EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning

Darko Anicic
FZI Research Center for Information Technology
Karlsruhe, Germany
darko.anicic@fzi.de

Paul Fodor
Stony Brook University
Stony Brook, NY, U.S.A
pfodor@cs.sunysb.edu

Sebastian Rudolph
Karlsruhe Institute of Technology
Karlsruhe, Germany
rudolph@kit.edu

Nenad Stojanovic
FZI Research Center for Information Technology
Karlsruhe, Germany
nenad.stojanovic@fzi.de

ABSTRACT

Streams of events appear increasingly today in various Web applications such as blogs, feeds, sensor data streams, geospatial information, on-line financial data, etc. Event Processing (EP) is concerned with timely detection of compound events within streams of simple events. State-of-the-art EP provides on-the-fly analysis of event streams, but cannot combine streams with *background knowledge* and cannot perform *reasoning* tasks. On the other hand, semantic tools can effectively handle background knowledge and perform reasoning thereon, but cannot deal with rapidly changing data provided by event streams.

To bridge the gap, we propose *Event Processing SPARQL* (*EP-SPARQL*) as a new language for complex events and Stream Reasoning. We provide syntax and formal semantics of the language and devise an effective execution model for the proposed formalism. The execution model is grounded on logic programming, and features effective event processing and inferencing capabilities over temporal and static knowledge. We provide an open-source prototype implementation and present a set of tests to show the usefulness and effectiveness of our approach.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Rule-Based Databases; D.1.6 [**Logic Programming**]:

General Terms

Languages, Algorithms

Keywords

ETALIS, Complex Event Processing, Semantic Web, Rule Systems, Streams

1. INTRODUCTION

In the recent decade, information representation on the Web has undergone a shift from static or quasi-static to dynamic. The average size of singular information items has become smaller (compare, e.g. blogs with tweets) and their mutual temporal relatedness gained in importance. In many domains the view on information has changed from a bag-of-knowledge to a stream-like, event-based perspective.

In general, an *event* is something that occurs, happens or changes the current state of affairs. For example, an event can be a tweet, geospatial data sent by a GPS-enabled device, a newly updated status of a Web 2.0 application user or the status of an object there. In order to describe more complex dynamic matters that involve several events, formalisms have been created which allow for combining (simpler) events into compound ones, so-called *complex events*, using different event operators and *temporal* relationships. The field of Event Processing[1] (EP) has the task of real-time processing streams of events with the goal of detecting complex events, according to meaningful event patterns.

Current EP systems are based on database stream technologies [1, 6, 8]. They provide on-the-fly analysis of data streams enabling real-time decisions and actions, but fall short of combining streams with higher-level knowledge representation and reasoning necessary for handling background knowledge describing the *context* or *domain* in which streaming data are interpreted. After all, both *semantic* as well as *temporal* relationships are needed for an appropriate description of complex events. On the other hand, standard Semantic Web technologies allow for handling background knowledge in the form of ontologies representing time-invariant or slowly evolving knowledge. However, the task of conjunctively reasoning over streaming data and background knowledge constitutes a new challenge known as *Stream Reasoning* [10].

The goal of this work is to provide a fundamental framework for *Event Processing* and *Stream Reasoning* exceeding the state-of-the-art in both EP and Semantic Web. We believe that such a framework is needed in order to address *dynamic aspects* in streaming knowledge, and move toward the *Real-time Semantic Web*. Our contribution involves:

- **EP-SPARQL: A language for Event Processing and Stream Reasoning.** We provide the syntax

[1]Also known as Complex Event Processing (CEP).

WWW 2011, March 28–April 1, 2011, Hyderabad, India.
ACM 978-1-4503-0632-4/11/03.

and a formal semantics of a new language called *Event Processing SPARQL* (EP-SPARQL). The language extends the SPARQL language with its event processing and stream reasoning capabilities;

- **A logic-based account for Event Processing and Stream Reasoning.** We provide the basic mechanism for Event Processing and Stream Reasoning, grounded in Logic Programming. EP-SPARQL is a high-level language based on this mechanism. Our approach is based on *event-driven backward chaining rules* that realize an effective event-driven inferencing. It features both effective event processing, and inference capabilities over *temporal* and *static* knowledge. We are aware of no approach that implements both features in one clean, logic framework;
- **Implementation and evaluation.** We provide an open-source prototype implementation of the proposed approach. The prototype is implemented in Prolog language (and could be realized in other declarative rule languages, too). We have conducted a set of tests to show the usefulness and effectiveness of our approach with respect to expressivity and run-time performance.

2. RELATED WORK

The work related to ours mainly fits into three areas: *Streaming Database* systems [1, 16, 8, 9], *temporal RDF* models [12, 17, 20], and *Stream Reasoning* approaches [4, 5, 21, 7].

Streaming Databases. Database approaches [1, 16, 8, 9] are based on languages with SQL-like syntaxes, and database execution models adapted to process streaming data. These approaches are dominant today due to their capability to handle large volumes of streaming data with low latency. As such database approaches are widely used in automated stock trading, logistic services, transaction management, business intelligence etc. However, they are not well suited for applications including Web structured data, ontologies, and other forms of knowledge bases (e.g., Linked Open Data) where support for *semantic-based* event processing and *reasoning* is required. We also showed on two examples (Esper and TelegraphCQ, which are systems that belong to this category of related work) that these systems do not necessarily perform better than our approach based on logic programming techniques.

Temporal RDF. The work in [12] introduces *time* as a new dimension in RDF graphs. The authors provide a semantics for temporal RDF graphs and a temporal query language for RDF, following concepts of temporal databases. They are concerned with evolution of RDF graphs through time, and provide a framework for temporal entailment and *querying* over changing graphs.

Our work differs from this approach in that our aim is to detect temporal complex patterns in (near) real time rather than just once posing a query and getting a singular response. We want to detect situations of interest *continuously* as soon as they happen. Hence patterns need to be continuously evaluated in order to process occurrences of relevant triples, and further to recognize complex events.

SPARQL-ST [17] is an extension of SPARQL for complex spatial and temporal queries. The language and a corresponding implementation deal with temporal data (and possible reasoning about that data). However as in [12], SPARQL-ST queries need to be triggered rather than being continuously active. The same argumentation also applies to other SPARQL approaches like Temporal SPARQL [20], stSPARQL [14], and T-SPARQL [11].

The work in [19] motivates the need for a semantic management of streaming data. Streaming data are represented in RDF format with the purpose of its exploitation in semantic-web applications (semantically annotated data and reasoning services). For this purpose, they propose a Time-Annotated RDF model and Time-Annotated SPARQL. However the authors explicitly mention that continuous queries, as one typical requirement of streaming data management systems, are not considered in that work.

Stream Reasoning. Continuous SPARQL (C-SPARQL) [5] is a language for continuous query processing and Stream Reasoning. It extends the SPARQL language by adding support for window and aggregation operations. In C-SPARQL, the set of currently valid RDF statements is determined based on a query (including its window specification), and classical reasoning on that RDF set is performed as if it were static. In our work, we focus more on detection of RDF triples in a specific *temporal* order (e.g., sequence versus conjunction). We strongly believe that additionally temporal relatedness between events (e.g., within the sliding window, an event happened before another event) as defined in streaming database systems [1, 8, 9] is required to capture more complex patterns over RDF streaming data. Additionally, in C-SPARQL queries are divided into static and dynamic parts. The static part is evaluated by a RDF triple storage, while a stream processing engine evaluates the dynamic part of the query. In such settings, these two parts act as "black boxes" and C-SPARQL cannot take advantage of a query pre-processing and optimizations over the unified (static and dynamic) data space. We propose an approach based on logic rules where the both parts are handled in a uniform framework.

