Zuerst ersch. in : ACM SIGMOD Record ; 39 (2010), 1. - S. 20-26
http://dx.doi.org/10.1145/1860702.1860705

Querying RDF Streams with C-SPARQL *

Davide Francesco Barbieri Daniele Braga Stefano Ceri
Emanuele Della Valle Michael Grossniklaus
Politecnico di Milano, Dipartimento di Elettronica e Informazione
Piazza L. da Vinci 32, 20133 Milano, Italy
firstname.lastname@elet.polimi.it

ABSTRACT

Continuous SPARQL (C-SPARQL) is a new language for continuous queries over streams of RDF data. C-SPARQL queries consider windows, i.e., the most recent triples of such streams, observed while data is continuously flowing. Supporting streams in RDF format guarantees interoperability and opens up important applications, in which reasoners can deal with knowledge evolving over time. Examples of such application domains include real-time reasoning over sensors, urban computing, and social semantic data. In this paper, we present the C-SPARQL language extensions in terms of both syntax and examples. Finally, we discuss existing applications that already use C-SPARQL and give an outlook on future research opportunities.

1. INTRODUCTION

Stream-based data sources such as sensors, feeds, click streams, and stock quotations have become increasingly important in many application domains. Streaming data are received continuously and in real-time, either implicitly ordered by arrival time, or explicitly associated with timestamps. As it is typically impossible to store a stream in its entirety, Data Stream Management Systems (DSMS) [14], e.g., [19, 12, 4, 1, 3], allow continuously running queries to be registered, that return new results as new data flow within the streams [15]. At the same time, reasoning upon very large RDF data collections is advancing fast, and SPARQL [23] has gained the role of standard query language for RDF data. Also, SPARQL engines are now capable of querying integrated repositories and collecting data from multiple sources. Still, the large knowledge bases now accessible via SPARQL (such as Linked Life Data[1]) are static, and knowledge evolution is not adequately supported.

*This work is supported by the European project LarKC (FP7-215535). Michael Grossniklaus's contribution was carried out under the SNF grant number PBEZ2-121230.
[1]http://www.linkedlifedata.com/

The combination of static RDF data with streaming information leads to **stream reasoning** [13], an important step to enable logical reasoning in real time on huge and noisy data streams in order to support the decision process of large numbers of concurrent users. So far, this step has received little attention by the Semantic Web community. C-SPARQL, that we introduced in [7], is an extension of SPARQL designed to express continuous queries, i.e., queries registered over both RDF repositories and *RDF streams*. C-SPARQL queries can be considered as inputs to specialized reasoners that use their knowledge about a domain to make real-time decisions. In such applications, reasoners operate upon knowledge snapshots, which are continuously refreshed by registered queries. It is important to note that, in this view, reasoners can be unaware of time changes and of the existence of streams. We have also explored the use of reasoners aware of the time-dependent nature of data in [6], where we propose an algorithm for the incremental maintenance of snapshots. Reasoning over streaming facts is also addressed by the authors of [29], who focus on the scalability of reasoning techniques. Another research related to ours is that by Law et al. [17], who put particular emphasis on the problem of mining data streams [18].

In this paper, we present a summary of the description of C-SPARQL published in [7] and apply the language to new use cases. We focus on how C-SPARQL extends SPARQL with functionality required to manage streams, in a way that is comparable to the approach taken by CQL [2]. Note that this paper neither discusses the evaluation and optimization of C-SPARQL queries, nor other entailment regimes beyond basic RDF entailment. Details on how we addressed these topics in the context of C-SPARQL can be found in [6, 7].

Bolles et al. [10] presented a first attempt to extend SPARQL to support streams, that can be considered an antecedent of our work. It introduced

a syntax for the specification of logical and physical windows in SPARQL queries by means of local grammar extensions. However, their approach is different from ours at least in two key aspects. First, they simply introduce RDF streams as a new data type, and omit essential ingredients, such as aggregates and timestamp functions. Second, the authors do not follow the established approach where windows are used to transform streaming data into non-streaming data in order to apply standard algebraic operations. Instead, they chose to change the standard SPARQL operators by making them timestamp-aware and, thereby, actually introduce a new language semantics.

