Sparkwave: Continuous Schema-Enhanced Pattern Matching over RDF Data Streams

Srdjan Komazec
srdjan.komazec@sti2.at

Davide Cerri
davide.cerri@sti2.at

Semantic Technology Institute Innsbruck
University of Innsbruck
Technikerstraße 21a
6020 Innsbruck, Austria

ABSTRACT

Data streams, often seen as sources of events, have appeared on the Web. Stream processing on the Web needs however to cope with the typical openness and heterogeneity of the Web environment. Semantic Web technologies, meant to facilitate data integration in an open environment, can help to address heterogeneities across multiple streams. In this paper we present Sparkwave, an approach for continuous pattern matching over RDF data streams. Sparkwave is based on the Rete algorithm, which allows efficient and truly continuous processing of data streams. Sparkwave is able to leverage RDF schema information associated to data streams to compute entailments, so that implicit knowledge is taken into account for pattern matching. In addition, it further extends Rete to support time-based sliding windows and static data instances, to cope with the streaming nature of processed data and real-world use cases.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information filtering, Selection process; H.2.4 [**Systems**]: Query processing

General Terms

Algorithms, Performance

Keywords

RDF, stream processing, stream reasoning, pattern matching, 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 Cosm,[1] have appeared. Techniques to process data streams while responding in a timely fashion are also following the trend. The most prominent areas in this respect are Complex Event Processing (CEP) [22] and Data Stream Management Systems (e.g., Aurora [1] and STREAM [6]).

The combination of data stream processing techniques with data streams distributed across the Web comes as a natural fit; data stream 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 open environments, thus can help to overcome these problems by using machine processable descriptions to resolve heterogeneities across multiple streams. For example, Semantic Sensor Web [25] represents an attempt to collect and process avalanches of sensor data using semantic Web technologies.

The application of semantic Web technologies to data stream processing opens also an opportunity to perform reasoning tasks over continuously and rapidly changing information, which was a trigger for the emergence of the stream reasoning research area [13]. Stream reasoning systems aim at preserving the core value of data stream processing, i.e. processing streaming data in a timely fashion, while providing a number of features: support for expressive queries/-patterns and complex schemas, integration of static background knowledge with streaming data, support for temporal operators, time-based windows and various consumption strategies. In addition, stream reasoning systems need to take into account entailed knowledge, which results in higher complexity and performance penalties.

In this paper we present Sparkwave, a solution for continuous schema-enhanced pattern matching over RDF data streams. The aim of Sparkwave is to achieve and retain high-throughput RDF graph pattern matching while providing a number of stream reasoning features such as support for fairly expressive pattern definitions, time-based sliding windows and schema-entailed knowledge. Having performance as a primary goal, Sparkwave includes limited support for background knowledge (schema and static data instances) and supports only a limited set of schema constructs; it is therefore complementary to other solutions which offer such functionalities but in the context of less stringent performance requirements.

The rest of this paper is structured as follows. In Section 2

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DEBS '12, July 16–20, 2012, Berlin, Germany.
Copyright 2012 ACM 978-1-4503-1315-5 ...$15.00.

[1]http://cosm.com/

we give an overview of the related work. In Section 3 we first define basic notions needed to establish the environment for RDF stream processing, and then we focus on Sparkwave, describing its architecture and in particular the support for schema-based entailments, time-based sliding windows, and static instances. The section is concluded with a comprehensive example. In Section 4 we present evaluation results. Finally, in Section 5 we outline future work directions, and in Section 6 we conclude the paper.

2. RELATED WORK

A solution for pattern matching over RDF data streams needs to address issues along at least two different dimensions. First, the presence of RDF schema entailments can materialise RDF statements at runtime, which is not the case in conventional data stream 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 Complex Event Processing systems but are not part of the usual notion of RDF pattern matching (e.g. in SPARQL). A number of languages extending SPARQL with temporal constructs have already been proposed (e.g., τ-SPARQL [27], SPARQL-ST [23] and Streaming SPARQL [10]).

A first RDF stream processing engine operating on top of SPARQL has been reported in [18]. The engine evaluates graphs compliant to the proposed streaming SPARQL algebra in a dataflow way. The graphs, produced as result of SPARQL query transformation into the algebra, intertwine operators which perform intra-triple (e.g., triple pattern operator) and inter-triple (e.g., join, filter, and optional operators) checks. The engine also includes a framework for logical and physical optimisations (e.g., join operator reordering, pushed upward filters, and hash-based selection of triple pattern operators), however the proposed algebra lacks typical stream processing features such as time-based windows.

ETALIS [5] 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 [4], 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. Static 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 the rich set of ETALIS features. However, ETALIS does not provide native support for entailments based on RDF schema, which may hinder its applicability in case of complex pattern expressions and high frequency streams due to performance reasons.[2]

C-SPARQL [8] 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 background knowledge needs to be combined with data streams (e.g., coming from heterogeneous 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 [7]. In terms of temporal operators, C-SPARQL provides a simple `timestamp()` function to express temporal relationships between RDF patterns. Since queries are periodically evaluated, C-SPARQL does 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 can become significant.

CQELS [21] represents a recent solution for RDF stream processing built on top of the notion of Linked Stream Data [24]. The solution offers a native way to interpret and implement common stream processing features (time window operator, relational database-like join and union operators, and stream generation operator) in an RDF data stream processing environment. In addition, CQELS is equipped with a flexible query execution framework capable of dynamically adapting to changes in input data (e.g., operator reordering to improve query execution).

3. SPARKWAVE

Sparkwave is a solution to perform continuous schema-enhanced pattern matching over RDF data streams. More precisely, the goal of Sparkwave is to provide efficient pattern matching functionalities on RDF streams in a truly continuous way, enabling the expression of temporal constraints in the form of time windows and taking into account RDF schema entailments. Unlike other existing solutions, Sparkwave meets this goal by taking as a foundation an efficient pattern matching algorithm: *Rete* [15]. 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 over a set of objects.

The intrinsic dataflow nature of Rete goes in favour of using it also over data streams. However, in order to meet the goal, the basic Rete algorithm needs to be extended to properly address RDF schema entailments and the temporal requirements of data stream processing. Sparkwave resolves the first issue by extending Rete with an additional network called *ε-network*. The ε-network is positioned in front of the normal Rete network and is responsible for generating triples following schema entailments. The ε-network nodes are connected to the appropriate α-network nodes in a dataflow style. Regarding the second issue, Sparkwave provides support for time windows. This is realised through an extension of the functionalities of β-network nodes, which are responsible for checking if partial or complete pattern matches fall into the scope of the designated time window. An high-level overview of Sparkwave architecture is presented in Figure 1.