Streaming Knowledge Bases [21] is a reasoner dealing with streaming RDF triples and computation of RDFS closures with respect to an ontology. For instance, the reasoner can identify a triple from a stream having a subject that is an instance of a certain class (or any of its subclasses, defined in an ontology). The approach is based on TelegraphCQ [8] that is an efficient data stream system. In order to speed up stream reasoning, the authors propose to pre-compute all inferences in advance, and to store them in a database. Although this is an interesting approach, we believe that stream reasoning demands both: on-the fly-inference capabilities, and scalability.

The work in [7] introduces Streaming SPARQL. The approach is built on temporal relational algebra, and the authors provide an algorithm to transform SPARQL queries to that algebra. Similarly as in [5], the approach does not detect sequences of RDF triples occurring in a specific order.

The same holds for [4] where the authors propose stream reasoning based on incremental maintenance of materializations. Streaming RDF triples (as they occur) trigger an inference procedure that maintains materializations. Although promising, it is not clear how this approach works for multiple queries with different time window definitions (in [4], materializations are assumed to be maintained only for one query).

3. PROBLEM STATEMENT

We argue – and the above review of current approaches in the literature clearly witnesses this – that Event Processing and Stream Reasoning in the Semantic Web as two research disciplines may contribute to and complement each other, and hence open new possibilities in direction of the Real-time Semantic Web. These areas served as design principles we followed when proposing EP-SPARQL.

We see the following dimensions where current Stream Reasoning research can greatly benefit from Event Processing (EP):

Support for Rapidly Changing Information on the Web. While existing semantic technologies and reasoning engines are constantly being improved in dealing with *time invariant* domain knowledge, they lack in support for processing *real-time* streaming data (events). Real-time Web data is valuable only if it is captured, processed, and delivered instantly. EP is a set of techniques and tools that help us understand and control real-time and event-driven systems [15]. As such, it is a technology that can help in processing real-time data on the Web, too.

Information Push versus Pull. According to Wikipedia, the Real-Time Web is a set of technologies and practices which enable users to receive information as soon as it is published by its authors, rather than requiring periodic updates. Hence, there is no need to *pull* information, it will be delivered to users nearly at the moment it is published. For instance, no more waiting for web services to communicate from one polling instance to another. We notice a paradigm shift from information *pull* to information *push*; or from *request-response* based web services to *event-driven* web services with possibly unforeseen consequences.

On the other hand, Semantic Web technologies clearly surpass current EP approaches in the following aspects:

Structured Events. Various sensors, gps-enabled devices and the Internet of Things are all sources of events that can easily be structured. Today event stream systems do not use ontologies to specify common-agreed vocabularies for events and event-driven applications. An important contribution of the Semantic Web to EP is to provide *structured events*. This will enable not only *knowledge-based* EP with Stream Reasoning capabilities, but also easier *communication* between event-driven applications and services, as well as, simpler *integration* of heterogeneous data streams and their *interpretation* on the Web.

Stream Reasoning in Knowledge-based EP. As mentioned above, current EP systems [1, 6, 8] do real-time pattern matching over unbound event streams. But they cannot combine data streams with *evolving knowledge*, and they cannot perform *reasoning* tasks over streaming data. Semantic technologies can enhance today's EP by providing and evaluating domain knowledge (e.g., in order to *enrich* recorded events, to detect more complex *situations* of interest, to propose certain intelligent *recommendations* on-the-fly etc.).

Consequently, the problem addressed in this paper combines tasks of Event Processing and Stream Reasoning. That is, we propose an approach to detect *complex events* (specified in an appropriate formal *language*) within a *stream of RDF triples* (atomic events). Detection of complex events may additionally require *reasoning* over background knowledge (expressed as an RDF graph or an RDFS ontology). We assume that the timeliness of this detection is crucial and algorithmically optimize our method towards a *continuous* evaluation of patterns and a fast response behavior.

4. EP-SPARQL

4.1 Syntax

In this section, we introduce the syntax of EP-SPARQL, our extension of the SPARQL querying language in order to enable stream-based querying that takes into account temporal situatedness of triple assertions. Thereby, we ensure syntactical and semantic downward-compatibility to plain SPARQL in the sense detailed below. We assume the reader to be familiar with RDF and SPARQL; For a thorough introduction to these formalisms see, e.g. [13].

Syntactically, we define EP-SPARQL to be SPARQL extended by the binary operators SEQ, EQUALS, OPTIONALSEQ, and EQUALSOPTIONAL used to combine graph patterns in the same way as UNION and OPTIONAL in pure SPARQL. Intuitively, all those operators act like a (left, right or full) join, but they do so in a selective way depending on how the constituents are temporally interrelated, as indicated by their naming: P_1 SEQ P_2 joins P_1 and P_2 only if P_2 occurs[2] strictly after P_1, whereas P_1 EQUALS P_2 performs the join if P_1 and P_2 are exactly simultaneous. OPTIONALSEQ and EQUALSOPTIONAL are temporal-sensitive variants of OPTIONAL.

Moreover, we add the function `getDURATION()` to be used inside filter expressions. This function yields a literal of type `xsd:duration` giving the length of the time interval associated to the graph pattern the FILTER condition is placed in. Likewise, we add functions `getSTARTTIME()` and `getENDTIME()` to retrieve the time stamps (of type `xsd:dateTime`) of the start and end of the currently described interval.

We provide a few examples to give some intuition on the newly introduced operators. The following EP-SPARQL query is supposed to search for companies whose stock price has decreased by over 30% and subsequently risen by more than 5% within a time frame of 30 days.

```
SELECT ?company WHERE
       { ?company hasStockPrice ?price1 }
   SEQ { ?company hasStockPrice ?price2 }
   SEQ { ?company hasStockPrice ?price3 }                    (1)
FILTER ( ?price2 < ?price1 * 0.7 && ?price3 > ?price1 * 1.05
         && getDURATION() < "P30D"^^xsd:duration)
```

The next EP-SPARQL query will identify companies with a more than 50% stock price drop and – in case some rating agency previously downrated this company, this rating agency will be indicated as well.

```
SELECT ?company ?ratingagency WHERE
  {      ?company downratedby ?ratingagency}
  OPTIONALSEQ                                               (2)
  {     { ?company hasStockPrice ?price1 }
    SEQ { ?company hasStockPrice ?price2 }}
FILTER (?price2 < ?price1 * 0.5)
```

It is worth mentioning that – just like for pure SPARQL – negation (i.e., requiring the *absence* of some triple pattern instead of its *presence*) is not an explicit part of the formalism, but can be expressed via OPTIONAL and FILTER. For instance, the following query asks for companies having a larger than 50% stock price increase in less than 15

[2]in a sense to be defined more precisely in the formal semantics

days without having acquired another company during that period.

```
SELECT ?company WHERE
       { ?company hasStockprice ?price1 }
   SEQ { { ?company hasAcquired ?othercompany }                (3)
         OPTIONALSEQ
         { ?company hasStockPrice ?price2 } }
FILTER ( ?price2 < ?price1 * 1.5 && !BOUND(?othercompany) &&
         getDURATION() < "P15D"^^xsd:duration )
```

Moreover, we allow for recursion by employing CONSTRUCT queries, conceiving them as a kind of production rule. Thereby, the result graph of such a query is assumed to be added to the RDF stream. For instance, the following statement gathers "temporally distributed" rating information to create a triple indicating an event of being downrated, which in turn can be used in other CONSTRUCT or SELECT queries.

```
CONSTRUCT ?company downratedby ?ratingagency
WHERE    { ?rating1 rater ?ratingagency ;
             rated ?company ; score ?score1 }                  (4)
     SEQ { ?rating2 rater ?ratingagency ;
             rated ?company ; score ?score2 }
FILTER ( ?score2 < ?score1 )
```

