Linked Stream Data Processing Engines – Facts and Figures

Danh Le-Phuoc*, Minh Dao-Tran, Minh-Duc Pham, Peter Boncz, Thomas Eiter and Michael Fink

*DERI – National University of Ireland, Galway
ISWC 2012, Boston
Outline

- Motivation for Linked Stream Data Processing
- State-Of-The-Art Linked Stream Processing Engines
- Challenges of cross-system evaluation
- Evaluation framework
- Experiments
- Findings
- Summary
Motivation for Linked Stream Data Processing
State–Of–The–Art Linked Stream Processing Engines

- **State–Of–The–Art Engines**
  - Streaming SPARQL (2008)
  - C–SPARQL (2010)
  - \( \text{SPARQL} \text{stream} \) (2010)
  - CQELS(2011)
  - Sparkware(2012)

- **Too few cross–system comparisons**
  - CQELS(2011)
  - Sparkwave(2012)
Challenges in cross-comparison of Linked Data Stream Processing engines

- Differences in query semantics
- Differences in execution mechanisms
- Running environment and experiment parameters can lead to different outputs in continual execution context.
Test suits:
- Functionality tests: query patterns supported
- Correctness tests: outputs are comparable to for cross-comparisons
- Performance tests: compare throughputs that meet comparability conditions

- Data generator: Stream Social network data Generator (S2Gen)
- Test drivers and Evaluators
Simulates data schema in social networks

- Stream data: GPS, Posts & comments, Photos
- Static data: User metadata (user profile, users’ relationships)
Data generator (S2Gen)

- A novel data generator for Linked Stream Data
- Models realistic data correlations and skewed data distributions in stream-based social networks
  - Use *window sliding* approach for generating social data streams
- Offers various parameters for different test cases
  - Scalability
  - Different stream input rates
  - Different correlation probabilities
Experiments

- Experiment Design

- Test runs
  - Functionality tests
  - Correctness tests
  - Performance tests
    - Execution throughput
    - Scalability tests
Experiment Design

- Engine as a black box
- Data streams via stream players
- Record output stream

Stream engine $E$

Stream rates ($R$)

Stream query $Q$

Static data

$S_1 \rightarrow r_1$

$S_m \rightarrow r_m$

$O(E, Q, R)$
## Functionality tests

- None of the engines support all the SPARQL query patterns

<table>
<thead>
<tr>
<th>Patterns covered</th>
<th>S</th>
<th>$N_P$</th>
<th>$N_S$</th>
<th>Engines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CQ</td>
<td>CS</td>
<td>JT</td>
<td></td>
</tr>
<tr>
<td>$Q_1$</td>
<td>✓</td>
<td>1</td>
<td>1</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>$Q_2$</td>
<td>✓</td>
<td>✓</td>
<td>2</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>$Q_3$</td>
<td>✓</td>
<td>✓</td>
<td>3</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>$Q_4$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>$Q_5$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓ ✓ Ø</td>
</tr>
<tr>
<td>$Q_6$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓ ✓ Ø</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Patterns covered</th>
<th>S</th>
<th>$N_P$</th>
<th>$N_S$</th>
<th>Engines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CQ</td>
<td>CS</td>
<td>JT</td>
<td></td>
</tr>
<tr>
<td>$Q_7$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓ 7 2</td>
</tr>
<tr>
<td>$Q_8$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>3 2 0</td>
</tr>
<tr>
<td>$Q_9$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>8 4 E 0</td>
</tr>
<tr>
<td>$Q_{10}$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>1 1 ✓ ✓</td>
</tr>
<tr>
<td>$Q_{11}$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>2 2 ✓ ✓</td>
</tr>
<tr>
<td>$Q_{12}$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>1 1 ✓ ✓</td>
</tr>
</tbody>
</table>

**F**: filter  
**J**: join  
**E**: nested query  
**N**: negation  
**T**: top k  
**U**: union  
**A**: aggregation  
**S**: uses static data

$N_P$: number of patterns, $N_S$: number of streams, $\odot$: syntax error, E: error, Ø: return no answer, ×: not supported

**CQ**: CQELS, **CS**: C-SPARQL, **JT**: JTALIS
Correctness tests

- All the engines don’t output the same numbers of results

<table>
<thead>
<tr>
<th></th>
<th>Rate :100 inputs/sec</th>
<th>Rate : 1000 inputs/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CQELS</td>
<td>C–SPARQL</td>
</tr>
<tr>
<td>Q1</td>
<td>68</td>
<td>604</td>
</tr>
<tr>
<td>Q2</td>
<td>68</td>
<td>124</td>
</tr>
<tr>
<td>Q3</td>
<td>533</td>
<td>1065</td>
</tr>
<tr>
<td>Q4</td>
<td>11984</td>
<td>125910</td>
</tr>
<tr>
<td>Q10</td>
<td>28021</td>
<td>205986</td>
</tr>
</tbody>
</table>

