Using a Streaming SPARQL Evaluator for Monitoring eBay Auctions

Sven Groppe, Jinghua Groppe, Stefan Werner, Matthias Samsel, Florian Kalis, Kristina Fell
Peter Kliesch, Markus Nakhlah
Institute of Information Systems (IFIS)
University of Lübeck
Lübeck, Germany
{groppe, groppej, werner}@ifis.uni-luebeck.de,{a,f }@fmnptu.de, mail@kristinafell.de,{p,m}@fmnptu.de

ABSTRACT: *Data streams are becoming an important concept and used in more and more applications. Processing of data streams needs a streaming engine. The streaming engine can start query processing once initial data is available. This capability is especially important for real-time computation and for long-relay transmission of data streams. In this work, we demonstrate a monitoring system of eBay auctions, which is based on our RDF stream engine and can analyze eBay auctions in a flexible way. Using our monitoring system, users can easily monitor the eBay auctions information of interest, analyze the behavior of buyers and sellers, predict the tendency of auctions and make more favorable decisions. Furthermore, each step during RDF stream processing can be visualized allowing a better and easier understanding of the internal processes. This paper is an extended version of [22].*

Keywords: Semantic Web, Streams, SPARQL, RDF

Received: 10 June 2011, Received 6 August 2011, Accepted 10 August 2011

1. Introduction

A growing number of applications in areas like network monitoring, sensor networks and auction industry are using continuous *data streams* rather than finite stored data sets. Processing and querying data streams require long-running *continuous queries* as opposed to one-time queries.

Data produced over time form data streams. Data streams having no end are called infinite data streams. (*Infinite*) *data streams* are generated from e.g., sensors, which constantly obtain data from their environment. In order to determine useful conclusions (like a probably upcoming earth quake) from data streams, we need to consider the infinite nature and support the computation of intermediate results based on a *window*, which contains the recent data of the infinite data stream. In many scenarios, the intermediate results must be calculated in a timely fashion, e.g., a probably upcoming earth quake must be detected as early as possible allowing no delays for the computations.

Streaming query engines operating on data streams can (a) discard irrelevant input as early as possible, and thus save processing costs and space costs, (b) build indices only on those parts of the data, which are needed for the evaluation of the query, and (c) determine partial results of a query earlier, and thus evaluate queries more efficiently.

Stream-based processing enables more efficient evaluation not only in local scenarios, where the data is stored and the query engines run on the same computer, but also in many other applications, e.g.

• in integrating data over networks like the Internet, in particular from slow sources. It is desirable to progressively process the input before all the data is retrieved.

• in continuous query processing over infinite data streams (e.g., [3]), generated by e.g., sensors. *Continuous query processing* (e.g., [3]) evaluates queries periodically.

• in selective dissemination of information, where RDF data has to be filtered according to requirements given in a query, and

• in pipelined processing, where data is sent through a chain of processors, and the input of each processor is the output of the preceding processor.

The data format of the Semantic Web is RDF [9], and a large amount of data is described using RDF. SPARQL [28] is the standard RDF querying language and has been extended by several contributions to support operations in infinite RDF data streams (see e.g., [10] and [7]). In this paper, we demonstrate our streaming query engine by a real-world case: monitoring the eBay auctions based on querying a real-time eBay RDF data stream. Our demonstration application is online available and can be downloaded from [13].

This paper is an extended version of [22]. The extensions include the formalization of a streaming SPARQL algebra (see Section 6), which extends the algebra in [21] by window and stream operators and the possibility to delete triples and solutions from operators as well as to compute periodic results of the whole query.

2. RDF and SPARQL

The Semantic Web uses RDF [9] as its data format and SPARQL [28] as the basic query language of RDF data.

The core element of RDF data is the *RDF triple*, and a set of RDF triples is called an *RDF graph*.

Definition 1 (*RDF triple*): Assume there are pairwise disjoint infinite sets I, B and L, where I represents the set of IRIs [12], B the set of blank nodes and L the set of literals. We call a triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ an *RDF triple*, where s represents the subject, p the predicate and o the object of the RDF triple. We call an element of $I \cup B \cup L$ an *RDF term*.