Finally, the forthcoming extended SPARQL standard[3] featuring *subqueries* and *expressions* allows for as complex mechanisms as aggregation over sliding windows. As an example we present a query monitoring the average stock price of a company ACME Inc. over the last 10 days. First, we use a construct rule that aggregates counts and sums of stock prices within the given time frame and feeds this information back into the stream. Thereby, the EQUALSOPTIONAL and filter part make sure that no price signal is left out.

```
CONSTRUCT  _:aaa   :hasCount ?count . _:aaa   :hasSum   ?sum .
{ SELECT  ?count AS ?prevcount + 1
          ?sum   AS ?prevsum + ?price
  WHERE   {{ ?point :hasCount ?prevcount .
             ?point :hasSum  ?prevsum    . }
           SEQ { :ACME  :hasStockPrice  ?price . }}
         EQUALSOPTIONAL                                        (5)
           {{ ?point :hasCount ?prevcount .
              ?point :hasSum  ?prevsum    . }
           SEQ { :ACME  :hasStockPrice  ?inbetween . }
           SEQ { :ACME  :hasStockPrice  ?price . }}
FILTER ( !BOUND(?inbetween) &&
         getDURATION() < "P10D"^^xsd:duration )}
```

Next, we calculate and output the average value as soon as the duration of our time window is exceeded.

```
SELECT ?sum / ?count AS ?average
WHERE {{  :ACME  :hasStockPrice  ?price . }
    SEQ { ?point :hasCount       ?prevcount .
          ?point  :hasSum  ?prevsum  . }}
    EQUALSOPTIONAL
      {{  :ACME  :hasStockPrice  ?price . }                    (6)
    SEQ { :ACME  :hasStockPrice  ?inbetween . }
    SEQ { ?point :hasCount       ?prevcount .
          ?point  :hasSum  ?prevsum  . }}
FILTER ( !BOUND(?inbetween) &&
         getDURATION() > "P10D"^^xsd:duration )
```

It may take some consideration and checking back with the formal semantics to verify that this realizes the intended behavior. In practice, additional constructs may be introduced as syntactic sugar to facilitate specification of patterns that are often used.

[3] `http://www.w3.org/TR/2009/WD-sparql-features-20090702/`

4.2 Formal Semantics

We define the formal semantics for EP-SPARQL along the same lines as it is done for pure SPARQL [18], that is, in a relational way. Thereby, the query is first translated into an algebraic expression. Recall that this conversion transforms simple graph patterns[4] (lists of triples) P into expressions of the form $BGP(P)$. Moreover, juxtapositions of graph triples are translated into the function *Join*, UNION into *Union*, OPTIONAL into *LeftJoin*. We reuse but extend this translation to map the new operators as follows: SEQ $\mapsto$ *SeqJoin*, EQUALS $\mapsto$ *EqJoin*, OPTIONALSEQ $\mapsto$ *SeqRightJoin*, and EQUALSOPTIONAL $\mapsto$ *EqLeftJoin*. Each of these functions is meant to return the result of the subquery it describes, which is a formal representation of the corresponding result table – as opposed to plain SPARQL, each row of these intermediary result tables is additionally associated with a time interval.

To make this more formal, note that we pose our query against an *RDF stream* which we define as a set S consisting of *triple occurrences* being pairs $\langle\langle s,p,o\rangle,t,t'\rangle$ where $\langle s,p,o\rangle$ is an RDF triple and t,t' are time stamps denoting the boundaries of the time interval of the occurrence. Now, we say that the tuple $\langle\mu,t_\alpha,t_\omega\rangle$ is a solution for an expression of the form "*BGP(* `list of triples` *)*" exactly if the following conditions are satisfied:

1. μ is a partial function the domain of which consists exactly of the variables that occur in the given list of triples.
2. for the triple set $\{\langle s_1,p_1,o_1\rangle,\ldots,\langle s_n,p_n,o_n\rangle\}$ obtained from substituting the variables in the triple list via μ, there exist time stamps $t_1,t_1',\ldots,t_n,t_n'$ such that
   - $\{\langle\langle s_1,p_1,o_1\rangle,t_1,t_1'\rangle,\ldots\langle\langle s_n,p_n,o_n\rangle,t_n,t_n'\rangle\}\subseteq S$,
   - $t_\alpha=\min(t_1,\ldots,t_n)$, and
   - $t_\omega=\max(t_1',\ldots,t_n')$.

Now define results for the other operators. A pair of solutions $\langle\mu,t_\alpha,t_\omega\rangle$ and $\langle\mu',t_\alpha',t_\omega'\rangle$ is said to be *compatible* if every variable that is mapped by both μ and μ' is also mapped to the same RDF term by both solutions. If this is the case, their combination $\langle\mu,t_\alpha,t_\omega\rangle\bowtie\langle\mu',t_\alpha',t_\omega'\rangle$ can be defined as the tuple $\langle\mu'',t_\alpha'',t_\omega''\rangle$ with $t_\alpha''=\min(t_\alpha,t_\alpha')$, $t_\omega''=\max(t_\omega,t_\omega')$, and

$$\mu''(x)=\begin{cases}\mu(x) & \text{if } x \text{ occurs in the domain of } \mu\\ \mu'(x) & \text{if } x \text{ occurs in the domain of } \mu'\\ \text{undefined} & \text{in all other cases}\end{cases}$$

Based on this, we define how to evaluate the introduced functions on sets Ψ,Ψ' of solutions. Thereby, we use $\sigma_{t_\alpha t_\omega}$ to denote the operator substituting `getDURATION()` by $t_\omega-t_\alpha$, `getSTARTTIME()` by t_α, and `getENDTIME()` by t_ω in filter expressions.

- *Filter*(F,Ψ) contains those $\langle\mu,t_\alpha,t_\omega\rangle\in\Psi$ for which the expression $\mu(\sigma_{t_\alpha t_\omega}(F))$ evaluates to `true`.
- *Join*(Ψ,Ψ') contains $\langle\mu,t_\alpha,t_\omega\rangle\bowtie\langle\mu',t_\alpha',t_\omega'\rangle$ for all compatible $\langle\mu,t_\alpha,t_\omega\rangle\in\Psi$ and $\langle\mu',t_\alpha',t_\omega'\rangle\in\Psi'$

[4] For the sake of brevity, we assume that the graph patterns do not contain blank nodes, as they can be replaced by (non-distinguished) variables without changing the semantics.

