Rinne, Mikko; Nuutila, Esko

Constructing Event Processing Systems of Layered and Heterogeneous Events with SPARQL

Published in:
Ontologies, Databases and Applications of Semantics (ODBASE), Amantea, Italy, October 27-31, 2014

DOI:
10.1007/978-3-662-45563-0_42

Published: 01/01/2014

Document Version
Peer reviewed version

Please cite the original version:
Rinne, M., & Nuutila, E. (2014). Constructing Event Processing Systems of Layered and Heterogeneous Events with SPARQL. In R. Meersman, H. Panetto, T. Dillon, M. Missikoff, L. Liu, O. Pastor, A. Cuzzocrea, & T. Sellis (Eds.), *Ontologies, Databases and Applications of Semantics (ODBASE), Amantea, Italy, October 27-31, 2014* (pp. 682-699). Springer. https://doi.org/10.1007/978-3-662-45563-0_42

Constructing Event Processing Systems of Layered and Heterogeneous Events with SPARQL

Mikko Rinne and Esko Nuutila

Department of Computer Science and Engineering,
Aalto University, School of Science, Finland
firstname.lastname@aalto.fi

Abstract. SPARQL was originally developed as a derivative of SQL to process queries over finite-length datasets encoded as RDF graphs. Processing of infinite data streams with SPARQL has been approached by using pre-processors dividing streams into finite-length windows based on either time or the number of incoming triples. Recent extensions to SPARQL can support interconnections of queries, enabling event processing applications to be constructed out of multiple incrementally processed collaborating SPARQL update rules. With more elaborate networks of queries it is possible to perform event processing on heterogeneous event formats without strict restrictions on the number of triples per event. Heterogeneous event support combined with the capability to synthesize new events enables the creation of layered event processing systems. In this paper we review the different types of complex event processing building blocks presented in literature and show their translations to SPARQL update rules through examples, supporting a modular and layered approach. The interconnected examples demonstrate the creation of an elaborate network of SPARQL update rules for solving event processing tasks.

Keywords: Complex event processing, SPARQL, heterogeneous events, stream processing

1 Introduction

Complex event processing, as pioneered by David Luckham [15], takes a layered approach to creating event processing systems. Using the core definitions from the *Event Processing Glossary* [16], *an event* is broadly defined as "anything that happens, or is contemplated as happening" and *a complex event* is "an event that summarizes, represents, or denotes a set of other events". Consequently the yardstick for complex event processing becomes the use of a layered model of event abstraction rather than the complexity of the underlying problem.

Event objects, representations of events for computer processing, may carry a variable number of parameters to describe an event with the accuracy required for a particular event processing application or set of applications [19]. In large

R. Meersman et al. (Eds.): OTM 2014, LNCS 8841, pp. 682–699, 2014.

heterogeneous systems such as smart cities built from hardware and software components provided by multiple suppliers and operated by multiple independent private, corporate and public actors there can be a lot of variation between the data supplied by different sensors, requiring a flexible representation of event data. The need for flexibility is further backed up by concepts like *a composite event* [16], which is created by combining a set of simple or complex events, always encapsulating the component events from which the composite event is derived.

Semantic web standards RDF[1] and SPARQL[2] offer a good set of tools to cope with distributed heterogeneous environments. RDF is built with an open-world assumption and together with OWL offers tools to manage disjoint vocabularies. RDF graphs, with help from property paths and blank nodes, provide flexible means to encode event objects using variable numbers of elements and layers. TriG[3] adds further structure by supporting the communication of RDF datasets in Turtle format. SPARQL, with support for federated queries, offers a natural way to enrich events with static background data available in the web e.g. as linked open data. Finally, the reasoning capabilities of different entailment regimes can be used to provide further semantic understanding of event data.

In [1] it is shown that an engine based on the Rete algorithm can incrementally process SPARQL queries offering competitive performance. The creation of an event processing application using multiple interconnected SPARQL queries has been described and tested in [18], while use-cases in the domain of semantic sensor systems have been further elaborated in [20]. In [8] a set of *event processing agents* to be used as building blocks in constructing event processing networks is proposed. The contribution of the present document is to demonstrate how the types of building blocks listed in [8] can be encoded in SPARQL to create an event processing network capable of handling heterogeneous events using standard unextended semantic web technologies.

The background of stream processing with SPARQL is explained in Section 2. Structured representations of heterogeneous events using RDF are reviewed in Section 3. SPARQL representations for different types of event processing agents are shown in Section 4. Conclusions and future areas of study are outlined in Section 5.

2 Stream Processing in SPARQL

The SPARQL query language, developed by W3C as the semantic web counterpart to SQL from the relational database world, was first released as a recommendation in January 2008. SPARQL is built to process queries over finite, possibly distributed datasets encoded in RDF. Event streams, such as the flows of measurements from sensors, are intrinsically infinite. Queries calculating aggregate values need to be computed over a finite partition of an event stream

[1] http://www.w3.org/RDF/
[2] http://www.w3.org/TR/sparql11-query/
[3] http://www.w3.org/TR/trig/

