An Execution Environment for C-SPARQL Queries

Davide Francesco Barbieri Daniele Braga

Stefano Ceri Michael Grossniklaus*

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

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

ABSTRACT

Continuous SPARQL (C-SPARQL) is proposed as new language for continuous queries over streams of RDF data. It covers a gap in the Semantic Web abstractions which is needed for many emerging applications, including our focus on Urban Computing. In this domain, sensor-based information on roads must be processed to deduce localized traffic conditions and then produce traffic management strategies. Executing C-SPARQL queries requires the effective integration of SPARQL and streaming technologies, which capitalize over a decade of research and development; such integration poses several nontrivial challenges.
In this paper we (a) show the syntax and semantics of the C-SPARQL language together with some examples; (b) introduce a query graph model which is an intermediate representation of queries devoted to optimization; (c) discuss the features of an execution environment that leverages existing technologies; (d) introduce optimizations in terms of rewriting rules applied to the query graph model, so as to efficiently exploit the execution environment; and (e) show evidence of the effectiveness of our optimizations on a prototype of execution environment.

1. INTRODUCTION

Data Stream Management Systems (DSMS) [13] process queries upon stream-based data sources, such as sensors, feeds, click streams, stock quotations, and so on. Streaming data are received continuously and in real-time, either implicitly ordered by arrival time, or explicitly associated with timestamps. It is typically impossible to store a stream in its entirety, therefore queries are continuously running and return new results as new data flow within the streams [14].

*This work is supported by the European project LarKC (FP7-215535). Michael Grossniklaus's contribution is carried out under the SNF grant number PBEZ2-121230.

EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.

Meanwhile, reasoning upon very large RDF data collections is widespreading, and SPARQL has gained the role of standard query language for RDF data. SPARQL-based systems 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.

The combination of static RDF data with streaming information yields to **stream reasoning**, an important step enabling reasoners to use rapidly changing data in addition to static knowledge, which has so far been neglected by the Semantic Web community. C-SPARQL is an extension of SPARQL designed to express continuous queries, i.e. queries registered over both RDF repositories and *RDF streams*. C-SPARQL computes queries such as "How many cars are continuously entering into the city center?", "How many of them come from the north-east district?", and so on. These queries can be considered as inputs to specialized reasoners for "Urban Computing" applications, capable of understanding traffic conditions in a global sense, and then decide policies for traffic management. In such applications, reasoners operate upon knowledge snapshots, which are continuously refreshed by continuous queries. It is important to note that, in this view, reasoners can be unaware of time changes and of the existence of streams. Urban Computing is approached in the context of the EU-funded LarKC project [32, 12].

DSMS and SPARQL systems already capitalize over at least a decade of research and development, therefore we have chosen to reuse existing technologies and systems for supporting C-SPARQL. However, the integration of DSMS and SPARQL systems is far from trivial, as it requires the automatic decomposition and transformation of C-SPARQL queries into suitable inputs for the two kinds of systems. By solving such challenge, we enable the development of an execution environment for C-SPARQL built on top of existing relational DSMS and SPARQL engines, using a plug-in approach which guarantees extensibility, portability, and good performance.

Thanks to a precise characterization of the C-SPARQL semantics, we map C-SPARQL queries to an internal model. We then use transformation methods in order to generate queries that distribute the work between DSMS and SPARQL engines. Transformations are inspired by classical re-

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

lation algebra optimizations, but we stress that rewritings occur prior to query decomposition, and therefore cannot be delegated to either kind of systems and must be explicitly managed outside them. After the transformation, a query can be answered by a suitable orchestration of SPARQL and DSMS engines, and each engine can further perform single or multi-query optimizations according to well known and established methods.

This paper is organized as follows. Section 2 presents C-SPARQL by introducing the new features relative to SPARQL, i.e. RDF stream data type, aggregates, windows management, and timestamps; their syntax is interleaved with Urban Computing examples. Then, Section 3 introduces the formal semantics of C-SPARQL as an extension of the SPARQL semantics defined by Perez et al. [26], and Section 4 presents an execution environment for C-SPARQL that leverages existing DSMS and SPARQL technologies. Section 5 describes a rule-based approach to query translation that optimizes the overall query performance. Finally, in Sections 6 and 7, related and future work, respectively, conclude the paper.

2. C-SPARQL

We present 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 [29] and the discussion of some examples of usage.

2.1 RDF Stream Data Type

C-SPARQL adds **RDF streams** to the data types supported by SPARQL.[2] An RDF stream is defined as an ordered sequence of pairs, where each pair is made of an RDF triple and its timestamp τ:

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

Timestamps can be considered as *annotations* of RDF triples; they are monotonically non-decreasing in the stream ($\tau_i \leq \tau_{i+1}$). They are not strictly increasing because timestamps 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 some positional order.

Example. In our example, taken from the Urban Computing scenario, data streams are associated with tollgates. In the streams every triple corresponds to the passing of a car through a tollgate. Each car is identified by means of its plate. The predicate of the triple (`t:registers`) is fixed, while the subject (`?tollgate`) and object (`?car`) parts of the triple are variable. Thus, a physical source for this stream has 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 the interpretation of C-SPARQL makes no specific assumptions nor requires restrictions on variable bindings relative to streaming triples. An example of stream with five cars passing through three different tollgates is given below.

[2]Similarly, the stream type has been introduced to extend relations in relational data stream management systems.

triple	Timestamp
`c:Distr1 t:registers "156"`	t_{100}
`c:Distr2 t:registers "75"`	t_{101}
`c:Distr1 t:registers "130"`	t_{102}
`c:Distr2 t:registers "95"`	t_{103}
`c:Distr3 t:registers "65"`	t_{104}

2.2 Aggregation

The official language specification of SPARQL does not include aggregation capabilities, and only some proprietary approaches[3] providing this functionality exist. Indeed, a continuous query language over streams without aggregates would not be practically useful. Therefore, our language extension includes aggregation as a fundamental characteristic of C-SPARQL.

Indeed, the addition of aggregation to the language is orthogonal w.r.t. the support for data streams, and the clauses that express aggregates can be syntactically and semantically added to SPARQL without referring to the presence of streams. This gives rise to a language extension which is fully autonomous and significant per se. Also, our extension is based on the conviction that in the context of RDF, knowledge should be extended rather than shrunk. Therefore, we propose to generate additional variable bindings and use them to annotate any existing variable binding that contributed to the aggregate value. This is in contrast to the conventional SQL grouping semantics that replaces all aggregated tuples with a single tuple representing the aggregate value. In this respect, our approach to aggregation is new, different from other existing ones, and more aligned with the baseline of the SPARQL semantics. Also, multiple independent aggregations are allowed within the same C-SPARQL query, with different grouping criteria and different partitions over the same set of bindings, thus pushing the aggregation capabilities beyond those of SQL. Aggregation clauses are added at the end of the query, and have the following syntax:

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

Every aggregation clause has the following three parts:

- The first part is a new variable (i.e., a variable not in the `WHERE` clause or in other aggregation clauses).
- The second part is an aggregation function (one of: `COUNT`, `MAX`, `MIN`, `SUM`, `AVG`); `COUNT` may have no argument, while the other functions take one of the variables occurring in the `WHERE` clause as argument.
- The third part is a set of one or more variables, which are chosen among those occurring in the `WHERE` clause. These variables express the grouping criteria.