- $Union(\Psi, \Psi') = \Psi \cup \Psi'$
- $LeftJoin(\Psi, \Psi', F)$ contains
  - every $\langle \mu, t_\alpha, t_\omega \rangle \bowtie \langle \mu', t'_\alpha, t'_\omega \rangle$ for every compatible $\langle \mu, t_\alpha, t_\omega \rangle \in \Psi$ and $\langle \mu', t'_\alpha, t'_\omega \rangle \in \Psi'$ with $(\mu \bowtie \mu')(\sigma_{t_\alpha t_\omega}(F)) = \texttt{true}$ and $t'_\omega < t_\omega$.
  - every $\langle \mu, t_\alpha, t_\omega \rangle \in \Psi$ for which there is no compatible $\langle \mu', t'_\alpha, t'_\omega \rangle \in \Psi'$ with $(\mu \bowtie \mu')(\sigma_{t_\alpha t_\omega}(F)) = \texttt{true}$ and $t'_\omega < t_\omega$.
- $SeqJoin(\Psi, \Psi')$ contains $\langle \mu, t_\alpha, t_\omega \rangle \bowtie \langle \mu', t'_\alpha, t'_\omega \rangle$ for all compatible $\langle \mu, t_\alpha, t_\omega \rangle \in \Psi$ and $\langle \mu', t'_\alpha, t'_\omega \rangle \in \Psi'$ additionally satisfying $t_\omega < t'_\alpha$
- $SeqRightJoin(\Psi', \Psi, F)$ contains
  - all solutions from $Filter(F, SeqJoin(\Psi', \Psi))$ as well as
  - all $\langle \mu, t_\alpha, t_\omega \rangle \in \Psi$ for which there is no compatible $\langle \mu', t'_\alpha, t'_\omega \rangle \in \Psi'$ with both $(\mu \bowtie \mu')(\sigma_{t_\alpha t_\omega}(F)) = \texttt{true}$ and $t_\alpha > t'_\omega$.
- $EqJoin(\Psi, \Psi')$ contains $\langle \mu, t_\alpha, t_\omega \rangle \bowtie \langle \mu', t'_\alpha, t'_\omega \rangle$ for all compatible $\langle \mu, t_\alpha, t_\omega \rangle \in \Psi$ and $\langle \mu', t'_\alpha, t'_\omega \rangle \in \Psi'$ additionally satisfying $t_\alpha = t'_\alpha$ and $t_\omega = t'_\omega$
- $EqLeftJoin(\Psi, \Psi', F)$ contains
  - all solutions from $Filter(F, EqJoin(\Psi, \Psi'))$ as well as
  - all $\langle \mu, t_\alpha, t_\omega \rangle \in \Psi$ for which there is no compatible $\langle \mu', t'_\alpha, t'_\omega \rangle \in \Psi'$ with all $(\mu \bowtie \mu')(\sigma_{t_\alpha t_\omega}(F)) = \texttt{true}$ and $t_\alpha = t'_\alpha$ and $t_\omega = t'_\omega$.

We would like to add the following remarks to justify some aspects of our definition of the EP-SPARQL semantics. First, we endorse the principle of *timewise monotonicity*: the querying formalism is intended to work on triple streams (i.e., triples continuously enter the system in the order of their associated time stamps) and query results are supposed to be output as soon as they are detected. This leads to the straightforward requirement that it should not be possible that query results once obtained get invalidated by later triple inputs. More formally, we have to guarantee that for any EP-SPARQL query and any RDF stream S all solutions for the stream $\{\langle \mu, t_\alpha, t_\omega \rangle \mid t_\omega < t_1\}$ are also solutions for the stream $\{\langle \mu, t_\alpha, t_\omega \rangle \mid t_\omega < t_2\}$ given that $t_1 \leq t_2$. Note that a hypothetical constructor SEQOPTIONAL (specifying a mandatory pattern followed by an optional one) defined as the inverse version of OPTIONALSEQ would violate this principle since the solution $\langle \{x \mapsto a\}, 0, 0 \rangle$ would be a solution to the query

```
SELECT ?x ?y WHERE ?x ?x ?x . SEQOPTIONAL ?y ?y ?y .        (7)
```

when posed against the stream $\{\langle \langle a, a, a \rangle, 0, 0 \rangle\}$ but not when posed against the augmented stream $\{\langle \langle a, a, a \rangle, 0, 0 \rangle, \langle \langle b, b, b \rangle, 1, 1 \rangle\}$. As second principle, we obtain *downward compatibility* in the following sense: as a consequence of the syntax definition, every (pure) SPARQL query q is also an EP-SPARQL query. Now, given an RDF graph $\{\langle s_1, p_1, o_1 \rangle, \ldots, \langle s_n, p_n, o_n \rangle\}$, we obtain μ as a result of the SPARQL query q if and only if we obtain $\langle \mu, t, t \rangle$ as a solution to the EP-SPAQL query against the RDF stream $\{\langle \langle s_1, p_1, o_1 \rangle, t, t \rangle, \ldots \langle \langle s_n, p_n, o_n \rangle, t, t \rangle\}$ for any fixed time stamp t.

To finish the semantics definition, we have to consider the CONSTRUCT rules. Given such a statement q with the graph pattern P_q following the CONSTRUCT command and the set Ψ_q^S of solutions to the WHERE part w.r.t. some given stream S, let $\Psi_q^S(P_q)$ denote the set of tuples $\langle \langle s, p, o \rangle, t, t' \rangle$ for which there is a solution $\langle \mu, t, t' \rangle \in \Psi_q^S$ such that (1) μ has as domain at least all variables occurring in P_q, and (2) $\langle s, p, o \rangle \in \mu(P_q)$. Now, given an RDF stream S and a set Q of CONSTRUCT statements, we define the *Q-closure* of S ($clos_Q(S)$) as the smallest set for which both $S \subseteq clos_Q(S)$ as well as $\Psi_q^{clos_Q(S)}(P_q) \subseteq clos_Q(S)$ for every $q \in Q$. We can see the *Q-closure* of S as the stream S enriched by the triple occurrences following from (possibly iterated) application of the CONSTRUCT rules. Consequently, in the presence of such rules, SELECT-queries get evaluated not against the pure input stream but against its Q-closure. Moreover, in the case of SELECT queries, after calculating the solution set, it is further processed (with respect to variable projection and output formatting) like for normal SPARQL.

5. EXECUTION MODEL

This section describes how complex events, as defined in Section 4, are computed at run-time. We describe an execution mechanism that is based on *event-driven backward chaining* (EDBC) rules, introduced in [2]. EP-SPARQL queries are compiled into EDBC rules, which enable timely, event-driven, and incremental detection of complex events (i.e., answers to EP-SPARQL queries). EDBC rules are logic rules,[5] and hence can be mixed with other background knowledge (domain knowledge that is used for Stream Reasoning). Therefore, we provide a unified execution mechanism for Event Processing and Stream Reasoning which is grounded in Logic Programming (LP).

For our encoding, we use a simple correspondence between RDF triples of the form $\langle s, p, o \rangle$ and Prolog predicates of the form $triple(s', p', o')$ so that s', p', and o' correspond to the RDF symbols s, p, and o, respectively. This means that whenever a triple $\langle s, p, o \rangle$ is satisfied, the corresponding predicate $triple(s', p', o')$ is satisfied too, and vice versa. Consequently, a time-stamped RDF triple $\langle \langle s, p, o \rangle, t_\alpha, t_\omega \rangle$ corresponds to a predicate $triple(s', p', o', T'_\alpha, T'_\omega)$ where T'_α and T'_ω denote time stamps. Time stamps are assigned to triples either by a triple source (e.g., a sensor or an application that generates triple updates) or by an EP-SPARQL engine (e.g., our prototype implementation). They facilitate time-related processing, and do not necessarily need to be kept once the stream has been processed (e.g., the pure RDF part could be persisted in a RDF triple store without time stamps).

Sequence operator (*SeqJoin*). Let us consider a sequence of three triples (events), represented by Example (1) (in Section 4.1) when the FILTER expression is omitted. This EP-SPARQL query can be represented by rule (8), where the SEQ operator has the identical meaning, i.e., $triple(\tau, T_1, T_6)$ is detected[6] when $triple(\tau_1, T_1, T_2)$ occurs in a data stream, followed by $triple(\tau_2, T_3, T_4)$, and $triple(\tau_3, T_5, T_6)$. We can always represent this pattern with rule (9).

