C-SPARQL: SPARQL for Continuous Querying

Davide Francesco Barbieri   Daniele Braga   Stefano Ceri
Emanuele Della Valle   Michael Grossniklaus

{ dbarbieri, braga, ceri, dellavalle, grossniklaus } @elet.polimi.it

Politecnico di Milano – Dipartimento di Elettronica e Informazione
Piazza L. da Vinci, 32 - 20133 Milano – Italy

ABSTRACT

C-SPARQL is an extension of SPARQL to support continuous queries, registered and continuously executed over RDF data streams, considering *windows* of such streams. Supporting streams in RDF format guarantees interoperability and opens up important applications, in which reasoners can deal with knowledge that evolves over time. We present C-SPARQL by means of examples in Urban Computing.

1. C-SPARQL IN A NUTSHELL

RDF repositories are scaling up in the time-invariant domain, and SPARQL engines support complex queries over knowledge from multiple sources. However, the combination of static (or relatively slowly changing) knowledge with rapidly changing (or "streaming") data has been so far neglected or forgotten. We introduce *RDF streams* as the natural extension of the RDF data model to the new scenario and Continuous SPARQL (or simply *C-SPARQL*) as the extension of SPARQL for querying RDF streams. C-SPARQL bridges data streams to reasoning and enables stream reasoning, a new, unexplored, high impact research area. C-SPARQL is defined in terms of orthogonal extensions to be added to the standard SPARQL grammar [2], in such a way that a regular SPARQL query is also a C-SPARQL query.

RDF streams – Similar to RDF graphs, every RDF stream is identified by using an IRI (a locator of the actual streaming data source), but instead of being a static collection of triples a stream is a sequence of RDF triples that are continuously produced and annotated with a *timestamp*. Timestamps can be considered as *annotations* of RDF triples, and are monotonically non-decreasing.

Windows – The introduction of RDF streams as a new type of input data requires the ability to *identify* such data and apply *selection* criteria over them. As for *identification*, we assume that each data stream is associated with a distinct IRI. As for *selection*, given that streams are intrinsically infinite, we introduce the notion of windows, whose types and characteristics are inspired by those of the windows in continuous query languages such as CQL [1]. A window extracts the *last* data stream elements. The extraction can be *physical* (a given number of triples) or *logical* (a variable number of triples occurring in a given time interval). Identification and windowing are expressed in C-SPARQL by means of the `FROM STREAM` clause:

```
FromStrClause  → 'FROM' ['NAMED'] 'STREAM' StreamIRI
                 '[ RANGE' Window ']'
Window         → LogicalWindow | PhysicalWindow
LogicalWindow  → Number TimeUnit WindowOverlap
TimeUnit       → 'MSEC' | 'SEC' | 'MIN' | 'HOUR' | 'DAY'
WindowOverlap  → 'STEP' Number TimeUnit | 'TUMBLING'
PhysicalWindow → 'TRIPLES' Number
```

Logical windows are *sliding* when progressively advanced of a `STEP` that is shorter than the window's time interval; they are *non-overlapping* (or `TUMBLING`) when they are advanced of exactly their time interval at each iteration. With tumbling windows every triple of the data stream is included into one window, whereas with sliding windows some triples can be included into several windows. The optional `NAMED` keyword, like in the standard SPARQL `FROM` clause, tracks the provenance of triples binding the IRI of the stream to variables later accessible via the `GRAPH` clause.

Registration – C-SPARQL produces as output the same types as SPARQL: boolean answers, variable bindings, new RDF triples, or RDF descriptions of resources. These outputs are continuously renewed with each query execution when a statement is registered as `QUERY`:

```
Registration → 'REGISTER' ('QUERY'|'STREAM') QName 'AS' Query
```

Only a `CONSTRUCT` or `DESCRIBE` query can be registered as `STREAM`, to produce RDF triples that, once associated with timestamps, yield to new RDF streams. In this case, every query execution produces from a minimum of one triple to a maximum of an entire RDF graph, depending on the construction pattern.

Aggregation – The SPARQL specification lacks aggregation capabilities, although some SPARQL implementations already support it. A continuous query language without aggregates would not be practically useful, therefore, we also provided C-SPARQL with aggregation. This extension is orthogonal w.r.t. the othersand gives rise to an extension of SPARQL which is significant per se. We also allow multiple independent aggregations within the same query, thus pushing the aggregation capabilities beyond those of SQL.

```
AggregateClause →
    ( 'AGGREGATE { (' var ',' Function ',' Group ')' [Filter] '}' )*
Function → 'COUNT' | 'SUM' | 'AVG' | 'MIN' | 'MAX'
Group → var | '{' var ( ',' var )* '}'
```

Every aggregation clause has the following parts: (a) a new variable (i.e. a variable not occurring in the `WHERE` clause or in other aggregation clauses); (b) an aggregation function (one of: `COUNT`, `MAX`, `MIN`, `SUM`, `AVG`); (c) a set of one or more variables, occurring in the WHERE clause, that express the grouping criteria; and (d) an optional FILTER clause.

The semantics of a query with aggregate functions consists in adding to the regular variable bindings computed by the WHERE clause some new bindings, one for each of the new variables introduced by the AGGREGATE clauses. The solution constructed in this way may be further filtered by the FILTER clause. The evaluations of aggregate functions are all independent from each other and take place after the computation of the bindings provided by the WHERE clause.