Every clause may also have an optional fourth part, a `FILTER` clause.

The semantics of a query containing aggregates consists in adding to the regular variable bindings, computed by the `WHERE` clause, some new bindings, one for each of the new

[3]More details will be given in Section 6

variables introduced by the AGGREGATE clauses; the query result constructed in this way may be further filtered by a standard FILTER clause, which may refer to all the variables introduced in the WHERE and AGGREGATE clauses.

The evaluations of aggregate clauses are all independent from one another and take place after the computation of the bindings provided by the WHERE clause. We deliberately constrain C-SPARQL aggregates to use only the variables in the WHERE clause, and not other variables bound by other AGGREGATE clauses. This limitation is in line with the choice of keeping aggregations independent from one another: permitting a grouping clause to reference a variable bound within another grouping clause would introduce dependencies among the clauses.

Example. Given that aggregation does not depend on stream management, we show a query having aggregates but no streams. The query counts the number of sensors statically placed in the streets and returns the street and the number of sensors, filtering out those streets that have five or less sensors. The query is clearly not continuous.

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

SELECT DISTINCT ?street ?total-sensor
WHERE { ?sensor c:placedIn ?street . }
AGGREGATE {( ?total-sensor, COUNT, {?street} )
           FILTER (?total-sensor > 5)}
```

The query is executed as follows. First, all pairs of bindings of sensors with their street are extracted by matching the pattern in the WHERE clause against the triples in the repository. Then, the number of sensors located at each street is counted and bound to the new variable named ?total-sensor, and the resulting pairs that satisfy the filter condition are retained. Finally, distinct pairs of street and total sensors are projected.

2.3 Windows

The introduction of data streams in C-SPARQL requires the ability to *identify* such data sources and to specify *selection* criteria over them.

As 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.

As for *selection*, given that streams are intrinsically infinite, we introduce the notion of windows upon streams, whose types and characteristics are inspired by those of the windows in continuous query languages for relational streaming data, such as CQL[3].

Identification and selection are expressed in C-SPARQL by means of the FROM STREAM clause, whose 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 from the stream the *last* data stream elements, which are considered by the query. Such extraction can be *physical* (a given number of triples) or *logical* (all the triples which occur during a given time interval, the number of which is variable over time).

Logical windows are *sliding* [16] when they are progressively advanced of a given STEP (i.e. a time interval 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 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. A classic urban computing query counts the number of cars entering into the city center through all the tollgates; the query considers the last 10 minutes, while the sliding window is modified every minute.

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

SELECT DISTINCT ?tollgate ?passages
FROM STREAM <http://streams.org/citytollgates.trdf>
            [RANGE 10m STEP 1m]
WHERE { ?tollgate t:registers ?car . }
AGGREGATE { (?passages, COUNT, {?tollgate}) }
```

The query is executed as follows. First, all pairs of gates and cars are extracted from the current window over the stream, then the total number of cars for each gate is counted and bound to the new variable ?passages. Thus, every pair of bindings is extended with the number of passages it contributed to, and finally the bindings are projected as distinct pairs of tollgates and number of passages. The window considers all the stream triples in the last 10 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 result of the aggregation does not change during the slide interval, therefore also the query result does not change during the slide interval. It changes instead at every slide change.

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. Each C-SPARQL query is registered through the following statement:

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

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.[4]

Example. Assume that a (classic, static) RDF repository stores (a) the city districts, (b) the streets of each district, and (c) the tollgates located in each street. We now show a query that combines static knowledge (from the triples in the repository) and dynamic knowledge (from the streaming

[4]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.

triples) in order to periodically count how many cars have entered the city from each district in the last 30 minutes. In this example the window is sliding with a step of five minutes. From now on, the c: and t: prefixes will be omitted for brevity.

```
REGISTER QUERY CarsEnteringCityCenterPerDistrict
         COMPUTED EVERY 5m AS

SELECT DISTINCT ?district ?passages
FROM STREAM <http://streams.org/citytollgates.trdf>
            [RANGE 30m STEP 5m]
WHERE { ?tollgate t:registers ?car .
        ?tollgate c:placedIn ?street .
        ?district c:contains ?street . }
AGGREGATE {(?passages, COUNT, {?district})}
```

The query is executed as follows. As in the previous query, all pairs of bindings of tollgates with the car they register are extracted from the current window over the stream, and joined to a graph pattern used to extract from the RDF repository the pair of bindings of tollgates with their district. Then, the number of cars registered by the tollgates in each district is counted into the new variable `passages`. Finally, pairs of distinct districts and passages are projected.

2.5 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[5] can be registered as generators of RDF streams, as they produce RDF triples, associated with a timestamp as an effect of the query execution.

Example. The following example shows the construction of a new RDF data stream by means of the registration of a CONSTRUCT query. We consider again the previous example, and modify it so as to generate a stream:

```
REGISTER STREAM CarsEnteringCityCenterPerDistrict
         COMPUTED EVERY 5m AS

CONSTRUCT {?district t:has-entering-cars ?passages}
FROM STREAM <http://streams.org/citytollgates.trdf>
            [RANGE 30m STEP 5m]
WHERE { ?tollgate t:registers ?car .
        ?district c:contains  ?street .
        ?tollgate c:placedIn  ?street . }
AGGREGATE {(?passages,
             COUNT, {?district, ?tollgate, ?car})}
```

This query uses the same logical conditions as the previous one, but constructs the output in the format of a stream of RDF triples. Every query execution may produce from a minimum of one triple to a maximum of an entire graph, but the timestamp is always dependent on the query execution time only. Thus, in the former case, a different timestamp is assigned to every triple, while in the latter case the same timestamp is assigned to all the triples of a graph. In both cases timestamps are system-generated in monotonic non-decreasing order. Results of two evaluations of the previous query are presented in the table below.

triple	Timestamp
c:Distr1 t:has-entering-cars "100"	t_{400}
c:Distr2 t:has-entering-cars "75"	t_{400}
c:Distr1 t:has-entering-cars "130"	t_{401}
c:Distr2 t:has-entering-cars "95"	t_{401}
c:Distr3 t:has-entering-cars "65"	t_{401}

The first evaluation occurs at t_{400}. Suppose that only data from two sources (i.e., c:Distr1 and c:Distr2) are present in the window. Then, the evaluation generates two triples with the same timestamp (i.e., t_{400}).

The second evaluation occurs at t_{401}. Suppose that part of the data elaborated by the previous query are still in the window and that new data related to uc:Distr3 entered in the window. Then, the evaluation produces 3 triples; all of them have the same new timestamp t_{401}.

2.6 Multiple Streams

C-SPARQL queries can combine triples from more than one RDF stream, as shown in the next example.

