High-performance Distributed Stream Reasoning using S4

Jesper Hoeksema and Spyros Kotoulas

Dept. of Computer Science, VU University Amsterdam, the Netherlands
{jehoekse,kot}@few.vu.nl

Abstract. In this paper, we present a high-throughput system for reasoning over RDF streams. Recognising the inherent limitations of sequential approaches, we turn to parallel methods, utilizing a compute cluster and the Yahoo S4 framework. We are presenting a set of low-level predicates operating on a stream and an encoding of RDFS reasoning and C-SPARQL query answering using these predicates. We perform experiments on up to 32 compute nodes. Our results indicate that our method scales well and can process hundreds of thousands of triples per second.

1 Introduction

Stream reasoning has gained traction as a way to cope with rapidly changing information on the Semantic Web [5]. Centralized approaches pose limitations on the computational resources that can be utilized for processing this information. As the number of information producers and the rate at which they generate data increases, centralization puts a barrier on the applicability of stream reasoning.

In this paper, we are aspiring to remove this barrier by proposing a method to distribute the task over a cluster of computers. We focus on two sub-tasks: *calculating the closure* and *running continuous queries* over an RDF stream.

Closure computation. We show that an approach based on naive distribution of triples across nodes does not scale. Besides incurring unnecessary computation, it creates load balancing issues that hamper the scalability of our system. Working on the assumption that, typically, schema triples are relatively few and inserted in the system first, we are presenting an improved scheme for distributing triples. With this scheme, we are (a) reducing the number of times each triple needs to be processed, (b) improve the load balancing properties of our system and (c) limit the number of processing elements a triple needs to pass through, by grouping operations.

Query answering. Queries over RDF streams are continuous, with the prominent language for expressing them being C-SPARQL [2]. We have developed a set of stream operators that can be used to support key features in C-SPARQL.

For the implementation of our methods, we turn to a distributed stream processing framework, Yahoo S4 [8]. We encode the operators we have developed in the primitives used by S4 and we evaluate our system on 32 compute nodes. Our preliminary results indicate that our method can process hundreds of thousands triples per second and scales fairly linearly with the number of nodes.

2 Related work

2.1 RDF Streams

Traditional RDF repositories contain a static set of information which can be queried. An RDF stream contains a continuous changing flow of information. In most streams, the latest information is the most relevant, as it describes the current state of some dynamic system. An RDF stream consists of an ordered sequence of pairs, where each pair consists of a triple and a timestamp [2]. Reasoning over RDF streams is more complex than reasoning over static RDF data, due to the fact that streams are usually read through a sliding window of either a fixed time or a fixed number of statements. Triples that have exited the window are no longer valid in the context of the application reading the stream, implying that triples derived from them should also exit the window.

C-SPARQL (Continuous SPARQL) is an extension to SPARQL that allows querying RDF streams[2]. It allows users to register queries which will periodically output their gathered data, possibly using aggregation. An example C-SPARQL query is shown in Listing 1.1.

```
1  REGISTER QUERY CarsEnteringCityCenterPerDistrict AS
2  SELECT DISTINCT ?district ?passages
3  FROM STREAM <www.uc.eu/tollgates.trdf> [RANGE 30 MIN STEP 5 MIN]
4  WHERE {
5     ?toll t:registers ?car .
6     ?toll c:placedIn ?street .
7     ?district c:contains ?street .
8   }
9  AGGREGATE { (?passages, COUNT, {?district }) }
```

Listing 1.1: Example C-SPARQL query (from Barbieri *et al.* [2]) counting the number of cars passing through toll gates

Barbieri *et al.* [1] propose and implement a C-SPARQL execution environment that works by transforming the C-SPARQL to a SPARQL query and a continuous query, then passing the queries to a sesame repository and STREAM (a relational stream processor) respectively. In contrast, we are using a single parallel environment to process C-SPARQL queries and our approach vastly outperforms the proposed approach.

Barbieri *et al.* [3] further propose a method of maintaining derived statements from a stream by adding an expiration time. Our system uses a similar algorithm to maintain a rolling window over derived statements but does so in a parallel environment.

2.2 Distributed Reasoning

MaRVIN[7, 9] brings forward a method named *divide-conquer-swap* to do inferencing through forward chaining: The input is divided into several partitions. These partitions are loaded by a number of peers and the closure of the data contained in these partitions is calculated. The closure and the original data are re-partitioned and the process

repeated until no new triples are derived. Although this approach also outputs results as they are calculated, it can not be used to process queries neither can it handle changes in the schema or ordering.

WebPIE [10, 11] is a high-performance OWL reasoner using the MapReduce [4] parallel programming model. It performs RDFS and OWL-horst reasoning on a cluster. It deals with issues of load-balancing, data distribution and unnecessary computation through a series of optimisations. In contrast to our work, this method is implemented as a batch job over static data and, additionally, can not process queries.

2.3 Yahoo S4

Yahoo S4 [8] is a streaming platform that allows developers to create distributed parallel streaming applications to process continuous unbounded streams of data. This is accomplished by writing small subprograms, called Processing Elements (PEs), that are able to consume events from a certain stream, and either publish their results directly, or emit new events on another stream, which may then be consumed by other PEs. Streams are distributed across PEs according to a given key (i.e. all events of a given stream and a given key will be processed by the same PE). This means that the PEs can be spread across different nodes, allowing for a parallel approach. When the keys are assigned intelligently, it will result in an even partitioning of the stream events, resulting in a highly scalable system.

3 Naive reasoning using S4

To illustrate the use of S4, let us describe a fairly simple, incomplete, RDFS reasoner in S4. The basic structure of this application is illustrated in Figure 1.

Triples arrive at the input stream *InputTriple*. This input stream does not have a key property (indicated by the asterisk in the consuming SplitterPE). This means that every S4 Processing Node will have exactly one SplitterPE which will process every triple arriving on the InputTriple stream for that node.