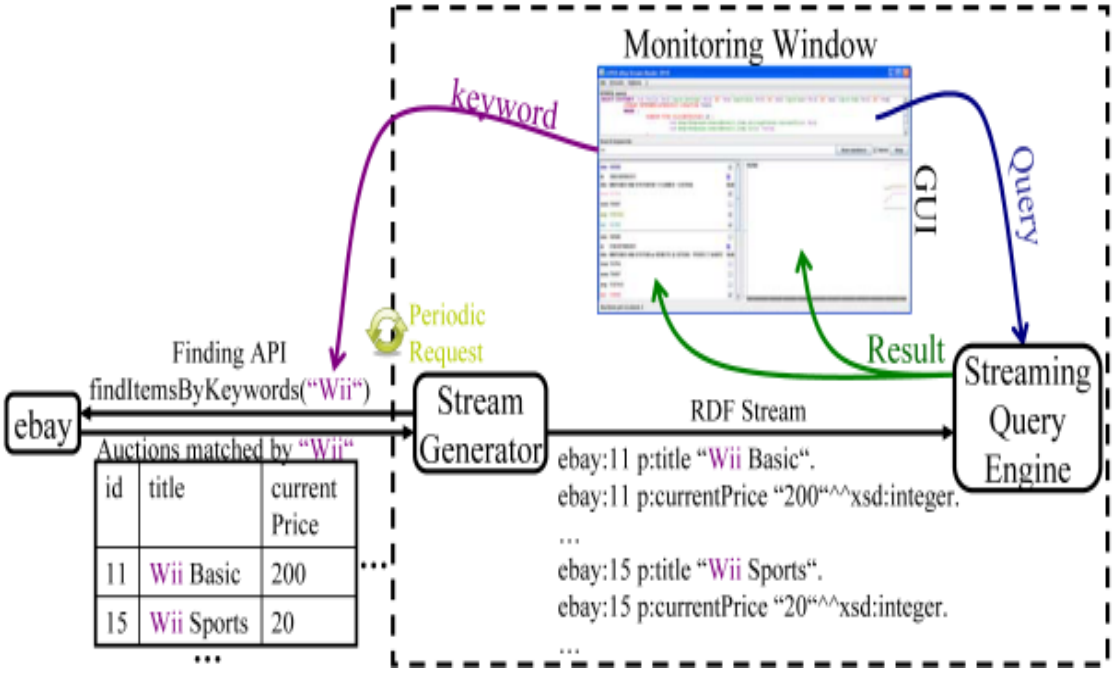

Figure 1. Monitoring System of eBay Auctions

SPARQL queries are evaluated on *RDF graphs*, and select data based on the matching of graph patterns of SPARQL. The core component of SPARQL graph patterns is a set of triple patterns *s p o*. *s p o* corresponds to the subject (*s*), predicate (*p*) and object (*o*) of an RDF triple, but they can be variables as well as RDF terms. Within a SPARQL query, the user specifies the known RDF terms of triples and leaves the unknown ones as variables in triple patterns.