Example. We now consider, in addition to tollgates, the presence of cameras as a second means of traffic control, placed on top of cross lights. Data from cameras flow within a second stream. We then consider a query in which cars seen by cameras or passing through tollgates are summed up, in order to return all the streets which have been full for more than 80% of their capacity in the last 5 minutes.

```
REGISTER QUERY FullStreets AS

SELECT ?street ?passages
FROM STREAM <http://streams.org/citytollgates.trdf>
            [RANGE 5m TUMBLING]
FROM STREAM <http://streams.org/citycameras.trdf>
            [RANGE 5m TUMBLING]
WHERE {
  ?street   c:hasCapacity ?capacity .
  {
  GRAPH <http://streams.org/citytollgates.trdf> {
    ?tollgate t:registers ?car .
    ?tollgate c:placedIn  ?street  . }
  }
  UNION
  {
  GRAPH <http://streams.org/citycameras.trdf> {
    ?camera   t:registers ?car .
    ?camera   c:placedAt  ?light .
    ?light    c:crossing  ?street . }
  }
}
AGGREGATE { ( ?passages, COUNT, {?street} )
          FILTER ( ?passages > (0.8 * ?capacity))}
```

The query is executed as follows. Pairs of bindings of tollgates and cars are extracted from the first graph, using a window over the tollgate stream, and from the second graph, using a window over the control camera stream. Also, the capacity of each street is extracted from the RDF static repository. The bindings are combined following the semantics of the UNION patter evaluation in SPARQL, and the new variable ?passages can count the cars registered by the tollgates and the cameras. Finally the streets that satisfy the filter predicate are selected, and distinct pairs of street and passages are projected.

[5]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 [28] for further explanations.

2.7 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 by pattern matching with an RDF triple of that stream.
- 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 now show a variant of the previous example. Now the goal is to detect all cars turning from one street (Palm Street) into another (Oak Avenue), by means of two cameras that are installed on the same traffic light. The query in C-SPARQL is the following:

```
REGISTER STREAM AllCarsTurningFromPalmIntoOak
         COMPUTED EVERY 1m AS

SELECT DISTINCT ?car1
FROM STREAM <http://streams.org/citycameras.trdf>
            [RANGE 5m STEP 1m]
WHERE { ?camera1 c:monitors c:Oak-Avenue .
        ?camera2 c:monitors c:Palm-Street.
        ?camera1 c: placedAt ?tr_light .
        ?camera2 c: placedAt ?tr_light .
        ?camera1 t: registers ?car1 .
        ?camera2 t: registers ?car2 .
        FILTER ( timestamp(?car1)>timestamp(?car2)
                 && ?car1 = ?car2 )}
```

Note that we use the two different variables (?car1 and ?car2) to refer to the same car, as stated in the FILTER clause. This is done in order to extract the two different timestamps and check that the car is first seen by ?camera1 and then by ?camera2. In this way, we only match cars that are actually turning in the specified direction, and not the other way round.

3. FORMAL SEMANTICS OF C-SPARQL

This section provides the formal semantics of C-SPARQL. In order to do this, we build on the work of Pérez et al. [26], and extend it with the formalization of aggregates, windows and the timestamp function. We address the reader to [26] for all the details and summarize here, for the sake of readability, the basic aspects of their formalization.

The semantics of a C-SPARQL query is formalized via the concept of mapping. We denote as I, B, L, V respectively the domains of IRIs, blank nodes, literals, and variables which are all disjoint. We also define $T = (I \cup B \cup L)$. A *mapping* μ is a partial function $\mu : V \rightarrow T$ which computes the bindings for all the variables of a query. This computation occurs when the *graph pattern* (denoted as P) in the query is matched against an RDF dataset (D). P is a set of triple patterns $t = (s, p, o)$ such that $s, p, o \in (V \cup T)$. We then define $dom(\mu)$ as the subset of V where μ is defined (i.e., the domain of μ), and $deg(\mu)$ as the cardinality of $dom(\mu)$.

Two mappings μ' and μ'' are said to be *compatible* if $\forall x \in dom(\mu') \cap dom(\mu'')$, then $\mu'(x) = \mu''(x)$.

Let Ω_1 and Ω_2 be sets of mappings. Then the basic operators for the composition of mappings are:

$$\Omega_1 \bowtie \Omega_2 = \{ \mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible } \}$$

$$\Omega_1 \cup \Omega_2 = \{ \mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2 \}$$

$$\Omega_1 \setminus \Omega_2 = \{ \mu \in \Omega_1 | \forall \mu' \in \Omega_2, \mu \text{ and } \mu' \text{ are not compatible } \}$$

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

The evaluation of a graph pattern P over a dataset D, is compactly as $[[P]]_D$, and is defined recursively, as follows:

1. $[[t]]_D = \{ \mu \mid dom(\mu) = var(t) \wedge \mu(t) \in D \}$, where t is a triple pattern and $var(t)$ is the set of variables occurring in t.
2. $[[(P_1 \text{ AND } P_2)]]_D = [[P_1]]_D \bowtie [[P_2]]_D$
3. $[[(P_1 \text{ OPT } P_2)]]_D = [[P_1]]_D \bowtie [[P_2]]_D$
4. $[[(P_1 \text{ UNION } P_2)]]_D = [[P_1]]_D \cup [[P_2]]_D$

3.1 Aggregates

We start by extending the binary operators (UNION, AND, OPT, and FILTER) with the new operator AGG (short for AGGREGATE). An *aggregation pattern* is denoted as $A(v, f, p, G)$, where v is the name of the new variable, f is the name of the aggregation function to be evaluated, p is the parameter of f, and G is the set of the grouping variables. We extend the evaluation of $[[P]]_D$ by adding a fifth rule to the above definition to deal with aggregation patterns:

5. $[[(P \text{ AGG } A)]]_D = [[P]]_D \bowtie [[A]]_D$, where P is a standard graph pattern and $A(v_a, f_a, p_a, G_a)$ is an aggregation pattern.

The evaluation of $[[A]]_D$ is defined by a mapping $\mu_a : V \rightarrow T$, where $dom(\mu_a) = v_a \cup G_a$; also, $deg(\mu_a) = deg(G_a) + deg(v_a) = deg(G_a) + 1$. This extension fully conforms to the notion of compatibility between mappings. Indeed, $v_a \notin dom(P)$ and, therefore, calling μ_p the mapping that evaluates $[[P]]_D$, μ_p and μ_a are compatible.

The result of the evaluation of μ produces a table of bindings, having one column for each variable $v \in dom(\mu)$. We can refer to a specific row in this table as $\mu_{(i)}$, and to a specific column as $\mu[v]$. The i-th binding of v is therefore $\mu_{(i)}[v]$.

The values to be bound to variable v_a are computed as
$\forall i \in [1, deg(\mu)]$, $\mu_{(i)}[v_a] = f_a(p_a, \mu[G_a])$

where $f(p_a, \mu[G_a])$ is the evaluation of the function $f_a \in$ (SUM, COUNT, AVG, MAX, MIN) with parameters p_a over the groups of values in $\mu[G_a]$. The set of groups of values in $\mu[G_a]$ is made of all the distinct tuples $\mu_{(i)}[G_a]$, i.e., the subset of the mapping $\mu[G_a]$ without duplicate rows.

3.2 Windows

We define an *RDF stream* as $R = \{(\langle subj, pred, obj \rangle , \tau) \mid \langle subj, pred, obj \rangle \in ((I \cup B) \times I \times (I \cup B \cup L)), \tau \in \mathbb{T}\}$ where $\mathbb{T}$ is the infinite set of timestamps. Note that triple patterns are enclosed in round brackets while triples are enclosed in angular brackets.

Operator	Meaning and Properties
Stream	Accesses an RDF Stream identified by its IRI.
Window	Breaks down an RDF Stream using a window ω and returns an RDF graph.
Aggregation	Based on an input set of variable bindings, computes the aggregate values given by $A(v, f, p, G)$.
Filter	Based on an input set of variable bindings, filters out the bindings that do not match the filtering condition R.