In contrast to more generic stream reasoning solutions, Sparkwave operates over a fixed RDF schema, provides limited support for static data instances (besides the schema), and provides limited reasoning functionalities to support pattern matching. We believe that a fixed schema does not significantly limit the applicability of our solution, since in

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

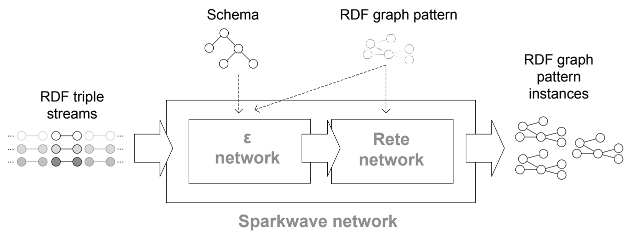


Figure 1: Sparkwave architecture

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 limitations on background knowledge and schema expressivity come from architectural choices motivated by performance reasons. Since Sparkwave is a memory-intensive solution, large amounts of static data instances that should be constantly kept in memory would hinder performance with respect to processing streamed data instances. The constraint over schema expressivity (i.e. RDFS constructs plus inverse and symmetric properties – see Section 3.3) are also not a significant limiting factor: a recent survey [16] of used RDF(S)/OWL features, performed over a large corpus of crawled RDF documents, shows that the top six features are those that form the core of RDFS, whereas sophisticated OWL features are rarely used.

In cases where a large amount of background knowledge (e.g. an external knowledge base) or complex reasoning capabilities are needed, Sparkwave can still be useful. Sparkwave can indeed be used as the entry block of a larger system, where more complex operations and semantics, possibly depending on static background knowledge, can be performed by subsequent blocks which can have less stringent performance requirements, benefiting from the fact that they operate on streams that have already been "filtered" by Sparkwave.

3.1 RDF Data Streams and Graph Patterns

Sparkwave considers a *time instant* τ as a value taken from a set of discrete equidistant values. A *time interval* T is defined as a pair of time instants (τ_s, τ_e) for which it holds that $\tau_s < \tau_e$.

A *triple* t is a tuple $\langle s, p, o\rangle$ consisting of subject s, predicate p, and object o, as defined in [20]. A *streamed triple* st is a pair (t, τ) consisting of a triple t and a time instant τ. The time instant τ plays the role of a time-stamp and its value is implicitly assigned when the triple enters Sparkwave. An *RDF data stream* ST is an unbounded sequence of streamed triples st_i $(1 \leq i \leq n)$. The sequence is ordered over the monotonically non-decreasing time-stamp value τ_i, where $\tau_i \leq \tau_{i+1}$:

$$\begin{gathered} \dots \\ (\langle s_i, p_i, o_i\rangle, \tau_i) \\ (\langle s_{i+1}, p_{i+1}, o_{i+1}\rangle, \tau_{i+1}) \\ \dots \end{gathered}$$

A *triple pattern tp* is a triple which may have a variable declaration for any of the three stream triple constituents, e.g., $\langle ?x, a, ?y\rangle$. A triple fulfils a triple pattern if all defined triple pattern values are equal to the corresponding triple values. A *graph pattern* G_p is a conjunction of triple patterns tp_i $(1 \leq i \leq n)$, as in the following example:

$$\langle ?x,\ a,\ b\rangle \wedge \langle ?x,\ c,\ ?y\rangle \wedge \langle ?y,\ m,\ n\rangle$$

This graph pattern consists of three triple patterns. The first triple pattern matches all triples which have in the predicate position the value a and in the object position the value b, while the value in the subject position is left unbound, i.e., it is declared as the variable $?x$. The graph pattern forms a conjunction in which every repeated declaration of a variable is a join over the respective streamed triples, in which variable values must be equal. For example, streamed triples fulfilling the first and the second triple pattern can be joined only if the values in the subject positions (declared with the shared variable $?x$) are identical.

A *graph* G is a set of streamed triples st_k $(1 \leq k \leq n)$. A graph G fulfils a graph pattern G_p if for every $tp_k \in G_p$ there exists $st_m \in G$ which fulfils tp_k and viceversa.

A set of functions can be applied over a graph G. Function $\tau^G_{start}(G)$ returns the time instant value of the earliest streamed triple contributing to the graph G, while function $\tau^G_{end}(G)$ returns the time instant value of the latest streamed triple contributing to the graph G_i. Function $T^G(G)$ returns the time interval (τ_s, τ_e) associated to the graph G:

$$T^G(G) = (\tau^G_{start}(G), \tau^G_{end}(G))$$

For the RDF data stream given in Listing 1(a) and the graph pattern shown above, all possible graph instances are given in Listing 1(b).

$$\begin{array}{l} (\langle t11, a, b\rangle, 10) \\ (\langle t21, m, n\rangle, 20) \\ (\langle t12, a, b\rangle, 40) \\ (\langle t11, c, t21\rangle, 60) \\ (\langle t22, m, n\rangle, 60) \\ (\langle t21, l, o\rangle, 70) \\ (\langle t12, c, t22\rangle, 80) \\ (\langle t22, l, o\rangle, 90) \\ (\langle t31, p, q\rangle, 120) \\ (\langle t32, p, q\rangle, 170) \end{array} \qquad \begin{array}{l} G_1 = \{(\langle t11, a, b\rangle, 10), \\ \qquad (\langle t11, c, t21\rangle, 60), \\ \qquad (\langle t21, m, n\rangle, 20)\} \\ \\ G_2 = \{(\langle t12, a, b\rangle, 40), \\ \qquad (\langle t12, c, t22\rangle, 80), \\ \qquad (\langle t22, m, n\rangle, 60)\} \\ \\ \quad T(G_1) = (10, 60) \\ \quad T(G_2) = (40, 80) \end{array}$$

(a) (b)

Listing 1: An RDF data stream and pattern matches

3.2 Rete Network

As mentioned at the beginning of Section 3, Sparkwave architecture is based on the Rete algorithm, which efficiently performs pattern matching over a set of objects. In the case of Sparkwave, objects are streamed triples $st_i \in ST$. An example of a Rete network built on top of the example graph pattern shown in Section 3.1 is given in Figure 2.

The algorithm performs discrimination of streamed triples through the α-network. In Figure 2, the Rete network has five α-network nodes checking whether a specific value can be found in a streamed triple in a specific position (e.g., whether predicate value is equal to a, and object value is equal to b). Each streamed triple compliant to the tests is stored in the associated α-memory αM_x. The algorithm preserves intermediate matches in β-network in the form of tokens. A token k is a pair (k_{parent}, st) which refers to a parent token k_{parent} and a streamed triple st stored in