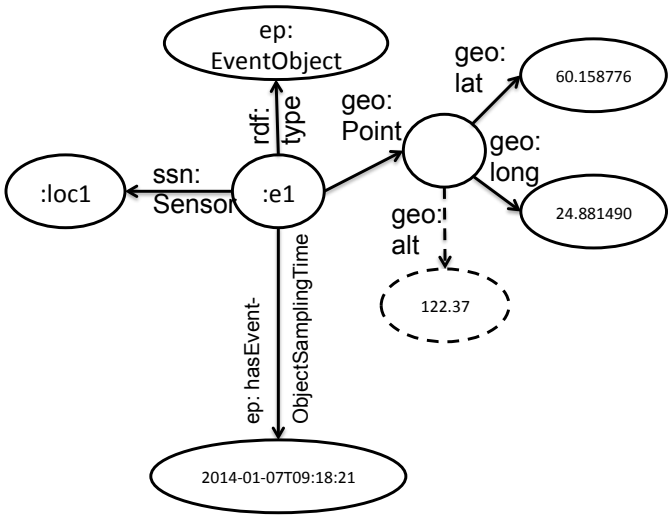

Fig. 1: Example Event for Sensor Location Reporting

to be solvable in finite time known as stream windows [16]. The first platform to demonstrate window-based stream processing in SPARQL was *C-SPARQL*[4] [3–5]. In C-SPARQL RDF streams are built out of time-annotated triples. C-SPARQL provides a window mechanism, where the window size can be defined based on either time or the number of triples. Aggregation operators COUNT, MAX, MIN, SUM and AVG (later incorporated into SPARQL 1.1 Query specification) can be used to compute aggregate values from the windows. There is also a timestamp() function to access the timestamp of a variable. *CQELS*[5] [14] was later introduced as another implementation of window-based streaming SPARQL.

Both C-SPARQL and CQELS assume repeated processing of queries over windows based on either time or number of triples. As pointed out in [18], this approach has some challenges:

– **No support for heterogeneous events:** Delimiting windows by time (with timestamps assigned to individual triples) or the number of triples carries a strong assumption that each event is represented by a single triple, an approach sometimes referred to as data stream processing [13]. Event objects consisting of a variable number of triples would be split across window borders (or merged, if the timestamps are identical). Objects consisting of a static number of triples could be windowed based on the number of triples, but this approach easily gets out-of-sync due to a missing triple.

[4] http://streamreasoning.org/download
[5] http://code.google.com/p/cqels/

– **Challenges in window dimensioning:** When a query involves multiple events in an event stream, the window has to be large enough to capture all the required events. To avoid missing event patterns due to window borders, the windows have to overlap, resulting in duplicate processing and duplicate detections. Faster window repetition helps to decrease detection delay, but leads to more duplicate processing and detections. In an event processing system these conflicting requirements lead to unwanted compromises between duplicate processing, duplicate detections and notification delay.
– **Wasted resources:** Queries are processed repeatedly over the defined windows, even when there are no new matching events.

EP-SPARQL [2] is a streaming environment focusing on the detection of RDF triples in a specific temporal order. The published examples also support heterogeneous event formats, create aggregation over sliding windows using subqueries and expressions, and layer events by constructing new streams from the results of queries. The prolog-based ETALIS[6] incorporates more functionality than the EP-SPARQL front-end.

Another approach to stream processing is to incrementally and asynchronously process each incoming event against a pre-defined set of queries. Instead of windows based on time or number of triples, all processing is based on events, which contribute new input to the queries. When all the required inputs of a query are present, the result is immediately available. Algorithms familiar from production rule systems, such as Rete [9, 10], can be adapted to use SPARQL to describe the rules.

Streaming SPARQL was first presented in [11] using a network highly similar to Rete. Komazec and Cerri [13] apply SPARQL queries to RDF data using an extended Rete-algorithm in a system called *Sparkwave*[7]. Their focus is on supporting selected RDF and RDFS inference rules through the use of a preprocessing ϵ network and fast processing of data streams consisting of individual triples instead of multi-triple events. Rete has also been used for processing SPARQL queries by Depena and Miranker in [7, 17], where the focus is on nested access of dereferenced URIs, exploiting the constructive analogy with forward-chaining production systems.

A SPARQL-based event stream processing platform denoted INSTANS[8] has been developed in our research group. The query language is SPARQL 1.1, implementing also selected properties of SPARQL 1.1 Update, such as *INSERT* and *DELETE*. In [18] we have described the creation of event processing applications using interconnected SPARQL update rules. The concept of collaborating SPARQL queries for stream processing is discussed also in [22], with Teymourian et al preferring a backward-chaining algorithm over the forward-chaining type used in Rete.

[6] http://code.google.com/p/etalis/
[7] https://github.com/skomazec/Sparkweave
[8] Incremental eNgine for STANding Sparql, http://instans.org/

3 Event Objects in RDF

In *data stream processing* it is assumed that each triple carries a standalone event. This can typically be a single reading from a temperature sensor or the license plate of a car from a tollgate. To minimize redundant data transfer and processing in the system, periodic measurements should be filtered as close to the source as possible, elevating to a higher level of abstraction, where only sensor readings falling outside of previously defined reporting thresholds or otherwise unexpected results are forwarded upstream as events. Such event objects typically carry more information than a single sensor reading and timestamp. RDF is better motivated as the data format for heterogeneous asynchronously triggered events than frequent periodic measurements of a single parameter, for which there are more compact representations. Different sensors, or even dif-

```
BASE <http://instans.org/>
PREFIX : <http:instans.org/default#>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX ep: <http://www.ontologydesignpatterns.org/cp/owl/eventprocessing.owl#>
<Eve1> {
   <Eve1> a ep:EventObject;
          ssn:Sensor :loc1;
          geo:Point [ geo:lat 60.158776 ; geo:long 24.881490 ; ] ;
          ep:hasEventObjectSamplingTime "2014-01-07T09:18:21"^^xsd:dateTime }
```

Fig. 2: Prefixes and an example event

ferent driver software versions of the same sensor, may include different types of auxiliary data in addition to the base parameters used by our event processing application. Also in [8] the assumed structure of an event includes a variable-length header, which the event processing system can process, and a body, which is transported as unstructured payload. These structural requirements have been addressed by the *event processing ontology*[9] [19]. To save space, the event objects in this paper are simple events without separate header and body parts, but the extensions to support header, body and composite events from the event processing ontology can be used to extend the examples presented herein as demonstrated in [19].

