[2.3] C-SPARQL
Emanuele Della Valle
emanuele.dellavalle@polimi.it
http://emanueledellavalle.org
Share, Remix, Reuse — Legally

- This work is licensed under the Creative Commons Attribution 3.0 Unported License.

- **Your are free:**
  - [Image of icon] **to Share** — to copy, distribute and transmit the work
  - [Image of icon] **to Remix** — to adapt the work

- **Under the following conditions**
  - [Image of icon] **Attribution** — You must attribute the work by inserting
    - “© streamreasoning.org” at the end of each reused slide
    - a credits slide stating
      - These slides are partially based on “Streaming Reasoning for Linked Data 2011” by Emanuele Della Valle and Jeff Z. Pan
        [Link](http://streamreasoning.org/sr4ld2011)

- To view a copy of this license, visit [Link](http://creativecommons.org/licenses/by/3.0/)
Agenda

- RDF Stream
- C-SPARQL
  - FROM STREAM Clause
  - Query and Stream Registration
  - Query Composability
  - Multiple Streams
  - TimeStamp Function
- An Execution Environment for C-SPARQL
- Applications of C-SPARQL
  - Social Media Analysis
  - Traffic Management
  - Environmental Monitoring
  - Financial Analysis
- C-SPARQL Queries Optimization
- Formal Semantics of C-SPARQL
- Conclusions
MEMO: Data Model Used in the Examples

- **sioc:UserAccount**
  - `sioc:id(xsd:string)`
- **sioc:Post**
  - `sioc:content(xsd:string)`
- **geo:SpatialThing**
  - `geo:lat(xsd:float)`
  - `geo:long(xsd:float)`
- **sr:NamedPlace**
- **sr:TwitterUser**
  - `sr:screenName(xsd:string)`
  - `sr:messageID(xsd:string)`
  - `sr:messageTimeStamp(xsd:string)`
- **sr:Tweet**
  - `sr:NamedPlace`
- **twd:post**
- **twd:discuss**
- **sr:following**
- **sr:retweet**
- **sr:reply**
- **sr:talksAbout**
  - `sr:NamedPlace`
  - `sr:Topic`
- **sr:talksAboutNeutrally**
- **sr:talksAboutPositively**
- **sr:talksAboutNegatively**
- **sioc:creator_of**
- **sioc:has_creator**
Streaming vs. Background Information

User related background knowledge

Point of Interest related background knowledge

data stream
RDF Stream Data Type

- Ordered sequence of pairs, where each pair is made of an RDF triple and its timestamp

\[
\ldots
\begin{align*}
(\langle subj_i, pred_i, obj_i \rangle, & \tau_i) \\
(\langle subj_{i+1}, pred_{i+1}, obj_{i+1} \rangle, & \tau_{i+1}) \\
& \ldots
\end{align*}
\]

- Timestamps are not required to be unique, they must be non-decreasing

- E.g.,

\[
\begin{align*}
(\langle Alice :posts :post1 \rangle, & 2010-02-12T13:34:41) \\
(\langle post1 :talksAboutPositively :LaScala \rangle, & 2010-02-12T13:34:41) \\
(\langle Bob :posts :post2 \rangle, & 2010-02-12T13:36:28) \\
(\langle post2 :talksAboutNegatively :Duomo \rangle, & 2010-02-12T13:36:28)
\end{align*}
\]
MEMO: SPARQL

Query Form:
- CONSTRUCT
- DESCRIBE
- SELECT
- ASK

Dataset Clause:
- FROM
- FROM NAMED

Where Clause (Graph Pattern):
- FILTER
- OPTIONAL
- AND
- UNION

TRUE - FALSE
Where C-SPARQL Extends SPARQL
Where C-SPARQL Extends SPARQL

FROM STREAM Clause
Where C-SPARQL Extends SPARQL

FROM STREAM Clause

- Processing RDF streams in C-SPARQL requires
  - to **identify** such data sources and
  - to **specify selection criteria** over them.