As mentioned above, any PE is in essence a small subprogram that processes events from one or more assigned streams. The SplitterPE is no exception. It receives Triple objects from the InputTriple stream, and emits (publishes) every received triple to three new streams: SplitTripleS, SplitTripleP and SplitTripleO, which are illustrated in Figure 1 with solid, dashed and dotted lines respectively.

These three streams each have a key property assigned: SplitTripleS is assigned the subject as key, SplitTripleP has the predicate assigned as key, and SplitTripleO the object. This means that each triple passing through one of these streams will be partitioned according to each of its terms.

For each distinct value that S4 encounters for a key property, a separate instance of the receiving PE class is instantiated. For example, the triple *(<Bessie> rdf:type <Cow>)* will be received by the RuleProcessingPE for *Bessie* (through the SplitTripleS stream), the RuleProcessingPE for *rdf:type* (through SplitTripleP), and the RuleProcessingPE for *Cow* (Through SplitTripleO).

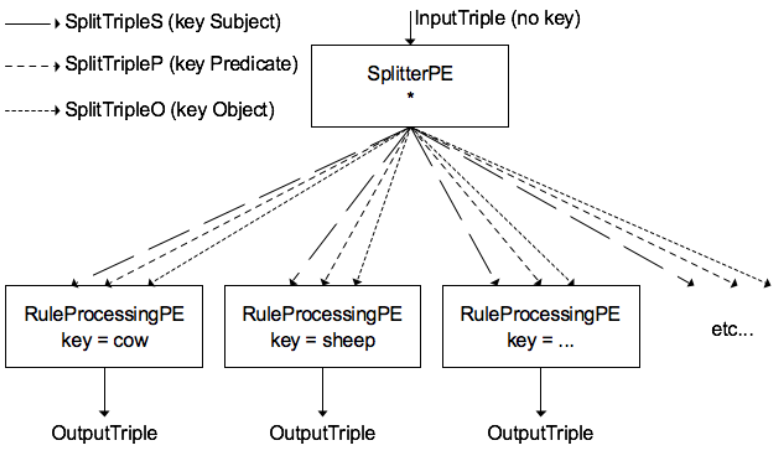

Fig. 1: A naive RDFS reasoner program in S4

If we look at this from another viewpoint, we can see that every RuleProcessingPE will receive all triples that have its key either in its subject, its predicate, or its object property, and it will receive no triples that do not contain its key value. Also note that for each distinct key, there is only one PE instance in the whole S4 cluster (for example the RuleProcessingPE for *cow* can be situated on node 1, while the RuleProcessingPE for *sheep* is running on node 8).

The RuleProcessingPE will simply remember all triples it receives, and for every new incoming triple check if it matches one of the RDFS entailment rules, either on its own or together with one of the earlier received triples. If this is the case, the resulting triple from that entailment rule is emitted on the OutputTriple stream. Essentially, this configuration performs all the pairwise joins for the input.

Obviously, the reasoner in this example suffers from a number of problems: It creates an excessive number of PEs, since it also remembers triples that will never lead to any inference, it is wasting resources, since it is also performing joins that will not lead to any inference, it is incomplete, since each triple is not fed back to the input and it creates duplicate derivations, since multiple rules can generate the same inference. In the following section, we will present a complete and more efficient method for RDFS reasoning.

4 Efficient RDFS reasoning

In this section, we introduce a number of specialized reasoning PEs, as well as a splitter PE that will distribute the triples over multiple streams in a more efficient way. The PEs and the streams between them are illustrated in Figure 2. The workings of each of these PEs will be described in the sections below.

In this paper, we will assume that schema triples only affect the closure for the triples that have been inserted after them. We believe that this is a reasonable assumption, since typically, the schema is not part of the stream but it is inserted separately. Furthermore, our reasoning process is *eventually complete*, meaning that are guaranteed to derive all possible conclusions only after a given time period has passed since the last schema triple was inserted or derived. Intuitively, this time period is equal to the amount of time it takes our system to clear its input and output buffers, but we leave a more precise estimate and a proof to future work.

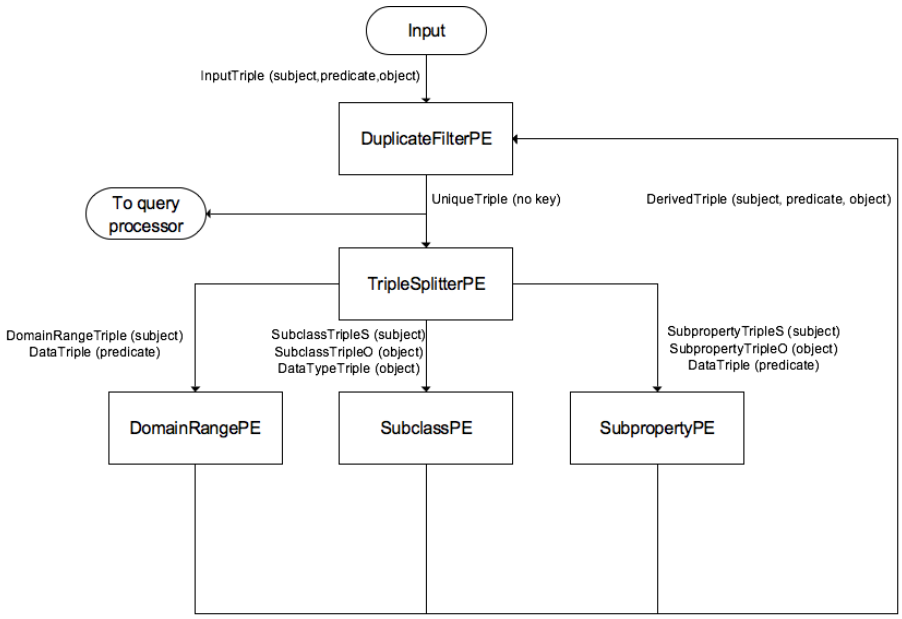

Fig. 2: RDFS reasoning flow

DuplicateFilterPE The DuplicateFilterPE is eliminating duplicate triples. The keys of both input streams (InputPE and DerivedPE) for this PE are <subject,predicate,object>, which means that for every unique triple one PE instance is created.

