[7] http://www.c3world.de/

5. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation, World Wide Web Consortium (2007), `http://www.w3.org/TR/rdf-sparql-query/`
6. Krämer, J.: Continuous Queries over Data Streams - Semantics and Implementation. PhD thesis, University of Marburg (2007)
7. Bettini, C., Dyreson, C.E., Evans, W.S., Snodgrass, R.T., Wang, X.S.: A glossary of time granularity concepts. In: Temporal Databases, Dagstuhl, pp. 406–413 (1997)
8. Dayal, U., Goodman, N., Katz, R.H.: An extended relational algebra with control over duplicate elimination. In: PODS 1982: Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems, pp. 117–123. ACM Press, New York (1982)
9. Hayes, P., McBride, B.: RDF Semantics. W3C Recommendation (2004)
10. Ghanem, T.M., Hammad, M.A., Mokbel, M.F., Aref, W.G., Elmagarmid, A.K.: Query processing using negative tuples in stream query engines. Technical report, Purdue University (2004)
11. Arasu, A., Babu, S., Widom, J.: An abstract semantics and concrete language for continuous queries overstreams and relations. In: 9th Interantional Workshop on Database Programming Languages, Stanford University, pp. 1–11 (2003)
12. Carabus, I., Fischer, P.M., Florescu, D., Kraska, D.K.T., Tamosevicius, R.: Extending xquery with window functions. Technical report, ETH Zürich (2006)
13. Sullivan, M.: Tribeca: A stream database manager for network traffic analysis. In: Vijayaraman, T.M., Buchmann, A.P., Mohan, C., Sarda, N.L. (eds.) VLDB 1996, Proceedings of 22th International Conference on Very Large DataBases, Mumbai (Bombay), India, September 3-6, p. 594. Morgan Kaufmann, San Francisco (1996)
14. Carney, D., Çetintemel, U., Cherniack, M., Lee, C.C.S., Seidman, G., Stonebraker, M., Zdonik, N.T.S.B.: Monitoring streams - a new class of data management applications. In: Bressan, S., Chaudhri, A.B., Li Lee, M., Yu, J.X., Lacroix, Z. (eds.) CAiSE 2002 and VLDB 2002. LNCS, vol. 2590, pp. 215–226. Springer, Heidelberg (2003)
15. Seshadri, P., Livny, M., Ramakrishnan, R.: Seq: A model for sequence databases. In: ICDE 1995: Proceedings of the Eleventh International Conference on Data Engineering, pp. 232–239. IEEE Computer Society, Washington, DC, USA (1995)
16. Grawunder, M.: DYNAQUEST: Dynamische und adaptive Anfrageverarbeitung in virtuellen Datenbanksystemen. PhD thesis, University of Oldenburg (2005)
17. Ghanem, T.M., Aref, W.G., Elmagarmid, A.K.: Exploiting predicate-window semantics over data streams. SIGMOD Rec. 35(1), 3–8 (2006)