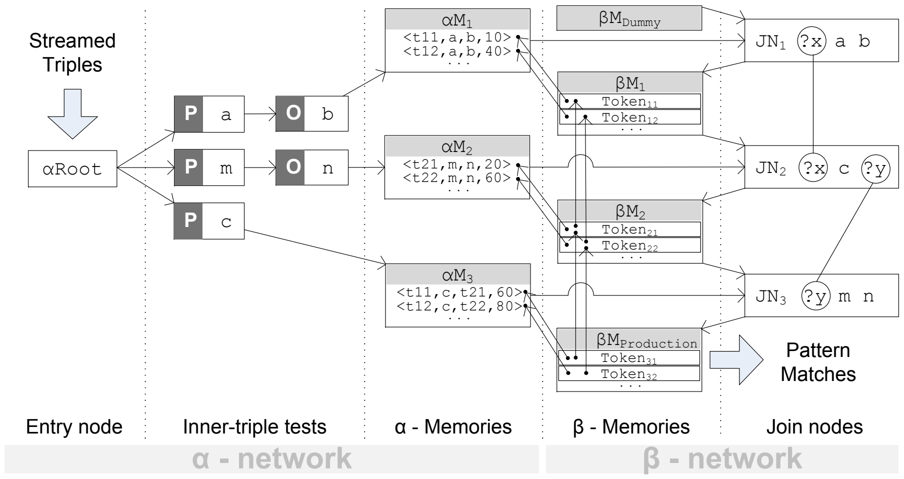


Figure 2: An example of Rete network

an α-memory αM. Each partial match is thus represented as a linked list of tokens. Since a token at the same time represents a graph G, we can define the functions $\tau^k_{start}(k)$, $\tau^k_{end}(k)$, and $T^k(k)$ in the same way as $\tau^G_{start}(G)$, $\tau^G_{end}(G)$, and $T^G(G)$ (see Section 3.1).

In the example given in Figure 2, β-memory βM_2 holds two tokens, $Token_{21}$ and $Token_{22}$, which point to the corresponding streamed triples and parent tokens. Whenever a new streamed triple appears in α-memory, a join node JN_x will be activated to examine the possibility to enlarge the currently holding partial matches, i.e., to produce new tokens. The enlargement of a partial match is possible if the new streamed triple passes the join node tests. The tests evaluate the possibility of inter-triple variable bindings. For example, the join node JN_2 checks if a newly streamed triple can be joined with the streamed triple pointed by a token in βM_1 over equality of values in the subject position. If such possibility exists, a new token is created and placed in βM_2.

Tests in join nodes must be performed over all respective elements in α- and β-memories possibly connected over inter-triple variable bindings. In the case of high-frequency streams and large time windows, the corresponding α- and β-memories may hold large numbers (e.g., tens of thousands) of triples and associated tokens. In such a realm, a naive implementation based on a nested loop join algorithm[3] would certainly hinder the performance goals that Sparkwave strives to meet. Therefore, Sparkwave implementation is based on a hash join algorithm[4] in which α- and β-memories hold hash tables in which hash values are computed over triple values corresponding to inter-triple variable bindings. Since join node tests are usually extensively repeated, the computational price paid to compute hash values is minor with respect to the price of selecting appropriate triples and tokens.

3.3 Entailments (ε-Network)

The ε-network is responsible for streamed triples following schema entailments, so that α- and β-networks can find matches to patterns taking into account implicit knowledge.

[3]http://en.wikipedia.org/wiki/Nested_loop_join
[4]http://en.wikipedia.org/wiki/Hash_join

The RDF semantics specification [19] includes a set of entailment rules to derive new statements from known ones (see Table 1[5]). Since in Sparkwave 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 p 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 (i.e. using rdfs:Resource as type), rules rdfs4a, rdfs4b and rdfs8 are not relevant, since their output is not used as input by any other rule. 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 ε-network is built, based on the schema and the patterns.

In addition to RDFS entailments, Sparkwave supports a few constructs from OWL that fit in the approach and can be useful in adding more expressivity to schemas. In particular, Sparkwave supports owl:inverseOf and owl:SymmetricProperty, through the entailment rules shown in Table 2. Rule inv1 is again schema-only, whereas rules inv2 and sym are relevant at runtime.

A similar discussion (limited to RDFS rules) can be found in [26], which distinguishes between rules for online and offline computation, with the latter used to compute schema closure. Similarly, in Sparkwave we can pre-compute schema closure, and use it in building the ε-network. 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. Working on

[5]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⟩

Table 2: Extra entailment rules from OWL

Rule name	*If*	*Then add*
inv1	⟨p owl:inverseOf q⟩	⟨q owl:inverseOf p⟩
inv2	⟨p owl:inverseOf q⟩ ⟨x p y⟩	⟨y q x⟩
sym	⟨p rdf:type owl:SymmetricProperty⟩ ⟨x p y⟩	⟨y p x⟩

a fixed schema, and limiting the supported entailment rules to the listed ones, which are always activated by a single triple from the stream, allow the ε-network to be stateless (i.e., no state is kept between subsequent triples entering the ε-network).

Given the schema, the ε-network is built as follows. First, a node is created for each property and each class in the schema:[6]

- for each *property* p in the schema, a node with the triple pattern ⟨? p ?⟩ is created (property node);
- for each *class* c in the schema, a node with the triple pattern ⟨? rdf:type c⟩ is created (class node).

Nodes can be connected by four 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);
- *SO-links*, which transfer both the subject and the object of the triple to the following property node;
- *OS-links*, which transfer both the subject and the object of the triple to the following property node swapping them (i.e., subject becomes object and vice versa).

Links between nodes in the ε-network are created as follows:

- for each *subclass* statement in the schema, an S-link between the corresponding class nodes is created (from the subclass to the superclass);
- for each *subproperty* statement in the schema, an SO-link between the corresponding property nodes is created (from the subproperty to the superproperty);
- 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;
- for each *inverse property* statement in the schema, an OS-link between the corresponding property nodes is created (so that two nodes representing a pair of inverse properties p and q are connected by a pair of links, one from p to q and one from q to p);
- for each *symmetric property* statement in the schema, an OS-link is created from the corresponding property node to itself.

An example of how to build the ε-network given a schema is presented in Section 3.8.

3.4 Schema Pre-Processing

Conceptually, an existing ontology that describes data instances in the streams could be directly used to build the ε-network. However, such ontology could contain a significant amount of axioms of no interest with respect to the patterns. Directly using that ontology would thus be inefficient, because a large amount of unneeded entailed triples could be generated, wasting processing time and memory both in the ε- and α-network. For this reason, Sparkwave performs a pre-processing step of the given schema before using it to build the ε-network. The reduced schema resulting from the pre-processing step is a subset of the original one, and depends on the patterns.

Only class and property definitions for classes and properties relevant to the given patterns are kept in the reduced schema. Relevant classes are all classes present in the patterns and their subclasses, whereas relevant properties are all properties present in the patterns and their subproperties, all properties whose domain or range is a relevant class and their subproperties, all properties which are inverse of a relevant property and their subproperties. A further effect of the pre-processing stage is that the class and property hierarchies in the reduced schema are "flattened": for example, if the original schema contains three classes c, d and e, where ⟨c rdfs:subClassOf d⟩ and ⟨d rdfs:subClassOf e⟩, but

[6]Actually, only the subset of schema that is relevant for the given patterns is considered, as explained in Section 3.4.

only class e is present in the patterns, there is no interest in inferring that something of type c has also type d. Therefore, this schema would be reduced to ⟨c rdfs:subClassOf e⟩ and ⟨d rdfs:subClassOf e⟩, "flattening" all subclasses of e to the same level.