- For identification,
  - each RDF stream is associated with a distinct **IRI**

- For selection
  - streams are intrinsically infinite,
  - the notion of **windows upon RDF streams** allows for loading in the dataset the most recent triples
  - Windows over streams are specified through the **FROM STREAM** clause
Where C-SPARQL Extends SPARQL

FROM STREAM Clause - Types of Window

- **physical**: a given number of triples
- **logical**: a variable number of triples which occur during a given time interval (e.g., 1 hour)
  - **Sliding**: they are progressively advanced of a given STEP (e.g., 5 minutes)
  - **Tumbling**: they are advanced of exactly their time interval
Grammar of the `FROM STREAM` clause

```
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
```

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.
extract all restaurants (i.e., all POIs whose category is restaurant) the followers of Alice have been talking about positively in the last 15 minutes. Have the window sliding every minute

PREFIX sr:  <http://www.streamreasoning.org/sr4ld2011/onto#>
PREFIX srd:  <http://www.streamreasoning.org/sr4ld2011/data#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX yago: <http://dbpedia.org/class/yago/>
SELECT ?poi
FROM STREAM <http://www.streamreasoning.org/sr4ld2011/stream#>
  [RANGE 15m STEP 1m]
WHERE {
  ?poi a sr:NamedPlace ;
  skos:subject yago:Restaurant .
  ?friend sr:following srd:Alice ;
  sr:posts ?tweet .
}
Evaluation of Window-based Selection vs FILTER-base Selection
Where C-SPARQL Extends SPARQL

Query and Stream Registration
C-SPARQL extends SPARQL with query and stream registration features:

1. **All C-SPARQL queries over RDF streams are continuous**
   - Registered through the `REGISTER QUERY` statement
   - Run at a frequency specified by the optional `COMPUTED EVERY` clause

2. **The output of queries is in the form of**
   - **Instantaneous** tables of variable bindings
   - **Instantaneous RDF graphs**
   - **RDF stream**

3. **Only queries in the `CONSTRUCT` or `DESCRIBE` form can be registered as generators of RDF streams**

Syntax for query registration:

```
Registration → 'REGISTER' (('QUERY'| 'STREAM') QueryName
              ['COMPUTED EVERY' Number TimeUnit] 'AS' Query)
```

4. **Composability:**
   - Query results registered as streams can feed other registered queries just like every other RDF stream
Create a stream of the restaurants (i.e., all POIs whose category is restaurant) the followers of Alice have been talking about positively in the last 30 minutes. Have the window sliding every 5 minutes.

REGISTER STREAM RestaurantsAlicesFollowersLike COMPUTED EVERY 5m AS
PREFIX sr: <http://www.streamreasoning.org/sr4ld2011/onto#>
PREFIX srd: <http://www.streamreasoning.org/sr4ld2011/data#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX yago: <http://dbpedia.org/class/yago/>
CONSTRUCT { ?friend sr:talksAboutPositively ?poi } FROM STREAM <http://www.streamreasoning.org/sr4ld2011/stream#> [RANGE 30m STEP 5m]
WHERE {
  ?poi a sr:NamedPlace ;
  skos:subject yago:Restaurant .
  ?friend sr:following srd:Alice ;
  sr:posts ?tweet .
}
The output is constructed in the format of an RDF stream.

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 patterns in the WHERE clause.
For each user, every 5 minutes, count the overall number of tweets posted in the last 30 minutes and the number of distinct point of interest to which the tweets refer.

REGISTER QUERY GlobalCountOfTweets
COMPUTED EVERY 5m AS
PREFIX sr:<http://www.streamreasoning.org/sr4ld2011/onto#>
SELECT ?user (COUNT(?tweet) as ?numberOfTweets)
            (COUNT(DISTINCT ?poi) as ?numDifferentPois)
FROM STREAM <http://www.streamreasoning.org/sr4ld2011/stream#>
            [RANGE 30m STEP 5m]
WHERE {
}
GROUP BY ?user
Where C-SPARQL Extends SPARQL

Query Composability - Example

- Which are the current best 10 restaurants according to Alice’s followers?

- The answer can be computed using the stream of results of the RestaurantsAlicesFollowersLike C-SPARQL query, i.e., the restaurants Alice’s followers have been talking about positively in the last 30 minutes

