Part II:
Linked Stream Data Processing - Building a Processing Engine

Danh Le-Phuoc, Josiane X. Parreira, and Manfred Hauswirth
DERI - National University of Ireland, Galway

Reasoning Web Summer School 2012
Outline

- **Part I: Basic Concepts & Modeling (Josi)**
  - Linked Stream Data
  - Data models
  - Query Languages and Operators
  - Choices/Challenges when designing a Linked Stream Data processor

- **Part II: Building a Linked Stream Processing Engine (Danh)**
  - Analysis of available Linked Stream Processing Engines
    - Design choices, implementation
    - Performance comparison
    - Open Challenges
“Why should I care?”

- As an application developer
  - What can I do with it?
  - How does it work?
  - How to choose one?

- As a processing engine developer
  - How to build one?
  - How to build/identify a better one?

- As a researcher
  - What have been done? What left?
  - Is there any interesting research problem?
  - How to find room to improvement?
Why a continuous query processing engine is needed?

Separation of concerns

- Focus of application logic
- Let the experts deal with data operations on streams
- Minimize the learning efforts
  - Learn simple APIs using the engine
  - Learn a simple query language
How a stream-based application is built with a stream processing engine?

1. Initialize the engine (less than 5 lines of code)

2. Write and register the queries to the engine (1 line for 1 query)

3. Write codes for wiring output streams to the application logic (depends on the application logic but the each wiring code snippet ≈5 lines of code)

4. Connect input streams to engine (1-3 lines for each stream)

Just 10-20 lines of code!!!
What need to be done to build a processing engine?

- **Data model**: relational, object-oriented, etc
- **Query model**:
  - Logical operators: sliding windows, relational algebras
  - Query language: CQL, C-SPARQL, CQELS, etc
- **Build a processing engine**
  - Handling input streams
  - Implement the execution engine
  - Schedule the executions
  - Optimization
Building blocks of a query processing engine

- Query
- Optimizer
- Executor
  - Access methods
    - database
    - datastream
  - Operator implementations
  - Execution
  - database
  - datastream
Algorithms/technologies/solutions for stream processing

- Handling live and push-based data stream sources
  - Time management
  - Load shedding for bursty streams

- Operator implementation for execution engine
  - Data structure and physical storage
  - Handling the new stream elements/expired ones
  - Incremental execution
  - Memory overflow

- Optimization

- Scheduling
Handling input streams

The main contributions of this paper are:

1. We describe how to estimate tuple transmission latencies from general enough to be applicable in a wide variety of environments.
2. Our algorithm is simple, and it can move all tuples with timestamps greater than a given threshold.
3. We demonstrate our approach on example data stream processors.

The main contributions of this paper are:

In general, each tuple must eventually be moved from the source to the query processor. For example, consider a sensor application and suppose there are two separate streams for temperature and pressure readings. Further, suppose that the sensors are programmed to generate a tuple when a reading occurs.

For the case when parameter values cannot be specified in advance, we describe how they may be estimated. Our approach to heartbeat generation is to quantify certain properties of the environment as bursty rate, skew, and latency.

Heartbeat generation algorithm (Section 5).

We discuss implementation and scalability issues of our approach to heartbeat generation (Section 6).

We formalize the problem and define the parameters to the query plans for execution (Section 7).

We demonstrate our approach on example data stream processors (Section 8).

Finally, we survey related work in Section 9.

Figure 1 depicts an abstraction of the environment we consider. Continuous queries (CQ) are registered with the DSMS. These are executed over input data sources to the DSMS over a network which may have some transmission latency, upper-bounded by τ.

Heartbeat τ

Strictly temporal order

Unordered stream elements

Buffered tuples > τ

Stream arrival

Strictly temporal order

Load shedding for bursty rate

Network

Stream emission

DSMS

Query Processor

Query Plans

Query

Processor

CQ₁

CQ₂

CQₘ

Input Manager

Tuples ≤ τ

Buffered tuples > τ

Tuple transmission latencies from general enough to be applicable in a wide variety of environments.

Our algorithm is simple, and it can move all tuples with timestamps greater than a given threshold.

We demonstrate our approach on example data stream processors.