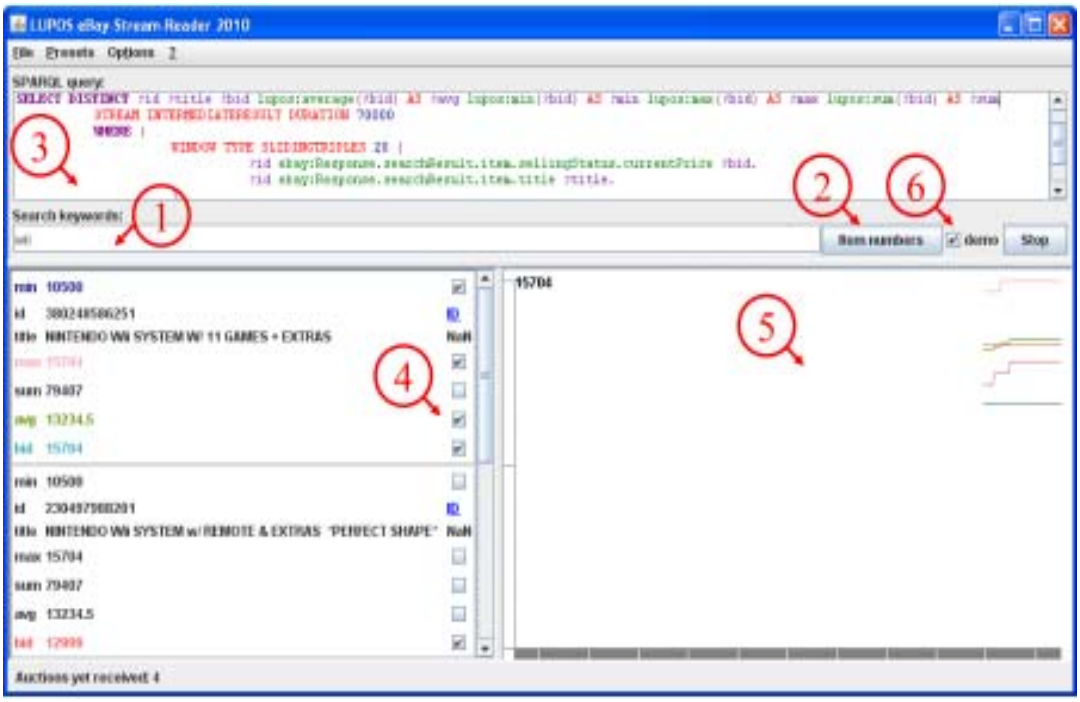

Figure 2. Main window of our demonstration

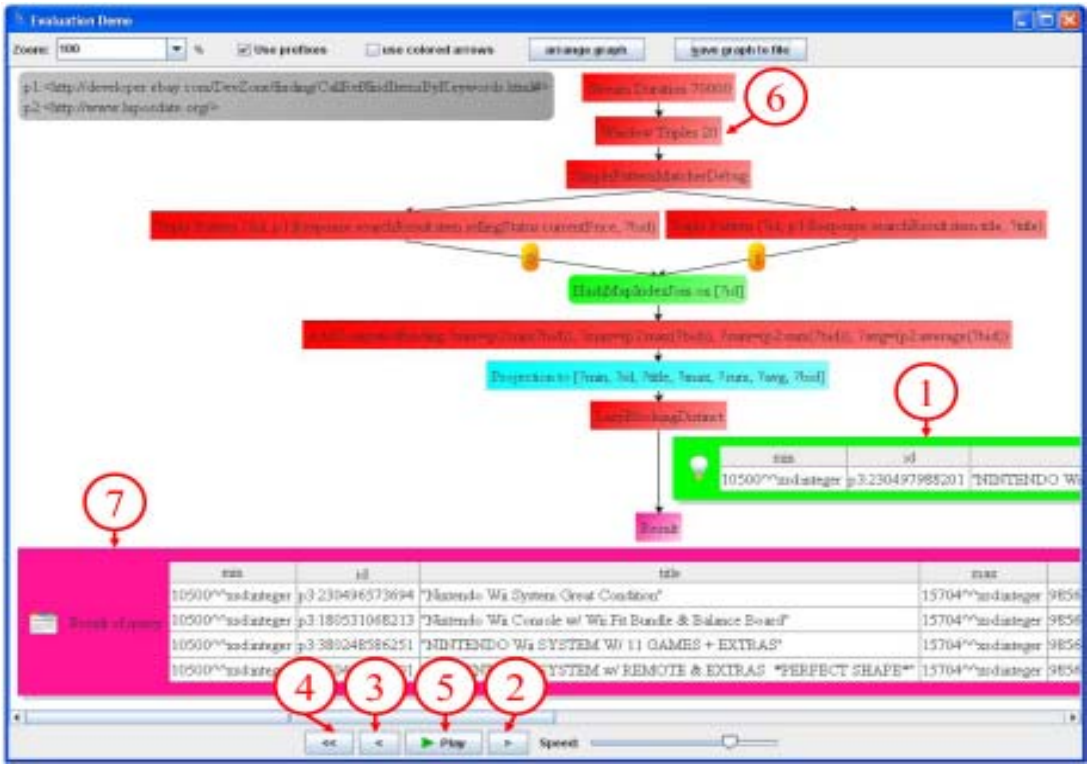

Figure 3. Demonstrating the evaluation of RDF streams

The same variables can occur in multiple triple patterns and thus imply joins. A triple pattern matches a subset of the RDF data, where the RDF terms in the triple pattern correspond to the ones in the RDF data.

Figure 4 presents the example RDF data Records.rdf with 3 triples, and Figure 5 an SPARQL query DLCRecords.sparql for this RDF data. DLCRecords.sparql consists of a SELECT clause and a WHERE clause. The SELECT clause identifies the variables to appear in the query results, i.e., the variable bindings of ?a and ?c. The WHERE clause contains two triple patterns, which identify the constraints on the input RDF data. The triple pattern ?a <records> ?c matches the first two triples of Records.rdf, and thus its result is {(?a=<ID1>, ?c=<ID6>),(?a=<ID2>, ?c=<ID5>)}. The triple pattern ?a <origin> <DLC> matches the last triple of Records.rdf, and thus its result is {(?a=<ID2>)}. The two triple patterns impose a join over the common variable ?a, and their result is hence {(?a=<ID2>, ?c=<ID5>)}, which is the final query result.

```
<ID1>  <records>  <ID6>
<ID2>  <records>  <ID5>
<ID2>  <origin>   <DLC>
```

Figure 4. Example RDF data Recods.rdf

```
SELECT  ?a  ?c
WHERE  {
            ?a<records>?c.
            ?a<origin><DLC>.
       }
```

Figure 5. SPARQL query DLCRecords.sparql

3. EBAY

eBay (http://www.ebay.com/) is a popular on-line auction and shopping website. Through eBay individuals and business sell and buy a wide variety goods and services, and millions of items are auctioned daily. Furthermore, the eBay Developers Program (http://developer.ebay.com/) offers several eBay web services, with which new applications, tools and value-added services can be created in order to meet the diverse needs of buyers and sellers on eBay.

The eBay web services use domains and aspects describe the auctioned items. A domain represents a kind of items, the aspects describe the characteristics of items in a given domain, and items are instances of a domain. For example, book can be a domain, and the title, author, pages and price can be the aspects of the book domain. A book entitled “Stream Processing” auctioned in eBay is an instance of the book domain. The information model1 used by eBay is very similar to the RDF data model.

Therefore, we can use RDF to describe eBay auctions, and thus leveraging SPARQL and RDF tools to query and process auction data. While the eBay’s Finding API supports the functionality of searching and browsing items listed on eBay, the RDF query language SPARQL provides more powerful capabilities than the eBay Finding API.

4. Monitoring EBAY Auctions

By monitoring the real-time eBay auctions of interest, the buyers and sellers can predict auction tendencies and make better decisions. Furthermore, it also helps the economists and researchers in analyzing various aspects of buying and selling behavior. In this section, we demonstrate how the users of our system can easily query and monitor the eBay auction information in which they are interested.

4.1 Monitoring System

Figure 1 describes our system of monitoring eBay Auctions. Our stream generator interacts with the eBay platform using the eBay web services. In this figure, the stream generator calls the eBay server with the function findItemsByKeywords(“Wii“), which retrieves and returns the auction information matched by the keyword “Wii”. Once the first data element arrives, the

[1]http://developer.ebay.com/DevZone/finding/Concepts/FindingAPIGuide.html

stream generator transforms it into the RDF format and sends it to the streaming query engine. The streaming query engine processes the SPARQL query on the RDF data stream and the results of the query are displayed in the monitoring window.

4.2 Demonstration

Figure 2 is a screen snapshot of the main window of our demonstration system. Users can specify the search keywords (see (1) of Figure 2) or eBay item numbers or the webpage address of the eBay auction (see (2) of Figure 2) for retrieving related information from eBay. The eBay item number can be found at its auction webpage. A SPARQL expression is used (see (3) of Figure 2) to query the data returned by the eBay server.Several pre-defined SPARQL queries can be obtained by clicking the menu item *Presets* in the top menu.