A sample event format, for the purpose of updating the location of a sensor, is illustrated in Figure 1. The dotted altitude field exemplifies an optional field, which may not be included by all sensors. The turtle serialization encapsulated in TriG is shown in Figure 2. Even though RDF triples can be used to build graphs of infinitely complex structures, there is no clear and reliable way to express the boundaries of a "data record" such as an event object, as pointed out in [12]. This

[9] http://ontologydesignpatterns.org/wiki/Submissions:EventProcessing

may cause problems when event objects in a stream contain optional elements. A query often produces different results depending on whether some optional elements are present, but there is no rule for how long the optional triples should be waited for in the case of an infinite stream. INSTANS can be parametrized to jointly process a "block" of data, which in the case of Turtle jointly processes all consecutive triples connected by a common subject or blank nodes according to rule #6 ("triples") of the Turtle grammar[10]. However, something as simple as sending the latitude and longitude coordinates of Figure 2 before the rest of the event may cause problems: A first block terminates already after the coordinates which, depending on the associated queries, may trigger results different from the case where the coordinates are sent after the event. The issue is further emphasized in a rule-based system like INSTANS, where the user has no explicit control over the order in which simultaneously matching rules are executed and therefore the order in which triples are output.

TriG [6] defines a way of communicating datasets as Turtle by encapsulating graphs. With this approach each event object can be defined as a graph, and each graph processed as a block with no ambiguity. This has clear benefits in matching incoming event objects having optional or unknown content. In [19] unknown content is matched using nested OPTIONAL clauses tracking triples connected by blank nodes. With TriG the situation is greatly simplified, because the entire incoming event object - independent of structural complexity or order of triples - can be matched simply with "?s ?p ?o", as e.g. in Figure 5. The tradeoff is that some of the granularity in using named graphs is lost: Each event object is encapsulated into a named graph and since SPARQL doesn't offer regular expression matching for graphs in one step, the incoming event objects have to be matched to variables and filtered in two separate steps, as shown in the examples in Section 4. As TriG is a new specification, there is currently no specified way to construct TriG output in SPARQL. INSTANS implements a small extension to enable construction of graphs, in TriG-format, using the regular CONSTRUCT operator.

4 Event Processing Agents in SPARQL

The basic building blocks of an *Event Processing Network* (EPN) are shown in Figure 3. *Event Processing Agents* (EPA, 3.) operate on events, which are transported between EPA:s over *Event Channels* (2.). Two special types of EPA:s are shown in the figure: *Event Producer* (1.), which is a source of events and has no inputs in this EPN and *Event Consumer*, which is an event sink and has no outputs in this EPN. It should be noted, however, that in a system with multiple EPN:s the Event Consumer of one EPN can typically be an Event Producer in another. When translated to SPARQL, EPA:s are implemented as SPARQL Update rules (or networks of rules). Event Channels can be modelled as named graphs, providing isolated connection points for event object transfer between EPA:s. The types of event processing agents from [8] are shown in Figure 4. In the

[10] http://www.w3.org/TR/turtle/#sec-grammar

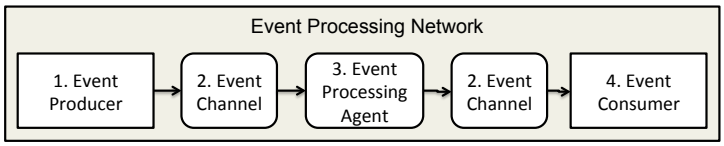

Fig. 3: Building Blocks of an Event Processing Network

following subsections the SPARQL implementation of each type, complemented by the set of filter types from [21], are examined to show the implementations as SPARQL update rules.

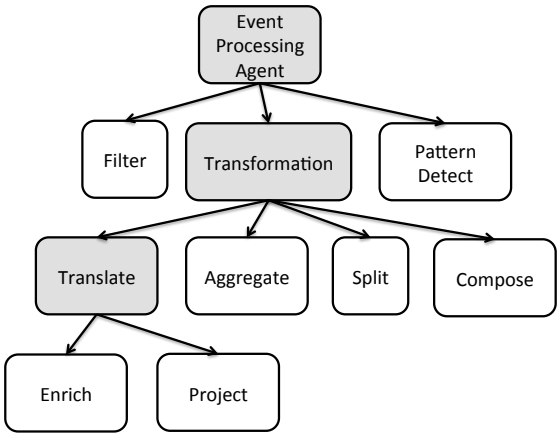

Fig. 4: Types of Event Processing Agents (from [8])

4.1 Filters

Filters select interesting events or eliminate uninteresting ones. A forwarding decision is made based on a test or set of tests. In a stateless filter the decision is fully based on the current event object. An example **(EPA 1)** of a stateless filter to pass through all events generated during business hours is in Figure 5. Since only the time attribute is needed for filtering, it is the only explicitly matched object. Otherwise the entire incoming event object is matched with "?s ?p ?o" and copied in case it passes the filter. The `str(<>)` function produces the BASE URI as a string. It is needed because the graph and event names are prefixed with the BASE URI when read from a file. The BIND-statement is used to prefix the output graph name with "Poststateless", separating it from incoming event objects and thereby creating a new event channel. A stateful filter utilizes

```
INSERT { GRAPH ?poststateless { ?s ?p ?o } }
WHERE { GRAPH ?g { FILTER (strStarts(str(?g),concat(str(<>),"Eve")))
    ?s ?p ?o . ?event ep:hasEventObjectSamplingTime ?time }
  FILTER ( ( HOURS(?time) > 8 ) && ( HOURS(?time) < 17 ) )
  BIND (IRI(concat("Poststateless-",strAfter(str(?event),str(<>))))
                                                    AS ?poststateless) }
```

Fig. 5: EPA 1: A stateless filter passing through events generated during business hours.