[5] which we represent in a Prolog-like syntax

[6] For brevity, we use τ with possible super- and subscripts to denote triplets s, p, o.

$$triple(\tau, T_1, T_6) \leftarrow triple(\tau_1, T_1, T_2) \text{ SEQ } triple(\tau_2, T_3, T_4) \text{ SEQ } triple(\tau_3, T_5, T_6). \quad (8)$$

$$triple(\tau, T_1, T_6) \leftarrow (triple(\tau_1, T_1, T_2) \text{ SEQ } triple(\tau_2, T_3, T_4)) \text{ SEQ } triple(\tau_3, T_5, T_6). \quad (9)$$

We refer to this rewriting as *binarization* of patterns. Effectively, in binarization we introduce triples that are *binary intermediate goals*. For example, now we can rewrite rule (8) as an intermediate $triple(\tau', T_1, T_4) \leftarrow triple(\tau_1, T_1, T_2)$ SEQ $triple(\tau_2, T_3, T_4)$, and $triple(\tau, T_1, T_6) \leftarrow triple(\tau', T_1, T_4)$ SEQ $triple(\tau_3, T_5, T_6)$. Every monitored pattern (capturing either a triple or a derived triple) will be associated with one or more *logic rules*. Rules are fired as soon as certain triples occur, hence they are driven by streaming triples (events). Use of binarization eases construction of these (event-driven) rules for three reasons: First, it is easier to implement an event operator when events are considered on a "two by two" basis (and to create an *automated* procedure that compiles EP-SPARQL expressions into executable rules). Second, the binarization increases the possibility for *computation sharing* of common subexpressions among complex patterns (when the granularity of intermediate goals is reduced). Third, the binarization eases the *management* of rules. Each new triple (being monitored in a pattern) amounts to appending one or more rules to the existing rule set. However, what is important for the management of rules, we do not need to modify existing rules when adding new ones.

Here we presented a left-associative binarization (events and goals are coupled from left to right). The left-associative binarization is a good choice when the rightmost event(s) in a pattern rule have a higher occurrence rate than the others (e.g., event $triple(\tau_3, T_5, T_6)$ occurs more frequently than event $triple(\tau_1, T_1, T_2)$), since in that situation event $triple(\tau_3, T_5, T_6)$ is joined later. It is also possible to do such a coupling from right to left. The right-associative coupling is beneficial when the leftmost event(s) have a higher rate of occurrence(s). Other combinations are possible, too. See for example bushy plan and inner plan in [16]. These, and similar plans and cost optimizations as proposed in [16] are applicable in our framework. They are, however, out of scope of this paper, and will be addressed in our future work.

In the following, we give more details about rule assignment for each monitored triple, and sketch the execution model for a sequence of triples.

$$triple(\tau, T_1, T_4) \leftarrow triple(\tau_1, T_1, T_2) \text{ SEQ } triple(\tau_2, T_3, T_4). \quad (10)$$

Rule 10 represents a binary sequence goal, and rules (11) and (12) represent the pair of *event-driven backward chaining rules* (EDBC) into which (10) is translated.

$$\begin{array}{l} triple(\tau_1, T_1, T_2) \leftarrow \\ \quad assert(\\ \quad\quad goal(triple(\tau_2, \_, \_), triple(\tau_1, T_1, T_2), triple(\tau, \_, \_)) \\ \quad). \end{array} \quad (11)$$

$$\begin{array}{l} triple(\tau_2, T_3, T_4) \leftarrow \\ \quad goal(triple(\tau_2, \_, \_), triple(\tau_1, T_1, T_2), triple(\tau, \_, \_)), \\ \quad T_2 < T_3, \\ \quad retract(\\ \quad\quad goal(triple(\tau_2, \_, \_), triple(\tau_1, T_1, T_2), triple(\tau, \_, \_)) \\ \quad), \\ \quad triple(\tau, T_1, T_4). \end{array} \quad (12)$$

In general, the EDBC rules created by our translation can be grouped in two different classes of rules. We refer to the first class as *goal-insertion rules*. The second class corresponds to *checking rules*. For example, rule (11) belongs to the first class as it asserts a goal. This rule will fire when $triple(\tau_1, T_1, T_2)$ occurs, and the meaning of the goal it inserts is as follows: "an event $triple(\tau_1, T_1, T_2)$ has occurred at $[T_1, T_2]$, and we are waiting for $triple(\tau_2, \_, \_)$ to happen in order to detect $triple(\tau, \_, \_)$". Obviously, the goal does not carry information about times for $triple(\tau_2, \_, \_)$ and $triple(\tau, \_, \_)$, as we do not know when they will occur. In general, the *second* event in a goal always denotes the event (triple) that has just occurred. The role of the *first* event is to specify what we are waiting for to detect an event that is on the *third* position (i.e., a derived triple).

Rule (12) belongs to the second class, being a *checking rule*. It checks whether certain prerequisite goals already exist in the knowledge base, in which case it triggers a more complex event. For example, that rule will fire whenever $triple(\tau_2, T_3, T_4)$ (the triple from the rule head) occurs. It checks whether $goal(triple(\tau_2, \_, \_), triple(\tau_1, T_1, T_2), triple(\tau, \_, \_))$ already exists (meaning that $triple(\tau_1, T_1, T_2)$ has previously happened), in which case it triggers $triple(\tau, T_1, T_4)$ by calling that triple. The triple occurrence time span (i.e. $[T_1, T_4]$) is defined based on the occurrence of constituting events (i.e. $triple(\tau_1, T_1, T_2)$, and $triple(\tau_2, T_3, T_4)$, see Section 4.2). Calling $triple(\tau, T_1, T_4)$, this event is effectively either propagated further (if it is an intermediate event) or triggered as a finished complex event.

We see that our *backward* chaining rules compute goals in a *forward* chaining manner. The goals are crucial for computing complex events *incrementally*. Goals can persist over a period of time. It is worth noting that *checking rules* can also delete goals (see *retract* predicate in rule 12). Once a goal is "consumed", it is removed from the knowledge base.[7] In this respect, goals are kept persistent as long as (but not longer) they are needed.

So far, we have explained how the SEQ operator is implemented with EDBC rules. OPTIONALSEQ is implemented similarly. The operator allows information to be added to the answer where certain triples are available, but do not reject the answer when some part of the query pattern does not match. Hence the functionality of OPTIONALSEQ operator is the same as for SEQ operator, and OPTIONALSEQ subpatterns are translated into event-driven rules and computed in the same way as the mandatory part. However, at the end, the pattern is detected when all mandatory conditions are satisfied (regardless whether an optional sub-pattern has been satisfied by that moment or not). The same applies for the EQUALS and EQUALSOPTIONAL operators.

Filters. A FILTER expression in an EP-SPARQL query can be represented as a rule, too.[8] The head of that rule may be part of a goal. When the goal gets evaluated, the corresponding rule will be evaluated, too. For example, let us consider again rule (1) from Section 4.1 and its FILTER expression. We said that $triple(\tau, T_1, T_6)$ can be represented as $triple(\tau, T_1, T_6) \leftarrow triple(\tau', T_1, T_4)$ SEQ $triple(\tau_3, T_5, T_6)$

[7] Removing "consumed" goals is often needed for space reasons but might be omitted if events are required in a log for further processing or analyzing.

