Operator-aware Approach for Boosting Performance in Processing RDF streams

Danh Le Phuoc | Open Distributed Systems, Technical University of Berlin, Germany

SMARTER Project – Grant No. 661180 (H2020-MSCA-IF-2014)
Agenda

- Overview of RDF Processing
- On Boosting The Processing Throughput
- Challenges in Incremental Evaluation
- Operator-Aware Approach
- Evaluation
- Summary
Overview of RDF Stream Processing (RSP)

Stream data in RDF

\[
\begin{align*}
S_{\text{pickup}} &= \{ 
:ride_1 :\text{taxi} : 89...CF4 \\
:ride_1 :\text{pickupTime} \ "2013-01-01\ 15:11:48". \} \\
S_{\text{dropoff}} &= \{ 
:ride_1 :\text{dropoffTime} \ "2013-01-01\ 15:18:10". \\
:ride_1 :\text{tripTime} 382. 
\} \\
S_{\text{fare}} &= \{ 
:trans_1 :\text{fare} 7. \\
:trans_1 :\text{pickupTime} \ "2013-01-01\ 15:11:48". 
\}
\end{align*}
\]
The query for continuous computation: "hourly riding rate of active taxies of last 1000 payment transactions"

\[
\begin{align*}
S_{\text{pickup}} &= \{ :\text{ride}1 :\text{taxi} : 89 \ldots : \text{CF4} \\
&\quad :\text{ride}1 :\text{pickupTime} "2013-01-01 15:11:48". \} \\
S_{\text{dropoff}} &= \{ :\text{ride}1 :\text{dropoffTime} "2013-01-01 15:18:10". \\
&\quad :\text{ride}1 :\text{tripTime} 382. \} \\
S_{\text{fare}} &= \{ :\text{trans}1 :\text{fare} 7. \\
&\quad :\text{trans}1 :\text{pickupTime} "2013-01-01 15:11:48". \}
\end{align*}
\]

Stream data in RDF
"hourly riding rate of active taxis of last 1000 payment transactions"

SELECT ?taxi (AVERAGE(?fare/(?tripTime/3600))) AS ?hourlyRate
FROM NAMED WINDOW :W1 ON nyc такси:fare [COUNT 1000]
FROM NAMED WINDOW :W2 ON nyc такси:pickup
   [RANGE PT2H @:pickupTime]
FROM NAMED WINDOW :W3 ON nyc такси:dropoff
   [RANGE PT1H @:dropoffTime]
WHERE {
   WINDOW :W3{ ?ride :tripTime ?tripTime}
} GROUP BY ?taxi

Stream data in RDF


"hourly riding rate of active taxies of last 1000 payment transactions"

SELECT ?taxi (AVERAGE(?fare/(?tripTime/3600)) AS ?hourlyRate)
FROM NAMED WINDOW :W1 ON nyctaxi:fare [COUNT 1000]
FROM NAMED WINDOW :W2 ON nyctaxi:pickup
  [RANGE PT2H @:pickupTime]
FROM NAMED WINDOW :W3 ON nyctaxi:dropoff
  [RANGE PT1H @:dropoffTime]
WHERE {
  WINDOW :W3{ ?ride :tripTime ?tripTime}
}
GROUP BY ?taxi

Continuous Query in SPARQL-like language

Stream data in RDF
On boosting the processing throughput of RSP engines

- Experience from Implementation CQELS Execution Framework: **Using off-the-shelf data structures and algorithms are not enough!!??**
  - Hardly can reach 10000 operator executions/second on large windows (100k-1M entries)
  - Big overhead of using row-based data structures
- **Bottom-up perspective:** investigating closely to data structures and algorithms
  - **Highly efficient data structures** for maintaining processing states
  - Sophisticate **incremental evaluation algorithms** of query operators
Incremental Evaluation for continuous operators in a Nutshell

stateful sliding window operators: reuse previous computing effort

Overhead of maintaining previous processing states
Incremental Evaluation of Continuous Queries over RDF Stream: Issues and Challenges