2. EXAMPLES OF C-SPARQL

A simple Query with Aggregation – Aggregation is orthogonal w.r.t. the other extensions, so we start with a query having aggregates but no streams. It counts the number of sensors placed in every street and returns those with more than 5 sensors. The query is not continuous and requires no registration.

```
PREFIX  c: <http://linkedurbandata.org/city#>
SELECT DISTINCT ?street ?sensors
WHERE { ?sensor c:placedIn ?street . }
AGGREGATE {( ?sensors, COUNT, {?street} ) FILTER (?sensors > 5)}
```

The query is executed by first extracting all pairs of bindings of sensors with their street, then the number of sensors in each street is counted into the new variable sensors and each resulting pair is extended into a triple, then the triples which satisfy the filter predicate are selected, and finally distinct pairs of street and sensor numbers are projected.

A simple Query over a Stream – A classic example in Urban Computing is counting the cars enter the city center passing through tollgates. The next query counts how many cars went through each tollgate in the last 10 minutes, sliding the window every minute.

```
REGISTER QUERY CarsEnteringCityCenterPerTollgate AS
PREFIX  t: <http://linkedurbandata.org/traffic#>
SELECT DISTINCT ?tollgate ?passages
FROM STREAM <www.uc.eu/tollgates.trdf> [RANGE 10 MIN STEP 1 MIN]
WHERE { ?tollgate t:registers ?car . }
AGGREGATE {(?passages, COUNT, {?tollgate})}
```

First, all pairs of bindings of tollgates with the car they register are extracted from the current window, then the number of cars is counted into the new variable passages for each tollgate (and each resulting pair is extended into a triple), and finally the result is projected as distinct pairs of tollgate and passages. Note that at every new minute new triples enter into the window and old triples exit, and the query result does not change during the slide interval; it changes only at every slide change (i.e., at every minute).

In this stream, as in all the streams that we will use in the examples of this paper, the predicate of the triple (e.g. t:register) is fixed while the subject and object part of the triple (e.g., ?tollgate and ?car) are variable. Thus, a physical source for this stream will have items consisting of pairs of values. This arrangement is coherent with RDF repositories whose predicates are taken from a small vocabulary constituting a sort of schema, but C-SPARQL makes no assumption on variable bindings of its stream triples.

Combining Static and Streaming Knowledge – A more complex example counts the number of car entering the city center from each district. The RDF repository stores (a) which districts a city is divided in, (b) which streets belong to each district, and (c) which street each tollgate is placed in. The window is set to 30 minutes and slides every 5 minutes. For brevity, the declaration of prefixes c: and t: will be omitted in the next examples.

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

As in the previous query, all pairs of bindings of tollgates with the cars are extracted. Also, a graph pattern also extracts the pair of bindings of tollgates with the district they are in. Here the cars are counted based on the district.

Streaming the Results of a Query – Continuous queries renew their output at each query execution; such output could be periodically transferred to another system for further analysis (e.g., to plot the traffic as a function of time). In addition, C-SPARQL allows the construction of new RDF data streams, by supporting the possibility to register CONSTRUCT and DESCRIBE queries. We can register the previous query to generate a stream of RDF triples:

```
REGISTER STREAM CarsEnteringCityCenterPerDistrict AS
CONSTRUCT {?district t:has-entering-cars ?passages}
FROM STREAM <www.uc.eu/tollgates.trdf> [RANGE 30 MIN STEP 5 MIN]
WHERE { ?toll t:registers ?car  .  ?toll c:placedIn ?street .
        ?district c:contains ?street . }
AGGREGATE { (?passages, COUNT, {?district}) }
```

Every query execution may produce from a minimum of one triple to a maximum of an entire graph. In the former case, a different timestamp is assigned to every triple; in the latter case, the same timestamp is assigned to all the triples of the graph. In both cases, timestamps are system-generated in monotonic order.

Combining Multiple Streams – We now also consider traffic control cameras registering cars at traffic lights, originating a different stream. The next query finds the streets that have been over 80% of their capacity in the last 5 minutes and shows the number of cars (cars seen by cameras and passing through tolls are summed up).

```
REGISTER QUERY FullStreets AS
SELECT { ?street ?passages }
FROM STREAM <www.uc.eu/tollgates.trdf> [RANGE 5 MIN TUMBLING]
FROM STREAM <www.uc.eu/cameras.trdf> [RANGE 5 MIN TUMBLING]
WHERE { GRAPH <http://stream.org/milantollgates.trdf> {
          ?toll t:registers ?car . ?toll c:placedIn ?street .
        } UNION
        GRAPH <http://stream.org/milancameras.trdf> {
          ?camera t:registers ?car . ?camera c:placedAt ?light .
          ?light  c:crossing  ?street .
        } UNION { ?street   c:hasCapacity ?capacity . }
AGGREGATE { ( ?passages, COUNT, {?street} )
          FILTER (?passages > (0.8*?capacity))}
```

Here, the bindings over the different graphs are combined following the semantics of the UNION pattern evaluation in SPARQL, and it becomes possible to count in the new variable passages the cars registered either by the tollgates or by the cameras in each street.

Acknowledgement

This work is supported by the FP7-215535 integrated project (LarKC) funded by the E.U. Dr. Grossniklaus's work is carried out under SNF grant number PBEZ2-121230. We acknowledge Ioana Manolescu for her contributions to the initial discussions on the potential impact of RDF streams on several use cases.

3. REFERENCES

[1] 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.

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