[8] Here we focus on filters without time constraints. Time constrained filters will be explained later in this section.

where $triple(\tau', T_1, T_4)$ is an intermediate triple, specified as $triple(\tau', T_1, T_4) \leftarrow triple(\tau_1, T_1, T_2)$ SEQ $triple(\tau_2, T_3, T_4)$. When the FILTER expression is considered, throughout the binarization process $triple(\tau, T_1, T_6)$ becomes $triple(\tau, T_1, T_6) \leftarrow triple(\tau'', T_1, T_6)$ SEQ $condition(Price1, Price2, Price3)$, and $triple(\tau'', T_1, T_6) \leftarrow triple(\tau', T_1, T_4)$ SEQ $triple(\tau_3, T_5, T_6)$ where *condition* is defined as the following rule.[9]

$$\begin{aligned} &condition(Price1, Price2, Price3) \\ &\quad \leftarrow P_1 \ is \ (Price1 * 0.7), \ P_1 > Price2, \\ &\quad\quad P_2 \ is \ (Price1 * 0.5), \ Price3 > P_2. \end{aligned} \tag{13}$$

Background knowledge. To enable detection of more complex events, our approach combines streams with background knowledge. This knowledge describes the context (domain) in which complex events are detected. As such, it enables detection of real-time situations that are identified based on explicit data (e.g., events) as well as on implicit knowledge (derived from the background knowledge).

The background knowledge may be specified as a Prolog knowledge base or as an RDFS ontology.[10] This enables our execution model to have all relevant parts expressible in a unified (logic rule) formalism, and ultimately to reason over a unified space. For example, while detecting a sequence of two events, we may check whether their joined attribute is an instance of a certain class (or any of its subclasses) defined in an ontology (e.g., see Test 2 in Section 6). To prove this on-the-fly, an inference procedure needs to be executed. In this respect, our execution model detects time relations among streaming triples (events), and performs reasoning tasks when necessary. Since all components of an EP-SPARQL query (including background knowledge) are represented as (Prolog) rules, we can use a Prolog inference engine to serve as an EP-SPARQL engine.

Equals operation (*EqJoin*). Two events are equal if they happen right at the same time. Hence, in order to implement the EQUALS operator we again use the rules of the type (11)-(12). Additionally, we use the rule (14) to check whether the occurrence intervals of two events are equal, i.e., the rule compares whether the *start* of the first interval (TI_1_S) is equal to the *start* of the second interval (TI_2_S). The same check is done for the *end* of the two intervals.

$$\begin{aligned} &equals(TI_1, TI_2) \leftarrow \\ &\quad TI_1 = [TI_1\_S, TI_1\_E], validTimeInterval(TI_1), \\ &\quad TI_2 = [TI_2\_S, TI_2\_E], validTimeInterval(TI_2), \\ &\quad TI_1\_S = TI_2\_S, TI_1\_E = TI_2\_E. \end{aligned} \tag{14}$$

$$\begin{aligned} &validTimeInterval(TI) \leftarrow \\ &\quad TI = [TI\_S, TI\_E], TI\_S@ < TI\_E. \end{aligned} \tag{15}$$

Rule (15) is an auxiliary rule which makes sure that parameters of rule (14) are valid time intervals.

Other operators, such as juxtapositions of graph triples and UNION, are similarly translated into EDBC rules. Due to space limitations, we cannot provide details here. Instead, the interested reader is referred to the conjunction (AND) and disjunction (OR) operations, described in [3].

Memory management and time windows. We have implemented two memory management techniques to *prune* outdated events.

The first technique modifies the binarization step by pushing time window constraints (set by FILTER expressions with time constraints,[11] e.g., 30 days in EP-SPARQL query (1) from Section 4.1). The technique ensures that time window constraints are checked during the incremental event detection. Therefore, unnecessary intermediary sub-complex events will not be generated if time constraints are violated (i.e., time expired).

The second solution prunes expired events (goals) by using system generated events (SGE). Similar to the first technique, rules are defined with time window constraints, and the binarization pushes the constraints to sub-components. This technique, however, does not check its constraints at each step in the event detection incrementally. Instead, events (goals) are pruned periodically as SGE are triggered.

For *time sliding windows*, we also need to prune expired events. This has been realized by using one of the two mentioned memory management techniques. The system generates frequent SGE, which prune outdated events so that an aggregation function can be recomputed on the set of valid events. An output-aggregation event is triggered whenever a new valid event occurs. Frequency of SGEs can be specified for every EP-SPARQL pattern, individually.

6. EXPERIMENTAL RESULTS

To evaluate our approach, we have implemented the system called ETALIS.[12] ETALIS is implemented in Prolog and runs on many Prolog systems, including YAP, SWI, SICStus, and XSB Prolog. The test cases presented here were carried out on a workstation with Intel Core Quad CPU Q9400 2,66GHz, 8GB of RAM, running Ubuntu 9.10. Our engine automatically loads the RDF knowledge base into Prolog and compiles RDF stream triples into Prolog. All tests were ran on a single dedicated CPU core (no multicore platform is currently deployed).

Performance tests include: (1) event processing evaluation; (2) stream reasoning capabilities; (3) implementation of two example applications that use both (1) and (2).

Test 1: Event Processing. We compared ETALIS with Esper 3.3.0.[13] Esper is a state-of-the-art data stream engine (also used for commercial purposes). We chose Esper as it is a freely available open source engine.

In Figure 1(a), we measured the throughput for the EP-SPARQL queries (5) and (6) from Section 4.1. The query is run over an RDF stream in ETALIS (with YAP Prolog) and also with Esper. The stream was generated so that every streaming triple contributes to the derivation of complex events. The RDF stream in the XML format for Esper was pre-parsed before the test as we wanted to exclude the pre-processing time. Figure 1(b) shows the evaluation of a similar EP-SPARQL query, with the difference that the window size is determined by a specified number of events. Both queries are typical EP patterns, and both show dominance of ETALIS over Esper. For tests that involve the stream reasoning (test (2) and test (3)) Esper cannot be used, hence we show the experimental results for ETALIS only.

[9] Note that $Price1, Price2, Price3$ are contained in τ_1, τ_2, τ_3.

[10] In the latter case, we utilize an existing library `www.swi-prolog.org/pldoc/package/semweb.html` to transform an RDFS ontology into Prolog rules and facts.

[11] Users are encouraged to specify time constraints in queries, as it enables the system to regularly free up its memory.

[12] ETALIS: `http://code.google.com/p/etalis/`

[13] Esper: `http://esper.codehaus.org`

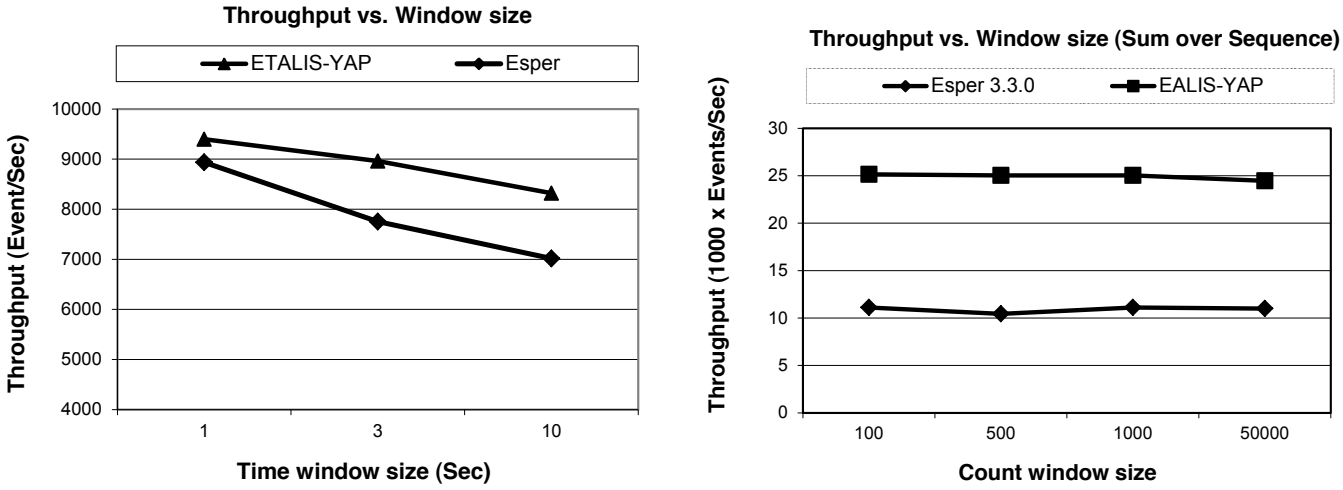

Figure 1: Event Processing: (a) Time sliding window (b) Aggregation over count sliding window