- **Row-based data structure** is not suitable for:
  - very small RDF data elements (encoded as fixed-size integers)
  - unusually large number individual data points (millions of mappings/RDF nodes are generated/evicted per second)

- **Timestamping or negative-tuple solutions** for incremental computation of RDF data elements and mappings have technical issues:
  - Auxiliary data (**extra timestamps or negative tuples**) might be bigger than original data
  - Other limitations of state-of-art techniques (**double computation in evicting expired computing state**)
Incremental Evaluation of Continuous Query: sliding data flow

```
SELECT ?taxi (AVG(?fare/?tripTime/3600)) AS ?hourlyRate
FROM NAMED WINDOW :W1 ON nyctaxi:fare [COUNT 1000]
FROM NAMED WINDOW :W2 ON nyctaxi:pickup
  [RANGE PT2H @:pickupTime]
FROM NAMED WINDOW :W3 ON nyctaxi:dropoff
  [RANGE PT1H @:dropoffTime]
WHERE {
  WINDOW :W3 { ?ride :tripTime ?tripTime}
}
GROUP BY ?taxi
```

The diagram illustrates the sliding window auxiliary state machine with three windows, W1, W2, and W3, and their respective timelines (1, 3, and 4). The SELECT statement calculates the average hourly rate based on the fare and trip time, using the windows to process incoming data streams.
Incremental Evaluation of Continuous Query: sliding data flow

```
SELECT ?taxi (AVG(?fare/(?tripTime/3600)) AS ?hourlyRate)
FROM NAMED WINDOW :W1 ON nyctaxi:fare [COUNT 1000]
FROM NAMED WINDOW :W2 ON nyctaxi:pickup
    [RANGE PT2H @:pickupTime]
FROM NAMED WINDOW :W3 ON nyctaxi:dropoff
    [RANGE PT1H @:dropoffTime]
WHERE {
    WINDOW :W3{ ?ride :tripTime ?tripTime}
}
GROUP BY ?taxi
```
**Incremental Evaluation of Continuous Query: sliding data flow**

```
SELECT ?taxi (AVERAGE(?fare/(?tripTime/3600)) AS ?hourlyRate)
FROM NAMED WINDOW :W1 ON nyctaxi:fare [COUNT 1000]
FROM NAMED WINDOW :W2 ON nyctaxi:pickup
  [RANGE PT2H @:pickupTime]
FROM NAMED WINDOW :W3 ON nyctaxi:dropoff
  [RANGE PT1H @:dropoffTime]
WHERE {
  WINDOW :W3{ ?ride :tripTime ?tripTime}
}
GROUP BY ?taxi
```

![Diagram showing the sliding data flow with window W1, W2, and W3, and the final aggregation with AVG function.](image)
SELECT ?taxi (AVERAGE(?fare/(?tripTime/3600)) AS ?hourlyRate)
FROM NAMED WINDOW :W1 ON nyctaxi:fare [COUNT 1000]
FROM NAMED WINDOW :W2 ON nyctaxi:pickup
    [RANGE PT2H @:pickupTime]
FROM NAMED WINDOW :W3 ON nyctaxi:dropoff
    [RANGE PT1H @:dropoffTime]
WHERE {
    WINDOW :W3{ ?ride :tripTime ?tripTime}
}
GROUP BY ?taxi
Incremental Evaluation of Continuous Query: sliding data flow

```
SELECT ?taxi (AVERAGE(?fare/(?tripTime/3600)) AS ?hourlyRate)
FROM NAMED WINDOW :W1 ON nyctaxi:fare [COUNT 1000]
FROM NAMED WINDOW :W2 ON nyctaxi:pickup
  [RANGE PT2H @:pickupTime]
FROM NAMED WINDOW :W3 ON nyctaxi:dropoff
  [RANGE PT1H @:dropoffTime]
WHERE {
  WINDOW :W3{ ?ride :tripTime ?tripTime}
}
GROUP BY ?taxi
```
Incremental Evaluation of Continuous Query: generating processing state
Incremental Evaluation of Continuous Query: generating processing state