In stream processing, aggregation is an important functionality. When we started working on C-SPARQL, we based it on SPARQL 1.0 which does not contain any support for aggregates. In previous publications, we therefore also introduced our own syntax and semantics for aggregates in C-SPARQL that does not shrink results in the presence of grouping [7]. In the meantime, it is foreseeable that the upcoming SPARQL 1.1 specification will include aggregation functionality similar to the one known from SQL. For this paper and future work on C-SPARQL, we have chosen to align our notion of aggregates with the one proposed by the W3C and present all examples accordingly.

Furthermore, several SPARQL implementations support some form of proprietary aggregation functions and group definitions. OpenLink Virtuoso[2] supports COUNT, COUNT DISTINCT, MAX, MIN and AVG, with implicit grouping criteria. ARQ[3] supports COUNT and COUNT DISTINCT over groups defined through an SQL-like GROUP BY clause. ARC[4] also supports the keyword AS to bind variables to aggregated results. In [25], the authors study how grouping and aggregation can be defined in the context of queries over RDF graphs, taking into consideration the peculiarities of the data model, and providing an extension of SPARQL based on operational semantics.

This paper is organized as follows. Section 2 presents the distinguishing language extensions of C-SPARQL referring to a simple scenario of social data analysis. After introducing the RDF stream data type, we discuss the extensions for windows, stream registration, and query registration. Other application scenarios, beyond social data analysis, are presented in Section 3. Finally, an outlook on using C-SPARQL for enabling stream reasoning is presented in Section 4.

[2]http://virtuoso.openlinksw.com/
[3]http://jena.sourceforge.net/ARQ/
[4]http://arc.semsol.org/

2. C-SPARQL

In the following, we present a summary of C-SPARQL by progressively introducing its new features relative to SPARQL. We interleave the presentation of the new syntax, extended by adding new productions to the standard grammar of SPARQL [23], and the discussion of some examples. As a demonstration scenario, we have chosen queries that are relevant to a (highly simplified) case of social data analysis.

2.1 RDF Stream Data Type

C-SPARQL adds **RDF streams** to the SPARQL data types, in the form of an extension done much in the same way in which the stream type has been introduced to extend relations in relational data stream management systems. RDF streams are defined as ordered sequences of pairs, each pair being made of an RDF triple and a timestamp τ:

$$\begin{array}{c}\ldots\\(\langle subj_i, pred_i, obj_i\rangle\ ,\ \tau_i)\\(\langle subj_{i+1}, pred_{i+1}, obj_{i+1}\rangle\ ,\ \tau_{i+1})\\\ldots\end{array}$$

Timestamps can be considered as *annotations* of RDF triples, and are monotonically non-decreasing in the stream ($\tau_i \leq \tau_{i+1}$). More precisely, timestamps are not strictly increasing because they are not required to be unique. Any (unbounded, though finite) number of consecutive triples can have the same timestamp, meaning that they "occur" at the same time, although sequenced in the stream according to a positional order.

Example. The classes and properties that we consider in the social data analysis scenario are described in the schema of Figure 1. All class instances are identified by URLs.

Users also have names, and, by virtue of two properties, they *know* and *follow* other users. Using well-known Semantic Web vocabularies, the user name and the `foaf:knows` property can be described using the Friend of a Friend vocabulary (FOAF) [9]. For the `sioc:follows` property, we can use Semantically-Interlinked Online Communities (SIOC) [8].

Topics represent entities of the real world (such as movies or books, to give examples that are relevant for our scenario), with a name and a category.

Documents represent information sources on actual topics. Examples of documents are Web pages that *describe* topics like a particular book or movie. As vocabularies, we can use `rdfs:label` [11] for the names of documents and topics. Finally, the attribute `skos:subject` from the Simple Knowledge Organization System (SKOS) [22] connects a topic to its category, identified using YAGO [27].