After specifying the query information, clicking the button *Start* starts the communication with the eBay server, the generation of RDF data and the evaluation of the SPARQL query over the RDF data stream. It might take some time to finish these processes, depending on various factors, e.g., the speed of networks, and the size of transmitted data.

The query result is displayed in the main window (see (4) of Figure 2). If a checkbox is marked in the query result, the numerical values are displayed in a chart (see (5) of Figure 2). The data is periodically retrieved and processed from eBay, and the query result periodically updated. The old query result still remains in the charts when updated. Consequently, users can easily observe and monitor the changes over time.

4.3 Streaming SPARQL Engine

Our streaming SPARQL engine supports an extended version of SPARQL by allowing windows and the specification of the periods for updating the query result. Figure 6 describes such a query for the RDF stream. Line (5) specifies the query result to be updated every second and lines (7) to (9) specify a window of the recent 100 triples of the RDF stream to be queried by the triple patterns in lines (8) and (9). Additional to SPARQL 1.0 [28], we support aggregation functions (see lines (3) and (4)) like average, min, max and sum to determine the average, minimum, maximum and summation.

```
(1)  PREFIX ebay:<http://developer.ebay.com/DevZone/finding/CallRef/
findItemsByKeywords.html#>
(2)  PREFIX lupos: <http://www.luposdate.org/>
(3)  SELECT DISTINCT ?id ?title ?bid lupos:average(?bid) AS ?avg
(4)   lupos:min(?bid) AS ?min lupos:max(?bid) AS ?max lupos:sum(?bid)AS?sum
(5)  STREAM INTERMEDIATERESULT DURATION 1000
(6)  WHERE {
(7)     WINDOW TYPE SLIDINGTRIPLES 100 {
(8)    ?id ebay:Response.searchResult.item.sellingStatus.currentPrice ?bid.
(9)     ?id ebay:Response.searchResult.item.title ?title. }}
```

Figure 6. Example query for RDF streams

Our demonstration also shows the internals of stream processing. Before processing a query, our streaming SPARQL engine parses the query and transforms it into a logically and physically optimized operator graph. If the checkbox “demo” (see (6) in Figure 2) is enabled, a window of the evaluation demo will be popped after clicking the button *Start*. The window demonstrates single execution steps of the query processing, and displays the operator graph of the SPARQL query, and the information transmission between the operators (see (1) of Figure 3).

The user can navigate through the processing steps by clicking on the next (see (2) of Figure 3) or previous (see (3) of Figure 3) step button. The user can also directly navigate to the first step (see (4) of Figure 3) or watch an animation of the processing steps (see (5) of Figure 3). The processing of the RDF stream is initialized by sending a Start-Of-Evaluation-Message to each operator. Incoming triples are transmitted along the operator graph until a Triple Pattern operator, which is evaluated on the incoming triples. The result (see (1) of Figure 3) is then transmitted to succeeding operators. The Window operator (see (6) of Figure 3) handles the window of considered triples for query evaluation. If a triple is out of the window, then the Window operator transmits the information of deleting this triple downwards. This information might cause a succeeding Triple Pattern

operator to delete a certain solution. The streaming engine triggers the periodic computation of the query by a Compute-Intermediate-Result-Message, and the query result is displayed (see (7) of Figure 3).

5. Special Operators for Stream Processing

The SPARQL algebra must be extended by basically two types of operators for stream processing:

1. The first one called Stream operator triggers the periodic computation of query results. The Stream operator is the root in operator graphs of stream queries.

2. The second one called Window operator implements that query processing only considers certain recent data instead of all data.

Both operator types can be found in the operator graph presented in Figure 3.

5.1 Types of Stream Operators

Stream operators differ in the way they determine when they trigger succeeding operators to compute an intermediate query result:

• The Stream Triples operator triggers the computation of an intermediate query result after a given number of triples have been arrived (and processed) independent from the time of the last computation.

• The Stream Duration operator starts the computation of intermediate query results after a certain time is over independent from the number of arrived (and processed) triples.

Both operator types have their applications: While the Stream Duration operator guarantees up-to-date query results, the Stream Triples operator allows computing the query result only when there are many changes in the input.

5.2 Types of Window Operators

Window operators can differ how one can define the triples to be considered:

• The Window Triples operator considers only the last recent triples (up to a specified number of triples) for query processing. The Window Triples operator finds its applications e.g. whenever new triples update the values of older ones and these newer triples should be only considered during query processing.

• The Window Duration operator considers only those triples, which have been arrived in recent time (up to a specified time period), for query processing. The Window Duration operator is a necessity to compute aggregation functions over a certain time period, e.g., the average temperature of the last hour.

More types of Window operators exist. We refer the interested reader to e.g. [3].

6. Streaming SPARQL Algebra

There are two ways to describe the streaming SPARQL algebra. The trivial variant uses the traditional (nonestreaming) SPARQL algebra and extends it by Stream and Window operators, which define the times of periodic query evaluations and the RDF dataset to be considered. However, the trivial variant describes a normal query evaluation with the actual RDF dataset, but does not describe an incremental query evaluation, where triples are immediately processed by the operators in the operator graph and (together with their caused intermediate results) are released if they do not need to be considered any more. We describe the SPARQL algebra of this incremental variant.

We formalize the special operators for stream processing as well as some important operators for SPARQL processing in the following paragraphs. For this purpose, we define also the term environment, which is used to define an intermediate (and final) result of the operators. As one part of the environment is a binding of a value to a variable, we first define this term accordingly.