REGISTER QUERY Top10Restaurants4AlicesFollowers
COMPUTED EVERY 5m AS
PREFIX sr:<http://www.streamreasoning.org/sr4ld2011/onto#>
SELECT ?poi (COUNT(DISTINCT ?follower) as ?numOfFollowers)
FROM STREAM <http://www.streamreasoning.org/sr4ld2011/RestaurantsAlicesFollowersLike/> [RANGE 30m STEP 5m]
WHERE {
}
GROUP BY ?poi
ORDER BY DESC (?numOfFollowers)
LIMIT 10
- C-SPARQL queries can combine triples from multiple RDF streams.
- Let’s add to our running example a second stream concerning the entrance of people into the restaurants (e.g., obtained by RDF-izing Foursquare check-ins).
- The next query takes as input the stream generated by the query `RestaurantsAlicesFollowersLike` and this second stream, and only consider the advice of followers who, in the last week, actually checked-in the restaurants they positively talked about.
REGISTER QUERY Top10Restaurants4TrustableAlicesFollowers
COMPUTED EVERY 5m AS
PREFIX sr:>PREFIX fs:<http://www.streamreasoning.org/
foursquare/onto#>
SELECT ?poi (COUNT(DISTINCT ?follower) as ?numOfFollowers)
FROM STREAM <http://www.streamreasoning.org/sr4ld2011/
RestaurantsAlicesFollowersLike/> [RANGE 30m STEP 5m]
FROM STREAM <http://www.streamreasoning.org/sr4ld2011/
foursquareCheckIns/> [RANGE 7d]
WHERE {
   ?follower fs:checksIn ?poi .
}
GROUP BY ?poi
ORDER BY DESC (?numOfFollowers)
LIMIT 10
Where C-SPARQL Extends SPARQL

**TimeStaMp Function**
The timestamp of a triple can be bound to a variable using a `timestamp()` function, which has two arguments:
- the name of a variable, introduced in the `WHERE` clause
- The (optional) IRI of the stream, that can be obtained through SPARQL `GRAPH` clause.

**Semantics**

<table>
<thead>
<tr>
<th>?var</th>
<th><code>Timestamp(?var, [?graph])</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Not bound</td>
<td>Type Error</td>
</tr>
<tr>
<td>Bound once</td>
<td><code>Timestamp of ?var in ?graph</code></td>
</tr>
<tr>
<td>Bound more than once</td>
<td>The most recent timestamp of the bindings of ?var</td>
</tr>
<tr>
<td></td>
<td>• in ?graph, if it is specified</td>
</tr>
<tr>
<td></td>
<td>• in all the streams referenced in the query otherwise</td>
</tr>
</tbody>
</table>
Where C-SPARQL Extends SPARQL

TimeStamp Function - Example

- Who are the opinion makers? i.e., the users who are likely to influence the behavior of other users who follow them

- REGISTER STREAM OpinionMakers COMPUTED EVERY 5m AS
  PREFIX sr:<http://larkc.eu/csparql/sparql/jena/ext#>
  PREFIX f:<http://larkc.eu/csparql/sparql/jena/ext#>
  CONSTRUCT { ?opinionMaker sd:about ?resource } 
  FROM STREAM <http://www.streamreasoning.org/sr4ld2011/stream#> [RANGE 30m STEP 5m]
  WHERE {
    FILTER ( f:timestamp(?follower) > f:timestamp(?opinionMaker) 
              && ( ?opinion = sr:talksAboutPositively || 
                  ?opinion = sr:talksAboutNegatively ) )
  } 
  HAVING ( COUNT(DISTINCT ?follower) > 3 )
An Execution Environment for C-SPARQL

• The execution environment relies entirely on existing technologies
  • Integration of DSMSs and standard SPARQL engines

• Simple, modular architecture to evaluate queries and our the optimization strategies
Applications of C-SPARQL

- The scenario of social data analysis is just one example of many possible applications of C-SPARQL.

- Other examples include:
  - Traffic Management
  - Environmental Monitoring
  - Financial Analysis
Detect cars turning from one street into another

REGISTER STREAM CarsTurningFromGolgiIntoCeloria
    COMPUTED EVERY 1 MIN AS

PREFIX c: <http://linkedurbandata.org/city#>
PREFIX t: <http://linkedurbandata.org/traffic#>

SELECT DISTINCT ?car1
FROM STREAM <http://stream.org/milancameras.trdf>
    [RANGE 5 MIN STEP 1 MIN]
WHERE {
  ?camera1 c:monitors c:MI-via-Celoria .
  ?camera1 c:placedAt ?tr_light .
  ?camera2 c:placedAt ?tr_light .
  FILTER ( timestamp(?car1)<timestamp(?car2)
            AND ?car1 = ?car2 )
}