The main contributions of this paper are:

In general, each tuple must eventually be moved from the source to the query processor. For example, consider a sensor application and suppose there are two separate streams for temperature and pressure readings. Further, suppose that the sensors are programmed to generate a tuple when a reading occurs.

For the case when parameter values cannot be specified in advance, we describe how they may be estimated. Our approach to heartbeat generation is to quantify certain properties of the environment as bursty rate, skew, and latency.

Heartbeat generation algorithm (Section 5).

We discuss implementation and scalability issues of our approach to heartbeat generation (Section 6).

We formalize the problem and define the parameters to the query plans for execution (Section 7).

We demonstrate our approach on example data stream processors (Section 8).

Finally, we survey related work in Section 9.
Data structure and physical storages for high-update-rate processing buffers

Handling the new data stream elements/expired ones

Operators && Incremental execution
- Stateless
- Stateful
  - Duplicate elimination
  - Window Join
  - Negation
  - Aggregation

Memory overflow

Dynamic Optimization of the continuous execution

Schedule execution for fluctuate execution settings
Incremental execution of windowing operators

(a) Stateless

(b) Window Join

(c) Duplicate elimination

(d) Aggregate

(e) Negation
What have “I” learned?

- 12-15 years of techniques/algorithms/solutions for general stream processing (DSMS)
- Only few prototypes and commercial products
  - STREAM, Borealis/Aurora, etc
  - StreamBase, IBM InfoSphere streams, etc
- Don’t take for granted!!
- DSMS is not mature as DBMS (>40 years)
Processing Linked Stream Data In A Nutshell

A. Linked Open Data cloud

B. Query

SELECT ?person
FROM ... [NOW]
WHERE {
  ?person ...
}

C. Sensor Stream Data

pre-processing

optimization

execution

answer
Black-box approach

- Query
  - Optimizer
    - Operator implementations
      - Executor
        - Access methods
  - Query rewriter
    - Orchestrator
    - Data transformation
      - Overhead
        - Execution
C-SPARQL
C-SPARQL execution 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))
```

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 aggregation1 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 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 rewriting rule.
C-SPARQL query rewriting

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 .
    ?tx x:with ?amount .
    FILTER (?country = 'CH' && ?amount >= 10)}
AGGREGATE { (?total, SUM(?amount), ?broker) }

Query Graph Model (QGM) [27]. A SQGM is a directed graph that represents SPARQL queries as a directed graph. Operators are used for both query evaluation and debugging. The QGM provides a visual representation of the query, which can be used to understand the flow of data in the query. The QGM is based on the work of P´erez et al. [26] and includes an O-Graph, which is used for visual query representation.

Logical windows (a) contain the most recent triples in a time interval of length 24 hours. A window is said to be as logical if it is sliding with range (24 hours). Some of these bindings may be used in C-SPARQL within the perspective of SQGM.

Logical windows (b) are evaluated for a window, which returns an RDF graph. A logical window is defined as a function which counts the items in a sliding window and returns a set of bindings coming from the stream. Logical windows are necessary because filter clauses can also be used in C-SPARQL within the perspective of SQGM. We denote the set of bindings as a triple 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 perspective of SQGM.

The operational semantics presented in this section is the basis for the visual query representation which is called the QGM. We also propose to represent the SPARQL filter clause as a 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 perspective of SQGM.
Fig. 1. Ontology-based streaming data access service

To transform a SPARQL Stream query, expressed in terms of the ontology, into queries in terms of the data sources, a set of mappings must be specified. These mappings are expressed in S2O, an extension of the R2O mapping language, which supports streaming queries and data, most notably window and stream operators (see Section 4.2). This transformation process is called query translation, and the target is the continuous query language SNEEql, which is expressive enough to deal with both streaming and stored sources.

After the continuous query has been generated, the query processing phase starts, and the evaluator uses distributed query processing techniques [14] to extract the relevant data from the sources and perform the required query processing, e.g., selection, projection, and joins. Note that query execution in sources such as sensor networks may include in-network query processing, pull or push based delivery of data between sources, and other data source specific settings.

The result of the query processing is a set of tuples that the data translation process transforms into ontology instances.

This approach requires several contributions and extensions to the existing technologies for continuous data querying, ontology-based data access, and SPARQL query processing. This paper focuses on a first stage that includes the process of transforming the SPARQL Stream queries into queries over the streaming data sources using SNEEql as the target language. The following sections provide the syntax and semantics for the querying of streaming RDF data and the mappings between streaming sources and an ontology. We will then provide details of an actual implementation of this approach.

4 Query and Mapping Syntax

In this section we introduce the SPARQL Stream query language, an extension to SPARQL for streaming RDF data, which has been inspired by previous proposals such as c-sparql [9] and sneeql [12]. However, significant improvements...
An example query of SPARQL_stream