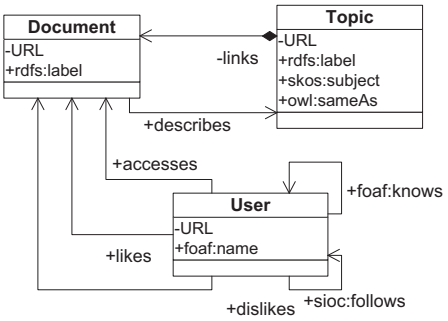

Figure 1: Example data schema

All the knowledge described so far is static (or, more precisely, slowly changing), meaning that the applications we are willing to consider can assume this information as invariant in a period comparable with the size of a window. Of course, updates of this information are also allowed, e.g., to state that a new friendship holds after the addition of an instance of the `foaf:knows` property.

The running example also uses streaming knowledge, and namely streams of notifications that capture the behavior of users with respect to documents (and therefore, transitively, to topics). The *accesses*, *likes*, and *dislikes* properties represent events that occur at the time in which users access a document or express their opinion about it.[5] Quite straightforwardly, any interaction of a user with a document generates in the stream a triple of the form $\langle U, \textit{sd:accesses}, D\rangle$, where U and D respectively represent a generic user and a generic document. Also, selected interactions of users with documents generate triples of the form $\langle U, \textit{sd:likes}, D\rangle$ and $\langle U, \textit{sd:dislikes}, D\rangle$. It is worth noting that in the stream the predicates can only assume one of the three values exemplified above, while the subjects and objects may freely vary in the space of users and documents. This is coherent with RDF repositories whose predicates are taken from a small vocabulary constituting a sort of schema. However, the interpretation of C-SPARQL makes no specific assumptions nor requires restrictions on variable bindings relative to any part of the streaming triples. An example of possible triples in a stream of interactions and opinions is given below.

triple	Timestamp
`c:Usr1 sd:accesses c:movie1`	t_{400}
`c:Usr2 sd:accesses c:movie1`	t_{401}
`c:Usr1 sd:likes c:movie2`	t_{402}
...	...

[5]In the rest of this paper, we refer to this vocabulary with the prefix `sd` (for "social data").

2.2 Windows

The introduction of data streams in C-SPARQL requires the ability to *identify* such data sources and to specify *selection* criteria over them.
For *identification*, we assume that each data stream is associated with a distinct IRI, that is a locator of the actual data source of the stream. More specifically, the IRI represents an IP address and a port for accessing streaming data.
For *selection*, given that streams are intrinsically infinite, we introduce a notion of windows on streams, whose types and characteristics are inspired by the ones defined for relational streaming data.

Identification and selection are expressed in C-SPARQL by means of the `FROM STREAM` clause. The syntax is as follows:

FromStrClause ⟶ 'FROM' ['NAMED'] 'STREAM' *StreamIRI* '[RANGE' *Window* ']'
Window ⟶ *LogicalWindow* | *PhysicalWindow*
LogicalWindow ⟶ *Number TimeUnit WindowOverlap*
TimeUnit ⟶ 'ms' | 's' | 'm' | 'h' | 'd'
WindowOverlap ⟶ 'STEP' *Number TimeUnit* | 'TUMBLING'
PhysicalWindow ⟶ 'TRIPLES' *Number*

A window extracts the *last* data elements from the stream, which are the only part of the stream to be considered by one execution of the query. The extraction can be *physical* (a given number of triples) or *logical* (all triples occurring within a given time interval, whose number is variable over time).

Logical windows are *sliding* [16] if they are progressively advanced by a given `STEP` (i.e., a time interval that is shorter than the window's time interval). They are *non-overlapping* (or `TUMBLING`) if they are advanced in each iteration by a time interval equal to their length. With tumbling windows every triple of the stream is included exactly into one window, whereas with sliding windows some triples can be included into several windows.