timestamp() function allows for detecting that the car is turning in one specific direction and not vice versa.
The following C-SPARQL query detects if a weather station is observing a blizzard, i.e., a severe storm, characterized by low temperatures, strong winds, and heavy snow, lasting for 3 hours or more.

```
REGISTER STREAM BlizzardDetection COMPUTED EVERY 10m AS
PREFIX w:<http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX so: :<http://knoesis.wright.edu/ssw/>
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 ) } 

Get the total amount of money transferred by each Swiss broker in the last 24 hours (ignore small transactions)

REGISTER QUERY TotalAmountPerDayAndBroker AS

SELECT DISTINCT ?broker SUM(?amount)
FROM <http://brokerscentral.org/brokers.rdf>
FROM STREAM <http://stockexchange.org/market.trdf>
[RANGE 24h TUMBLING]
WHERE {
  ?broker b:is_from ?country .
  ?transaction x:with ?amount .
  FILTER (?country = 'CH' && ?amount >= 10)
}

GROUP BY ?broker
C-SPARQL Queries Optimization

The rationale behind the optimizations:

• Move the computation as near to the data source as possible

• Let the architecture modules do what they do best

• We need to:
  • Represent the query in an algebraic way
  • Recognize where data is coming from

• We have:
  • Implemented the push of filters and projection
  • Studied the problem of pushing aggregates

• We use two graphical formalisms to represent queries
  • The Denotational Graph (D-Graph)
  • The Operator Graph (O-Graph)
The Denotational Graph (D-Graph)

- Represents graph patterns, filters, and aggregations
- Used to represent data provenance
  - Dashed blocks as data sources
The rationale behind the optimizations:

- Move the computation as near to the data source as possible
- Let the architecture modules do what they do best
- We need to:
  - **Represent the query in an algebraic way**
  - Recognize where data is coming from
- We have:
  - Implemented the push of filters and projection
  - Studied the problem of pushing aggregates
- We use two graphical formalisms to represent queries
  - The Denotational Graph (**D-Graph**)
  - The Operator Graph (**O-Graph**)
C-SPARQL semantics formalization is based on SPARQL semantics formalization by Perez et al.
- Pérez, Arenas, Gutierrez: Semantics and Complexity of SPARQL, ISWC 2006

A mapping \( \mu \) is a partial function \( \mu : V \rightarrow T \) which computes the bindings for all the variables of a query, matching a pattern \( P \) against an RDF dataset \( D \).

Two mappings \( \mu' \) and \( \mu'' \) are said to be compatible if
- \( \forall x \in \text{dom}(\mu') \cap \text{dom}(\mu'') \), then \( \mu'(x) = \mu''(x) \)
- Where \( \text{dom}(\mu) \) is the subset of \( V \) where \( \mu \) is defined
Let $\Omega_1$ and $\Omega_2$ be sets of mappings.

- Operators for mappings composition:
  - $\Omega_1 \otimes \Omega_2 = \{ \mu_1 \cup \mu_2 \mid \mu \in \Omega_1, \mu \in \Omega_2 \text{ are compatible } \}$
  - $\Omega_1 \cup \Omega_2 = \{ \mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2 \}$
  - $\Omega_1 \setminus \Omega_2 = \{ \mu \in \Omega_1 \mid \forall \mu' \in \Omega_2, \mu \text{ and } \mu' \text{ are not compatible } \}$

- The left outer-join is a derived operator:
  - $\Omega_1 \bowtie \Omega_2 = (\Omega_1 \otimes \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$

- The evaluation of a graph pattern $P$ over a dataset $D$ ($[[P]]_D$) is defined recursively:
  - $[[t]]_D = \{ \mu \mid \text{dom}(\mu) = \text{var}(t) \land \mu(t) \in D \}$, where $t$ is a triple pattern and var($t$) is the set of variables occurring in $t$.
  - $[[\langle P_1 \text{ AND } P_2 \rangle]]_D = [[[P_1]]_D \otimes [[[P_2]]_D$}
  - $[[\langle P_1 \text{ OPT } P_2 \rangle]]_D = [[[P_1]]_D \otimes [[[P_2]]_D$}
  - $[[\langle P \text{ AGG } A \rangle]]_D = [[[P]]_D \otimes [[[A]]}_D$, where $A(v_a, f_a, p_a, G_a)$ is an aggregation pattern.
REGISTER QUERY CarsPassingThroughEachStreet
COMPUTED EVERY 5m AS
SELECT ?tollgate, (count(?car) AS ?numOfCars)
FROM STREAM <http://streams.org/gates.trdf> [RANGE 30m STEP 5m]
WHERE {
    ?tollgate c:placedIn ?street .
}
GROUP BY ?tollgate