3.5 Time Window

One of the crucial temporal aspects of stream processing is the support for time windows. A time window partitions an RDF stream in the context of time, thus reducing the number of triples over which pattern matching is performed, which is compliant to the nature of typical applications and limited computing resources. Keeping a windowed triple stream up-to-date (and deallocating resources occupied by staled triples) comes with additional processing penalties. In Sparkwave additional overhead is experienced due to the existence of the implicit knowledge derived by the ε-network.

A way to extend the Rete algorithm with time window support is described in [29]. This solution proposes an extension of β-network nodes, i.e. join nodes, with features for window evaluation. The extended nodes are continuously tracking window boundaries based on the latest injected events, on top of which event filtering is performed. The nodes are closely collaborating with the garbage collection thread, which tracks single event timeouts and initiates α- and β-network cleanups in a callback fashion.

Sparkwave implements a similar approach to perform time window checks as β-network nodes extensions, but in contrast to the reported solution Sparkwave continuously and incrementally updates the time interval value $T^k(k)$ of a partially fulfilled match represented by a token k, which is kept in the token itself. With this approach, time window boundaries are token-specific rather than join-node-specific. For this reason, the same join node can be potentially used by multiple patterns having different time window specifications, therefore different patterns can benefit from each other's partial matches. In addition, the interaction between the main Sparkwave components, i.e., the main Rete computation and garbage collection procedure (see Section 3.7), is simplified, thus enabling straightforward and efficient clean-up operations.

Since the time interval value of the token $T^k(k)$ is extensively considered during join node tests, the solution adopted by Sparkwave reduces the effort needed to compute it. The time interval value of a new token $T^k(k_{new})$ is computed on

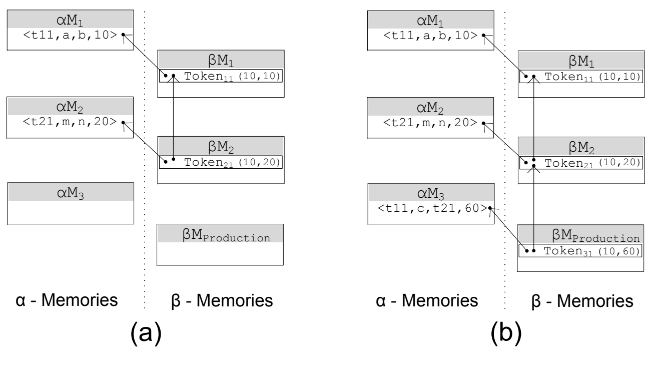


Figure 3: Incremental computation of time window

top of the time interval value of the parent token $T^k(k)$ and the time-stamp value τ of the streamed triple $st = (t, \tau)$:

$$T^k(k_{new}) = \begin{cases} (\tau, \tau^k_{end}(k)), & \tau < \tau^k_{start}(k) \\ (\tau^k_{start}(k), \tau^k_{end}(k)), & \tau^k_{start}(k) \le \tau \le \tau^k_{end}(k) \\ (\tau^k_{start}(k), \tau), & \tau > \tau^k_{end}(k) \end{cases}$$

The token k_{new} can be built only if its time interval value still falls inside of a defined time window. Figure 3 shows an example of an incremental update of a partial match which happens when a new streamed triple arrives at αM_3. In example (a) the current partial match is represented by $Token_{21}$, which spans over the time interval $(10, 20)$. In example (b) the incremented partial match represented by $Token_{31}$ includes the new streamed triple, and spans over the time interval $(10, 60)$.

3.6 Static Data Instances

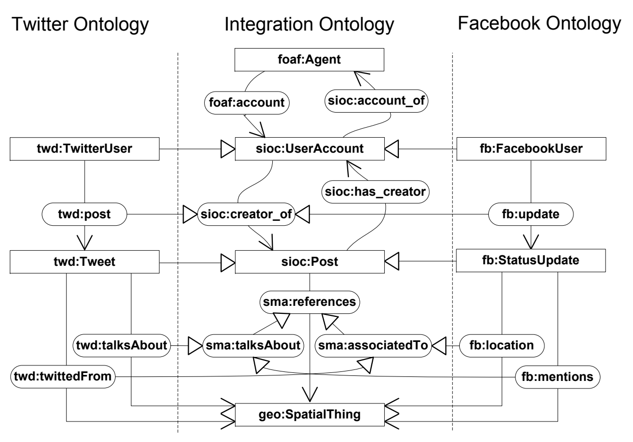
Sparkwave focuses on pattern matching on streams and it is not intended to be directly used in cases in which the data in the stream need to be joined with those in a large knowledge base. As already stated, in such cases Sparkwave can be used to pre-filter the streams, and it is complementary to other solutions which are designed to deal with large knowledge bases (e.g., see C-SPARQL [7]). In many cases, however, a fairly limited amount of background knowledge, in the form of static data instances, can be considered to be part of the domain definition together with the schema, and needs to be joined at runtime with the dynamic data coming from the streams. For example, in an application which processes streams of data regarding the status of parking spaces in several car parks, the definition of the car parks could constitute such static knowledge and be useful in defining the patterns. Sparkwave therefore provides support for static data by allowing the injection of static triples into the network before stream processing begins.

Different data nature calls for a different treatment of static triples in Sparkwave. Like streamed triples, static triples could also initiate entailment of implicit knowledge. Therefore, they need to pass through the ε-network and activate the subsequent α-network nodes. In contrast to streamed triples, static triples need to be continuously present in α-memory nodes, to be joined with upcoming stream triples and form tokens.

Sparkwave loads static triples in the network after building inner structures (ε-, α- and β-network nodes) but before processing streamed triples. Since static triples have unconstrained life-time, the garbage collector ignores their presence, and join nodes do not take them into account in computing time interval values of tokens.

3.7 Garbage Collection

The issue of garbage collection is closely related to the time-based window support. The garbage collector responsibility is to clear ε-, α- and β-network nodes of staled streamed triples and dependent tokens. The garbage collection algorithm is shown in Algorithm 1. As opposed to the solution presented in [29], which is based on timers run by a separate garbage collection thread triggering α- and β-network clean-up in a callback fashion, in Sparkwave the garbage collector runs as a part of a single Rete computational thread. This solution simplifies the internal Rete data structures since there is no need for thread synchronisation