The optional `NAMED` keyword works exactly like when applied to the standard SPARQL `FROM` clause for tracking the provenance of triples. It binds the IRI of a stream to a variable which is later accessible through the `GRAPH` clause.

Example. As a very simple first example, consider the query that extracts all books (i.e., all topics whose category is "book") seen by the friends of John in the last 15 minutes. The query considers the last 15 minutes, and the sliding window is modified every minute, so that the query result is renewed every minute.

```
SELECT DISTINCT ?topic
FROM STREAM <http://streamingsocialdata.org/
             interact.trdf> [RANGE 15m STEP 1m]
WHERE { ?user sd:accesses ?document .
        ?user foaf:knows ?john .
        ?john foaf:name "John" .
        ?document t:describes ?topic .
        ?topic skos:subject yago:Movies . }
```

The query joins static and streaming knowledge, and is executed as follows. First, all triples with `sd:accesses` as a predicate are extracted from the current window over the stream, to match the first triple pattern in the WHERE clause. Then the other triple patterns are matched against the static knowledge base, applying the "join" conditions expressed by the bindings of variables `?user` and `?document` to identify the observed `?topic`s. The window considers all the stream triples in the last 15 minutes, and is advanced every minute. This means that at every new minute new triples enter into the window and old triples exit from the window. Note that the query result does not change during the slide interval, and is only updated at every slide change. Triples arriving in the stream between these points in time are queued until the next slide change and do not contribute to the result until then.

2.3 Stream Registration

The result of a C-SPARQL query can be a set of bindings, but also a new RDF stream. In order to generate a stream, the query must be registered through the following statement:

Registration ⟶ 'REGISTER STREAM' *QueryName*
['COMPUTED EVERY' *Number TimeUnit*] 'AS' *Query*

Only queries in the CONSTRUCT and DESCRIBE form[6] can be registered as generators of RDF streams, as they produce RDF triples, associated with a timestamp as an effect of the query execution.

The optional COMPUTED EVERY clause indicates the frequency at which the query *should* be computed. If no frequency is specified, the query is computed at a frequency that is automatically determined by the system.[7]

[6] There are four query forms in SPARQL, different in the first clause: SELECT returns variables bound in a query pattern match. CONSTRUCT returns an RDF graph constructed by substituting variables in a set of triple templates. ASK returns a boolean indicating whether a query pattern matches or not. DESCRIBE returns an RDF graph that describes the resources found. Please refer to [23] for further explanations.

[7] Several data stream management systems are capable of self tuning the execution frequency of registered queries. This not only applies to queries with unspecified registration frequencies, but also whenever, due to peaks of workload, the execution frequency of all queries is reduced, so as to gracefully degrade the overall performances.

Example. The following example shows the construction of a new RDF data stream by means of the registration of a CONSTRUCT query. We consider the previous example again, and modify it so as to generate a stream by selecting all interactions that are of the "likes" type, that are performed by a friend of John, and that concern movies.

```
REGISTER STREAM MoviesJohnsFriendsLike
        COMPUTED EVERY 5m AS
CONSTRUCT {?user sd:likes ?document}
FROM STREAM <http://streamingsocialdata.org/
             interact.trdf> [RANGE 30m STEP 5m]
WHERE { ?user sd:likes ?document .
        ?user foaf:knows ?john .
        ?john foat:name "John" .
        ?document sd:describes ?topic .
        ?topic skos:subject yago:Movies . }
```

This query uses the same logical conditions as the previous one on static data, but only matches the `sd:likes` predicate. The output is constructed in the format of a stream of RDF triples. Every query execution may produce from a minimum of zero triples to a maximum of an entire graph. The timestamp is always dependent on the query execution time only, and is not taken from the triples that match the patters in the WHERE clause. Thus, even though in the example the output stream is a restriction of the input stream, a new timestamp is assigned to every triple. Also note that, if the window contains more than one matching triple with a `sd:likes` predicate, then also the result contains more than one triple, that are returned as a graph. In this case the same timestamp is assigned to all the triples of the graph. In all cases, however, timestamps are system-generated in monotonic non-decreasing order. Results of two evaluations of the previous query are presented in the table below, generating two graphs (one at $\tau = 100$ and one at $\tau = 101$).