Table 1: Additional SQGM operator types

A *logical* window is defined as:
$\omega_l(R, t_i, t_f) = \{(\langle s, p, o\rangle, \tau) \in R \mid t_i < \tau \leq t_f\}$
Let $c(R, t_i, t_f)$ be a function which counts the items in R which have timestamp in the range $(t_i, t_f]$.
$c(R, t_i, t_f) = \mid \{(\langle s, p, o\rangle, \tau) \in R \mid t_i < \tau \leq t_f\} \mid$
A *physical* window is defined as:
$\omega_p(R, n) = \{(\langle s, p, o\rangle, \tau) \in \omega_l(R, t_i, t_f) \mid c(R, t_i, t_f) = n\}$
A window ω can be *sliding*, with *range* ρ and *step* σ. For logical windows, ρ and σ take the form of a time interval. Logical windows (a) contain the most recent triples in a time interval of length ρ; and (b) are evaluated with frequency $1/\sigma$. For physical windows, ρ and σ are integers. Physical windows (a) contain the last ρ triples; and (b) are evaluated whenever σ new triples arrive in the stream. A window is said to be as *tumbling* with range ρ if it is sliding with range ρ and step $\sigma = \rho$.

3.3 Timestamp Function

A variable v can occur multiple times in a graph pattern P. When P is matched against D, v gets as many bindings as are its occurrences in P. Some of these bindings may derive from static data, others from streaming data. Each of the bindings coming from the stream R is characterized by the timestamp of the triple that matches one of the triple patterns $t \in P$ such that $v \in dom(t)$. We denote the set of timestamps associated with a variable by a triple pattern t as $TS_{set}(v, t)$ and the set of all timestamps associated with the variable by a graph pattern P as
$TS_{set}(v, P) = \{\tau \mid t \in P \wedge v \in dom(t) \wedge \tau \in TS_{set}(v, t)\}$
We can now define the timestamp function
$ts(v, P) = max(TS_{set}(v, P))$
which returns the *highest* timestamp associated with v among all bindings of v in P. The timestamp function returns a value only if v has been matched at least once over a triple $\langle s, p, o\rangle \in R$, $\perp$ otherwise.

3.4 Visual Representation

The operational semantics presented in this section is the basis for the visual query representation which is called the *Operator Graph* or *O-Graph* since its nodes correspond to operators. O-Graphs are used for both query evaluation and optimization and are, thus, the basis for Sections 4 and 5, respectively.

The definition of O-Graphs is based on the SPARQL Query Graph Model (SQGM) [20] which in turn is based on the Query Graph Model (QGM) [27]. A SQGM is a directed labeled graph with vertices representing operators and edges capturing the flow of data. In contrast to SQGM, however, that uses a proprietary notation to describe operators, we propose to use in the nodes the extended formal semantics of Pérez et al. [26] introduced in Section 3.

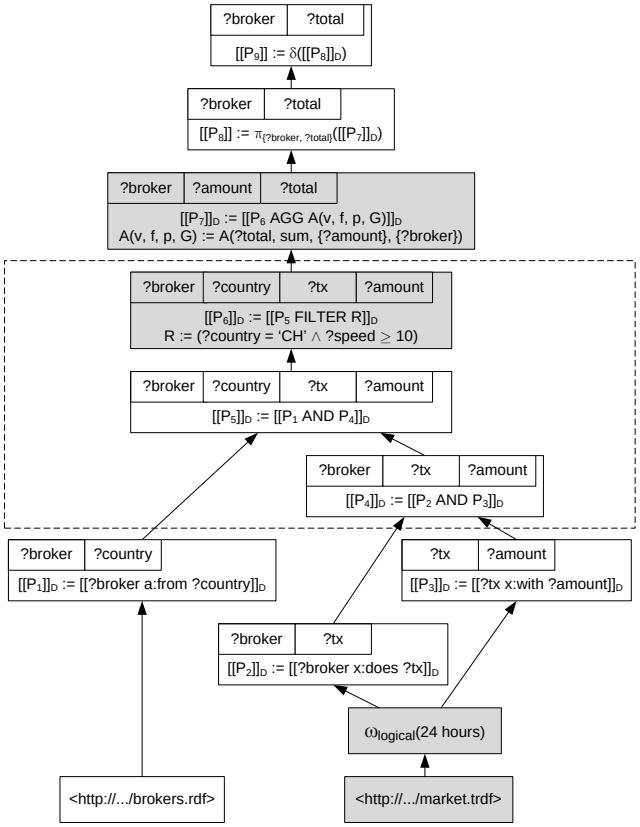


Figure 1: O-Graph of the example query

In order for SQGM to serve as a representation for all C-SPARQL queries, the set of operator types defined in [20] is extended with the additional operators given in Table 1. We also propose to represent the SPARQL filter clause as a node on its own instead of as an attribute of the graph pattern operator node as suggested in [20]. This modification of SQGM is necessary because filter clauses can also be used in C-SPARQL within the AGGREGATE clause, which is independent of the WHERE clause.