(a) An example of social media analytics ontology

```
REGISTER QUERY MultiplePlaceNotification AS
PREFIX twd: <http://twitter.example.org/ns#>.
PREFIX sma: <http://socialmedia.example.org/ns#>.
PREFIX sioc: <http://rdfs.org/sioc/ns#>.
PREFIX fb: <http://facebook.example.org/ns#>.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
SELECT ?tweet ?fbpost ?user
FROM STREAM <http://facebook.example.org/update/stream>
     [RANGE 5m]
FROM STREAM <http://twitter.example.org/update/stream>
     [RANGE 5m]
FROM <http://foaf.example.org/friends.rdf>
WHERE {
  ?tweet rdf:type twd:Tweet.
  ?tweet sma:talksAbout ?place.
  ?twitteraccount sioc:creator_of ?tweet.
  ?twitteraccount sioc:account_of ?user.
  ?fbpost rdf:type fb:StatusUpdate.
  ?fbpost sma:talksAbout ?place.
  ?fbaccount sioc:creator_of ?fbpost.
  ?fbaccount sioc:account_of ?user.
}
```

(b) An RDF graph pattern in a form of a C-SPARQL query

```
          .
          .
fb:JohnDoe fb:update fb:post123
fb:post123 fb:mentions geo:2775220
twd:johnd twd:post twd:tweet456
twd:tweet456 twd:talksAbout geo:2775220
          .
          .
```

(c) Streamed triples

```
twd:tweet456 rdf:type twd:Tweet
twd:tweet456 sma:talksAbout geo:2775220
twd:johnd sioc:creator_of twd:tweet456
fb:post123 rdf:type fb:StatusUpdate
fb:post123 sma:talksAbout geo:2775220
fb:JohnDoe sioc:creator_of fb:post123
```

(d) Inferred triples

```
fb:JohnDoe sioc:account_of foaf:JohnDoe
twd:johnd sioc:account_of foaf:JohnDoe
```

(e) Static triples

Figure 4: Example ontology, C-SPARQL-like query and examples of streamed, inferred and static triples

and deadlock handling, which further contributes to more efficient processing.

As shown in Algorithm 1, the garbage collection procedure first removes from the ε-network all tokens connected to already processed triples. Subsequently, the algorithm computes the *threshold* timepoint, serving as a decision point about staled triples. All such triples and associated tokens are removed from α- and and β-memories in case they are not part of the static data instances.

Algorithm 1 Garbage collection

```
for $triple \leftarrow epsilon.processedTriples$ do
    for $token \leftarrow triple.tokens$ do
        $token.removeTokenFromNode()$
    end for
    $epsilon.removeTokens(triple)$
end for
$threshold \leftarrow currentTime - timeWindow$
for $alphaMemory \leftarrow rete.alphaMemories$ do
    for $triple \leftarrow alphaMemory.triples$ do
        if $triple.timestamp < threshold$ then
            if $triple \notin staticTriples$ then
                triple.remove()
            end if
        end if
    end for
end for
```

3.8 Example

In this section we discuss a complete Sparkwave network instance based on the social media analytics example presented in [11]. Figure 4 shows a social media analytics ontology (a) and an RDF graph pattern in the form of a C-SPARQL query (b). Grounded in established ontologies (FOAF[7] and SIOC[8]), this ontology presents a possible way to integrate streams of data coming from different social networking platforms (in our example Facebook and Twitter). The pattern aims at detecting when the same person talks about the same place on both channels. It is worth noting that the pattern is written mostly by relying on the concepts which integrate data streams from different channels. Since streamed data annotations are channel-specific, Sparkwave schema-entailment capabilities are necessary for pattern matching. Examples of streamed (c), inferred (d) and static (e) triples are also shown in Figure 4.

Figure 5 shows two versions of the ε-network: the full network (a) and the corresponding reduced network (b). The full network is built taking all schema definitions from the ontology, without taking the pattern into account. In contrast, the schema pre-processing step (as described in Section 3.4) results in a significantly simpler network structure which holds only the nodes and links that are relevant and can contribute to the completion of the given RDF graph pattern. For example, the reduced network has two class nodes instead of eight, and four S-links instead of fifteen. Depending on the complexity of the given schema and pattern, the pre-processing step can significantly reduce the complexity of the ε-network.

Finally, Figure 6 shows an instance of the Rete network according to the RDF graph pattern. The α-network nodes are responsible for checking intra-triple conditions (e.g., whether a triple has as predicate rdf:type and as object fb:StatusUpdate) and for storing the triples passing the tests in the corresponding αM_n arrays. When a new triple appears in an

[7]http://www.foaf-project.org/
[8]http://sioc-project.org/

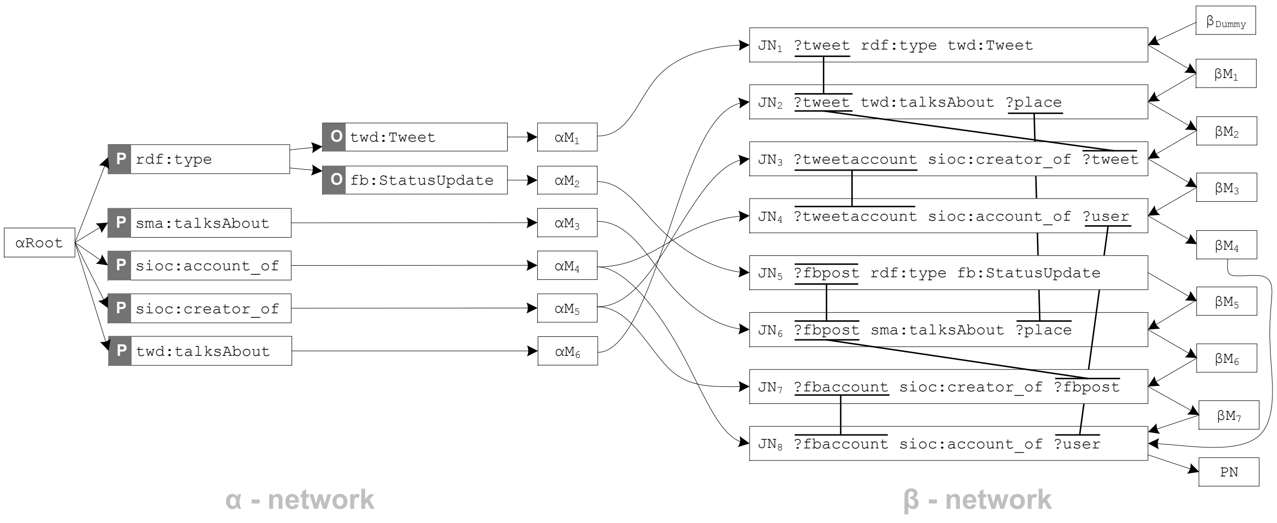

αRoot
rdf:type
sma:talksAbout
sioc:account_of
sioc:creator_of
twd:talksAbout
twd:Tweet
fb:StatusUpdate
αM1
αM2
αM3
αM4
αM5
αM6
JN1 ?tweet rdf:type twd:Tweet
JN2 ?tweet twd:talksAbout ?place
JN3 ?tweetaccount sioc:creator_of ?tweet
JN4 ?tweetaccount sioc:account_of ?user
JN5 ?fbpost rdf:type fb:StatusUpdate
JN6 ?fbpost sma:talksAbout ?place
JN7 ?fbaccount sioc:creator_of ?fbpost
JN8 ?fbaccount sioc:account_of ?user
βDummy
βM1
βM2
βM3
βM4
βM5
βM6
βM7
PN
α - network
β - network