Definition 2: A *binding* of a variable is a tuple (n,v) where n represents the name of the variable and *v* its current value.

Definition 3: An environment E is a set of bindings of variables, where each variable of the bindings in E has exactly one assigned value, i.e. $\forall (n, v_1) \in E: \forall (n, v_2) \in E: v_1 = v_2$.

In the following paragraphs, we enumerate the operators of the proposed streaming SPARQL algebra. The proposed operators provide core functionalities for SPARQL processing, which abstract from SPARQL syntax and equivalent language constructs, and are the basis for logical and physical optimization. The proposed operators for the streaming SPARQL engine are event-based. For every incoming triple of the RDF data stream the Stream operator triggers an *event* at its succeeding operators, which may trigger events at their succeeding operators as well. All operators, their succeeding and preceding operators span an *operator graph*. Figure 7 contains the operator graph of the streaming SPARQL query of Figure 6.

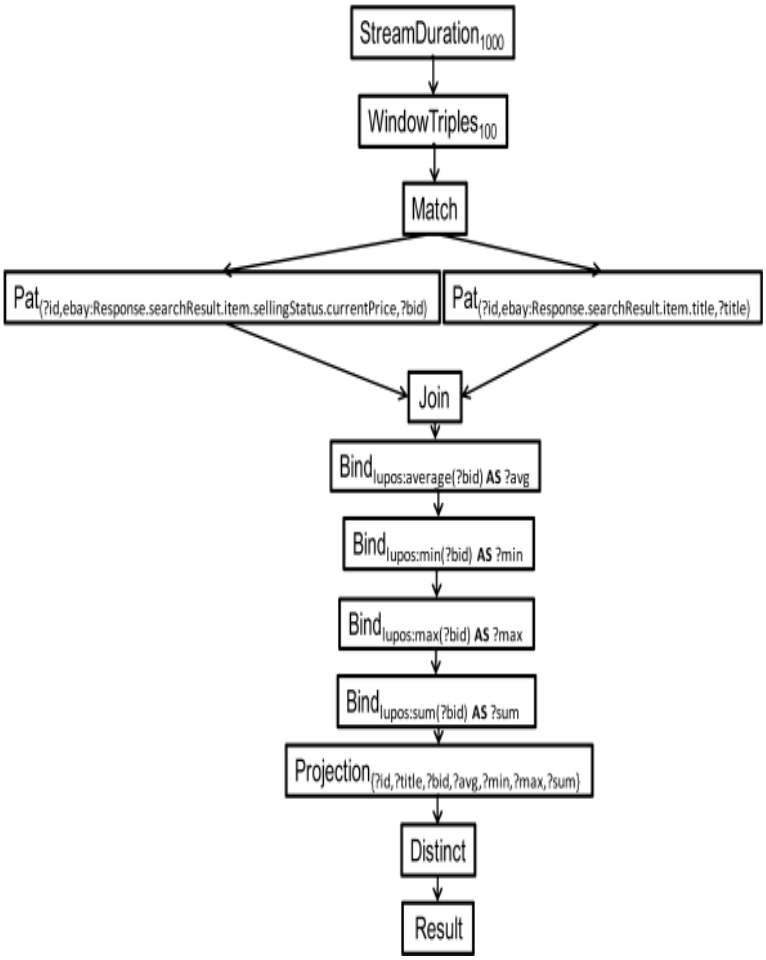

Figure 7. The operator graph of the streaming SPARQL query of Figure 6

For each operator, we describe the actions and the resulting events of the operators after receiving an event ti from the i-th preceding operator in pseudo code. Note that we describe the resulting event in a declarative way, i.e. we only describe what the resulting events look like and not how to compute the resulting events.

In the definitions of the operators, we describe the optional initialization after the keyword "Init", the events to receive after the keyword "Input", the actions to be done for each received event after the keyword "Actions" and the actions to be done after the RDF data stream has been closed, i.e. the RDF data stream has no incoming triples any more, after the keyword "Final" (optional) in pseudo code. The actions for computing a valid periodic result of the whole query are described after the computeResult keyword. If no *computeResult* keyword appears in the description of the operators, then the *computeResult* action just triggers the *computeResult* actions at its succeeding operators. The actions to be done for deletion of a triple or environment can be found after the keyword "Delete". We may define a subroutine after the keyword $Helper_{op}$, which supports the parameter op for defining a specific instruction (trigger or delete). If a subroutine $Helper_{op}$ occurs in the description of an operator, but no Actions and Delete actions, then the Actions and Delete actions are defined by the following statements:

```
Actions: Helper_trigger;
Delete: Helper_delete;
```

While t_i represents an event from the i-th preceding operator, we use *t* to represent an event if there is only one preceding operand or if the events of the preceding operators do not need to be distinguished. t_i.E represents an attached environment and t_i.*S* an attached triple of the event t_i.*num* represents the concrete number of operands of an operator. The instruction $trigger_p$ E triggers an event with attached environment/triple E at the operator *p* and trigger E triggers at all succeeding operators. Accordingly, the instruction $delete_p$ E deletes an environment/triple E at the operator *p* and deletes E deletes E at all succeeding operators. The instruction *computeResul*t triggers the actions for *computeResult* at all succeeding operators.

We may use a queue in the description of operators, which supports operations to add elements at the end of its elements, to access its first element, remove the first or any given element from its stored elements. Furthermore, we sometimes use multisets, which have the same operations like sets, but additionally consider the number of its elements when adding new elements or removing existing ones.