Comparisons on input/output throughputs are invalid
Output stream coverage

Not covered

OE₁  

OE₂

a₁ a₂ a₃  a₄ a₅  a₆ a₇  a₈ a₉ a₁₀

a₁₀ a₇ a₉ a₈ a₆

a partition of output stream

- Not full ordered output streams but the output partitions might be
- A partition of OE₁ can be covered by groups of elements from OE₂
- The output with the greater number of outputs might not cover the one with the smaller number.
Coverage checking for a output partition

<table>
<thead>
<tr>
<th>OE₁</th>
<th></th>
<th>Ai</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Check if covered by a slice

\[ OE₂ \mid j = b_j, \ldots, b_{n_2} \]

- iff for every \( a_{ik} \in A_i \), there exists \( b_t \in OE₂ \mid j \)

\[ a_{ik} = b_t \]

Position to start checking \( j \)

- Slice the stream output to output sequences to check if a output partition is covered by a sequence
- If not covered \( \Rightarrow \) compute the mismatch metric
Mismatch metric for non-coverage

\[ m = \sum_{i=1}^{m} \frac{|A_i| - |T_i|}{\sum_{i=1}^{m} |A_i|} \times 100\% \]

Mismatch metric

\[ T_i = P_i^{\text{match}(T_{i-1})} = A_i \cap O_{E_2 \backslash \text{match}(T_{i-1})} \]

Maximum subsequences that have matches

\[ \text{Next position for the output sequence to find the matches} \]

The output sequence to search for matches for \( A_i \)

\[ \text{OE}_1 \]

\[ \text{OE}_2 \]

\[ A_i \]
Some mismatch values vs. output size

<table>
<thead>
<tr>
<th></th>
<th>Rate: 100 (input elements/sec)</th>
<th>Rate: 1000 (input elements/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Output size</td>
<td>Mismatch (%)</td>
</tr>
<tr>
<td></td>
<td>CQ</td>
<td>CS</td>
</tr>
<tr>
<td>Q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>68</td>
<td>604</td>
</tr>
<tr>
<td>2</td>
<td>68</td>
<td>124</td>
</tr>
<tr>
<td>3</td>
<td>533</td>
<td>1065</td>
</tr>
<tr>
<td>4</td>
<td>11945</td>
<td>125910</td>
</tr>
<tr>
<td>10</td>
<td>28021</td>
<td>205986</td>
</tr>
</tbody>
</table>

Periodical and recurrent execution (CS–CPARQL) might output greater number of output but they are repeated and contained in output results of eager and incremental execution (CQ–CQELS, JT–JTALIS)
How fast they are?

<table>
<thead>
<tr>
<th></th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q10</th>
</tr>
</thead>
<tbody>
<tr>
<td>CQELS</td>
<td>24122</td>
<td>8462</td>
<td>9828</td>
<td>1304</td>
<td>7459</td>
<td>3491</td>
<td>2326</td>
</tr>
<tr>
<td>C-SPARQL</td>
<td>10</td>
<td>1.68</td>
<td>1.63</td>
<td>10</td>
<td>1.72</td>
<td>1.71</td>
<td>10</td>
</tr>
<tr>
<td>JTALIS</td>
<td>3790</td>
<td>3857</td>
<td>1062</td>
<td>99</td>
<td>—</td>
<td>—</td>
<td>87</td>
</tr>
</tbody>
</table>

- Twitter might deliver 20k–100k tweets/sec
- Berlin Benchmark for triple storages (Feb 2011): ~50–400 queries/sec
- ESPER: 200k–500k events/sec
- SASE: 10–150k events/sec
Maximum execution throughput when varying static data size

C-SPARQL and JTALIS could not scale with big static data size
Maximum execution throughput for multiple queries

None of the systems could share the processing among multiple queries
Findings

- **Expressiveness**: None of state-of-the-art continuous query languages support all SPARQL query patterns.

- **Validity**: Throughputs based on the number of outputs/inputs are invalid.

- **Performance**: Incremental & eager execution (JTALIS, CQELS) outperforms over periodical & recurrent execution (C–SPARQL).

- **Scalability**: Most of the engines have poor scalability.
  - On static data size: only CQELS can scale with static data sizes by pre-computing and caching static sub-queries.
  - On the number of queries: none of the engines apply multiple query optimisation techniques.
Summary

- **Evaluation Framework (LSBench)**
  - Test suits
  - Data generator for social networks

- **First extensive cross system evaluations**
  - Validity of evaluations
  - Evaluation methodologies

- **Findings**
  - Expressiveness
  - Performance
    - Execution throughput
    - Scalability
      - Static data size
      - Number of current queries