information external to the incoming event, e.g. information based on previously processed events or an external source. A stateful filter **(EPA 2)** to pass through maximally one location update for each even hour is shown in Figure 6. This filter, implemented as three SPARQL update rules, demonstrates how a graph is used as memory to save the hour of the previous event for comparison. Explicit memory initialization in EPA2-1 is used to avoid OPTIONAL statements in the other queries. It can be observed that implementing a stateful filter with

```
# EPA2-1 Initialize
INSERT DATA { GRAPH <memory> { :stateful :hour -1 ; :passEvent [] } };
# EPA2-2 Update memory, mark passing events
DELETE { GRAPH <memory> { :stateful :hour ?prevhour ;
                                    :passEvent ?prevPass } }
INSERT { GRAPH <memory> { :stateful :hour ?hour ;
                                    :passEvent ?event } }
WHERE { GRAPH ?g { FILTER ( strStarts(str(?g),"Poststateless-Eve") )
    ?event ep:hasEventObjectSamplingTime ?time }
  GRAPH <memory> { :stateful :hour ?prevhour ;
                             :passEvent ?prevPass }
  BIND ( HOURS(?time) as ?hour)
  FILTER ( ?hour != ?prevhour ) } ;
# EPA2-3 Pass events marked by EPA2-2
INSERT { GRAPH ?poststateful { ?s ?p ?o } }
WHERE {
  GRAPH <memory> { :stateful :passEvent ?event }
  GRAPH ?g { FILTER ( strStarts(str(?g),"Poststateless-Eve") )
    ?event ep:hasEventObjectSamplingTime ?time . ?s ?p ?o }
  BIND (IRI(concat("Poststateful-",strAfter(str(?event),str(<>))))
                                                AS ?poststateful) } ;
```

Fig. 6: EPA 2: A stateful filter passing through one event per even hour.

a streaming SPARQL system based on window repetition would be practically impossible in scenarios where one window should incorporate multiple events. Because all the events in a window are processed jointly as a batch, results of processing one event cannot influence the processing of the next one. Therefore

stateful filters such as the example in Figure 6 typically require a continuous engine, which processes each event as it arrives in a stream.

In principle a SPARQL-based event processing network can support all types of filters, which can be computed with the available arithmetic operators, taking into account that multiple filter stages can be chained like any other EPA:s. One clear limitation is the absence of square root from SPARQL, which causes the calculation of geographical distances to be usually processed with extension functions. As a collection of filter types Table 1 shows the filters listed in [21] as "input specifiers" together with the corresponding SPARQL FILTER statements. The first four filters are clearly stateless. The last one, *change*, does stateful comparison, but it would also be simpler to implement as a combination of an *aggregate* block (EPA6 below) to compute a sliding aggregate value (e.g. min, max, average) over the n previous readings and compare the output of that block to the latest value using a stateless filter.

Description:	**SPARQL filter**
Equal: Check for equality	FILTER (?value1 = ?value2)
About: Equality within a defined tolerance value	FILTER (abs(?value1 - ?value2) < tolerance)
Area: Interval between two values	FILTER ((?value > lower_bound) && (?value < upper_bound))
Greater / Less: Check if the value is greater or less than a defined value.	FILTER (?value > limit) or FILTER (?value < limit)
Change: Tracks changes compared to n previous readings. "Decrease" and "Increase" can be used to specify the direction of change.	Two rules recommended: An aggregate to compute "n previous readings" and a stateless filter to compare.

Table 1: SPARQL implementations of the filter types listed in [21]

4.2 Transformation

Transformation agents modify the content of received event objects. Subtypes are:

- **Translate**: Operate on each event object independently of preceding or subsequent objects using a single in - single out model. A special case is *Enrich*, which attaches additional information to an event object. SPARQL is particularly suitable for this purpose, since any data from a SPARQL endpoint can be queried as a federated query. An example is given in **EPA 3** (Figure 7), where our location events are enriched with preferred geographical labels from *FactForge*[11].

[11] http://factforge.net

– **Project**: Remove information from the incoming event. A simple example of this (**EPA 4**) would be the removal of the recently added location names. As this is a simple copy operation of an incoming event leaving out the unwanted parts, the SPARQL listing is omitted as trivial.
– **Split:** Split a single incoming event into multiple outgoing events. In **EPA 5** (Figure 8) incoming events are split to two channels.
– **Aggregate:** Output a function of incoming events in a multiple-in single-out model. Typical examples are count, min, max, average and sum. The example in **EPA 6** (Figure 9) counts incoming events per hour. It employs five rules to count the aggregate number of events per hour. EPA6-1 initializes the memory graph. EPA6-2 compacts SPARQL code lines by extracting the hour of the incoming event, used by all three subsequent queries. EPA6-3 increases the event counter, 6-4 outputs the event counts when the hour changes and 6-5 resets the counter. Since the output is triggered by on object of the next hour, the last counter result is never produced. This may not be critical in an infinite stream setting, but for recorded streams the final output can be triggered e.g. by a special end marker triple at the end of the file. Alternatively the count could be updated after every event object.
– **Compose:** Combine two or more incoming streams to a single output stream. An example is given as **EPA 7** in Figure 10, a very straightforward reversal of EPA5 with the contents of the *INSERT* and *WHERE* clauses reversed.

```
PREFIX omgeo: <http://www.ontotext.com/owlim/geo#>
PREFIX ff: <http://factforge.net/>
INSERT { GRAPH ?translated { ?event :locationName ?label . ?s ?p ?o } }
WHERE { GRAPH ?g { FILTER ( strStarts(str(?g),"Poststateful-Eve") )
    ?s ?p ?o . ?event a ep:EventObject .
    SERVICE <http://factforge.net/sparql> { # Retrieve location label
       ?location omgeo:nearby(?lat ?long "1km"); ff:preferredLabel ?label }
  BIND (IRI(concat("Translated-",strAfter(str(?event),str(<>))))
                                                    AS ?translated) } }
```

Fig. 7: EPA 3: Enriches incoming events with location labels.