Figure 6: Rete network

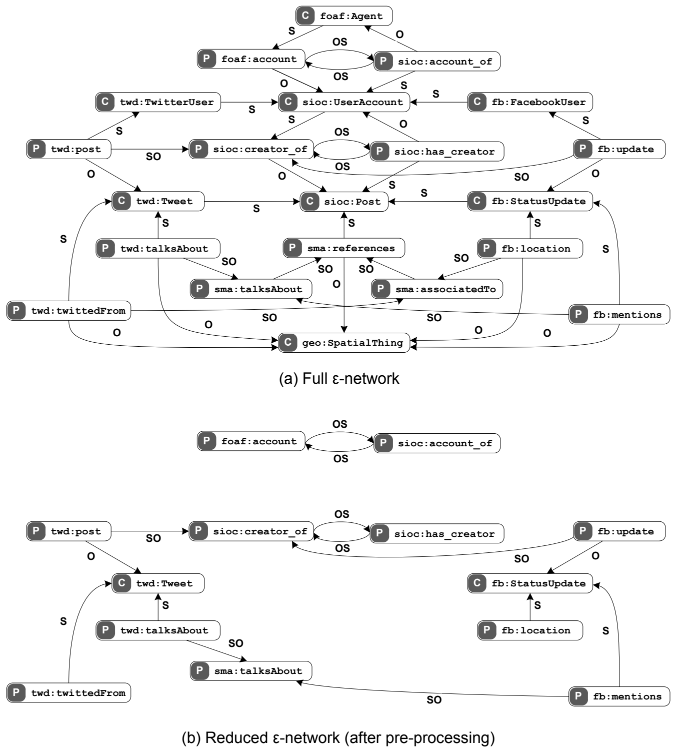


Figure 5: Epsilon network

αM_n, the associated β-network join node JN_n is activated in order to attempt to join the new entry with the existing triples according to variable bindings. In case such a join is possible, a new token is generated, linked with the existing token (representing an existing partial match) and stored in the associated βM_n node, thus representing an incremental update to the partial match. As explained in Section 3.5, JN_n nodes are also performing time-based window checks.

It is worth noting that Figures 5 and 6 are actually connected (not shown in the figures), i.e., outputs of ε-network nodes are inputs to the α-network nodes. Since there is enough knowledge on how a triple looks like in the ε-network, specific ε and α nodes are directly connected. For example, the output of the fb:StatusUpdate class node is directly connected to αM_2. These direct links potentially bring significant savings of computational resources. Streamed triples not amenable for ε-network processing enter the α-network at the level of $\alpha Root$.

4. EVALUATION

In order to evaluate the performance of Sparkwave, we defined some tests reusing part of the dataset from the Berlin SPARQL Benchmark (BSBM) [9]. The BSBM defines a suite of benchmarks to compare the performance of RDF stores and, in general, of storage systems that expose SPARQL endpoints. The benchmark is built around an e-commerce use case in which a set of products is offered by different vendors, and consumers have posted reviews about the products.

The BSBM is not intended for measuring stream processing systems and by itself does not provide data streams, so we had to adapt it to our needs. In particular, we reused the data describing product types and offers. The BSBM generates a single-rooted hierarchy of product types, whose depth and number of classes are determined by a scaling factor; each product is then assigned to a product type at the leaf level of the hierarchy. The hierarchy of product types constitutes for us the schema. The BSBM also generates a number of offers related to products, which for us constitute the data stream.[9] Given these schema and stream from the BSBM, we defined three graph patterns, shown in Listing 2:

1. In *pattern 1* we look for offers for products of a given type; for non-leaf types, inference needs to take place on the hierarchy of product types. There are no static instances.

2. Additionally to the conditions set in pattern 1, in *pattern 2* we only want offers for which the vendor is

[9]We include in the stream also triples to assert the type of the product of each offer (i.e. one extra triple per offer).

from a given country. The binding between vendors and countries is given through static instances.

3. Additionally to the conditions set in pattern 2, in *pattern 3* we only want offers for products for which the producer is from a given country. The binding between producers and countries is given through static instances.

```
–– Pattern 1:
REGISTER QUERY BSBSOfferDetection AS
PREFIX bsbm-voc:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX bsbm-inst:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
SELECT ?offer ?product ?vendor ?price ?from ?to ?delivery ?webpage
       ?publisher ?date
FROM STREAM <http://bsbm.org/stream>
FROM <http://members.sti2.at/~srdjank/ProductTypeHierarchy.rdf>
WHERE {
    ?offer rdf:type bsbm-voc:Offer.
    ?offer bsbm-voc:product ?product.
    ?offer bsbm-voc:vendor ?vendor.
    ?offer bsbm-voc:price ?price.
    ?offer bsbm-voc:validFrom ?from.
    ?offer bsbm-voc:validTo ?to.
    ?offer bsbm-voc:deliveryDays ?delivery.
    ?offer bsbm-voc:offerWebpage ?webpage.
    ?offer dc:publisher ?publisher.
    ?offer dc:date ?date.
    ?product rdf:type bsbm-inst:TargetProductType
}

–– Pattern 2: additionally to pattern 1, in WHERE clause:
?vendor bsbm-voc:country http://downlode.org/rdf/iso-3166/countries#GB

–– Pattern 3: additionally to pattern 1, in WHERE clause:
?vendor bsbm-voc:country http://downlode.org/rdf/iso-3166/countries#GB
?product bsbm-voc:producer ?producer
?producer bsbm-voc:country http://downlode.org/rdf/iso-3166/countries#DE
```

Listing 2: Product offer detection patterns

Sparkwave implementation is written in Java. In order to measure throughput, we use a small program that reads the input data (i.e. the offers) from a file and sends them to Sparkwave in N-Triples format through a Unix domain socket at the highest possible rate, and we measure the time needed to process all the data. The schema (i.e. the product type hierarchy) is loaded before starting to stream the data, since it is needed to build the ε-network. The static instances for patterns 2 and 3 are loaded after the schema. In some tests we used different schemas with different numbers of classes: the smallest one has 329 product types arranged in a 4-levels hierarchy (including the root level), whereas the largest one has 22,527 product types arranged in a 6-levels hierarchy. We used different target product types in the patterns, having different number of subclasses (from 0 to 1,608). The test stream contains 200,000 offers, corresponding to almost 2.2 million triples. Tests were run on a Linux machine with CPU Intel Core i7 620M (4 MB cache, 2.66 GHz) and 4 GB of memory.

Results of throughput tests for pattern 1 are shown in Figure 7: the chart shows the throughput in relation to the size of the time window, in the case of a schema with 329 classes and a target product type with 40 subclasses. In order to compare with CQELS and C-SPARQL, which do not support reasoning,[10] we created a second stream including also the triples that associate products to the corresponding non-leaf product types, and we tested CQELS and C-SPARQL, and also Sparkwave, with this second stream. The throughput of course decreases when the time window size increases, but the chart shows that Sparkwave achieves much

[10]C-SPARQL authors describe a technique for stream reasoning in [7], however the publicly available implementation of C-SPARQL does not provide support for reasoning on data streams.