To illustrate the construction of O-Graphs, we use an example query computing the daily sum of all transactions of at least 10 Euros done by Swiss brokers. In C-SPARQL:

```
REGISTER QUERY TotalAmountPerDayAndBroker AS

PREFIX b: <http://brokerscentral.org/accounts#>
PREFIX x: <http://stockexchange.org/exchanges#>

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

The O-Graph corresponding to the example query is shown

in Figure 1[6]. Since it is out of the scope of this paper to present the algorithm translating queries into O-Graphs, we refer the reader to [20] and limit the discussion to the issues specific to C-SPARQL. In the figure, the nodes introduced to support C-SPARQL are shaded in gray. The FROM STREAM clause of the example query is represented in the O-Graph as a *stream operator* followed by a *window operator*, as shown in the lower right-hand corner of the figure. The FILTER clause translates to a *filter operator*, whereas the AGGREGATE clause is represented by an *aggregation operator*.

4. EXECUTION ENVIRONMENT

An important contribution of this paper is the execution framework that we propose for C-SPARQL. With more than a decade of experience, there are highly optimized solutions for processing continuous queries over relational streams. Taking this into account, our approach is based on a plug-in architecture that leverages existing technology. Given that currently available stream sources do not manage RDF data streams—but rather relational streams—we investigated how the requirements of C-SPARQL can be covered with relational technologies, such as STREAM [2], Aurora/Borealis [1] and Stream Mill [5]. The experiments discussed in this paper were conducted with STREAM.

Figure 2 shows the architecture of the proposed framework which relies entirely on existing technologies. Whereas the SPARQL reasoner plug-in is used to evaluate the static part of the query, an existing relational data stream management system to evaluate both streams and aggregates. Note that this approach is only feasible, if aggregations can be performed by the DSMS. A parser parses the C-SPARQL query and hands it over to the orchestrator. The orchestrator is the central component of our approach and translates the query in a static and dynamic part. The static query is used to extract the static knowledge from the reasoner, while the dynamic query is registered in the DSMS. Note that this process is executed only once when a C-SPARQL query is registered as the continuous evaluation is handled consequently by the DSMS.[7]

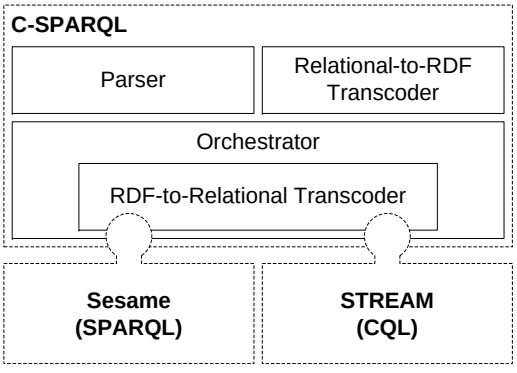


Figure 2: Architecture overview

When translating C-SPARQL queries into SPARQL and CQL, the orchestrator relies on the information captured by

[6]The purpose of the dashed box in the middle of the O-Graph of Figure 1 will be explained in Section 5.

[7]Due to space limitations, we omit the discussion of updates to the static knowledge base of the reasoner.

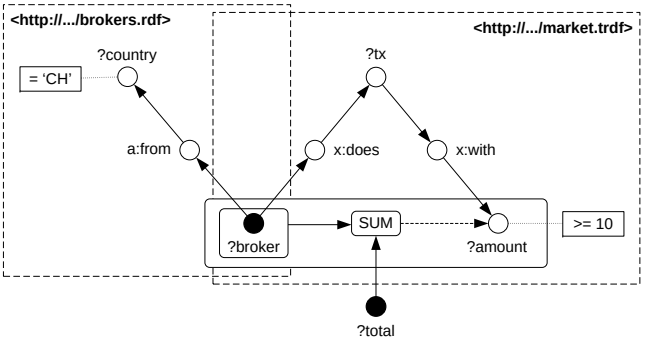


Figure 3: D-Graph of the example query

the so-called *Denotational Graph* or *D-Graph* to distinguish static from streaming knowledge. The D-Graph is defined as a view on the O-Graph and is constructed using the following algorithm based on the formalism used in [26].

1. Each variable $v \in V$, IRI $i \in I$ or literal $l \in L$ used in operator nodes of the O-Graph is a vertex of the D-Graph.
2. Each triple pattern (s, p, o) occurring in graph pattern operator nodes is represented as two directed edges (s, p) and (p, o) connecting the vertices of the graph.
3. Assuming conjunctive semantics, each filtering condition (generically referred to as R in the O-Graphs) appearing in a filter operator node is represented in the D-Graph as annotations on the corresponding vertices.
4. Each aggregation operator node of the O-Graph is represented as a "hyper-node" in the D-Graph containing another hyper-node for the group G that is linked to the aggregation function f. The aggregation function in turn is linked to the node representing the parameter p. Finally, the aggregation function is bound to the variable node v.
5. Any variable names occurring in a projection operator node of the O-Graph are represented in the D-Graph by filling the corresponding vertices.
6. Graph and stream operator nodes in the O-Graph are represented as dashed rectangles in the O-Graph, grouping the vertices that emerge from each source.

The D-Graph obtained by applying this algorithm to the O-Graph shown in Figure 1 is given in Figure 3.

Rather than a formal definition of transformations, we prefer to provide their description using the example given in Section 3.4. However, the method generalizes to arbitrary C-SPARQL queries, based on the partitioning of the graph into static and streaming nodes. An overview of this general query evaluation process given in Figure 4.

Referring back to the example of the previous section, the following query corresponds to the graph operator and graph pattern operator nodes located in the lower left branch of the O-Graph in Figure 1:

```
PREFIX b: <http://brokerscentral.org/accounts#>

SELECT DISTINCT ?broker ?country
FROM <http://brokerscentral.org/brokers.rdf>
WHERE { ?broker b:is_from ?country }
```

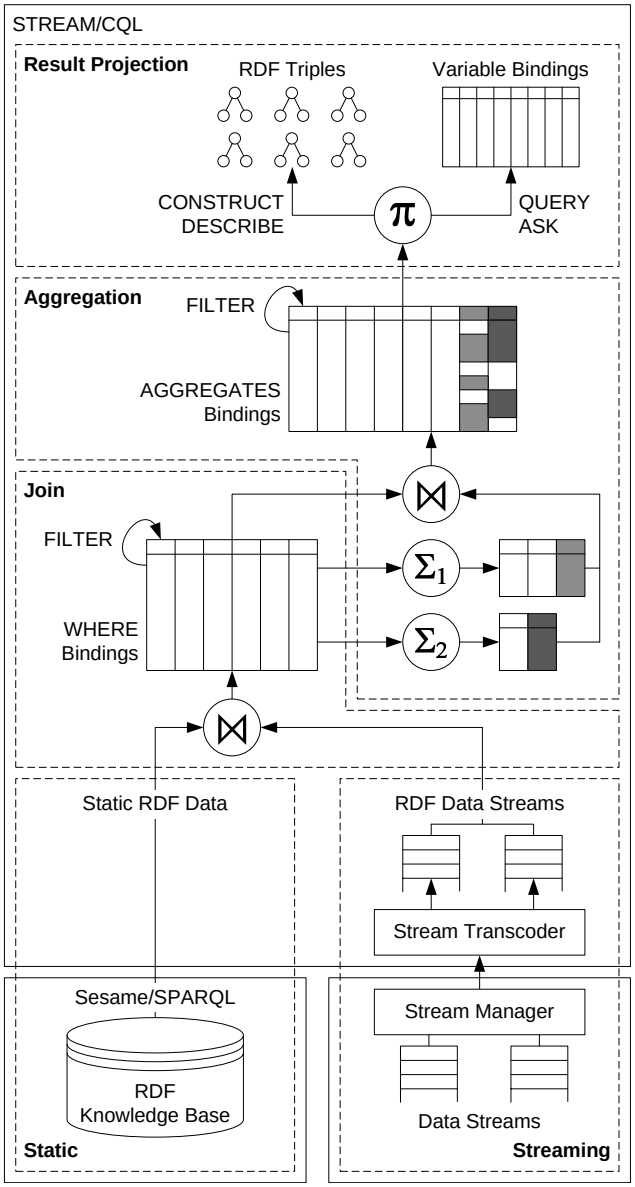


Figure 4: Query evaluation process

The variables bindings returned by this query are then translated into a relation and materialized within the DSMS. The statements required to do so are:

```
CREATE TABLE static ( broker    VARCHAR(32),
                      country   VARCHAR(32),
                      PRIMARY KEY(broker, country)
)
```

```
INSERT INTO static
execute_sparql("SELECT ?broker ?country
                WHERE { ... }")
```

Next, we will discuss the rewriting used to transform C-SPARQL queries into CQL queries. Again, the D-Graph is used to map the the RDF graph patterns to the schema of the underlying relational stream. For our example, we assume that this schema is given by:

market.trdf(broker: integer, tx: integer, amount: integer)

Then the following CQL statement corresponds to the streaming part of the example C-SPARQL query represented by the nodes in the lower right branch of the O-Graph. Note that the stream operator and window operator nodes of the O-Graph map to CQL features rather nicely, and therefore this translation is straightforward.

```
CREATE VIEW streaming AS
SELECT *
FROM <http://stockexchange.org/market.trdf> [24 hours]
```

As a next step, the join operator node of the O-Graph that combines the static and the streaming knowledge of the query has to be evaluated. To do so, we create a *comprehensive view* that corresponds to the bindings of the WHERE clause. At the same time, we use the view comprehensive to also evaluate the filter operator node following the join operator node in the O-Graph. The SPARQL FILTER clauses are computed by translating them into SQL/CQL WHERE clauses.