triple	Timestamp
`c:Usr1 sd:likes c:Movie1`	t_{100}
`c:Usr2 sd:likes c:Movie2`	t_{100}
`c:Usr1 sd:likes c:Movie2`	t_{101}
`c:Usr2 sd:likes c:Movie1`	t_{101}
`c:Usr3 sd:likes c:Movie3`	t_{101}

2.4 Query Registration

All queries over RDF data streams are denoted as *continuous queries*, because they continuously produce output in the form of tables of variable bindings or RDF graphs. In the section above we addressed the registration of RDF streams. Here, we address the registration of queries that do not produce a stream, but a result that is periodically updated. C-SPARQL queries are registered through the following statement:

Registration → 'REGISTER QUERY' *QueryName*
['COMPUTED EVERY' *Number TimeUnit*] 'AS' *Query*

The COMPUTED EVERY clause is the same as the one for stream registration.

Example. As a very simple example of a registered query that does not generate a stream, consider the following query. For each known user, the query counts the overall number of interactions performed in the last 30 minutes and the number of distinct topics to which the documents refer.

```
REGISTER QUERY GlobalCountOfInteractions
         COMPUTED EVERY 5m AS
SELECT ?user
       COUNT(?document) as ?numberOfInteractions
       COUNT(DISTINCT ?topic) as ?numDifferentTopics
FROM STREAM <http://streamingsocialdata.org/
             interact.trdf> [RANGE 30m STEP 5m]
WHERE { ?user sd:accesses ?document .
        ?document sd:describes ?topic . }
GROUP BY { ?user }
```

The query is executed by matching all interactions in the window, grouping them by ?user, and computing the aggregates. The result has the form of a table of bindings that is updated every 5 minutes.

All the examples considered so far have shown a join of static and streaming knowledge. As an example of query composability, we now show a query that takes as input the registered stream generated by the query shown in Section 2.3.

```
REGISTER QUERY GlobalCountOfInteractions
         COMPUTED EVERY 5m AS
SELECT ?user COUNT(?document) as ?numberOfMovies
FROM STREAM <http://streamingsocialdata.org/
     MoviesJohnsFriendsLike.trdf> [RANGE 30m STEP 5m]
WHERE { ?user sd:likes ?document }
GROUP BY { ?user }
```

The query counts, among the friends of John, the number of movies that each friend has liked in the last 30 minutes.

2.5 Multiple Streams

C-SPARQL queries can combine triples from multiple RDF streams, as shown in the following example.

Example. In addition to the stream of interactions, we now consider the presence of a second stream of data concerning the entrance of registered users into theaters to watch movies. The next query takes as input the stream of preferences of John's friends and the stream about people entering cinemas, and identifies friends who like a 3D movies, but only considering users who actually watched at least two 3D movies in the last week (so as to focus on the advice of "experts").

```
REGISTER QUERY JohnsFriendsToRecommend3DMovies AS
SELECT ?user
FROM STREAM <http://streamingsocialdata.org/
          MoviesJohnsFriendsLike.trdf> [RANGE 1h]
FROM STREAM <http://comingsoon.com/
       WatchedMovies.trdf>  [RANGE 7d]
WHERE { ?user sd:likes ?document .
        ?document sd:describes ?topic .
        ?topic skos:subject yago:3DMovies .
        { SELECT ?user
          WHERE { ?user sd:accesses ?document1 .
                  ?document1 sd:describes ?topic1 .
            ?topic1 skos:subject yago:3DMovies . }
          GROUP BY ?user
          HAVING COUNT(DISTINCT ?topic1) >= 2 } }
```

The query is executed as follows. Variable ?user is matched in the WHERE clause of the outer query among the friends of John. Also, the topic is checked to be a 3D movie (the stream is selected checking that the topics are classified as generic movies). The user is also checked to have the property of having seen at least two other 3D movies in the nested query. Note the use of the same ?user variable in the nested query so as to pass the binding and check the "aggregate" property.