We enumerate the operators in the following items:

1. Every operator graph has a *Stream or* $StreamTriples_{numberOfTriples}$ *or* $StreamDuration_{duration}$ operator as root node. The Stream operator receives the incoming triples of the RDF data stream and transmits these triples as events to its succeeding operators (which typically consist of pattern matcher operators or window operators). The Stream operator is used whenever no STREAM clause occurs in the query.

```
Operator Stream
Input:   RDF data stream with incoming triple s
Actions: s.setTimestamp(getTimestamp());
         trigger s;
Final:   computeResult;
```

The $StreamTriples_{numberOfTriples}$ operator triggers the computation of an intermediate query result after a given number of triples (numberOfTriples) have been arrived (and processed) independent from the time of the last computation.

```
Operator StreamTriples_numberOfTriples
Init:    count=0;
Input:   RDF data stream with incoming triple s
Actions: s.setTimestamp(getTimestamp());
         trigger s;
         count=count+1;
         if(count>=numerOfTriples)
         then { count=0; computeResult; }
Final:   computeResult;
```

The $StreamDuration_{duration}$ operator starts the computation of intermediate query results after a certain time (duration) is over independent from the number of arrived (and processed) triples.

```
Operator StreamDuration_duration
Init:    timestamp=getTimestamp();
Input:   RDF data stream with incoming triple s
Actions: interTimestamp = getTimestamp();
         s.setTimestamp(interTimestamp);
         trigger s;
         If(timestamp - interTimestamp >=duration)
         then { timestamp = interTimestamp; computeResult; }
Final:    computeResult;
```

2. The window operators specify the triples to be processed from its succeeding operators.

The $WindowTriples_{numberOfTriples}$ operator considers only the last recent triples (up to a specified number of triples (numberOfTriples)) for query processing.

```
Operator  WindowTriples_numberOfTriples
Init:     queue.init();
Input:    Event t with attached triple
Actions:  If(|queue| >= numberOfTriples)
          then { delete queue.first(); queue.removeFirst(); }
          queue.addLast(t);
          trigger t;
Final:    queue.release();
```

The $WindowDuration_{duration}$ operator considers only those triples, which have been arrived in recent time (up to a specified time period (duration)), for query processing.

```
Operator  WindowDuration_duration
Init:     queue.init();
Input:    Event t with attached triple
Actions:  now=getTimestamp();
          ∀ s∈queue:s.timestamp()- now>= duration:{queue.remove(s); delete s;}
          queue.addLast(t);
          trigger t;
Final:    queue.release();
```

3. The pattern matcher operator $Match_{Pats}$ triggers all triple pattern operators Pats with an incoming event with attached triple. Note that implementations of the pattern matcher may choose matching triple pattern operators in a more intelligent way for speeding up processing of SPARQL queries as part of the physical optimization.

```
Operator  Match_Pats
Input:    Event t with attached triple
Helper_op: ∀ p∈Pats:  op_p  t.S;
```

4. The triple pattern operator $Pat_{(p1, p2,p3)}$ represents a triple pattern ($p1$, $p2$, $p3$) (e.g. $Pat_{(?id,ebay:Response.searchResult.item.title, ?title)}$ for line (9) of Figure 6). The resulting event of $Pat_{(p1, p2,p3)}$ is triggered when the attached triple of the received event matches the triple pattern, i.e. all literals in the triple pattern are also in the attached triple at the same position and the variables are only bound to one value. An environment is attached to the resulting event, where the variables of the triple pattern are bound to the corresponding values of the considered triple. For example, $Pat_{(?id, ebay:Response.searchResult.item.title, ?title)}$ triggers an event with attachedenvironment{(*id,ebay:book*1),(*title,*"*title*1")}forthetriple(*ebay:book*1, *ebay:Response.searchResult.item.title,"title*1"), but triggers no event for the triple (*ebay:book*1, *ebay:price*, 22).

```
Operator     Pat_(p1,p2,p3)
Input:       Event t with attached triple t.S=(s_1,s_2,s_3)
Helper_op:   E={(x,v)|i∈{1,2,3}∧x=pi ∧pi is a variable∧v=s_i)};
             if((∀ j∈{1,2,3}:(p_j is a variable)∨(p_j=s_j))∧∀ (n,v_1)∈E:∀ (n,
             v_2)∀ E:v_1=v_2) then op E;
```

5. The join operator Join represents a join of environments of received events of several operands (e.g. one join operator for the triple pattern operators representing the triple patterns from line (8) to (9) in Figure 6). An environment of a received event of one operand is joined with all environments of previously received events of the other operands. The Join operator triggers events

with all joined environments of the received environment and all previously received environments of the other operands whenever the join condition is fulfilled. The join condition requires that variables with the same name (which are called *join partners*) are bound to the same value.

```
Operator Join
Init:     ∀i∈{1,...,num}:SE_i={}; // SE_i are multisets!
Input:    Event t_i with attached environment, where i∈{1,...,num}
Actions:  SE_i= SE_i∪t_i.E;
            Helper_trigger;
Delete:   SE_i = SE_i - t_i.E;
           Helper_delete;
Helper_op:∀E∈{ s_1∪...∪s_num|∀j∈{1,...,num}- {i}:s_j∈SE_j∧s_i=t_i.E∧∀(n,v_1)∈
             s_1∪...∪s_num:∀(n,v_2)∈s_1∪...∪s_num:v_1=v_2}:op E;
```