```
CREATE VIEW comprehensive AS
SELECT s.broker AS broker, country, tx, amount
FROM static s, streaming
WHERE s.broker = streaming.broker &&
      country = 'CH' && amount >= 10
```

The last step of the query evaluation is the computation of the aggregations specified in the C-SPARQL query. In C-SPARQL, the semantics of aggregation is different than in SQL and, incidentally, also CQL. Therefore, C-SPARQL AGGREGATE clauses cannot be directly translated into an SQL aggregation function together with a GROUP BY statement. The main difference between C-SPARQL and SQL is that in C-SPARQL, aggregation does not reduce the cardinality of the result set, whereas the SQL/CQL GROUP BY operation has this characteristic.However, the desired behavior can easily be emulated in CQL by computing the aggregation in a separate view and then using an outer join to "add a column" with the aggregated values to the comprehensive relation given above. The following two SQL/CQL statements evaluate the aggregation operator node of the example O-Graph according to these semantics.

```
CREATE VIEW arregation1 AS
SELECT broker, SUM(amount) AS total
FROM filtered
GROUP BY broker

CREATE VIEW result AS
SELECT *
FROM comprehensive c
        LEFT OUTER JOIN aggregation1 a1
        ON c.broker = a1.broker
```

The last two nodes of the O-Graph—the select result operator[8] and solution modifier operator nodes following the aggregation operator node—are evaluated by a final query consisting simply of a projection over the variables broker and total from the view result. If the query is registered at the STREAM/CQL environment, its continuous output is then produced.

5. OPTIMIZATIONS AND EVALUATION

Several transformations can be applied to the O-Graph, some recalling well known results from classical relational

[8]The name of this node is based on [20]. We would prefer "projection operator node".

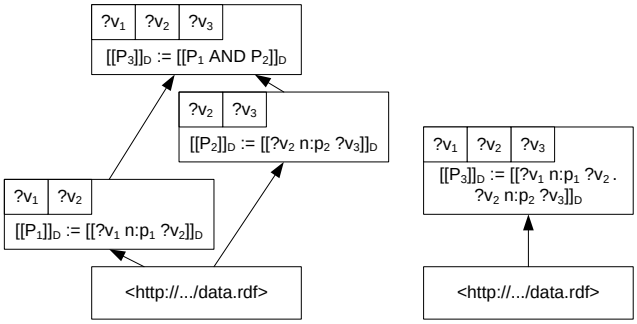


Figure 5: Application of the MergeJoinedGPOs transformation rule

optimization [31] and some being more specific to the domain of streams. More specifically, we first address the push of FILTERs and projections, analogous to those of the relational algebra, and then address the push of aggregates.

For C-SPARQL we have identified the following types of transformations:

5.1 Push of filters and projections

For the sake of briefness, we address the reader to [31] for a complete description of the rewriting rules that push selections and projections as near to the data sources as possible. In our data model duplicates are allowed, and therefore projections can be pushed with no restrictions.

We now exemplify such transformations on the query discussed in Section 3.4. For readability, we collapse the conjunction of several patterns into one node, thus applying the MergeJoinedGPOs transfomation rule, as defined in [20] and shown in Figure 5. The optimized version of the O-Graph of Figure 1 is shown in Figure 6, limited to the part in the dashed box. Two new projection blocks have been introduced right above the topmost occurrences of the variables that have been projected away (namely `?broker` and `?tx`). The two FILTER conditions have been pushed down to the first occurrences of the variables they apply to (`?amount` and `?country`). In the Figure, the only filtering block of the non-optimized version has been split into two blocks, also represented in gray in the optimized version. Note that one filter applies to static knowledge while the other one applies to streaming data.

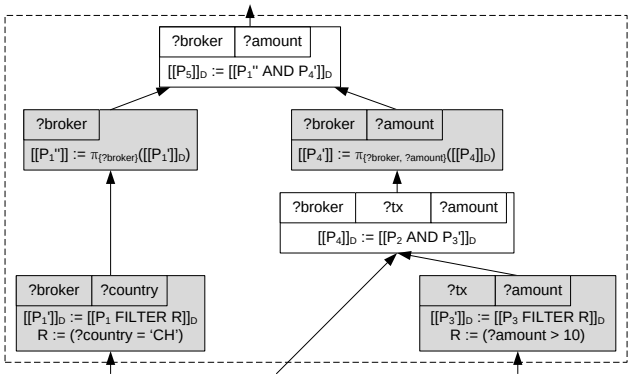
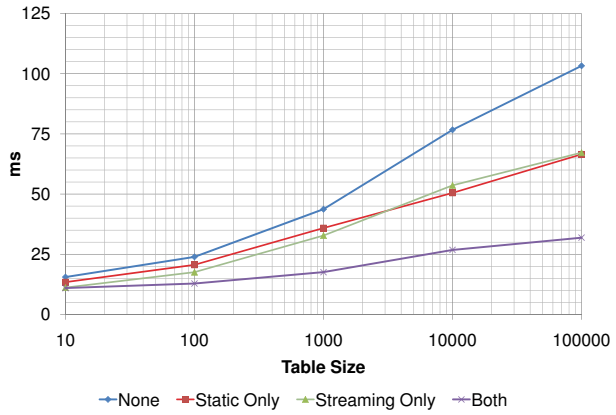
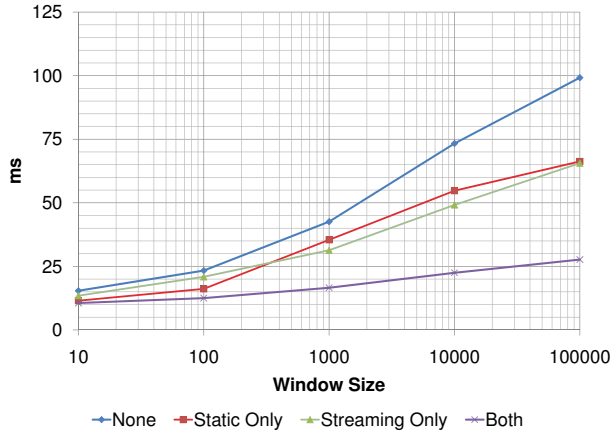


Figure 6: Optimized O-Graph of the example query

We have run experiments in order to evaluate the effect of pushing the FILTERs on the respective data sources, the results of which are shown in Figure 7. On both experiments, which respectively stress the size of the window and the size of the data set, the gain from the transformations is perceivable, and their effects sum up independently. At first glance these results may be surprising, as one could think that a relational engine should be able to perform push optimization by itself. However there is one peculiar aspect in the way in which the query is rewritten. The transformation occurs in C-SPARQL and depends on the distinction between static and streaming data. The ad-hoc architecture we designed is capable of applying algebraic transformations directly on the C-SPARQL query and then decompose it, so as to take advantage of the natural distribution of data.



(a) Increasing window size



(b) Increasing size of the data set

Figure 7: Performance results of pushing filters

5.2 Push of aggregates

Aggregate functions within a query, are all independent one from another and can be separated into orthogonal groups w.r.t. the streams they refer to (aggregates such that the union of the grouping variables and the argument of aggregation belongs exactly to one subset of the query streams). Aggregates can then be further classified as:

1. *totally distributable*: an aggregate such that the argument of aggregation and all the grouping variables belong to one or more streams, but each possible binding

value for the grouping variables belongs exactly to one stream. Then, the functions are distributed to streams and the query returns the union of their results. An example is the sum of amounts of transactions, grouped by sellers in each market, when each stream is associated with one market.

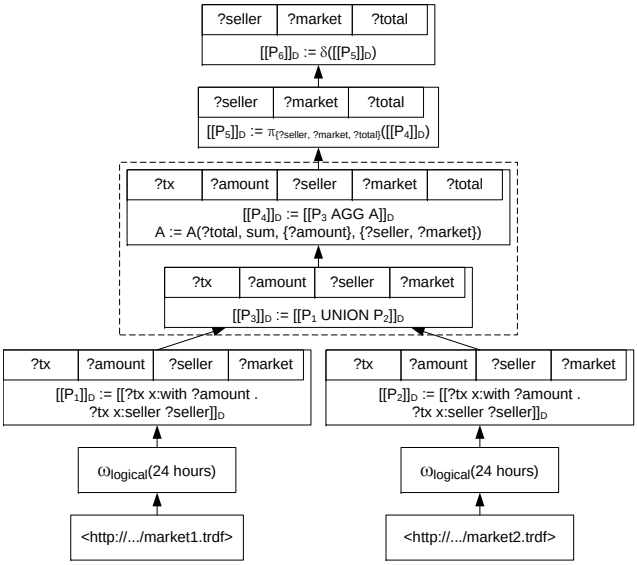
2. *partially distributable*: an aggregate such that the argument of the aggregation and only a subset of the grouping variables belong to one or more streams, and there exists a distributed computation of the aggregate function over the streams in the sense of [8] over fragments of relations. An example is the computation of sum of amounts of transactions grouped by seller, over all markets, when each stream is still associated with one market. In such case, each stream only provides a partial sum by seller and market, and then the sum of the sums—over all markets—yields to the sum of amounts for each seller.

3. *not distributable*: an aggregate such that neither of the above cases holds. An example is the count of distinct amounts of the various transactions.

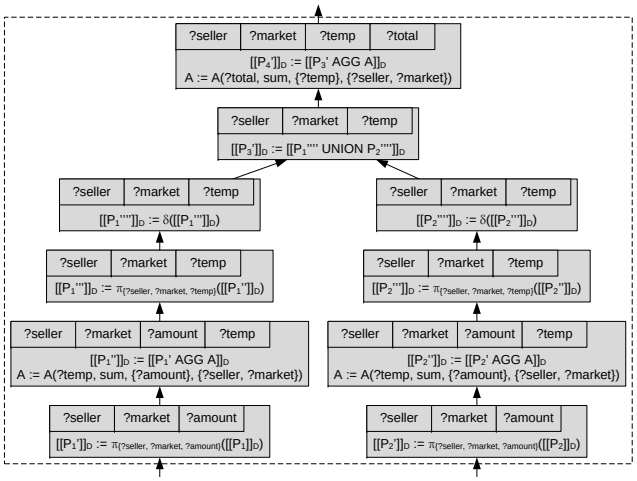
An example of a query in class 1 follows, while its initial and optimized O-Graphs are in Figure 8.