```sparql
PREFIX fire : <http://www.semsorgrid4env.eu#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT RSTREAM ?WindSpeedAvg 
FROM STREAM <www.semsorgrid4env.eu/SensorReadings.srdf> [FROM NOW − 10 MINUTES TO NOW STEP 1 MINUTE] 
FROM STREAM <www.semsorgrid4env.eu/SensorArchiveReadings.srdf> [FROM NOW − 3 HOURS TO NOW − 2 HOURS STEP 1 MINUTE] 
WHERE {
    
    
    SELECT AVG(?speed) AS ?WindSpeedAvg 
    WHERE 
    {
        
        GRAPH <www.semsorgrid4env.eu/SensorReadings.srdf> {
            ?WindSpeed a fire:WindSpeedMeasurement; 
            fire:hasSpeed ?speed; }
        } GROUP BY ?WindSpeed
    }
    
    SELECT AVG(?archivedSpeed) AS ?WindSpeedHistoryAvg 
    WHERE 
    {
        
        GRAPH <www.semsorgrid4env.eu/SensorArchiveReadings.srdf> {
            ?ArchWindSpeed a fire:WindSpeedMeasurement; 
            fire:hasSpeed ?archivedSpeed; }
        } GROUP BY ?ArchWindSpeed
    }
FILTER (?WindSpeedAvg > ?WindSpeedHistoryAvg)
}
```

every minute computes the average wind speed measurement for each sensor over the last 10 minutes if it is higher than the average of the last 2 to 3 hours.
An example of mapping rule in $S_2O$

S$_2$O declaration of a data stream schema and mapping from a stream schema to an ontology concept.

Relational stream to be mapped

Map the columns with ontological properties
EP-SPARQL

- **Execution mechanism**: Prolog-based event-driven backward chaining (EDBC) rules

- **Representation**
  - RDF triple \((s, p, o)\) ➔ predicate \(triple(s, p, o)\)
  - Time-stamped RDF triple \((s, p, o, t_1, t_2)\) ➔ predicate \(triple(s, p, o, T_1, T_2)\)

- **Operators rewriting**
  - Operators (SeqJoin, Filters, etc) are rewritten in Prolog rules
  - Two types of EDBC rules
    - Goal-insertion rules: to create intermediate goals of incoming events
    - Checking-rule: check if intermediate goals are triggered
Whitebox approach: Streaming SPARQL and CQELS

Query

Optimizer

Executor

Triple-based physical operators

Operator implementations

Data structures and physical storage for triple-based data elements

Access methods

RDF datasets

RDF stream
Extension of SPARQL physical operators for windowing graph patterns

Extends SweepArea for triple-based stream elements
Examples of executing physical operators of Streaming SPARQL engine

SELECT ?w ?x ?y ?z
FROM STREAM <http://src.net/graph.rdf>
WHERE {?w my:name ?x} UNION {?y my:power ?z}

PREFIX wtur: <http://iec.org/61400-25/root/ln/classes/WTUR#>
SELECT ?x ?y ?z
FROM STREAM <http://iec.org/61400-25/root/td.Rdf>
WINDOW RANGE 30 MINUTE SLIDE
WHERE {?x wtur:StrCnt ?y}.
OPTIONAL {?x wtur:StopCnt ?z}.

{WINDOW ELEMS 1500}
CQELS architecture for adaptive and native processing
Adaptive execution of CQELS

CONSTRUCT {?person1 lv:reachable ?person2}
FROM NAMED <http://deri.org/floorplan/>
WHERE {
  STREAM <http://deri.org/streams/rfid> [NOW] {?person1 lv:detectedAt ?loc1}
  STREAM <http://deri.org/streams/rfid> [RANGE 3s] {?person2 lv:detectedAt ?loc2}
  GRAPH <http://deri.org/floorplan/> {?loc1 lv:connected ?loc2} }

(a) From window now
(b) From window range 3s
System Comparisons

<table>
<thead>
<tr>
<th>Input</th>
<th>Query language</th>
<th>Extras</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming SPARQL</td>
<td>RDF stream</td>
<td></td>
</tr>
<tr>
<td>C-SPARQL</td>
<td>RDF Stream &amp; RDF</td>
<td>TF</td>
</tr>
<tr>
<td>EP-SPARQL</td>
<td>RDF Stream &amp; RDF</td>
<td>EVENT,TF</td>
</tr>
<tr>
<td>SPARQL_{stream}</td>
<td>Relational stream</td>
<td>NEST</td>
</tr>
<tr>
<td>CQELS</td>
<td>RDF Stream &amp; RDF</td>
<td>VoS,NEST</td>