Large number of intermediate processing states generated by copying binding values from buffers of downstream operators.
Incremental Evaluation of Continuous Query: generating processing state

Large number of intermediate processing states generated by copying binding values from buffers of downstream operators

Processing pipeline create traits to trace back to the binding values in the buffers of the bottom operators (leaves of operator tree)
Incremental Evaluation of Continuous Query: evicting processing state

```sql
SELECT ?taxi (AVG(?fare/(?tripTime/3600)) AS HourlyRate) 
FROM NAMED WINDOW :W1 ON nyc_taxi:fare [COUNT 1000] 
FROM NAMED WINDOW :W2 ON nyc_taxi:pickup 
WHERE {
  WINDOW :W3 {?ride :tripTime ?tripTime}
} 
GROUP BY ?taxi
```

In effect, each of these operators consumes a set of bags of mappings and re-derived data, and the processing state can be an original data storage or auxiliary data structures like a hash table or a B-tree.
SELECT ?taxi (AVG(?fare/(?tripTime/3600))) AS ?hourlyRate
FROM NAMED WINDOW :W1 ON nyctaxi:fare [COUNT 1000]
FROM NAMED WINDOW :W2 ON nyctaxi:pickup
    [RANGE PT2H 0:pickupTime]
FROM NAMED WINDOW :W3 ON nyctaxi:dropoff
    [RANGE PT2H 0:dropoffTime]
WHERE {
    WINDOW :W3 {?ride :tripTime ?tripTime}
}
GROUP BY ?taxi
Incremental Evaluation of Continuous Query: evicting processing state

```
SELECT ?taxi (AVG(?fare/(?tripTime/3600))) AS HourlyRate
FROM NAMED WINDOW :W1 ON nyctaxi:fare [COUNT 1000]
FROM NAMED WINDOW :W2 ON nyctaxi:pickup
  [RANGE PT2H 0:pickupTime]
FROM NAMED WINDOW :W3 ON nyctaxi:dropoff
  [RANGE PT1H 0:dropoffTime]
WHERE {
  WINDOW :W3 { ?ride :tripTime ?tripTime }
}
GROUP BY ?taxi
```

Figure 3: State-of-the-art stateful processing operators. In effect, each of these operators consumes a set of bagsof mappings and dedust data and the processing state can be an original data storage or auxiliary data structures like a hash table or a B-tree.
Incremental Evaluation of Continuous Query: evicting processing state

```
SELECT ?taxi (AVG(?fare/(?tripTime/3600)) as HourlyRate)
FROM NAMED WINDOW :W1 ON nyc-taxi:fare [COUNT 1000]
FROM NAMED WINDOW :W2 ON nyc-taxi:pickup
[GLOBAL RANGE PT2H 0:pickupTime]
FROM NAMED WINDOW :W3 ON nyc-taxi:dropoff
[GLOBAL RANGE PT1H 0:dropoffTime]
WHERE {
  WINDOW :W3 {?ride :tripTime ?tripTime}
}
GROUP BY ?taxi
```
Incremental Evaluation of Continuous Query: evicting processing state

```sql
SELECT ?taxi (AVERAGE(?fare/(?tripTime/3600)) AS HourlyRate)
FROM NAMED WINDOW :W1 ON nyc_taxi:fare [COUNT 1000]
FROM NAMED WINDOW :W2 ON nyc_taxi:pickup
   [RANGE PT2H 0:pickupTime]
FROM NAMED WINDOW :W3 ON nyc_taxi:dropoff
   [RANGE PT1H 0:dropoffTime]
WHERE {
   WINDOW :W3 { ?ride :trip_time ?tripTime }
}
GROUP BY ?taxi
```
Incremental Evaluation of Continuous Query: evicting processing state

```
SELECT ?taxi (AVG(?fare/(?tripTime/3600)) AS HourlyRate
FROM NAMED WINDOW :W1 ON nyctaxi:fare [COUNT 1000]
FROM NAMED WINDOW :W2 ON nyctaxi:pickup
   [RANGE PT2H 0:pickupTime]
FROM NAMED WINDOW :W3 ON nyctaxi:dropoff
   [RANGE PT2H 0:dropoffTime]
WHERE {
  WINDOW :W3 { ?ride :tripTime ?tripTime}
}
GROUP BY ?taxi
```