2.6 Timestamp Function

The timestamp of a stream element can be retrieved and bound to a variable using a timestamp function. The timestamp function has two arguments.

- The first is the name of a variable, introduced in the WHERE clause and bound to an RDF triple of that stream by pattern matching.
- The second (optional) is the URI of a stream, that can be obtained through SPARQL GRAPH clause.

The function returns the timestamp of the RDF stream element producing the binding. If the variable is not bound, the function is undefined, and any comparison involving its evaluation has a non-determined behavior. If the variable gets bound multiple times, the function returns the most recent timestamp value relative to the query evaluation time.

Example. In order to exemplify the use of timestamps within queries, we show a query that tries to discover causal relationships between different actions. More precisely, the query identifies users who are likely to influence the behavior of other users, by matching interactions of the same kind that occur on the same document *after* the first user has performed them. The query in C-SPARQL is the following:

```
REGISTER STREAM OpinionMakers
         COMPUTED EVERY 5m AS
SELECT ?opinionMaker
FROM STREAM <http://streamingsocialdata.org/
     interact.trdf>  [RANGE 30m STEP 5m]
```

```
WHERE { ?opinionMaker foaf:knows ?friend .
        ?friend ?opinion ?document.
        ?opinionMaker ?opinion ?document .
        FILTER ( timestamp(?friend) >
                 timestamp(?opinionMaker)
                   && ?opinion != sd:accesses ) }
GROUP BY ( ?opinionMaker )
HAVING ( COUNT(DISTINCT ?friend) > 3 )
```

Note that the timestamps are taken from variables that occur only once in patterns applied to streaming triples, thus avoiding ambiguity. Also, the query filters out actions of type "accesses", that are normally required before expressing an opinion such as "like" or "dislike".

3. APPLICATIONS

The scenario of social data analysis is just one example of many possible applications of C-SPARQL. In the last years, more and more effort has been put in trying to address problems that require reasoning on streaming data, and this has been done mainly with "classical" reasoning tools. For instance, Bandini et al. [5] worked on traffic monitoring and traffic pattern detection. Mendler et al. [21] applied constructive Description Logics to financial-transaction auditing. In the mobile telecommunication sector, Luther et al. [20] reported the need for reasoning over streams for situation-aware mobile services. Walavalkar et al. [29] worked on patient monitoring systems. All these application areas are natural settings for C-SPARQL. In the following, we provide more details about concrete applications of C-SPARQL in the cases of situation aware mobility and oil production. In Section 4, we will also outline how we are currently studying dedicated reasoning techniques for the interplay of C-SPARQL and reasoners, in order to efficiently carry out reasoning tasks over streams.

3.1 Situation-Aware Mobility

Mobility is one of the defining characteristics of modern life. Technology can support and accompany mobility in several ways, both for business and for pleasure. Mobile phones provide a good basis for challenging C-SPARQL use cases, as they are popular and widespread. In order to complete the adoption of such devices in our everyday life, mobile applications must fulfill real-time requirements, especially if we are to use them to make short-term decisions. Leveraging data from sensors, which is likely to be available in the form of streams, mobile applications may compute interesting answers by reasoning over streams.