</tr>
</tbody>
</table>

**TF:** built-in time functions  **EVENT:** event pattern  **NEST:** nested patterns  **VoS:** Variables on streams ID

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Re-execution</th>
<th>Scheduling</th>
<th>Optimisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming SPARQL</td>
<td>whitebox</td>
<td>periodical</td>
<td>Logical plan</td>
</tr>
<tr>
<td>C-SPARQL</td>
<td>blackbox</td>
<td>periodical</td>
<td>Logical plan</td>
</tr>
<tr>
<td>EP-SPARQL</td>
<td>blackbox</td>
<td>eager</td>
<td>Logic program</td>
</tr>
<tr>
<td>SPARQL_{stream}</td>
<td>blackbox</td>
<td>periodical</td>
<td>External call</td>
</tr>
<tr>
<td>CQELS</td>
<td>whitebox</td>
<td>eager</td>
<td>Adaptive physical plans</td>
</tr>
</tbody>
</table>
Experiment setup for performance comparisons

- **Conference scenario**: combine linked stream from RFID tags (physical relationships) with DBLP data (social relationships)

- **Setup**
  - **Systems**: CQELS vs ETALIS and C-SPARQL
  - **Datasets**
    - Replayed RFID data from Open Beacon deployments
    - Simulated DBLP by SP²Bench
  - **Queries**: 5 query templates with different complexities
    - Q1: selection,
    - Q2: stream joins, Q3,Q4: Stream and non-stream joins
    - Q5: aggregation
  - **Experiments**
    - Single query: generate 10 query instances of each template by varying the constants
    - Vary size of the DBLP ($10^4$-$10^7$ triples)
    - Multiple queries: register $2^M$ instances at the same time ($0 \leq M \leq 10$)
## Performance comparison - Query execution time

- **CQELS** perform faster by orders of magnitudes

<table>
<thead>
<tr>
<th></th>
<th>Query 1</th>
<th>Query 2</th>
<th>Query 3</th>
<th>Query 4</th>
<th>Query 5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CQELS</strong></td>
<td>0.47</td>
<td>3.90</td>
<td>0.51</td>
<td>0.53</td>
<td>21.83</td>
</tr>
<tr>
<td><strong>C-SPARQL</strong></td>
<td>332.46</td>
<td>99.84</td>
<td>331.68</td>
<td>395.18</td>
<td>322.64</td>
</tr>
<tr>
<td><strong>ETALIS</strong></td>
<td>0.06</td>
<td>27.47</td>
<td>79.95</td>
<td>469.23</td>
<td>160.83</td>
</tr>
</tbody>
</table>

**Simple selection : ETALIS perform best**

**Stream join :**
- 25 times faster than C-SPARQL
- 8 times faster than ETALIS

**Stream and non-stream joins :**
- >600 times faster than C-SPARQL
- 150-850 times faster than ETALIS

**Aggregation:**
- 15 times faster than C-SPARQL
- 8 times faster than ETALIS
Performance comparison – Scalability (non-stream data size)

Non-stream data size: logarithmic to size of static intermediate results

Graphs showing the average query execution time (ms) on a log scale for different numbers of triples from DBLP for queries Q1, Q3, and Q5, comparing COELS, C-SPARQL, and ETALIS.
Performance comparison - Scalability (number of queries)

No multiple query optimization
Performance comparison-Maximum input throughput

- Pattern 1, schema has 329 classes, target product type has 40 subclasses
- Sparkwave is significantly better only for smaller time windows
- Impact of reasoning over Sparkwave performance is limited (15-20%)

C-SPARQL gives better throughput than CQELS?
Performance comparison – Memory consumption

- **Pattern 1, schema has 329 classes, target product type has 40 subclasses**
  - Memory consumption is lower but rises
Are they ready for production?

- Not quite !!! (just 4-6 years)
- Why?
  - Functionality
  - Performance
  - Scalability
- But interesting to test/compete/extend/study
  - Vast amount of interesting of heterogeneous data streams and open Linked Data sources