TripleSplitterPE The TripleSplitterPE analyzes each input triple and emits it on one or more output streams, depending on the possible derivations in the RDFS ruleset[1]:

– If the triple's predicate is *rdfs:domain* or *rdfs:range*, dispatch it on stream DomainRangeTripleS, keyed on subject, to be consumed by DomainRangePE (rdfs rules 2 and 3)

[1] http://www.w3.org/TR/rdf-mt/

- If the triple's predicate is *rdfs:subPropertyOf*, dispatch it on streams SubpropTripleS, keyed on subject, and SubpropTripleO, keyed on object, to be consumed by SubpropertyPE (rdfs rules 5 and 7)
- If the triple's predicate is *rdfs:subClassOf*, dispatch it on streams SubclassTripleS, keyed on subject, and SubclassTripleO, keyed on object, to be consumed by SubclassPE (rdfs rules 9 and 11)
- If the triple's predicate is *rdf:type*, dispatch it on stream DataTypeTripleO, keyed on object, to be consumed by SubclassPE (rdfs rules 6,8,9,10,12,13)
- Dispatch all triples to DataTripleP, keyed on predicate, to be consumed by SubpropertyPE (rdfs rules 4a, 4b and 7) and DomainRangePE (rdfs rules 2 and 3)

DomainRangePE The DomainRangePE (Algorithm 1) computes the results of RDFS rules 2 and 3. It receives triples of the form (x rdfs:domain d) and (x rdfs:range r) from the DomainRangeTripleS stream, and triples of the form (s x o) from the DataTripleP stream. For both these keys, x is the key assigned to the specific PE instance, so one PE will receive all triples where x is assigned a certain value. Due to the fact that we assume schema triples (and thus triples from DomainRangeTripleS) will be entered into the system before any data triples, it will only be necessary to remember the triples from the schema stream in two lists - one for the domain triples and one for the range triples. Then, whenever a triple arrives from the data stream, a simple iteration over the two lists is sufficient to generate all resulting triples.

```
1  domainTriples ← ∅
2  rangeTriples ← ∅
3  processEvent(triple) begin
4      if (triple.predicate = "rdfs:domain") then
5          domainTriples.add(triple);
6      if (triple.predicate = "rdfs:range") then
7          rangeTriples.add(triple);
8      for t ∈ domainTriples do
9          emit("DerivedTriple",(triple.subject,"rdf:type",t.object));
10     if ¬ triple.object.type = literal then
11         for t ∈ rangeTriples do
12             emit("DerivedTriple",(triple.object,"rdf:type",t.object));
13 end
```

Algorithm 1: DomainRangePE

SubclassPE The SubclassPE computes the results of RDFS rules 6, 8, 9, 10, 11, 12 and 13. For data triples, it works analogously with the DomainRangePE. For schema triples, (i.e. triples from the SubclassTripleS and SubclassTripleO streams), when a join

can be made between the subject of one triple and the object of another (i.e. we have two triples originating from different streams with the same key), a new schema triple is generated.

SubpropertyPE The SubpropertyPE works similar to the SubclassPE described above, but instead generates triples concerning subproperties. The results of RDFS rules 4a, 4b, 5 and 7 are generated here.

5 C-SPARQL query processing

We have created a number of components that can be combined to translate a subset of C-SPARQL into a parallel execution plan. Figure 3 shows an example setup of PEs: Initially, the input is processed by a set of PatternMatcherPEs, which bind variables from the input stream. The output is sent to to a network of BindingJoinPEs, which perform the joins in the query. It is also possible to apply some filtering functions using a FilterPE placed after or before the BindingJoinPE. The final results are produced by an OutputTransformerPE. We will briefly describe each PE.

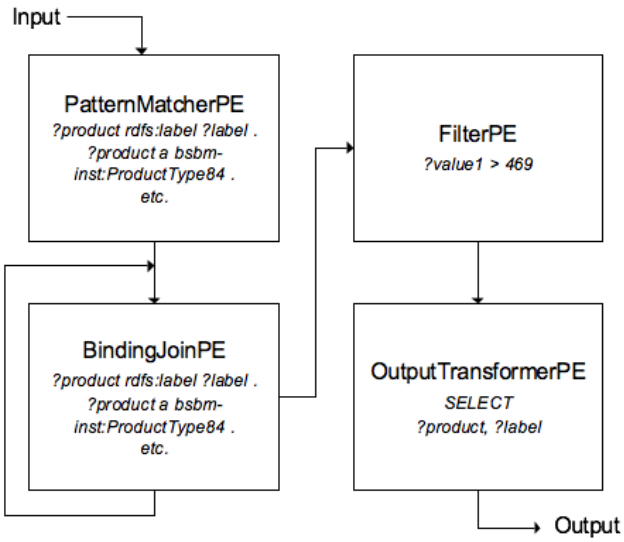

Fig. 3: Query processing

PatternMatcherPE The PatternMatcherPE filters out triples that do not match any triple pattern in the graph and creates variable bindings. Being a keyless PE, there is one instance of the PatternMatcherPE on each processing node, which receives all triples

that enter the SPARQL processing part on that specific node. This makes 'searching' for triples highly parallel. The PatternMatcher PE outputs variable bindings, sorted by variable name. The pseudocode for this PE can be found in Algorithm 2

```
1 triplePatterns ← configuration
2 outputStream ← configuration
3 processEvent(triple) begin
4     for p ∈ triplePatterns do
5         if p.matches(triple) ∧ p.numVars > 0 then
6             vb ← new VariableBinding (p.fillVariables(triple),p);
7             emit(outputStream,vb);
8 end
```

Algorithm 2: PatternMatcherPE

BindingJoinPE For each variable in the query, a set of PEs is created. Variables are given some order[2]. Each set of PEs processes the joins for a given variable. This is implemented by listening to all PatternMatcherPEs that emit bindings for patterns containing this variable and holding the bindings in one list per pattern. When a variable binding appears in all lists, a new binding is created and the emitted to the PEs for the next variable. This is repeated until there are no variables left. In essence, the query tree is evaluated bottom-up, and a sequence of PEs is created with each element corresponding to a join in the query. We have also developed a variant of the BindingJoinPE, the OptionalJoinPE, which implements outer joins.