Many streaming SPARQL platforms [3, 14] implement proprietary SPARQL extensions for computation of aggregate values over time windows. These tailor-made solutions result in compact queries, but currently suffer from problems in supporting heterogeneous events due to the window size definitions outlined in Section 2 and different timecodes in streams. There is currently no specified way to use e.g. ep:hasEventObjectSamplingTime from the sample events of this paper as the time base for these window operators, but W3C RDF Stream Processing Community Group[12] is working on defining common models. With manually generated aggregate calculation rules windowing can be based on any

[12] http://www.w3.org/community/rsp/

```
INSERT { GRAPH ?geograph { ?event a ep:EventObject ;
                              ep:hasEventObjectSamplingTime ?time ;
                              geo:Point [ geo:lat ?lat ; geo:long ?long ; ] }
         GRAPH ?sensorgraph { ?event a ep:EventObject ; ssn:Sensor ?sensor ;
                                    ep:hasEventObjectSamplingTime ?time } }
WHERE { GRAPH ?g { FILTER ( strStarts(str(?g),"Poststateful-Eve") )
    ?event a ep:EventObject ; ssn:Sensor ?sensor ;
           geo:Point [ geo:lat ?lat ; geo:long ?long ; ] ;
           ep:hasEventObjectSamplingTime ?time }
  BIND (IRI(concat("GeoEvents-",strAfter(str(?event),str(<>)))) AS ?geograph)
  BIND (IRI(concat("SensorEvents-",strAfter(str(?event),str(<>))))
                                                          AS ?sensorgraph) }
```

Fig. 8: EPA 5: Event objects split to two channels.

```
# EPA6-1 Initialize
INSERT DATA { GRAPH <memory> { :aggrHour :hour -1 ; :counter -1 } } ;
# EPA6-2 Extract incoming hour
INSERT { GRAPH <memory> { ?event :hasAggrHour ?hour } }
WHERE  { GRAPH ?g { FILTER ( strStarts(str(?g),concat(str(<>),"Eve")) )
    ?event ep:hasEventObjectSamplingTime ?time }
  BIND ( HOURS(?time) as ?hour ) } ;
# EPA6-3 Increase event counter
DELETE { GRAPH <memory> { ?x :counter ?oldcount . ?event :hasAggrHour ?hour }}
INSERT { GRAPH <memory> { ?x :counter ?newcount } }
WHERE  { GRAPH <memory> { ?x :hour ?memhour ;
    :counter ?oldcount . ?event :hasAggrHour ?hour }
  FILTER(?hour = ?memhour)
  BIND ( ?oldcount + 1 AS ?newcount ) } ;
# EPA6-4 Output event counts
CONSTRUCT { GRAPH <eventcounts> { # Insert counter event
    [] a ep:EventObject ; :eventType :eventCount ;
       :hour ?memhour ; :count ?count } }
WHERE { GRAPH <memory> { ?x :hour ?memhour ; :counter ?count .
    ?event :hasAggrHour ?hour }
  FILTER ( ?memhour != -1 && ?memhour != ?hour ) } ;
# EPA6-5 Reset memory
DELETE { GRAPH <memory> { ?x :hour ?memhour ; :counter ?count } }
INSERT { GRAPH <memory> { [] :hour ?hour ; :counter 0 } }
WHERE  { GRAPH <memory> { ?x :hour ?memhour ; :counter ?count .
    ?event :hasAggrHour ?hour }
  FILTER ( ?memhour != ?hour ) } ;
```

Fig. 9: EPA 6: Produce aggregate events by counting the number of incoming events per hour.

parameter (e.g. location), not just time, from the incoming RDF stream. Multiple aggregate values can also be computed jointly. In the example above the sum of latitude and longitude coordinates could also be recorded, after which it would be easy to obtain average locations.

```
INSERT { GRAPH ?combined {
    ?event a ep:EventObject ; ssn:Sensor ?sensor ;
           geo:Point [ geo:lat ?lat ; geo:long ?long ; ] ;
           ep:hasEventObjectSamplingTime ?time } }
WHERE { GRAPH ?g1 { FILTER(strStarts(str(?g1),"GeoEvents-"))
          ?event a ep:EventObject ;
                 geo:Point [ geo:lat ?lat ; geo:long ?long ; ] }
        GRAPH ?g2 { FILTER(strStarts(str(?g2),"SensorEvents-"))
          ?event a ep:EventObject ; ssn:Sensor ?sensor ;
                 ep:hasEventObjectSamplingTime ?time }
  BIND (IRI(concat("Combined-",strAfter(str(?event),str(<>)))) AS ?combined) }
```

Fig. 10: EPA 7: Combine the streams, which were earlier split to GeoEvents and SensorEvents

As INSTANS operates asynchronously, windowing based on a real-time clock would require the use of *timed events* ("pulse" event objects triggered based on a clock, in this case to trigger aggregate calculation). In many practical cases using timestamps from the incoming stream is a better approach, because it works on both live and recorded streams and produces more predictable and repeatable results than referencing the clock of the computer running the stream processing platform. The downside is that the absence of an out-of-window trigger may leave the current result waiting infinitely.

4.3 Pattern Detect

Pattern detect agents are searching for patterns in incoming events. They may either describe the pattern, pass through qualifying patterns of incoming event objects or both. Simple patterns could be detected e.g. with slightly augmented stateful filters. A **pattern detect example (EPA 8)** to detect a pattern of movement directions from consecutive location updates is shown here as an example:

1. **Transform location updates to a stream of directions:** Auxiliary query creating a stream of compass events (e.g. "NW") out of an incoming stream of location updates, Figure 11.
2. **Advance pattern index with positive match:** Compare incoming direction events with a pattern in a graph, advance index when they match, Figure 12.
3. **Reset pattern index when not matching:** If a direction event does not match the next item in the pattern, reset index in memory graph, Figure 13.
4. **Detect completed pattern:** When the pattern is complete, indicate detection and reset the index, Figure 14.