  - Plethora of use cases/applications
  - New interesting research problems
Open challenges

- Serialization of RDF-based stream elements
- Optimization & Scheduling
- How to measure and compare performances
- Early-stage
  - Expressiveness
  - Reasoning
- Lack of functionalities
  - Disk-based stream processing
  - Distributed processing for large-scale data
How to serialize the RDF stream elements - Graph-based stream layout
How to serialize the RDF stream elements

<table>
<thead>
<tr>
<th>triple</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>c:Distr1 t:has-entering-cars &quot;100&quot;</td>
<td>t_{400}</td>
</tr>
<tr>
<td>c:Distr2 t:has-entering-cars &quot;75&quot;</td>
<td>t_{400}</td>
</tr>
<tr>
<td>c:Distr1 t:has-entering-cars &quot;130&quot;</td>
<td>t_{401}</td>
</tr>
<tr>
<td>c:Distr2 t:has-entering-cars &quot;95&quot;</td>
<td>t_{401}</td>
</tr>
<tr>
<td>c:Distr3 t:has-entering-cars &quot;65&quot;</td>
<td>t_{401}</td>
</tr>
</tbody>
</table>

- Lack of standard way to serialize RDF Stream
- N-Triple-like representation is inefficient
  - 100-500 bytes to represent 1 integer reading
  - 100k triples/sec $\rightarrow$ 10-50MB/sec $\rightarrow$ 80-400Mbps bandwidth
- Is it necessary in text-line format? Binary format?
Optimization & Scheduling

- At logical plan level → inefficient and restricted on highly dynamic settings of the stream processing.

- Few at physical level but only with naïve/simplistic algorithms/strategies.

- None support multiple query optimization

☞ Needs more studies on optimization and scheduling graph-based query patterns
How to compare performance

- There are only few stream benchmarking systems
  - Linear Road benchmark (VLDB 2004)
  - LSBench (to appear at ISWC 2012)
  - SRBench (to appear at ISWC 2012)

- How to define the measurement to compare
  - Execution time/Response time?
  - Throughput?

- Too many elements to cause differences in outputs
  - Difference in semantics
  - The execution mechanisms
  - Execution environments and settings
Early-stage work

Expressiveness

- SPARQL extensions based on relational algebra is not expressive enough for stream/event processing applications
- Higher expressive continuous query language?
  - Recursive expression
  - Rule-based expression
  - Support uncertainty in matching pattern

Stream reasoning based on RDF data model

- Emerging topic with some early work
  - Barbieri et al. Incremental Reasoning on Streams and Rich Background Knowledge (ESWC’2010).
  - Komazec et al. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams (DEBS’2012)
- Complexity vs low latency ➞ need quantitative metrics to judge the advantages of each stream reasoner
Lack of functionalities

- **Disk-based stream processing**
  - Big windows
  - Big linked data sets

- **Distributed stream processing**
  - Some general distributed stream processing platform/systems
    - Borealis, StreamBase, IBM Stream Spheres, etc
    - S4, Storm, Kafka, etc
  - Use black-box approach delegate processing ➔ How to deal with the overhead and restriction of optimization?
  - Use whitebox approach ➔ which physical processing can be reused from such platform/system? Can it be better?
Summary

- What is Linked Stream Data?
- Data models for Linked Stream Data
- Query operators and query languages
- How to build a Linked Stream Processing Engine
- Comparisons and analysis of State-Of-The-Art systems
- Open challenges
  - Serialization of RDF-based stream elements
  - Optimization & Scheduling
  - How to measure and compare performances
  - Early-stage & Lack of functionalities