FilterPE For each incoming binding, the FilterPE checks whether it matches the filter condition provided in the configuration. If this matches, the binding is sent on to its output stream, otherwise it is ignored.

Unions In order to support alternative (union) graph patterns, no additional component needs to be introduced. The outputs from two PatternMatcher/BindingJoinPE pairs can use the same output stream, which will result in the union of the two being inserted into this output stream.

OutputTransformerPE The OutputTransformerPE will create a projection of the distinguished variables and, in case of a CONSTRUCT query, an extension. It is also responsible for formatting the results.

[2] Currently variables are ordered in alphanumeric order, but future work lies in optimizing the variable order (which corresponds to the join order)

6 Windowing and aggregation

C-SPARQL supports two types of windows: a window containing a fixed number of triples and a window containing all the triples that entered the stream in a fixed time period. The former is inappropriate for a distributed environment, since there is no practical way to determine which triple enters the system first (i.e. there are several machines receiving input from multiple sources in parallel). Hence, we have focussed our efforts on windows of a fixed time period.

To maintain the inferences up to date, we draw heavily from the work in [3]. Each triple inserted in the system is attached a timestamp and an expiration time. This way we can push multiple streams, all with a different lifetime, and even give static data an infinite lifetime. We then define that each triple is valid until the system's timestamp is greater than the triple's timestamp plus the triple's lifetime; invalid triples are purged.

The derived triples get as a timestamp the time they were derived and an expiration time corresponding to the triple which expires first. For triples that are derived again, the timestamp and expiration time are updated at the DuplicateFilterPE.

In C-SPARQL, aggregates generate additional variables to all bindings that contribute to their value. This is done based on the conviction that in the context of RDF, knowledge should be extended rather than shrunk[3]. This is contrary to traditional (SQL-based) grouping and aggregates which replaces every group of tuples with a single one containing the aggregate value.

To facilitate these aggregates, we provide a PE called AggregatePE which can be used in the query configuration file just before the OutputTransformerPE. The AggregatePE receives bindings that are keyed to the values of the variables to group on. This means that per group, one AggregatePE is instantiated, which simply keeps a window-aware list of all bindings it receives, and periodically calculates the aggregates, adds them to the bindings in the lists, and outputs them all at the same time. All aggregations are supported, such as SUM, AVG, COUNT, MIN and MAX. Results from an AggregatePE may be further filtered by sending it through a FilterPE.

In order to support multiple aggregates, we have developed variants of the above PE which tag triples and bindings by the aggregation being calculated.

7 Preliminary Evaluation

7.1 Experimental Setup

We have developed an open source[4] implementation using S4 version 0.3 and java 1.6.

Platform We have evaluated our implementation using the DAS-4[5] VU cluster, consisting of dual quad-core compute nodes with 24GB of RAM each, connected using 1Gbps ethernet.

[3] http://wiki.larkc.eu/c-sparql/sparql11-feedback
[4] http://svn.majorerror.com/thesis/
[5] http://www.cs.vu.nl/das4/

Datasets We have used the LUBM benchmark [6] to test reasoning and part of the BSBM benchmark[6] to test query answering. In both cases, we have split the generated datasets in 32 pieces and published them in parallel. Triples were given the timestamp of the time they were read from the input. In all experiments, we have limited the maximum throughput to 160.000 triples per second, due to limitations outside our system for publishing triples.

Evaluation criteria Our evaluation criteria were maximum triple throughput in the input (expressed in triples per second) and the number of processing nodes.

7.2 Experiments

No reasoning, passthrough query To evaluate the baseline performance of our system, we have used a passthrough query[7] with no reasoning, over the LUBM dataset. In Figure 4 (top left), we see that, even for a small number of processing nodes, the throughput of the system is very high and scales fairly linearly.

RDFS reasoning, passthrough query For this experiment, RDFS reasoning was enabled, and a simple passthrough query was used over the LUBM dataset. In this experiment, shown in Figure 4 (top right), one can also observe the high-throughput of the system and fairly linear scalability up to 8 nodes. for 16 and 32 nodes, we see that the throughput does not increase linearly, a fact meriting further investigation.

No reasoning, BSBM explore query 1 For this experiment, reasoning was disabled, and modified version of the first query from the BSBM query mix was used, which does not do aggregation (Listing 1.2). The ORDER and LIMIT constructs were omitted due to the fact that they are not supported by our streaming system and the STEP was set to 0 min (meaning that results were immediately put out). The window was set to 60 seconds, which means that for the maximum throughput in our experiment, we had around $throughput * 60 = 9.600.000$ triples in the window. Linear scalability is shown in Figure 4 (bottom left).

```
1  REGISTER QUERY BSBMExplore1 AS
2  SELECT DISTINCT ?product ?label
3  FROM STREAM < ... > [RANGE 60 SEC STEP 0 MIN]
4  WHERE {
5   ?product rdfs:label ?label .
6   ?product a bsbm-inst:ProductType84 .
7   ?product bsbm:productFeature bsbm-inst:ProductFeature2569 .
8   ?product bsbm:productFeature bsbm-inst:ProductFeature2587 .
9   ?product bsbm:productPropertyNumeric1 ?value1 .
10  FILTER (?value1 > 469)
11 }
```

Listing 1.2: Modified BSBM explore query 1

[6] www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/

[7] SELECT ?A ?B ?C WHERE {?A ?B ?C}.