Test 2: Stream Reasoning. To provide a performance evaluation for the stream reasoning functionality, we have reconstructed an experiment from [21]. The goal of the test is to listen to streaming triples, and to infer whether the subject of a triple is an instance of the class of concern (or any of its subclasses). The class of concern is `http://spire.umbc.edu/ontologies/EthanPlants.owl#Tracheobionta`, that has 40,080 subclasses with a maximum class-hierarchy depth of eight. As in [21], we measured delay caused by the automated reasoning process needed to determine whether an entry in a streaming triple is an instance of the class of concern. The work in [21] provides three implementations: the first based on Jena,[14] the second based on pre-computed inference results stored in a hash table, and the third based on a streaming database engine, TelegraphCQ [8] (none of which is available for download and testing). According to [21], the fastest implementation is the third one (which also pre-computes all inferences and stores them in a PostgreSQL database). Figure 2 shows results of the same test with ETALIS. Our system is more than 20 times faster. On one hand, we did the test on a faster machine. On the other hand, ETALIS was doing stream reasoning on-the-fly (with no persisted inferences), and still performed significantly faster.

In the future, we also plan to provide persistence of inferences (as in [21]) in order to speed up query processing. FILTER sub-patterns which demand access and reasoning over *static* knowledge (ontologies) can be pre-computed. These results can then be reused every time a query needs to be executed. This approach may be beneficial when large ontologies are used, and events are streamed with a high frequency (e.g., hundred thousands events per second).

Test 3: Example applications. We developed an application using both static RDF knowledge bases and RDF event streams. The application implements a Goods Delivery System in the city of Milan. The system comprises of a set of delivery agents that need to deliver the manufactured products to the consumers. Each of them has a list of locations that it needs to deliver goods to. While an agent is visiting a particular location, the system "knows" her next location and "*listens*" to traffic-update events on that route(s). If the agent requests the next route at the moment when the route is currently inaccessible, the system will find another route (calculating a transitive closure on-the-fly over the background ontology). We use a Milan ontology[15] to explore routes in Milan. The application has been implemented on top of EP-SPARQL and ETALIS. Due to space limitations we cannot show patterns from the whole application here. Instead, we show in Figure 3 results obtained for 1 and 10 delivery agents when visiting 20 locations (the time spent at a location is irrelevant for the test, and hence it is ignored). We simulated situations where more than 50% of the connections between the visiting locations were inaccessible, so that the system needed to recalculate the transitive closure frequently (as response to traffic-update events).

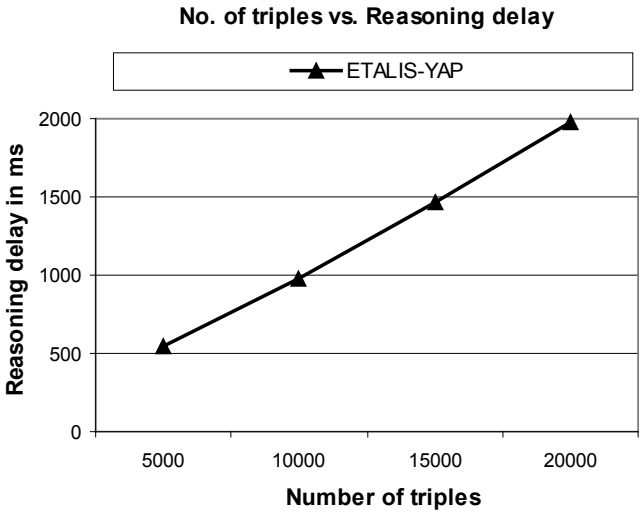

Figure 2: Delay caused by stream reasoning

[14]Jena: `http://jena.sourceforge.net/`

[15]Milan ontology was generously provided by AMAT Milano and CEFRIEL team: `http://www.larkc.eu/resources/published-data-sources/`

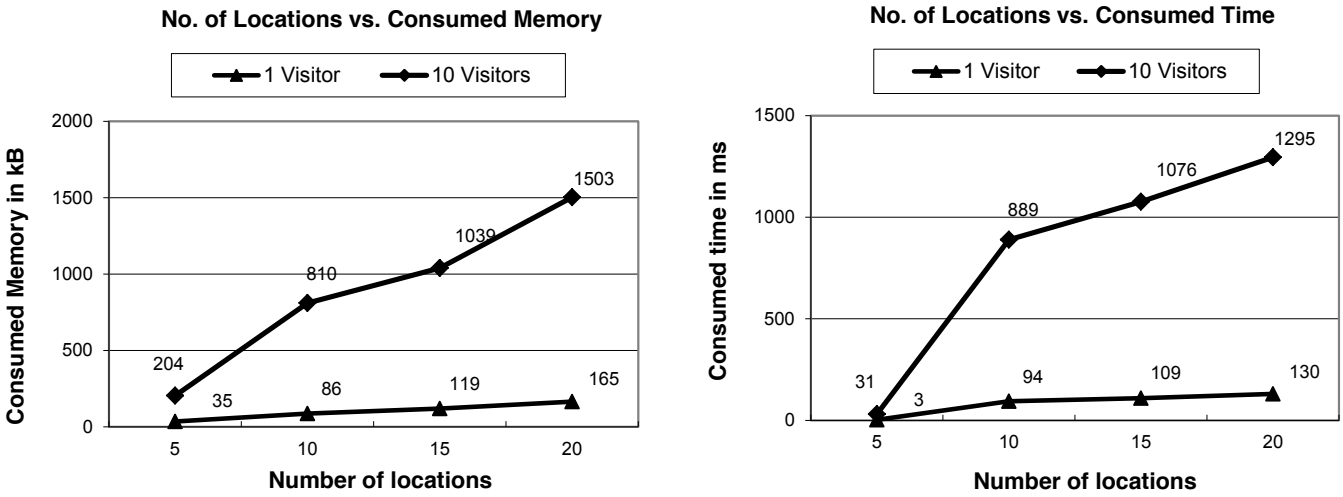

Figure 3: Milan Sightseeing: (a) Delay caused by processing (b) Memory consumption

The goal of the test was to show the usefulness of our formalism in a real-use scenario, as well as to show that the application scales well with the increase of number of agents (throughput for one agent is about 10 times higher than the throughput for 10 agents, indicating a *linear* relationship in the investigated range, see Figure 3 (a)). Similarly, Figure 3 (b) shows the memory consumption for the same test (likewise indicating a linear space dependency w.r.t. the number of agents).