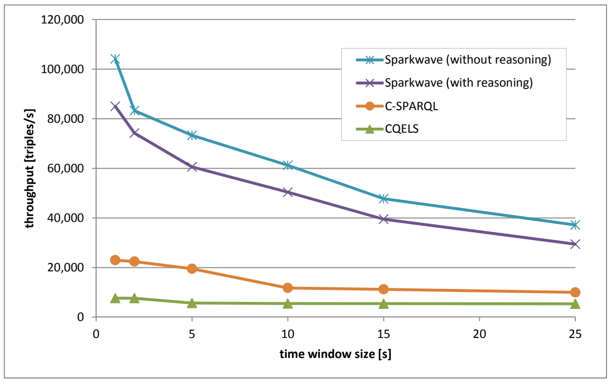


Figure 7: Sparkwave, CQELS and C-SPARQL throughput for pattern 1

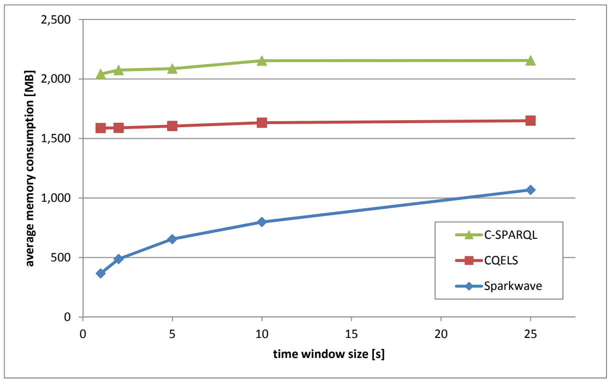


Figure 8: Sparkwave, CQELS and C-SPARQL memory consumption for pattern 1

higher throughput than C-SPARQL and CQELS, especially for smaller sizes of the window: with a time window of 5 seconds Sparkwave (with reasoning) has a maximum throughput 3.1 times higher than C-SPARQL and 10.6 times higher than CQELS, and with a time window of 25 seconds Sparkwave (with reasoning) has a maximum throughput 2.9 times higher than C-SPARQL and 5.4 times higher than CQELS. The advantage of Sparkwave over CQELS and C-SPARQL is even bigger when the systems are compared on the same set of features, i.e. without reasoning. The chart also shows that the impact of reasoning on the performance of Sparkwave is limited, causing a decrease of the maximum throughput around 15-20%.

Figure 8 shows the behaviour of Sparkwave, CQELS and C-SPARQL in terms of memory consumption, in the same test conditions used to measure throughput. Sparkwave memory needs increase when increasing the size of the time window, but are below those of CQELS and C-SPARQL.

In order to evaluate the impact of the size of the schema on Sparkwave performance, we repeated the test using different target product types, having from 4 to 1,608 subclasses. The result is that the throughput is basically unaffected. This was expected, because a larger schema implies more class nodes in the ε-network, but the processing is basically the same (also because schema pre-processing reduces all subclasses of the target class to direct subclasses, no matter how many levels are present in the original hierarchy).

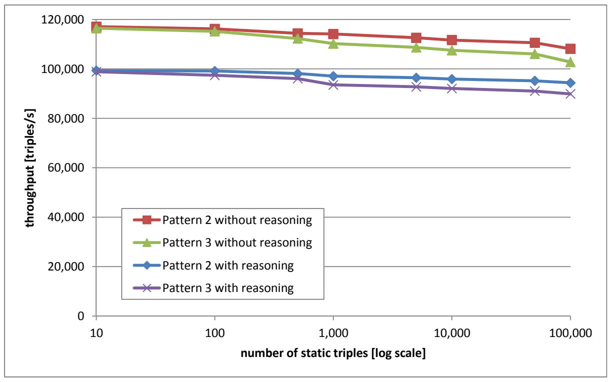


Figure 9: Sparkwave throughput for pattern 2 and 3

Finally, Figure 9 shows Sparkwave throughput for patterns 2 and 3 in relation to the amount of static triples in the system, with and without reasoning, in the case of a time window of 1 second. These results show that the loss of performance due to the presence of static knowledge is very small. Moreover, the difference between the two patterns does not have a significant impact: even if pattern 3 is slightly more complex than pattern 2, the throughput is basically the same, and shows some difference only in the cases with a larger amount of static triples. Finally, as in the case of pattern 1, the performance loss caused by reasoning is not dramatic, and as expected it does not depend on the amount of static instances.

5. FUTURE WORK

Sparkwave performs RDF graph pattern detection, i.e., detection of triple pattern conjunctions bound over joining variables (and occurring inside a time window). Our intention is to extend its capabilities towards supporting other *logical operators* (disjunction and negation), *data operators* (comparison and arithmetic) and *temporal operators* (support for Allen temporal operators [3] on top of interval-based semantics [2]). The existing work reported in [28, 30], which investigates the integration of temporal reasoning in an event-based Rete system, will be used as a foundation and will be further adapted to the temporal characteristics of RDF graph structures.

In parallel with the development of Sparkwave engine capabilities, we intend to formalise its RDF graph pattern expression language, by extending C-SPARQL and borrowing temporal operator elements from EP-SPARQL. The language will support the aforementioned logical and data operators to express patterns over data streams, as motivated by [17] and [14]. Similarly to C-SPARQL, the language will enable the declaration of multiple streams contributing to the pattern, but in contrast to it, the language will also allow the allocation of sub-patterns to different streams, thus providing means to precisely define which part of a graph pattern needs to be fulfilled by which stream. We believe that this feature will play an important role in supporting multiple stream aggregation and integration cases. Due to the same reasons, we intend to enable the definition of time-based windows at the level of sub-patterns, thus allowing cascading expression of time-based windows over a single pattern.

Event consumption strategies (also called parameter contexts) resolve the issue of multiple simultaneous matches to a pattern inside of a time window. For example, in a typical sensor application only the most recent reading of a sensor reflects the current state of the observed object. Sparkwave currently supports a simple strategy (known as “unrestricted”) which outputs a number of matches equal to all possibilities build on top of triple sets (streamed or entailed) contributing to a pattern in a time-based window. Besides this simple strategy, the literature also enumerates more restrictive approaches (e.g., “recent” and “chronological”) to deal with the problem [12, 2]. Our intention is to provide support for other consumption strategies, taking into account also the presence of entailed statements.