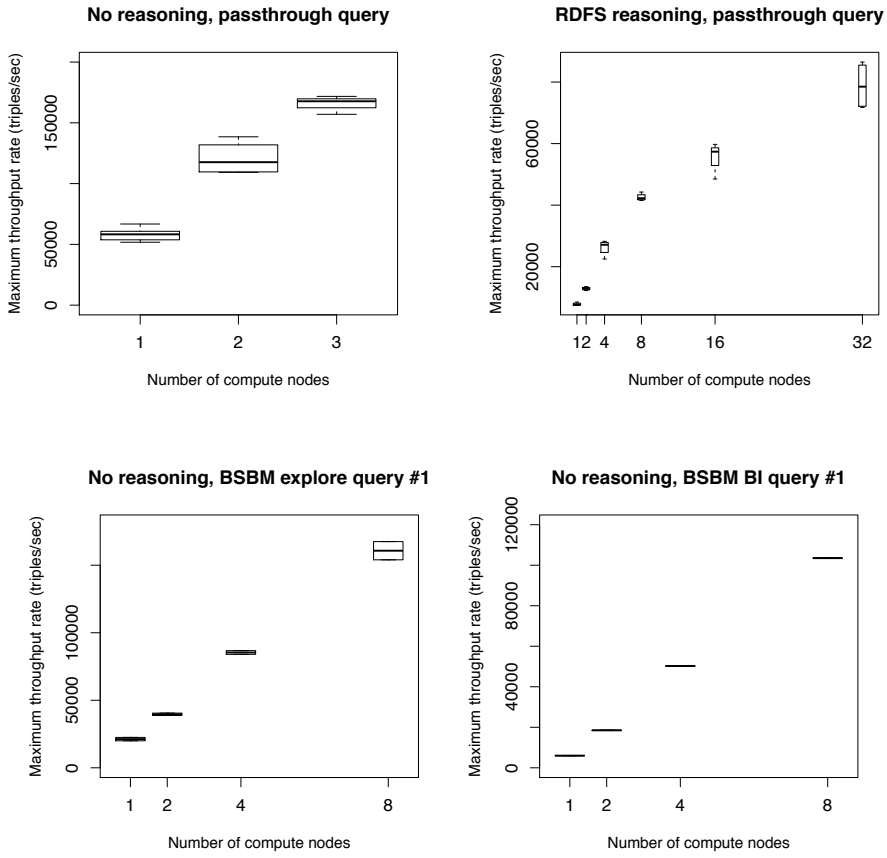

Fig. 4: Maximum query throughput by the number of nodes

No reasoning, BSBM business intelligence query 1 For this experiment, reasoning was disabled and the first query from the BSBM business intelligence query mix was used, which includes aggregation. The ORDER and LIMIT constructs were omitted due to the fact that they are not supported by our streaming system. As seen in Figure 4 (top right), for this query, our system again seems to scale fairly linearly.

8 Discussion and conclusions

In this paper, we have presented a parallel approach for stream reasoning using Yahoo S4. We have presented a series of components and optimizations to calculate the RDFS closure and C-SPARQL queries over a sliding window. Although a comparison is beyond the scope of this paper, our preliminary evaluation shows that our parallel

approach pushes the state-of-the-art to a reasoning throughput of tens or hundreds of thousands triples per second.

There is a series of issues that still need to be considered: reasoning only on the subset of the data that can be considered for the current queries (backwards reasoner), performing query optimisations for the C-SPARQL processor and further evaluation of our method, using a benchmark designed for stream reasoning and more queries.

The work in this paper has been partially funded by the EU project LarKC (FP7-215535).

References

[1] D. 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*, pp. 441–452. ACM, 2010.

[2] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, *et al.* C-sparql: Sparql for continuous querying. In *WWW*, pp. 1061–1062. 2009.

[3] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, *et al.* Incremental reasoning on streams and rich background knowledge. In L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, *et al.*, (eds.) *ESWC (1)*, vol. 6088 of *Lecture Notes in Computer Science*, pp. 1–15. Springer, 2010.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pp. 137–147. 2004.

[5] 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.

[6] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3:158–182, 2005.

[7] S. Kotoulas, E. Oren, and F. van Harmelen. Mind the data skew: distributed inferencing by speeddating in elastic regions. In M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, (eds.) *WWW*, pp. 531–540. ACM, 2010.

[8] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. *Data Mining Workshops, International Conference on*, 0:170–177, 2010.

[9] E. Oren, S. Kotoulas, G. Anadiotis, R. Siebes, *et al.* Marvin: distributed reasoning over large-scale semantic web data. *Journal of Web Semantics*, 2009.

[10] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, *et al.* Owl reasoning with webpie: calculating the closure of 100 billion triples. In *Proceedings of the Seventh European Semantic Web Conference*, LNCS. Springer, 2010.

[11] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen. Scalable distributed reasoning using mapreduce. In *Proceedings of the ISWC '09*. 2009.

Towards Efficient Schema-Enhanced Pattern Matching over RDF Data Streams

Srdjan Komazec and Davide Cerri

Semantic Technology Institute (STI) Innsbruck, University of Innsbruck
Technikerstraße 21a, 6020 Innsbruck, Austria
{srdjan.komazec,davide.cerri}@sti2.at

Abstract. Data streams, often seen as sources of events, have appeared on the Web. Event processing on the Web needs however to cope with the typical openness and heterogeneity of the Web environment. Semantic Web technology, meant to facilitate data integration in an open environment, can help to address heterogeneities across multiple streams. In this paper we discuss an approach towards efficient pattern matching over RDF data streams based on the Rete algorithm, which can be considered as a first building block for event processing on the Web. Our approach focuses on enhancing Rete with knowledge from the RDF schema associated with data streams, so that implicit knowledge can contribute to pattern matching. Moreover, we cover Rete extensions that cope with the streaming nature of the processed data, such as support for temporal operators, time windows, consumption strategies and garbage collection.

Keywords: RDF, data streams, pattern matching, event processing, semantic Web, inference, Rete.

1 Introduction

Data streams are becoming more and more common on the Web. Many streams regarding e.g. stock exchange movements, weather information, sensor readings, or social networking activity notifications are already present, and platforms to collect and share these streams, such as Patchube,[1] have appeared. These data streams can often be seen as sources of events, and the opportunity to efficiently detect event occurrences at the Internet scale while correlating different sources on this *"Web of Events"* could open the doors for new powerful applications.