The query can be expressed as: \( P = t_1 \text{ AND } t_2 \text{ AGG } A_1 \)
- \( t_1 = ?\text{tollgate t:registers ?car} \)
- \( t_2 = ?\text{tollgate c:placedIn ?street} \)
- \( A_1 = A(?\text{passages, count, ?car, ?tollgate}) \)

\( D = \omega_1(R; now - 10; now) \cup S \), being \( R \) the stream and \( S \) the static knowledge
P is evaluated as
\[
[[P]]_D = [[t_1]]_D \times [[t_2]]_D \times [[A_1]]_D
\]
\[
[[P]]_D = [[t_1]]_R \times [[t_2]]_S \times [[A_1]]_D
\]

The evaluation steps are
- \(T_1 = \text{Eval}(t_1)_R\)
- \(T_2 = \text{Eval}(t_2)_S\)
- \(T_3 = T_1 \text{ AND } T_2\)
- \(A_1 = \text{Eval}(A(?total, count, ?car, ?street))_D\)
- \(\text{Eval}(P)_D = T_3 \text{ AGG } A_1\)

<table>
<thead>
<tr>
<th>Eval((T_1))</th>
<th>Eval((T_2))</th>
<th>Eval((T_1 \text{ AND } T_2))</th>
<th>Eval((A_1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-130</td>
<td>T-130</td>
<td>Pine Park</td>
<td>Pine Park</td>
</tr>
<tr>
<td>T-142</td>
<td>CD234EF</td>
<td>Oak avenue</td>
<td>Oak avenue</td>
</tr>
<tr>
<td>T-142</td>
<td>EF567GH</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Eval((P))</th>
</tr>
</thead>
<tbody>
<tr>
<td>?tollgate</td>
</tr>
<tr>
<td>T-130</td>
</tr>
<tr>
<td>T-142</td>
</tr>
<tr>
<td>T-142</td>
</tr>
</tbody>
</table>
C-SPARQL Queries Optimization

The rationale behind the optimizations:

- Move the computation as near to the data source as possible
- Let the architecture modules do what they do best
- We need to:
  - Represent the query in an algebraic way
  - **Recognize where data is coming from**
- We have:
  - Implemented the push of filters and projection
  - Studied the problem of pushing aggregates
- We use two graphical formalisms to represent queries
  - The Denotational Graph (**D-Graph**)
  - The Operator Graph (**O-Graph**)

C-SPARQL Queries Optimization

The Operational Graph – O-Graph

- Directed graph where
  - Nodes represent operators
  - Edges represent the flow of data

- Used for both query evaluation and optimization

- Based on the SPARQL Query Graph Model (SQGM)

Output of the operator

Operator evaluation based on C-SPARQL semantics

C-SPARQL Formal semantics is based on
Pérez, Arenas, Gutierrez: *Semantics and Complexity of SPARQL*, ISWC 2006
C-SPARQL Queries Optimization

Example of Optimization

?-broker | ?total
[[P₆]] := δ([[P₆]])₉

?-broker | ?total
[[P₇]] := π(?-broker, ?total)([[P₇]])₉

?-broker | ?amount | ?total
[[P₇]] := [[P₆ AGG A(v, f, p, G)]]₉
A(v, f, p, G) := A(?total, sum, {?amount}, {?broker})

?-broker | ?country | ?tx | ?amount
[[P₆]] := [[P₆ FILTER R]]₉
R := (?country = 'CH' ∧ ?speed ≥ 10)

?-broker | ?country | ?tx | ?amount
[[P₅]] := [[P₁ AND P₄]]₉
Example of Optimization

C-SPARQL Queries Optimization

Push of filters and projections
- Push of filters and projections
As for selection, aggregates can be pushed towards the streaming engine, if some constraints hold.

Aggregate evaluations can then be classified as:

- **totally distributable**: all the data needed to compute every aggregation belongs to one stream
  - Example: The sum of amounts of transactions, grouped by sellers in each market, when each stream is associated with one market

- **partially distributable**: there exists a distributed computation of the aggregate function over the streams
  - Example: The sum of amounts of transactions grouped by seller, over all markets, when each stream is still associated with one market

- **not distributable**
  - Example: The count of distinct brokers who did at least one transaction
Conclusions

- We specified of C-SPARQL syntax
  - Incrementally, from existing specifications
    - Including windows, grouping, aggregates, timestamping

- We gave the formal **semantics** of C-SPARQL
  - Query registration, handling overloads
  - Order of evaluation, pattern matching over time, …

- We show **applicability to multiple** industrial sectors

- We investigated **efficiency of evaluation**
  - Defining a suitable algebra
  - Applying optimizations
References

[EDBT2010] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri and Michael Grossniklaus. An Execution Environment for C-SPARQL Queries. EDBT 2010