The following C-SPARQL query finds the locations of commuters having less than 30 minutes of travel time remaining. For each user, it retrieves the train number, its position in terms of the closest station, the city where the station is in, etc., by computing the transitive closure of relation `isIn`.

```
REGISTER QUERY WhereAlmomstToDestinationCommutersAre
         COMPUTED EVERY 1sec AS
SELECT DISTINCT ?user ?location
FROM <http://mobileservice.org/meansOfTransp.rdf>
FROM STREAM <http://mobileservice.org/
             positions.trdf> [RANGE 10sec STEP 1sec]
WHERE { ?user ex:isIn ?location .
        ?user a ex:Commuter .
        ?user ex:remainingTravelTime ?t .
        FILTER ( ?t >= "PT30M"^^xsd:duration )  }
```

It does so by continuously querying a stream of RDF triples that describe the users on trains, moving from a station to another, together with a static RDF graph, which describes where the stations are located, e.g., a station is in a city, which is in a region, which is in a state, etc. For further information about this application scenario, the reader is directed to [6].

3.2 Oil Production

Oil operation engineers base their decision process on real time data acquired from sensors on oil rigs, located at the sea surface and seabed. A typical oil production platform is equipped with about 400.000 sensors for measuring environmental and technical parameters. The problems they face include determining the expected time to failure whenever the barring starts vibrating, given the brand of the turbine, or detecting weather events from observation data. For details about this application scenario, the reader is directed to [26].

The C-SPARQL query below detects if a weather station is observing a blizzard. A blizzard is identified when a severe storm, characterized by low temperatures, strong winds, and heavy snow, lasts for 3 hours or more.

```
REGISTER STREAM BlizzardDetection
         COMPUTED EVERY 10m AS
CONSTRUCT {?s so:generatedObservation [a w:blizzard]}
FROM <http://oilprod.org/weatherStations.rdf>
FROM STREAM <http://oilprod.org/weatherObs.trdf>
  [RANGE 3h STEP 10m]
WHERE {
 ?s grs:point "66.348085,10.180662" ;
  so:generatedObservation [a w:SnowfallObservation] .
 { SELECT ?s
   WHERE { ?s so:generatedObservation ?o1
           ?o1 a w:TemperatureObservation ;
             so:observedProperty w:AirTemperature ;
             so:result [ so:value ?temperature  ] . }
   GROUP BY ( ?s )
   HAVING (AVG(?temperature)<"0.0"^^xsd:float) }
 { SELECT ?s
   WHERE { ?s so:generatedObservation ?o2
           ?o2 a w:WindObservation ;
             so:observedProperty w:WindSpeed ;
             so:result [ so:value ?speed  ] . }
   GROUP BY ( ?s )
   HAVING (MIN(?speed)> "40.0"^^xsd:float)  }
}
```

4. OUTLOOK

We believe that C-SPARQL and its corresponding infrastructure provide an excellent starting point for stream reasoning [13]. By providing an RDF-based representation of heterogeneous streams, C-SPARQL solves the challenge of giving reasoners an access protocol for heterogeneous streams. As RDF is the most accepted format to feed information to reasoners, C-SPARQL allows existing reasoning mechanisms to be further extended in order to support continuous reasoning over data streams and rich background knowledge. We already made a first step in this direction, investigating the incremental maintenance of ontological entailment materializations [6]. To do so, we annotate streaming knowledge with expiration times, which we manage in an auxiliary data structure, devoted to handle the limited validity of inference through time. Our reasoner is then capable of incrementally maintaining the entailments of transient knowledge, that are themselves transient, in an efficient way. In future work, we plan to extend this approach and to generalize it to more expressive languages.

Moreover, the extraction of patterns from data streams is subject of ongoing research in machine learning. For instance, results from statistical relational learning are able to derive classification rules from example data in very effective ways. In our future work, we intend to link relational learning methods with C-SPARQL to facilitate pattern extraction on top of RDF streams.

Finally, we envision the possibility to leverage recent developments in distributed and parallel reasoning [28, 24] for scaling up to large data streams and many concurrent reasoning tasks.

5. REFERENCES

[1] D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In *Proc. CIDR*, 2005.

[2] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2):121–142, 2006.

[3] Y. Bai, H. Thakkar, H. Wang, C. Luo, and C. Zaniolo. A Data Stream Language and System Designed for Power and Extensibility. In *Proc. CIKM*, 2006.