Event processing is an established computing paradigm that provides approaches and techniques to process event streams and to respond in a timely fashion. The combination of event processing techniques with data streams distributed across the Web comes as a natural fit; event processing on the Web needs however to cope with the typical openness and heterogeneity of the Web environment. Semantic Web technologies are meant to facilitate data integration in an open environment, thus can help to overcome these problems by using machine processable descriptions to resolve heterogeneities across multiple streams.

[1] http://www.patchube.com/

For example, Semantic Sensor Web [11] represents an attempt to collect and process avalanches of data about the world using semantic Web technologies.

While there are a few systems bridging semantic Web technologies and event processing, bringing together these two areas and providing an efficient solution for event processing on the Web is an open research topic. In this paper we present a work-in-progress solution towards efficient pattern matching over RDF data streams, which can be considered as a first building block in this vision.

2 Problem Statement and Related Work

A solution for pattern matching over RDF data streams in the context of event processing on the Web needs to address issues along two different dimensions. First, the presence of RDF schema entailments can materialise RDF statements at runtime, which is not the usual case in event processing systems. Entailed statements, even if they are not explicitly present in the input streams, can contribute to pattern completion. Second, the streaming nature of the data calls for support to express and match patterns taking into account the temporal dimension, and thus for operators that are common for event processing systems but are not part of the usual notion of RDF pattern matching (e.g. in SPARQL).

ETALIS [2] is a Complex Event Processing system providing a number of features such as evaluation of out-of-order events, computation of aggregate functions, dynamic insertion and retraction of patterns, and several garbage collection policies and consumption strategies. To operate on RDF streams ETALIS provides EP-SPARQL, an extension of SPARQL that introduces a number of temporal operators (compliant to ETALIS ones) used to combine RDF graph patterns along time-related dependencies. The presence of background knowledge, in the form of an RDFS ontology, is also supported. EP-SPARQL patterns are evaluated on top of ETALIS, and thus can benefit from all ETALIS features.

C-SPARQL [4] is a SPARQL-based stream reasoning system and language extended with the notions of RDF streams, time windows, handling of multiple streams and aggregation functions. It is suited for cases in which a significant amount of static knowledge needs to be combined with data streams (e.g., coming from heterogeneous and noisy data sensors) in order to enable time-constrained reasoning. The RDF statements from the streams are fed into pre-reasoners performing incremental RDF schema-based materialisation of RDF snapshots, which are further fed into reasoners to which SPARQL queries are submitted [3].

These approaches fall in our problem space, but neither of them provides a complete solution. Although it covers RDF data stream processing, ETALIS does not provide native support for entailments based on RDF schema.[2] On the other hand, C-SPARQL only provides a simple `timestamp()` function to express temporal relationships between RDF patterns, but it does not provide more expressive temporal operators like ETALIS, which can hinder its application in the area of event processing. Since queries are periodically evaluated, C-SPARQL does

[2] ETALIS translates an RDFS ontology into a set of Prolog rules via an external library.

not provide truly continuous querying, and in case of short time windows, high-frequency streams and rich background knowledge, the overhead to compute the incremental materialisation of RDF snapshots becomes significant. ETALIS suffers from performance issues in case of complex pattern expressions.

The goal of our system is to provide efficient pattern matching functionalities on RDF streams, enabling the expression of temporal dependencies and taking into account RDF schema entailments. Unlike C-SPARQL, our system operates over a fixed RDF schema, does not support static background knowledge (besides the schema), and does not provide generic reasoning capabilities. We believe that a fixed schema does not significantly limit the applicability of our solution, since in most applications only data statements come through the streams (e.g., sensor readings). This makes the inclusion of schema-driven entailments simpler, as it can be realised in a pre-running step. The exclusion of background knowledge is due to performance reasons. Our system can be used as the entry block of a bigger system and has performance as a primary concern, whereas more complex operations and semantics are left to subsequent blocks (which can have less stringent performance requirements, as they operate after the first "filtering").

3 Solution

Unlike the aforementioned solutions, our system takes as foundation an efficient pattern matching algorithm: *Rete* [8]. The Rete algorithm, originally designed as a solution for production rule systems, represents a general approach to deal with many pattern/many object situations. The algorithm emphasises on trading memory for performance by building comprehensive memory structures, called α- and β-networks, designated to check respectively intra- and inter-pattern conditions. The algorithm internally preserves intermediate matches in the form of tokens which point to all contributing objects. The intrinsic dataflow nature of Rete goes in favour of using it also over data streams. However, in order to cope with our requirements, the basic Rete algorithm needs to be extended to properly address the issues of RDF schema entailments and data stream processing.

3.1 Support for Schema Entailments in Rete

The RDF semantics specification [10] includes a set of entailment rules to derive new statements from known ones (Table 1[3]). Since in our system the schema is fixed, some rules have no impact at runtime: in particular, rules rdfs5, rdfs6, rdfs8, rdfs10, rdfs11, rdfs12 and rdfs13 have in their bodies only T-box statements, thus cannot be activated by statements in the streams. Rule rdf1 is also not relevant at runtime because, if the property is not in the schema, nothing else (i.e., domain, range, sub/super-properties) can be stated about it, thus no further entailments are possible. Finally, unless we look for resources, rules rdfs4a,

[3] Rules are named as in the specification. We omit rules dealing with blank nodes and literals, since they are not relevant to our discussion.