Evicting expired data might lead to double computation to update final outputs.
Operator-aware Approach

- Operator-aware data structures designed for:
  - Bookkeeping how the processing states were generated by the query operators
  - Indexing windowing buffers tailored for query operators’ behaviors
  - Algorithms for incremental evaluations driven by operator-aware data structures
Tree-based data structure for solution mappings
Tree-based data structure for solution mappings
Tree-based data structure for solution mappings
Tree-based data structure for solution mappings
Tree-based data structure for solution mappings
Evicting expired mappings

The diagram illustrates a process involving expired mappings. The mappings are divided into three ranges: [COUNT 1], [RANGE 10], and [RANGE 5]. Each range contains specific data, such as timestamps or identifiers, as indicated by the symbols within the boxes.

The process involves probing and consuming data, as well as maintaining and evicting expired mappings. The symbols and arrows represent the flow of data and the actions taken during this process.
Evicting expired mappings
Evicting expired mappings

In effect, each of these operators consumes a set of bags of...
Evicting expired mappings
Ring Indexes on bags of mappings

- Operator-aware indexes for quick lookup operations
- Low maintenance cost for fast insert/delete operations

![Diagram of ring indexes on bags of mappings](image)
Throughputs of Ring indexes

Hash outperforms over AVL Tree and Red-Black Tree

Insert throughput for 1M keys

Probing time for 1M-mapping windows
Inserting throughput: 500-900k

Probing/searching throughput: 1M-1.6M
Throughputs of Query Operators

- Operator-aware implementations outperform relational implementations
- ... are marginally faster than ad-hoc implementations of ESPER in most cases
Throughputs & Memory footprint

<table>
<thead>
<tr>
<th></th>
<th>SRBench (triples/sec)</th>
<th>LSBench (triples/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$Q_1$</td>
<td>$Q_4$</td>
</tr>
<tr>
<td>R-CQELS</td>
<td>1214</td>
<td>820</td>
</tr>
<tr>
<td>CQELS</td>
<td>25147</td>
<td>20161</td>
</tr>
</tbody>
</table>

Processing Throughputs: 5-12 times more than relational

<table>
<thead>
<tr>
<th></th>
<th>Multiway Join (MB)</th>
<th>SRBench (MB)</th>
<th>LSBench (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>R-CQELS</td>
<td>25.61</td>
<td>28.67</td>
<td>38.95</td>
</tr>
<tr>
<td>ESPER</td>
<td>8.93</td>
<td>12.04</td>
<td>15.13</td>
</tr>
</tbody>
</table>

Memory Footprint: twice less memory than relational and 20-50% less than ESPER
Summary

- Incremental evaluation algorithms based on operator-aware data structures:
  - Overcome technical issues on traditional incremental evaluation techniques/algorithms
  - Perform several orders of magnitude faster than relation-based implementations
- Throughputs on operator-aware operations on processing state:
  - Up to 1 million of updates/sec vs. 10k of relation-based one
  - Up to 1.6 million lookup operations/second
  - Outperform over relational operations by order of magnitudes
  - Consume twice less memory than relation-based implementations
- The implementation will be open sourced in the next release of CQELS (cqels.org)