For the second application, we have developed a real-time service for detection of tsunamis. A tsunamis gauge is designed to detect and report tsunamis based on buoy sensor data. Data is provided by the National Data Buoy Center (NDBC).[16] We have implemented a tsunami detection algorithm[17] which works by predicting the amplitudes of the pressure fluctuations within the tsunami frequency band and then testing these amplitudes against a threshold value. The prediction is calculated by the following formula:

$$H_p(t') = \sum_{i=0}^{3} w(i) H^*(t - i \cdot \Delta t)$$

where $w(i)$ are coefficients that come from Newton's formula for forward extrapolation. The NDBC uses the following values for these coefficients:

$$w(0) = 1.16818457031250$$
$$w(1) = -0.28197558593750$$
$$w(2) = 0.14689746093750$$
$$w(3) = -0.03310644531250$$

Buoy sensor data is updated every 15 seconds, providing the sea level pressure, air temperature, wind speed, wave hight etc. The asterisk H^* denotes average pressure. Four values are continuously produced over a 3 hour sliding window ($i = 0, .., 3$ where a new value is output every hour, i.e., $\Delta t = 1$ hour), and t' is the prediction time which is set to 5,25 minutes. A tsunami is detected if the difference between the observed pressure (current sensor value) and the prediction H_p exceeds a threshold (30mm for the North Pacific as prescribed by the NDBC). The difference magnitude was continuously calculated over historic NDBC data from May 2005 until September 2010. In this period, 44310 sensor readings were reported to ETALIS. The system detected pressure differences higher than 30mm only 3 times (all of them during 3 hours, on 23.03.2010). Results are shown as a histogram in Figure 4. The according sensor station[18] is located in the Bering Sea, close to Alaska ($55°0'40"N 171°58'50"W$).

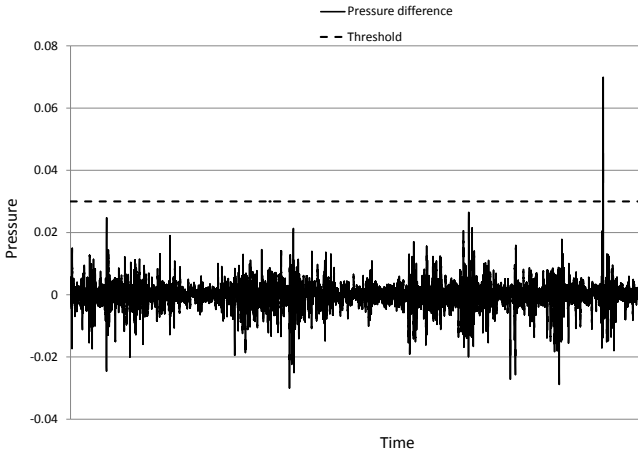

Figure 4: Tsunami detection histogram

Further on, we have utilized GeoNames as a worldwide geographical knowledge base. If a sensor detects a tsunami, GeoNames can provide all geographical places within a certain radius from the sensor location. These places can then automatically be warned of a detected tsunami. We have set up an on-line demo for this application[19] to continuously monitor live data provided by NDBC and detect tsunami warnings in real-time.

[16] NDBC : http://www.ndbc.noaa.gov/
[17] http://www.ndbc.noaa.gov/dart/algorithm.shtml
[18] http://www.ndbc.noaa.gov/station_page.php?station=46073
[19] http://etalis.fzi.de

7. CONCLUSIONS AND FUTURE WORK

Addressing dynamics and notification on the Web has recently become an important area of research. Real-time data is generated by multiple social networks, sensor networks, various on-line services etc. The challenge is to get advantage of real-time data, and recognise important situations of interest in a timely fashion.

We proposed a new language, EP-SPARQL, for Event Processing and Stream Reasoning. EP-SPARQL specifies complex events by temporally situating real-time streaming data, and uses background ontologies to enable stream reasoning. We defined the language with a clear syntax and a formal semantics. Further, our contribution includes an execution model which efficiently derives complex RDF events in (near) real-time. We also provided an implementation of our approach and conducted several types of tests.

We foresee several extensions to this work. Promising candidates for further investigation are support for stream reasoning over more expressive formalisms (e.g., OWL and its different profiles). Also adaptive optimizations as well as distributed stream reasoning are affirmative areas of future work.

8. ACKNOWLEDGMENTS

This work was partially supported by the European Commission funded project PLAY (FP7-20495) and by the ExpresST project funded by the German Research Foundation (DFG). We thank Ahmed Khalil Hafsi and Jia Ding for their help in implementation and testing of ETALIS.

9. REFERENCES

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proceedings of the 28th ACM SIGMOD Conference*, pages 147–160, 2008.

[2] D. Anicic, P. Fodor, S. Rudolph, R. Stühmer, N. Stojanovic, and R. Studer. A rule-based language for complex event processing and reasoning. In *Proceedings of the 4th International Conference on Web Reasoning and Rule Systems (RR 2010)*, pages 42–57, 2010.

[3] D. Anicic, P. Fodor, S. Rudolph, R. Stühmer, N. Stojanovic, and R. Studer. Etalis: Rule-based reasoning in event processing. In S. Helmer, A. Poulovassilis, and F. Xhafa, editors, *Reasoning in Event-based Distributed Systems, Studies in Computational Intelligence series*. LNCS, Springer Verlag, 2011.

[4] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. Incremental reasoning on streams and rich background knowledge. In *Proceedings of the 7th Extended Semantic Web Conference (ESWC'10)*, pages 1–15, 2010.

[5] D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus. An execution environment for C-SPARQL queries. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT'10)*, pages 441–452, 2010.

[6] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR'07)*, pages 363–374, 2007.

[7] A. Bolles, M. Grawunder, and J. Jacobi. Streaming SPARQL - extending SPARQL to process data streams. In *Proceedings of the 5th European Semantic Web Conference (ESWC'08)*, pages 448–462, 2008.

[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR'03)*, 2003.

[9] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR'03)*, 2003.

[10] E. Della Valle, S. Ceri, F. van Harmelen, and D. Fensel. It's a streaming world! Reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24(6):83–89, 2009.

[11] F. Grandi. T-SPARQL: a TSQL2-like temporal query language for RDF. In *International Workshop on on Querying Graph Structured Data*, pages 21–30, 2010.

[12] C. Gutierrez, C. A. Hurtado, and A. A. Vaisman. Introducing time into rdf. *The IEEE Transactions on Knowledge and Data Engineering*, 19(2):207–218, 2007.

[13] P. Hitzler, M. Krötzsch, and S. Rudolph. *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, August 2009.

[14] M. Koubarakis and K. Kyzirakos. Modeling and querying metadata in the semantic sensor web: The model stRDF and the query language stSPARQL. In *Proceedings of the 7th Extended Semantic Web Conference (ESWC'10)*, pages 425–439, 2010.

[15] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, Reading, MA, USA, 2002.

[16] Y. Mei and S. Madden. Zstream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of the 29th ACM SIGMOD Conference*, pages 193–206, 2009.

[17] M. Perry, A. P. Sheth, and P. Jain. SPARQLST: Extending SPARQL to support spatiotemporal queries. In *Technical Report. KNOESIS-TR-2009-01*, 2008.

[18] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. In *http://www.w3.org/TR/rdf-sparql-query/*, 2008.

[19] A. Rodriguez, R. E. McGrath, and J. Myers. Semantic management of streaming data. In *Workshop on Semantic Sensor Nets at ISWC '09*, 2009.

[20] J. Tappolet and A. Bernstein. Applied temporal RDF: Efficient temporal querying of RDF data with SPARQL. In *Proceedings of the 6th European Semantic Web Conference (ESWC'09)*, pages 308–322, 2009.

[21] O. Walavalkar, A. Joshi, T. Finin, and Y. Yesha. Streaming knowledge bases. In *International Workshop on Scalable Semantic Web Knowledge Base Systems*, 2008.