Table 1. RDF/RDFS entailment rules

Rule name	*If*	*Then add*
rdf1	(x p y)	(p rdf:type rdf:Property)
rdfs2	(p rdfs:domain c),(x p y)	(x rdf:type c)
rdfs3	(p rdfs:range c),(x p y)	(y rdf:type c)
rdfs4a	(x p y)	(x rdf:type rdfs:Resource)
rdfs4b	(x p y)	(y rdf:type rdfs:Resource)
rdfs5	(p rdfs:subPropertyOf q),(q rdfs:subPropertyOf r)	(p rdfs:subPropertyOf r)
rdfs6	(p rdf:type rdf:Property)	(p rdfs:subPropertyOf p)
rdfs7	(p rdfs:subPropertyOf q),(x p y)	(x q y)
rdfs8	(c rdf:type rdfs:Class)	(c rdfs:subClassOf rdfs:Resource)
rdfs9	(c rdfs:subClassOf d),(x rdf:type c)	(x rdf:type d)
rdfs10	(c rdf:type rdfs:Class)	(c rdfs:subClassOf c)
rdfs11	(c rdfs:subClassOf d),(d rdfs:subClassOf e)	(c rdfs:subClassOf e)
rdfs12	(p rdf:type rdfs:ContainerMembershipProperty)	(p rdfs:subPropertyOf rdfs:member)
rdfs13	(x rdf:type rdfs:Datatype)	(x rdfs:subClassOf rdfs:Literal)

rdfs4b and rdfs8 are not relevant, since their output is not used as input by other rules. Therefore, only rules rdfs2, rdfs3, rdfs7 and rdfs9, i.e. inference based on the hierarchies of classes and properties together with their domain and range, need to be considered at runtime. The other rules are relevant only when the extended Rete network is built, based on the schema and the patterns.

A similar discussion can be found in [12], which distinguishes between rules for online and offline computation (the latter used to compute schema closure). Similarly, in our system we can pre-compute schema closure, and use it in building our extended Rete network. Our approach extends the Rete architecture with an additional network of entailment nodes, called *ε-network*, responsible for generating network objects following schema entailments. Given a schema and a set of patterns, the ε-network consists of an optimised set of nodes arranged according to the dataflow paradigm. In contrast to the typical usage of Rete in forward-chaining reasoners, i.e. detecting patterns corresponding to the bodies of entailment rules, our approach encodes in the ε-network schema-driven property hierarchies with specified domain and range definitions connected to class hierarchies. The triples that constitute the schema are not part of the data streams, therefore they are not present as data items in the network and are not used as such in the pattern-matching process. The outputs of the ε-network are connected to the appropriate inputs of the α-network. This approach retains the efficiency of Rete, and fits appropriately with the support for data stream issues.

The ε-network is built as follows. Each property and each class in the schema correspond to a node:[4] for each property p in the schema a node with the triple pattern (? p ?) is created, and for each class c in the schema a node with the triple pattern (? rdf:type c) is created. Nodes are connected by three different types of links: *S-links*, which transfer the subject of the triple to the following class node, *O-links*, which transfer the object of the triple to the following class node (where it becomes the subject), and *SO-links*, which transfer both the subject and the object of the triple to the following property node. For each subclass statement in the schema, an S-link between the corresponding class nodes is

[4] Actually, only the subset of the schema that is relevant for the patterns is considered.

created. For each subproperty statement in the schema, an SO-link between the corresponding property nodes is created. For each domain statement in the schema, an S-link from the corresponding property node to the corresponding class node is created. For each range statement in the schema, an O-link from the corresponding property node to the corresponding class node is created. A simple example of ε-network and the corresponding ontology fragment are shown in Figure 1.

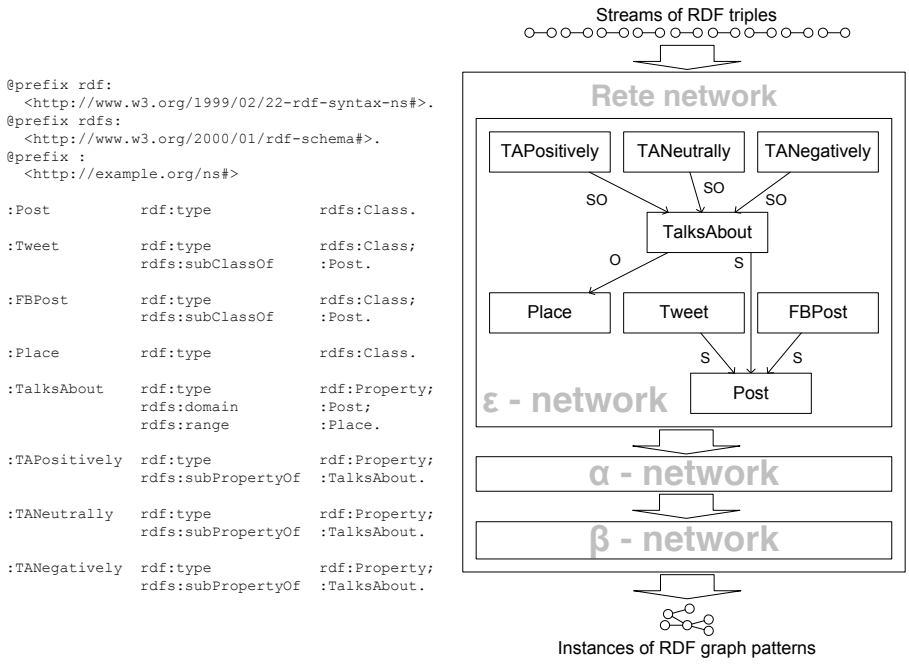

Fig. 1. Extended Rete with ε-network

When a triple enters the ε-network, it activates the node with the corresponding pattern (if it exists), in the same way as in the α-network in basic Rete. When a node is activated, it emits the corresponding triple to the α-network and it transfers the subject and/or the object of the triple (depending on the link type) to the following nodes, activating them. The activation mechanism is governed by tokens (as in basic Rete), which allows detecting multiple inference paths, in order to avoid passing the same inferred triple to the α-network more than once.

3.2 Support for Data Stream Issues in Rete

Our solution intends to incorporate and adapt the following stream processing extensions to Rete: temporal operators in RDF patterns, time windows, and data consumption strategies coupled with appropriate garbage collection approaches.