```
REGISTER QUERY SellerMarketMovementsPerDay AS

PREFIX x: <http://stockexchanges.org/exchanges#>

FROM <http://brokerscentral.org/brokers.rdf>
FROM STREAM <http://stockexchange.org/market.trdf>
            [RANGE 24 hours TUMBLING]

SELECT DISTINCT ?seller, ?market, ?Tot
WHERE {
     GRAPH ?market {
        ?Transaction x:seller ?seller .
        ?Transaction x:with ?amount .
     }
}
AGGREGATE { (?tot, SUM(?amount), {?seller, ?market}) }
```

Distributing the computation of aggregates over streams always yields to an increase of performance and therefore should be performed when possible. Moreover, the gain is high when the DISTINCT modifier is specified in SELECT clause, because it can be applied to intermediate results coming from the computation of aggregates over each stream.



(a) After application of MergeJoinedGPOs



(b) After pushing aggregates towards streams

Figure 8: O-Graphs of the second example query

6. RELATED WORK

This section illustrates previous work on the SPARQL language and then on data streams.

6.1 SPARQL

The most authoritative source about the syntax and semantics of the SPARQL language is the W3C recommendation [28].

Cyganiak [11] presents a relational model of SPARQL. The author uses relational algebra operators (join, left outer join, projection, selection, etc.) to model the SPARQL SELECT clauses. A translation system between SPARQL and SQL is outlined. The system extensively resorts to the use of COALESCE and IS NULL in order to express some SPARQL features. Harris [19] presents an implementation of SPARQL queries over a relational database engine. The use of relational algebra operators is similar to that of [11].

In [17], Gutierrez et al. discuss the semantics and the computational complexity of a conjunctive query language for RDF with basic patterns, which is a formal and unambiguous basis for defining the semantics of SPARQL queries evaluation. In [26], Perez et al. consider simple RDF graphs (without special semantics for literals) and a simplified version of filters; these assumptions allow them to provide a compositional semantics, to prove that there is a normal form in which, under certain constraints over variable bindings, a wide range of queries can be expressed, to fix some complexity bounds, and to discuss optimization opportunities. Haase et al. [18] present a comparison of functionalities of pre-SPARQL query languages, many of which gave inspirations to the definition of SPARQL.

A previous attempt to extend SPARQL to support data streams has been presented by Bolles et al. [7]. Their paper represents an antecedent of our work as it introduces 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 the work presented in this paper in several key aspects. First, Bolles et al. simply introduce RDF streams as a new data type, and omit essential ingredients, such as aggregate and timestamp functions. With these limitations, the resulting expressive power is not sufficient to express most practical queries. 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, Bolles et al. have chosen to change the standard SPARQL operators by making them timestamp-aware and, thereby, effectively introduce a new language semantics. Finally, their approach allows window clauses to appear within SPARQL group graph pattern expressions. On the one hand, this makes the query syntax more intricate, as window clauses can appear in multiple places. On the other hand, it complicates query evaluation. Since window operations are no longer required to be at the leaves of the query tree, they need to be interleaved with standard SPARQL operations, possibly interfering with the separation of concerns between stream management and query evaluation.

Even though the SPARQL specification contains no aggregates definition, several implementations support some forms of aggregation functions and group definitions. OpenLink Virtuoso[9] supports COUNT, COUNT DISTINCT, MAX, MIN and AVG, with implicit grouping criteria. ARQ[10] supports COUNT and COUNT DISTINCT over groups defined through an SQL-like GROUP BY clause. ARC[11] also supports the keyword AS to bind variables to aggregated results.

In [30], the authors study how aggregation and grouping 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. Their approach is different from ours w.r.t. both the semantics and the syntax of the proposed extension of SPARQL. More specifically, the extension proposed by Seid and Mehrotra changes the semantics of the SPARQL SELECT clause, while in C-SPARQL all new bindings are defined by the AGGREGATE clauses.

In [33], the authors describe an approach to reasoning over streaming facts. Their work is complementary to ours, as they focus on the scalability of reasoning techniques, rather than on query decomposition, query processing, and stream management.

6.2 Data Streams

One of the first proposed models for data streams was the Chronicle data model [21]. It introduced the concept of chronicles as append-only ordered sequences of tuples, together with a restricted view definition language and an algebra that operates over chronicles as well as over traditional relations. OpenCQ [24] and NiagaraCQ [9] addressed continuous queries for monitoring persistent data sets spread over wide-area networks. Another data stream management system is Aurora [6], which in turn evolved into the Borealis project [1], which addresses distribution issues.

In [4], Babu et al. tackle the problem of continuous queries over data streams addressing semantic issues as well as efficiency concerns. They specify a general and flexible architecture for query processing in the presence of data streams. This research evolved into the specification and development of a query language tailored for data streams, named CQL [2, 3]. Further optimizations are discussed in [25].

Another stream of research was developed by Law et al. [22], putting particular emphasis on the problem of mining data streams [23]. Another project that addresses data mining issues is the Stream Mill project [5], which extensively considered the problem of data aggregation. Its query language (ESL) efficiently supports physical and logical windows (with optional slides and tumbles) on both built-in aggregates and user-defined aggregates. The constructs introduced in ESL extend the power and generality of DSMSs.

The problem of processing delays is one of the most critical issues and at the same time a strong quality requirement for many data stream applications, since the value of query results decreases dramatically over time as the delays sum up. In [10], the authors address the problem of keeping delays below a desired threshold in situations of overload, which are common in data stream systems. The framework described in the paper is built on top of the Borealis platform.