[4] H. Balakrishnan et al. Retrospective on Aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[5] S. Bandini, A. Mosca, and M. Palmonari. Common-sense spatial reasoning for information correlation in pervasive computing. *Applied Artificial Intelligence*, 21(4&5):405–425, 2007.

[6] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. Incremental Reasoning on Streams and Rich Background Knowledge. In *ESWC*, 2010.

[7] D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus. An Execution Environment for C-SPARQL Queries. In *Proc. EDBT*, 2010.

[8] U. Bojars, J. G. Breslin, A. Finn, and S. Decker. Using the semantic web for linking and reusing data across web 2.0 communities. *Web Semantics*, 6(1):21–28, 2008.

[9] U. Bojars, J. G. Breslin, V. Peristeras, G. Tummarello, and S. Decker. Interlinking the social web with semantics. *Intelligent Systems*, 23(3):29–40, 2008.

[10] A. Bolles, M. Grawunder, and J. Jacobi. Streaming SPARQL – Extending SPARQL to Process Data Streams. In *Proc. ESWC*, 2008.

[11] D. Brickley and R. Guha. RDF Vocabulary Description Language 1.0: RDF Schema, W3C Working Draft. Technical report, W3C, 2002.

[12] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. SIGMOD*, 2000.

[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(6):83–89, 2009.

[14] M. Garofalakis, J. Gehrke, and R. Rastogi. *Data Stream Management: Processing High-Speed Data Streams (Data-Centric Systems and Applications)*. Springer-Verlag New York, Inc., 2007.

[15] L. Golab, D. DeHaan, E. D. Demaine, A. López-Ortiz, and J. I. Munro. Identifying Frequent Items in Sliding Windows over On-line Packet Streams. In *IMC*, 2003.

[16] L. Golab and M. T. Özsu. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In *Proc. VLDB*, 2003.

[17] Y.-N. Law, H. Wang, and C. Zaniolo. Query Languages and Data Models for Database Sequences and Data Streams. In *Proc. VLDB*, 2004.

[18] Y.-N. Law and C. Zaniolo. An Adaptive Nearest Neighbor Classification Algorithm for Data Streams. In *Proc. PKDD*, 2005.

[19] L. Liu, C. Pu, and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Trans. Knowl. Data Eng.*, 11(4):610–628, 1999.

[20] M. Luther, Y. Fukazawa, M. Wagner, and S. Kurakake. Situational reasoning for task-oriented mobile service recommendation. *Knowledge Eng. Review*, 23(1):7–19, 2008.

[21] M. Mendler and S. Scheele. Exponential Speedup in UL Subsumption Checking relative to general TBoxes for the Constructive Semantics. In *Proc. DL*, 2009.

[22] A. Miles, B. Matthews, M. Wilson, and D. Brickley. SKOS core: Simple Knowledge Organisation for the Web. In *Proc. Intl. Conf. on Dublin Core and metadata applications*, Madrid, Spanien, 2005.

[23] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`.

[24] A. Schlicht and H. Stuckenschmidt. Distributed resolution for expressive ontology networks. In *Web Reasoning and Rule Systems*, 2009.

[25] D. Y. Seid and S. Mehrotra. Grouping and Aggregate queries Over Semantic Web Databases. In *Proc. Intl. Conf. on Semantic Computing (ICSC)*, 2007.

[26] H. Stuckenschmidt, S. Ceri, E. Della Valle, and F. van Harmelen. Towards expressive stream reasoning. In *Proceedings of the Dagstuhl Seminar on Semantic Aspects of Sensor Networks*, 2010.

[27] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: a core of semantic knowledge unifying wordnet and wikipedia. In *Proc. WWW*, 2007.

[28] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. Bal. OWL reasoning with WebPIE: calculating the closure of 100 billion triples. In *Proc. ESWC*, 2010.

[29] O. Walavalkar, A. Joshi, T. Finin, and Y. Yesha. Streaming Knowledge Bases. In *Proc. SSWS*, 2008.