An event pattern specification language usually introduces a number of operators to build comprehensive pattern expressions. The operators can be divided into *logical operators* (conjunction, disjunction and negation), *data operators* (comparison and arithmetic), and *temporal operators*. Basic Rete provides only support for conjunctions between objects in a pattern. Support for the remaining logical and data operators has been introduced soon after the algorithm [9, 7], and our system follows the same design and implementation guidelines. Temporal operators allow to express time-dependent constraints over a set of objects in a pattern. Our system treats pattern occurrences in accordance to interval-based semantics [1], in which an occurrence spans a time interval (unlike point-based semantics in which the last contributing object defines the time of the pattern occurrence). Motivated by [5] and [14], we consider to implement the temporal operator semantics through extensions of join-nodes behaviour. The extensions allow nodes to perform interval-based comparisons to establish a temporal relationship between the joining patterns (i.e., partial graph pattern matches) by executing basic timepoint and time interval arithmetics. In order to perform the join, the node further detects if the established relationship is in accordance to the one designated by the pattern.

Rete implementations usually provide no time-window support. An initial work [13] proposes a method to automatically discard objects no longer needed by computing temporal dependency matrices over the Rete network tokens. The work in [16, 15] gives an implementation that adds new time-aware Rete nodes checking time-window boundaries during the matching process, based on temporal token dependency computations. Our system follows a similar approach by extending β-node behaviour to include time-window boundary checking. Tokens falling out of the time-window boundaries are excluded from further matching results and flagged as candidates for garbage collection. The checks are performed in the same way as for temporal operators (i.e., by establishing a relationship between the intervals, where one interval defines the time window boundaries).

Event consumption strategies (also called parameter contexts) [6, 1] resolve the issue of multiple simultaneous matches to a pattern inside of a time window. Rete implementations usually lack support for these strategies; we will consider further extensions to the join-node behaviour to enable their support. The issue of garbage collection is closely related. In [14] it is proposed to compute an object lifetime during which the object must be retained due to the participation in partial matches inside the time window. Our system will consider the same approach by implementing an incremental garbage collection process using the lifetime information to schedule deletion of objects.

4 Conclusions and Further Work

An efficient mechanism for schema-enhanced pattern detection on RDF data streams is required to address the challenges of event processing on the Web. In this paper we proposed a system based on the Rete algorithm, extended to include statements materialised through schema entailments and to support the

temporal nature of data streams. We consider several dimensions as our further work. First, more precise answers to the issues of how to integrate the chosen temporal and entailment approaches into Rete will need to be defined. Second, special attention will be devoted to the analysis of mutual influence of temporal and entailment extensions. Third, performance evaluation will be conducted in order to ensure the overall system responsiveness under different conditions.

Acknowledgement. This work has been supported by the European Commission through the SUPPORT project (EU FP7 Project 242112).

References

1. Adaikkalavan, R., Chakravarthy, S.: SnoopIB: Interval-Based Event Specification and Detection for Active Databases. Data Knowl. Eng. 59, 139–165 (October 2006)
2. Anicic, D., Fodor, P., Rudolph, S., Stühmer, R., Stojanovic, N., Studer, R.: ETALIS: Rule-Based Reasoning in Event Processing. In: Reasoning in Event-Based Distributed Systems, Studies in Computational Intelligence, vol. 347, pp. 99–124. Springer (2011)
3. Barbieri, D., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Incremental Reasoning on Streams and Rich Background Knowledge. In: The Semantic Web: Research and Applications, Proc. of 7th Extended Semantic Web Conference (ESWC 2010). LNCS, vol. 6088, pp. 1–15. Springer (2010)
4. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: a Continuous Query Language for RDF Data Streams. Int. J. Semantic Computing 4(1), 3–25 (2010)
5. Berstel, B.: Extending the RETE Algorithm for Event Management. In: Proc. of 9th Int. Symp. on Temporal Representation and Reasoning (TIME'02). pp. 49–51. IEEE Computer Society (2002)
6. Chakravarthy, S., Krishnaprasad, V., Anwar, E., Kim, S.K.: Composite Events for Active Databases: Semantics, Contexts and Detection. In: Proc. of 20th Int. Conf. on Very Large Data Bases. pp. 606–617. VLDB '94, Morgan Kaufmann (1994)
7. Doorenbos, R.J.: Production Matching for Large Learning Systems. Ph.D. thesis, Carnegie Mellon University, Pittsburg, PA (1995)
8. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence 19, 17–37 (1982)
9. Grady, C., Highland, F., Iwaskiw, C., Pfeifer, M.: System and Method For Building a Computer-Based RETE Pattern Matching Network. Tech. rep., IBM Corp., Armonk, N.Y. (1994)
10. Hayes, P.: RDF Semantics. W3C recommendation, W3C (Feb 2004)
11. Sheth, A., Henson, C., Sahoo, S.S.: Semantic Sensor Web. IEEE Internet Computing 12(4), 78–83 (2008)
12. Stuckenschmidt, H., Broekstra, J.: Time-Space Trade-offs in Scaling up RDF Schema Reasoning. In: WISE Workshops. LNCS, vol. 3807, pp. 172–181. Springer (2005)
13. Teodosiu, D., Pollak, G., Souvenirs, A.M., Piaf, E.: Discarding Unused Temporal Information in a Production System. In: Proc. of ISMM Int. Conf. on Information and Knowledge Management. pp. 177–184. CIKM-92 (1992)

14. Walzer, K., Breddin, T., Groch, M.: Relative temporal constraints in the Rete algorithm for complex event detection. In: Proc. of 2nd Int. Conf. on Distributed Event-Based Systems. pp. 147–155. DEBS '08, ACM (2008)
15. Walzer, K., Groch, M., Breddin, T.: Time to the Rescue – Supporting Temporal Reasoning in the Rete Algorithm for Complex Event Processing. In: Proc. of 19th Int. Conf. on Database and Expert Systems Applications. pp. 635–642. DEXA '08, Springer-Verlag (2008)
16. Walzer, K., Schill, A., Löser, A.: Temporal constraints for rule-based event processing. In: Proc. of ACM 1st Ph.D. Workshop in CIKM. pp. 93–100. PIKM '07, ACM (2007)