It is worth noting that the WHERE-clauses in the two queries of Figure 14 are completely identical. The only reason why they cannot be merged is that SPARQL currently does not allow DELETE and CONSTRUCT in the same query.

```
# EPA8-Transform Initialize memory with 1st incoming point from ?sensor
INSERT { GRAPH <memory> { [] a :transformPrevPoint ; ssn:Sensor ?sensor ;
                          geo:Point [ geo:lat ?lat ; geo:long ?long ; ] } }
WHERE { GRAPH ?g { FILTER (strStarts(str(?g),concat(str(<>),"Eve")))
    ?event a ep:EventObject ; ssn:Sensor ?sensor ;
           geo:Point [ geo:lat ?lat ; geo:long ?long ; ] }
  FILTER NOT EXISTS {
    GRAPH <memory> { ?x a :transformPrevPoint ; ssn:Sensor ?sensor } } } ;
# EPA8-Transform subsequent points to compass directions
DELETE {  GRAPH <memory> { # Delete previous point
  ?mem geo:Point ?blnk . ?blnk geo:lat ?prevlat ; geo:long ?prevlong } }
INSERT { GRAPH <memory> { # Write new point
    ?mem geo:Point [ geo:lat ?lat ; geo:long ?long ; ] }
  GRAPH <directions> { # Output direction
    ?event a ep:EventObject ; ssn:Sensor ?sensor ;
           ep:hasEventObjectSamplingTime ?time ; :direction ?dir } }
WHERE { GRAPH ?g { FILTER (strStarts(str(?g),concat(str(<>),"Eve")))
    ?event a ep:EventObject ; ssn:Sensor ?sensor ;
           ep:hasEventObjectSamplingTime ?time ;
           geo:Point [ geo:lat ?lat ; geo:long ?long ; ] . }
  GRAPH <memory> { # Retrieve previous point
    ?mem a :transformPrevPoint ; ssn:Sensor ?sensor ;
         geo:Point ?blnk . ?blnk geo:lat ?prevlat ; geo:long ?prevlong }
  BIND ( IF ( ?lat > ?prevlat, "N", "") as ?latdir1) # Test for directions
  BIND ( IF ( ?lat < ?prevlat, "S", "") as ?latdir2)
  BIND ( IF ( ?long > ?prevlong, "E", "") as ?longdir1)
  BIND ( IF ( ?long < ?prevlong, "W", "") as ?longdir2)
  BIND ( concat(?latdir1, ?latdir2, ?longdir1, ?longdir2) as ?tdir )
  BIND ( IF ( !bound(?tdir) || strlen(?tdir)=0, "0", ?tdir ) as ?dir ) }
```

Fig. 11: Transform: Generate direction events out of subsequent locations

5 Conclusions

The benefits of using SPARQL for event processing include all the benefits of semantic web in general, e.g. agreed specifications, conceptual compatibility through shared ontologies, compatibility with the current tool base, support for loosely coupled heterogeneous systems, straightforward connectivity to linked open data for enriching event information and a solid base of inference mechanisms to enhance reasoning over events. The principle of processing heterogeneous events using a network of SPARQL queries and update rules was first presented in [18] through a practical event processing example solved by three interconnected rules and a result output query. This paper takes a more systematic approach by showing working examples of every type of event processing agent found in [8]. The SPARQL implementation of the "input modifiers" (filter types) mentioned in [21] is also reviewed. Building on top of the principles introduced in [19], the benefits of TriG input over plain Turtle in encapsulating events objects are explained and demonstrated. While the examples in this paper have

```
# EPA8-Pattern-Initialize memory with 1st direction entry from sensor
INSERT { GRAPH <memory> { # Initialize a new entry to memory
      [] a :patternIndexEntry ; ssn:Sensor ?sensor ;
           :basedOnEvent 0 ; :patternIndex 0 } }
WHERE { GRAPH <directions> { ?event a ep:EventObject ; ssn:Sensor ?sensor }
  FILTER NOT EXISTS { GRAPH <memory> {
    ?mem a :patternIndexEntry ; ssn:Sensor ?sensor } } } ;
# EPA8-Pattern-Advance index when direction matches with pattern
DELETE { GRAPH <memory> { # Clear old index from memory
    ?mem :basedOnEvent ?oldEvent ; :patternIndex ?oldIndex } }
INSERT { GRAPH <memory> { # Write new index to memory
    ?mem :basedOnEvent ?event ; :patternIndex ?newIndex } }
WHERE { GRAPH <directions> { ?event a ep:EventObject ;
                                ssn:Sensor ?sensor ; :direction ?dir }
  FILTER ( ?dir != "0" )
  GRAPH <memory> { # Retrieve index from memory
      ?mem a :patternIndexEntry ; ssn:Sensor ?sensor ;
           :basedOnEvent ?oldEvent ; :patternIndex ?oldIndex }
  FILTER ( !sameTerm(?event,?oldEvent) ) # Only once per event
  GRAPH <pattern> { # Retrieve pattern index and value
    ?pattern :index ?ptIndex ; :value ?ptValue }
  BIND ( ?oldIndex+1 as ?newIndex )
  FILTER ( (?dir = ?ptValue) && (?newIndex = ?ptIndex) ) }
```

Fig. 12: EPA 8-1: Advance an index if the incoming direction matches with a pre-defined pattern

```
DELETE { GRAPH <memory> { # Remove index from memory
    ?mem a :patternIndexEntry ; ssn:Sensor ?sensor ;
         :basedOnEvent ?oldEvent ; :patternIndex ?oldIndex } }
WHERE { GRAPH <directions> { # Match directional input
    ?event a ep:EventObject ; ssn:Sensor ?sensor ; :direction ?dir }
  GRAPH <memory> { # Retrieve index from memory
      ?mem a :patternIndexEntry ; ssn:Sensor ?sensor ;
           :basedOnEvent ?oldEvent ; :patternIndex ?oldIndex }
  FILTER ( !sameTerm(?event,?oldEvent) ) # Must be different events
  GRAPH <pattern> { ?pattern :index ?ptIndex ; :value ?ptValue }
  FILTER ( (?dir != ?ptValue) && (?oldIndex+1 = ?ptIndex) && (?oldIndex>0) ) }
```

Fig. 13: EPA 8-2: Reset index if the latest direction does not match with the pattern

been prepared manually, it would also be possible to create end-user tools to synthesize the appropriate rules and queries.