As for the join over data streams, rewriting techniques are proposed in [15] for streaming aggregation queries, studying the conditions under which joins can be optimized and providing error bounds for results of the rewritten queries. The basis of the optimization is a theory in which constraints over data streams can be formulated and the result error bounds are specified as functions of the boundary effects incurred during query rewriting.

7. CONCLUSION

In this paper we addressed the optimization of the execution of stream reasoning queries; our plug-in architecture capitalizes on the use of existing DSMS and SPARQL engines, whose optimized orchestration exhibits ideal performance. Our experiments have been performed using Sesame and STREAM as representative DSMS and SPARQL engines, but the approach is general and as such can be ported to different component systems.

In our future work, we intend to focus on the optimal deployment of multiple continuous queries over streams in distributed and heterogeneous environments, where RDF repositories and data streams will be managed by different systems, and stream managers may exhibit limited data management capabilities. We believe that generalizing some of the results presented in this paper in a multi-query, heterogeneous, and distributed context is possible, although far from trivial.

8. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. Intl. Conf. on Innovative Data Systems Research (CIDR 2005)*, 2005.

[2] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: The Stanford Stream Data Manager (Demonstration Description). In *Proc. ACM Intl. Conf. on Management of Data (SIGMOD 2003)*, page 665, 2003.

[3] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations

[9] http://virtuoso.openlinksw.com/

[10] http://jena.sourceforge.net/ARQ/

[11] http://arc.semsol.org/

and Query Execution. *The VLDB Journal*, 15(2):121–142, 2006.

[4] S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Rec.*, 30(3):109–120, 2001.

[5] Y. Bai, H. Thakkar, H. Wang, C. Luo, and C. Zaniolo. A Data Stream Language and System Designed for Power and Extensibility. In *Proc. Intl. Conf. on Information and Knowledge Management (CIKM 2006)*, pages 337–346, 2006.

[6] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on Aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[7] A. Bolles, M. Grawunder, and J. Jacobi. Streaming SPARQL – Extending SPARQL to Process Data Streams. In *Proc. Europ. Semantic Web Conf. (ESWC 2008)*, pages 448–462, 2008.

[8] S. Ceri and G. Pelagatti. Correctness of Query Execution Strategies in Distributed Databases. *ACM Trans. Database Syst.*, 8(4):577–607, 1983.

[9] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proc. ACM Intl. Conf. on Management of Data (SIGMOD 2000)*, pages 379–390, 2000.

[10] Y. cheng Tu, S. Liu, S. Prabhakar, and B. Yao. Load Shedding in Stream Databases: A Control-based Approach. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB 2006)*, pages 787–798, 2006.

[11] R. Cyganiak. A Relational Algebra for SPARQL. Technical report, HP-Labs.

[12] D. Fensel, F. van Harmelen, B. Andersson, P. Brennan, H. Cunningham, E. D. Valle, F. Fischer, Z. Huang, A. Kiryakov, T. K. il Lee, L. School, V. Tresp, S. Wesner, M. Witbrock, and N. Zhong. Towards LarKC: a Platform for Web-scale Reasoning . In *Proc. IEEE Intl. Conf. on Semantic Computing (ICSC 2008)*, 2008.

[13] 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.

[14] 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 *Proc. Intl. Conf. on Internet Measurement (IMC 2003)*, pages 173–178, 2003.

[15] L. Golab, T. Johnson, N. Koudas, D. Srivastava, and D. Toman. Optimizing Away Joins on Data Streams. In *Proc. Intl. Workshop on Scalable Stream Processing System (SSPS 2008)*, pages 48–57, 2008.

[16] L. Golab and M. T. Özsu. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB 2006)*, pages 500–511, 2003.

[17] C. Gutierrez, C. Hurtado, and A. O. Mendelzon. Foundations of Semantic Web Databases. In *Proc. ACM Symp. on Principles of Database Systems (PODS 2004)*, pages 95–106, 2004.

[18] P. Haase, J. Broekstra, A. Eberhart, and R. Volz. A Comparison of RDF Query Languages. In *Proc. Intl. Semantic Web Conf. (ISWC 2004)*, pages 502–517, 2004.

[19] S. Harris. SPARQL Query Processing with Conventional Relational Database Systems. In *Proc. Intl. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2005)*, pages 235–244, 2005.

[20] O. Hartig and R. Hesse. The SPARQL Query Graph Model for Query Optimization. In *Proc. Europ. Semantic Web Conf. (ESWC 2007)*, pages 564–578, 2007.

[21] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View Maintenance Issues for the Chronicle Data Model. In *Proc. ACM Symp. on Principles of Database Systems (PODS 1995)*, pages 113–124, 1995.

[22] Y.-N. Law, H. Wang, and C. Zaniolo. Query Languages and Data Models for Database Sequences and Data Streams. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB 2004)*, pages 492–503, 2004.

[23] Y.-N. Law and C. Zaniolo. An Adaptive Nearest Neighbor Classification Algorithm for Data Streams. In *Proc. Europ. Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD 2005)*, pages 108–120, 2005.

[24] 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.

[25] K. Munagala, U. Srivastava, and J. Widom. Optimization of Continuous Queries with Shared Expensive Filters. In *Proc. ACM Intl. Symp. on Principles of Database Systems (PODS 2007)*, pages 215–224, 2007.

[26] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. In *Proc. Intl. Semantic Web Conf. (ISWC 2006)*, pages 30–43, 2006.

[27] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proc. ACM Intl. Conf. on Management of Data (SIGMOD 1992)*, pages 39–48, 1992.

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

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

[30] D. Seid and S. Mehrotra. Grouping and aggregate queries over semantic web databases. *International Conference on Semantic Computing*, 0:775–782, 2007.

[31] J. M. Smith and P. Y.-T. Chang. Optimizing the performance of a relational algebra database interface. *Commun. ACM*, 18(10):568–579, 1975.

[32] E. D. Valle, S. Ceri, D. F. Barbieri, D. Braga, and A. Campi. A First Step Towards Stream Reasoning. In *Proc. of the Future Internet Symposium (FIS 2008)*, 2008.

[33] O. Walavalkar, A. Joshi, T. Finin, and Y. Yesha. Streaming Knowledge Bases. In *Proc. Intl. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2008)*, 2008.