6. CONCLUSION

An efficient mechanism for schema-enhanced pattern detection on RDF data streams is required to address the challenges of stream processing on the Web. In this paper we presented Sparkwave, a system based on the Rete algorithm, extended to include statements materialised through schema entailments and to support the temporal nature of data streams. The schema entailments support is provided through the introduction of ε-network which precedes Rete network. We described Sparkwave constituent building blocks and presented a way to build the network. The temporal nature of data streams is tackled through time-based window support, implemented by extending the behaviour of β-network nodes to perform time window constraint checks. We evaluated our approach by reusing part of the dataset from the Berlin SPARQL Benchmark, taking into account the size of the window and the amount of static instances, and compared Sparkwave performance with CQELS and C-SPARQL. In the future, we plan to extend Sparkwave with more features borrowed from Complex Event Processing systems, such as a richer set of temporal operators and consumption strategies.

7. ACKNOWLEDGEMENTS

This work has been supported by the European Commission through the SUPPORT project (EU FP7 Project 242112). The authors wish also to acknowledge Michael Rogger for his insightful comments and effort spent on implementing Sparkwave.

8. REFERENCES

[1] D. J. Abadi, D. Carney, U. ÃĞetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12:120–139, 2003.

[2] R. Adaikkalavan and S. Chakravarthy. SnoopIB: Interval-Based Event Specification and Detection for Active Databases. In *Advances in Databases and Information Systems*, volume 2798 of *Lecture Notes in Computer Science*, pages 190–204. Springer Berlin / Heidelberg, 2003.

[3] J. F. Allen and G. Ferguson. Actions and Events in Interval Temporal Logic. Technical report, University of Rochester, 1994.

[4] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: A Unified Language for Event

Processing and Stream Reasoning. In *Proc. of the 20th Int, Conf. on World Wide Web*, WWW '11, pages 635–644, New York, NY, USA, 2011. ACM.

[5] D. Anicic, P. Fodor, S. Rudolph, R. Stühmer, N. Stojanovic, and R. Studer. ETALIS: Rule-Based Reasoning in Event Processing. In *Reasoning in Event-Based Distributed Systems*, volume 347 of *Studies in Computational Intelligence*, pages 99–124. Springer, 2011.

[6] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford Data Stream Management System. Technical Report 2004-20, Stanford InfoLab, 2004.

[7] D. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. Incremental Reasoning on Streams and Rich Background Knowledge. In *Proc. of 7th Extended Semantic Web Conference (ESWC 2010)*, volume 6088 of *LNCS*, pages 1–15. Springer, 2010.

[8] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-SPARQL: a Continuous Query Language for RDF Data Streams. *Int. J. Semantic Computing*, 4(1):3–25, 2010.

[9] C. Bizer and A. Schultz. The Berlin SPARQL Benchmark. *International Journal On Semantic Web and Information Systems*, 5(2):1–24, 2009.

[10] A. Bolles, M. Grawunder, and J. Jacobi. Streaming SPARQL - Extending SPARQL to Process Data Streams. In *The Semantic Web: Research and Applications*, volume 5021 of *Lecture Notes in Computer Science*, pages 448–462. Springer Berlin / Heidelberg, 2008.

[11] I. Celino, D. Dell'Aglio, E. Della Valle, Y. Huang, T. Lee, S. Park, and V. Tresp. Making Sense of Location Based Micro-posts Using Stream Reasoning. In *Proceedings of the 1st Workshop on Making Sense of Microposts (#MSM2011)*, pages 13–18, May 2011.

[12] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *Proc. of 20th Int. Conf. on Very Large Data Bases*, VLDB '94, pages 606–617. Morgan Kaufmann, 1994.

[13] E. Della Valle, S. Ceri, F. van Harmelen, and D. Fensel. It's a Streaming World! Reasoning upon Rapidly Changing Information. *IEEE Intelligent Systems*, 24:83–89, November 2009.

[14] R. J. Doorenbos. *Production Matching for Large Learning Systems*. PhD thesis, Carnegie Mellon University, Pittsbrurg, PA, 1995.

[15] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[16] B. Glimm, A. Hogan, M. Krötzsch, and A. Polleres. OWL: Yet to arrive on the Web of Data? In *Proceedings of Linked Data on the Web (LDOW2012) Workshop*. CEUR Workshop Proceedings, 2012.

[17] C. Grady, F. Highland, C. Iwaskiw, and M. Pfeifer. System and Method For Building a Computer-Based RETE Pattern Matching Network. Technical report, IBM Corp., Armonk, N.Y., 1994.

[18] S. Groppe, J. Groppe, D. Kukulenz, and V. Linnemann. A SPARQL Engine for Streaming RDF Data. In *Proceedings of the 2007 Third International IEEE Conference on Signal-Image Technologies and Internet-Based System*, pages 167–174, Washington, DC, USA, 2007. IEEE Computer Society.

[19] P. Hayes. RDF Semantics. W3C Recommendation, W3C, Feb. 2004.

[20] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, W3C, Feb. 2004.

[21] D. Le-Phuoc, M. Dao-Tran, J. Xavier Parreira, and M. Hauswirth. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In *Proceedings of the 10th International Semantic Web Conference*, volume 7031 of *Lecture Notes in Computer Science*, pages 370–388. Springer Berlin / Heidelberg, 2011.

[22] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[23] M. Perry, P. Jain, and A. P. Sheth. SPARQL-ST: Extending SPARQL to Support Spatiotemporal Queries. In *Geospatial Semantics and the Semantic Web*, volume 12 of *Semantic Web and Beyond*, pages 61–86. Springer US, 2011.

[24] J. F. Sequeda, O. Corcho, and A. Gomez-Perez. Linked Stream Data: a short paper. In *2nd Semantic Sensor Network Workshop*. CEUR Workshop Proceedings, 2009.

[25] A. Sheth, C. Henson, and S. S. Sahoo. Semantic Sensor Web. *IEEE Internet Computing*, 12(4):78–83, july-august 2008.

[26] H. Stuckenschmidt and J. Broekstra. Time-Space Trade-offs in Scaling up RDF Schema Reasoning. In *WISE Workshops*, volume 3807 of *LNCS*, pages 172–181. Springer, 2005.

[27] J. Tappolet and A. Bernstein. Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL. In *The Semantic Web: Research and Applications*, volume 5554 of *Lecture Notes in Computer Science*, pages 308–322. Springer Berlin / Heidelberg, 2009.

[28] K. Walzer, T. Breddin, and M. Groch. Relative temporal constraints in the Rete algorithm for complex event detection. In *Proceedings of the second international conference on Distributed event-based systems*, DEBS '08, pages 147–155, New York, NY, USA, 2008. ACM.

[29] K. Walzer, M. Groch, and T. Breddin. Time to the Rescue - Supporting Temporal Reasoning in the Rete Algorithm for Complex Event Processing. In *Proceedings of the 19th international conference on Database and Expert Systems Applications*, DEXA '08, pages 635–642, Berlin, Heidelberg, 2008. Springer-Verlag.

[30] K. Walzer, T. Heinze, and A. Klein. Event Lifetime Calculation based on Temporal Relationships. In *Proceedings of the International Conference on Knowledge Engineering and Ontology Development (KEOD 2009)*, pages 269–274. INSTICC Press, October 2009.