Based on the examples it is observed that every type of event processing agent found in the referenced literature can be expressed using a SPARQL update rule or network of rules. While the examples do not extend to prove that every event processing task would have a corresponding solution in SPARQL, they serve to demonstrate a systematic approach up to a level of complexity,

```
# EPA8-Pattern-Reset index when pattern is complete
DELETE {  GRAPH <memory> { # Remove last index from memory
    ?mem a :patternIndexEntry ; ssn:Sensor ?sensor ;
         :basedOnEvent ?oldEvent ; :patternIndex ?oldIndex } }
WHERE { GRAPH <pattern> { :pattern :length ?length }
  GRAPH <memory> {
      ?mem a :patternIndexEntry ; ssn:Sensor ?sensor ;
           :basedOnEvent ?oldEvent ; :patternIndex ?oldIndex }
  FILTER ( ?oldIndex = ?length ) } ;
# EPA8-Pattern-Output result
CONSTRUCT { GRAPH <PatternDetect> {
    [] a ep:EventObject ; :patterndetected "Pattern detected!" ;
       ssn:Sensor ?sensor ; :lastEvent ?oldEvent } }
WHERE { GRAPH <pattern> { :pattern :length ?length }
  GRAPH <memory> {
      ?mem a :patternIndexEntry ; ssn:Sensor ?sensor ;
           :basedOnEvent ?oldEvent ; :patternIndex ?oldIndex }
  FILTER ( ?oldIndex = ?length ) } ;
```

Fig. 14: EPA 8-3: Output result when the designated pattern is complete. Reset pattern index.

which would be sufficient for many real-life scenarios. SPARQL is observed to be capable of serving as the foundation for event processing systems. However, limitations especially in arithmetic operators are frequently forcing the inclusion of proprietary library functions, e.g. for the calculation of distance between two geographical points.

The experiment has also revealed in SPARQL 1.1 some structural shortcomings, which could be addressed in future releases of the specification set:

1. *Generating dataset output from a query:* Even though TriG is specified as a format to convey datasets and SPARQL can query TriG input, there is currently no way to produce dataset output with a query. We have added to CONSTRUCT the capability to generate graphs as TriG output and have observed no negative impacts either to the coherence of the syntax or the INSTANS implementation.
2. *Combination of output and update:* As we saw in Figure 14, the SPARQL code could be made more compact by allowing multiple result processing operators (e.g. CONSTRUCT and INSERT) in the same query. Even more importantly, it is currently very difficult to chain any operation to take place after output processing, because SELECT or CONSTRUCT queries cannot produce any changes detectable by other queries.

In EPA6 (Figure 9) some codelines were reduced by adding an extra query to extract a parameter used by three other queries. INSTANS automatically merges identical parts of queries in the internal Rete representation. Therefore this reduction is only significant in terms of SPARQL text; the compiled rule processing network is the same. Due to limits in the capabilities of a single query, similar

parts often need to be used in multiple queries. A macro mechanism would reduce the need to replicate the same SPARQL code to multiple places, compacting the source code and reducing opportunities for inconsistency.

The individual example rules presented in this paper connect to each other, forming the event processing network shown in Figure 15. This size of network would already be capable of solving quite complex event processing tasks. The complete set of rules is available in executable form in the INSTANS github repository[13]. Memory management was not discussed in detail in this document, but the complete query set in the repository avoids garbage buildup by two means:

1. An operational policy of INSTANS removes input triples after all the rules, which match with a given input, have been processed.
2. Explicit "cleanup"-queries delete old events from the channels between queries.

While putting all of these rules into a single Rete engine serves to demonstrate that complex networks can be created, it is not the most efficient solution. No EPA shown here is exchanging any other information than event objects with other EPA:s, in which case it would be better to run each EPA in a separate instance of INSTANS. This modularizes the approach and ensures that there will not be any unwanted side-effects due to unplanned matching of rules between different agents.

The challenges of applying synchronous query repetition over windows in event processing tasks were discussed in [18]. In this document it was additionally observed that the creation of stateful filters has further challenges in windowed environments because conventional SPARQL matches graph patterns over the complete addressable dataset, in this case window, before acting on the results. Unless it can be guaranteed that a window contains precisely one event object (thereby precluding cases using multiple event objects from the same stream as input), the results of processing one event cannot impact the processing of the next. The results of stateful filtering can also be sensitive to repeated processing, causing symptoms with overlapping (non-tumbling) windows.

After translation of event processing tasks to SPARQL, the next step will be to describe the translation of SPARQL to a query network. Not all aspects of SPARQL are applicable to continuous event-triggered processing, and in some cases the semantics need to be adjusted. The aspects of synchronized aggregate calculation and processing of timed events in general also need to be elaborated further. While the synchronized computation of aggregate values is not seen as a key aspect of an asynchronous event processing platform, some reporting tasks and cleaning of noisy sensor input may require aggregate calculations. Timed events are also essential in detecting missing events and need to be addressed in more detail in future work.