6. The filter operator Selexpression evaluates a filter expression expression based on the environment of the received event in order to transmit the environment to its succeeding operators or to discard this environment.

```
Operator   Sel_expression
Input:     Event t with attached environment
Helperop: if(expression(t.E)) then op t.E;
```

7. The *Optional* operator joins the environments, which are attached to the events of its two operands, where the second operand is transformed from an optional graph pattern, and triggers the joined environments afterwards. After the RDF data stream has been closed, the *Optional* operator triggers the environments received from events of the first operand, which have not been joined so far with the environments from the second operand.

```
Operator Optional
Init:      SE_1={};SE_2={};SE_joined={}; // SE_1,SE_1 and SE_joined are multisets!
Input:    Event t_i with attached environment, where i∈{1,2}
Actions:  SE_i = SE_i∪t_i.E;
           ∀(e_1,e_2)∈{(s_1,s_2)|∀j∈{1,2}-{i}:s_j∈SE_j
             ∧s_i=t_i.E∧∀(n,v_1)∈s_1∪s_2:∀(n,v_2)∈s_1∪s_2:
              v_1=v_2}:{SE_joined = SE_joined∪e_1;trigger e_1∪e_2;}
Delete:    SE_i=SE_i-t_2.E;
           ∀(e_1,e_2)∈{(s_1,s_2)|∀j∈{1,2}-{i}:s_j∈SE_j
           ∧s_i=t_i.E∧∀(n,v_1)∈s_1∪s_2:∀(n,v_2)∈s_1∪s_2:
             v_1=v_2}:{SE_joined = SE_joined-e_1;delete e_1∪e_2; }
Final:     ∀E∈SE_1-SE_joined:trigger E;
           computeResult: ∀E∈SE_1-SE_joined: trigger E;
         computeResult;
           ∀E∈SE_1-SE_joined: delete E;
```

The projection operator Projection_V excludes those bindings of an environment of a received event, which are not contained in V. For example, we use $\text{Projection}_{\{?id,\ ?title,\ ?bid,\ ?avg,\ ?min,\ ?max,\ ?sum\}}$ for the SELECT clause (except DISTINCT and the select-expressions for aggregation functions for which we have own operators) in line (3) and (4) of Figure 6.

```
Operator Projection_V
Input:   Event t with attached environment
Helperop:op{(n,v)|(n,v)∈t.E∧n∈V};
```

8. The distinct operator Distinct represents the option DISTINCT in SELECT clauses (as in line (3) of Figure 6). The distinct operator triggers the succeeding operator(s) with the environments of received events, if the environment is different from environments of previously received events.

```
Operator Distinct
Init:  E_previous={};
Input: Event t with attached environment
Actions: if(t.E∉E_previous) then {E_previous=E_previous∪t.E; trigger t.E;}
Delete: if(t.E∈E_previous) then {E_previous=E_previous -t.E;delete t.E;}
```

9. The union operator Union triggers the unchanged environments of each received event of different operands.

```
Operator Union
Input:   Event t with attached environment
Helper_op:opt.E;
```

10. The Bind operator adds a binding to the environments of each received event. The binding typically contains the computation of a select-expression such as lupos:average(?bid) **AS** ?avg in line (3) of Figure 6.

```
Operator Bind_expression AS ?v
Input:   Event t with attached environment
Helper_op:opt.E∪{(v,expression(t.E))};
```

11. Every operator graph has a Result operator as leaf. The events for the output operator Result contain the results of the operator graph, such that the operator Result triggers the application with the environments of each received event in order to transmit the resultant bindings of the operator graph. If the operator retrieves a computeResult event, then the application is informed that the current result is a valid intermediate result (which considers also the correct result of optional operators).

```
Operator Result
Input:   Event t with attached environment
Helper_op:opt.E;
```

7. Related Work

We divide the related contributions into those dealing with data streams in general and those especially for Semantic Web streams:

7.1 Data Streams in general

The Chronicle [23] data model introduced data streams by describing chronicles as append-only ordered sequence of tuples and an algebra operating over chronicles as well as over traditional relations. Distributed stream management is supported in OpenCQ [26], NiagraCQ [11] and Aurora [6], which evolved into the Borealis project [1]. [4] addresses continuous queries over data streams, which evolved into the development of the CQL [2] [3] [27] query language tailored for data streams. [24], [25] and [5] deal with mining data streams.

Especially [5] extensively considers data aggregation in streams. Rewriting techniques for streaming aggregation queries are discussed in [14].

7.2 Semantic Web Streams

We have proposed a SPARQL engine processing finite data streams in [21] and have there defined corresponding logical and physical operators. To the best of our knowledge, our contribution in [21] reports the first streaming SPARQL engine. At that time, we did not support window functions. [10] firstly introduced a window-based processing of RDF streams. [7] and [8] have further extended the syntax of SPARQL by aggregates and timestamp functions, but restrict the functionalities by allowing only one Window per stream.Apart from supporting the aggregates and timestamp functions, we also allow several windows per stream. [28] describes a first approach to reasoning on data streams.

8. Summary and Conclusions

Our demonstration [13] shows the importance of streaming query engines in data-stream applications. By using the RDF data stream and its querying language SPARQL, our monitoring system obtains big benefits in realtime data processing, and it provides users with the capabilities of observing, analyzing and predicating the behavior and pattern of eBay buyers and sellers. Furthermore, our system can also stepwise display the internal processes of querying RDF streams, which helps ones better and easier understand the RDF stream processing technology.

References

[1] Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A. S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y. and Zdonik, S. (2005). The Design of the Borealis Stream Processing Engine. *CIDR*.
[2] Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., Widom, J. (2003). STREAM: The Stanford Stream Data Manager (Demonstration Description). *SIGMOD 2003*, p. 665.
[3] Arasu, A., Babu, S., Widom, J. (2006). The CQL continuous query language: semantic foundations and query execution. *VLDB Journal*, 15 (2) 121-142.
[4] Babu, S. and Widom, J. (2001). Continuous Queries over Data Streams. *SIGMOD Rec.*, 30 (3) 109-120.
[5] Bai, Y., Thakkar, H., Wang, H., Luo, C., Zaniolo, C. (2006). A Data Stream Language and System Designed for Power and Extensibility. *CIKM*, p. 337-346.
[6] Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Galvez, E., Salz, J., Stonebraker, M., Tatbul, N., Tibbetts, R. and Zdonik, S. (2004). Retrospective on Aurora. *The VLDB Journal*, 13 (4) 370- 383.
[7] Barbieri, Davide Francesco., Braga, Daniele., Ceri, Stefano., Valle, Emanuele Della., Grossniklaus, Michael. (2009). CSPARQL: SPARQL for continuous querying, *WWW*, Madrid, Spain.
[8] Barbieri, Davide Francesco., Braga, Daniele., Ceri, Stefano., Valle, Grossniklaus, Michael, (2010). An Execution Environment for C-SPARQL Queries, *EDBT*, Lausanne, Switzerland.
[9] Beckett, D. (editor), (2004). RDF/XML Syntax Specification (Revised). W3C Recommendation, 10th February.
[10] Bolles, A., Grawunder, M., Jacobi, J. (2008). Streaming SPARQL - Extending SPARQL to Process Data Streams. *ESWC*, Tenerife, Spain.
[11] Chen, J., DeWitt, D. J., Tian, F., Wang, Y. (2000). NiagaraCQ: A Scalable Continuous Query System for Internet Databases. SIGMOD, p. 379-390.
[12] Dürst, M., Suignard, M. (2005). Internationalized Resource Identifiers (IRIs), http://www.ietf.org/rfc/rfc3987.txt, *W3C Memo*.
[13] Fell, Kristina., Kalis, Florian., Samsel, Matthias (2010). LUPOS eBay-Stream-Reader 2010, http://lupos.metawort.de/.
[14] Golab, L., Johnson, T., Koudas, N., Srivastava, D., Toman, D. (2008). Optimizing Away Joins on Data Streams. *In:* Proc. SSPS, p. 48-57.
[15] Groppe, S. (2011). Data Management and Query Processing in Semantic Web Databases, Springer.
[16] Groppe, J., Groppe, S. (2011). Parallelizing Join Computations of SPARQL Queries for Large Semantic Web Databases, *In:*26th Symposium On Applied Computing (ACM SAC 2011), Tai Chung, Taiwan.
[17] Groppe, J., Groppe, S. (2009). Ebers and V. Linnemann, Efficient Processing of SPARQL Joins in Memory by Dynamically Restricting Triple Patterns, *ACM SAC*.
[18] Groppe, J., Groppe, S., Schleifer, A., Volker Linnemann, (2009). LuposDate: A Semantic Web Database System, *ACM CIKM*, Hong Kong, China.
[19] Groppe, S., Groppe, J. (2010). External Sorting for Index Construction of Large Semantic Web Databases, *ACM SAC*.
[20] Groppe, S., Groppe, J. (2009). LUPOSDATE Demonstration, http://www.ifis.uniluebeck. de/index.php?id=luposdate-demo.
[21] Groppe, S., Groppe, J., Kukulenz, D., Linnemann, V. (2007). A SPARQL Engine for Streaming RDF Data. SITIS, Shanghai, China.
[22] Groppe, Sven., Groppe, Jinghua., Werner, Stefan., Samsel, Matthias., Kalis, Florian., Fell, Kristina., Kliesch, Peter., Nakhlah, Markus (2011). Monitoring eBay Auctions by querying RDF Streams, *In:*Proceedings of the Sixth International Conference on Digital Information Management (ICDIM 2011), Trinity College, The University of Melbourne, Australia.
[23] Jagadish, H. V., Mumick, I. S., Silberschatz, A. (1995). View Maintenance Issues for the Chronicle Data Model. *PODS*, p. 113-124.
[24] Law, Y.-N., Wang, H., Zaniolo, C. (2004). Query Languages and Data Models for Database Sequences and Data Streams. *VLDB*, p. 492-503.
[25] Law, Y.-N. , Zaniolo, C. (2005). An Adaptive Nearest Neighbor Classification Algorithm for Data Streams. PKDD, p. 108-120.

[26] Liu, L., Pu, C., Tang, W. (1999). Continual Queries for Internet Scale Event-Driven Information Delivery, *IEEE Trans. Knowl. Data Eng.*, 11 (4) 610-628.
[27] Munagala, K., Srivastava, U., Widom, J. (2007). Optimization of Continuous Queries with Shared Expensive Filters. *PODS*, p. 215-224.
[28] Prud'hommeaux, E., Seaborne, A. (2008). SPARQL Query Language for RDF, W3C Recommendation.
[29] Walavalkar, O., Joshi, A., Finin, T., Yesha, Y. (2008). Streaming Knowledge Bases. *SSWS*.