[13] https://github.com/aaltodsg/instans/tree/master/tests/input/CEP2SPARQL

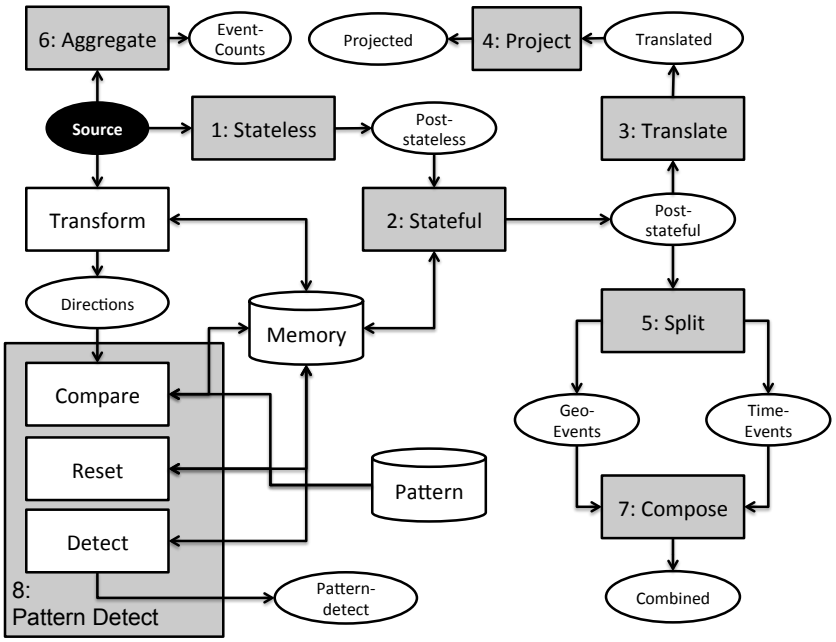

Fig. 15: The example event processing network created in the paper.

Acknowledgments

This work has been carried out in Spaceify, SPIRE and TrafficSense projects funded by European Commission through the SSRA (Smart Space Research and Applications) activity of EIT ICT Labs[14], Tekes and Aalto University.

References

1. Abdullah, H., Rinne, M., Törmä, S., Nuutila, E.: Efficient matching of SPARQL subscriptions using Rete. In: Proceedings of the 27th Symposium On Applied Computing. Riva del Garda, Italy (Mar 2012)
2. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: Proceedings of the 20th international conference on World wide web (WWW'11). pp. 635–644. ACM (2011)
3. Barbieri, D.F., Braga, D., Ceri, S., Grossniklaus, M.: An execution environment for C-SPARQL queries. In: Proceedings of the 13th International Conference on Extending Database Technology - EDBT '10. p. 441. Lausanne, Switzerland (2010)
4. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: C-SPARQL: A Continuous query language for RDF data streams. International Journal of Semantic Computing 04, 3 (2010)

[14] http://eit.ictlabs.eu/ict-labs/thematic-action-lines/smart-spaces/

5. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: Querying RDF streams with C-SPARQL. ACM SIGMOD Record 39, 20 (Sep 2010)
6. Bizer, C., Cyganiak, R.: RDF 1.1 TriG, http://www.w3.org/TR/trig/
7. Depena, R.K.: Diamond : A Rete-Match Linked Data SPARQL Environment (M.Sc. Thesis). Ph.D. thesis, University of Texas at Austin (2010)
8. Etzion, O., Niblett, P., Luckham, D.: Event Processing in Action. Manning Publications (Jul 2010)
9. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence 19(1), 17–37 (Sep 1982)
10. Forgy, C.L.: On the efficient implementation of production systems. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1979), aAI7919143
11. Groppe, S., Groppe, J., Kukulenz, D., Linnemann, V.: A SPARQL Engine for Streaming RDF Data. In: Third International IEEE Conference on Signal-Image Technologies and Internet-Based System. pp. 167–174. IEEE (Dec 2007)
12. Keskisärkkä, R., Blomqvist, E.: Event Object Boundaries in RDF Streams - A Position Paper. In: OrdRing 2013 - 2nd International Workshop on Ordering and Reasoning, CEUR workshop proceedings (2013)
13. Komazec, S., Cerri, D.: Towards Efficient Schema-Enhanced Pattern Matching over RDF Data Streams. In: 10th ISWC. Springer, Bonn, Germany (2011)
14. Le-Phuoc, D., Dao-Tran, M., Parreira, J.X., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: ISWC'11. pp. 370–388. Springer-Verlag Berlin (Oct 2011)
15. Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley Professional, 1 edn. (May 2002)
16. Luckham, D., Schulte, R.: Event Processing Glossary Version 2.0 (Jul 2011), http://www.complexevents.com/
17. Miranker, D.P., Depena, R.K., Hyunjoon, J., Carlos, R., Sequeda, J.F.: Diamond: A SPARQL Query Engine, for Linked Data Based on the Rete Match. In: Gueret, C., Sharffle, F., Ienco, D., Villata, S. (eds.) 1st International Workshop on Artificial Intelligence meets the Web of Data (ECAI 2012). pp. 12–17. Montpellier (2012)
18. Rinne, M., Abdullah, H., Törmä, S., Nuutila, E.: Processing Heterogeneous RDF Events with Standing SPARQL Update Rules. In: Meersman, R., Dillon, T. (eds.) OTM 2012 Conferences, Part II. pp. 793–802. Springer-Verlag (2012)
19. Rinne, M., Blomqvist, E., Keskisärkkä, R., Nuutila, E.: Event Processing in RDF. In: Proceedings of WOP2013. p. 13. CEUR Workshop Proceedings (2013)
20. Rinne, M., Törmä, S., Nuutila, E.: SPARQL-Based Applications for RDF-Encoded Sensor Data. In: 5th International Workshop on Semantic Sensor Networks (2012)
21. Taylor, K., Leidinger, L.: Ontology-Driven Complex Event Processing in Heterogeneous Sensor Networks. In: 8th Extended Semantic Web Conference (ESWC). pp. 285–299. Springer Berlin Heidelberg, Heraklion, Crete, Greece (2011)
22. Teymourian, K., Rohde, M., Paschke, A.: Fusion of background knowledge and streams of events. In: Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems - DEBS '12. pp. 302–313. ACM Press, New York, New York, USA (2012)