

SLD Revolution: A Cheaper, Faster yet More Accurate Streaming Linked Data Framework

Marco Balduini, Emanuele Della Valle, and Riccardo Tommasini^(✉)

DEIB, Politecnico di Milano, Milano, Italy
{marco.balduini,emanuele.dellavalle,riccardo.tommasini}@polimi.it

Abstract. The RDF Stream Processing (RSP) is gaining momentum. The RDF stream data model is progressively adopted and many SPARQL extensions for continuous querying are converging to a unified RSP query language. However, the RSP community still has to investigate when transforming data streams in RDF streams pays off. In this paper, we report on several experiments on a revolutionized version of our Streaming Linked Data framework (namely, SLD Revolution). SLD Revolution (i) operates on time-stamped generic data items (events, tuples, trees and graphs), and (ii) it applies a lazy-transformation approach, i.e. it processes data according to their nature as long as possible. SLD Revolution results to be a cheaper (it uses less memory and has a smaller CPU load), faster (it reaches higher maximum input throughput), yet more accurate (it provides a smaller error rate in the results) solution than its ancestor SLD.

1 Introduction

RDF Stream Processing (RSP) is gaining momentum. The RSP W3C community group¹ has just reached 100 members. It is actively working on a report that will present the RDF Stream data model, the possible RDF Stream serializations and the syntax and semantics of the RSP-QL query language.

We started using RSP in 2011, when we won the Semantic Web Challenge with Bottari [1]. A key ingredient of our solution was the Streaming Linked Data framework [2] (SLD). SLD is a middleware that extends RSP engines with adapters, decorators and publishers. All these components observe and push RDF streams to a central RDF stream bus. The adapters are able to ingest any kind of external data, transforming them into RDF streams modeled as time-stamped RDF Graphs and push them to the bus. A network of C-SPARQL queries [3] analyzes the RDF streams in the bus, elaborate them, and push the results to other internal streams. Decorators can semantically enrich RDF streams using user defined functions (e.g., in Bottari to add the opinion that a social media user expresses about a given named entity in a micro-post). Publishers push data to the bus encoded in the Streaming Linked Data format [4].

¹ <https://www.w3.org/community/rsp/>.

SLD it is currently a key component of Fluxedo², i.e. a commercial solution for monitoring topics on social media. A typical deployment of SLD in Fluxedo processes in real-time thousands of micro-posts per minute on a 50 *e*/month machine in the cloud (8 GB of RAM and 4 cores), providing semantic analysis and sophisticated visualizations.

In five years of SLD usage, we learned that using RDF streams is valuable when *(i)* data are naturally represented as graphs, i.e. micro-posts in the larger social graph, and when *(ii)* the availability of popular vocabularies makes easy writing adapters that semantically annotate the incoming data, e.g. we wrote adapters that annotated streams from the major social networks using SIOC [5].

However, we have also found out several weaknesses of the approach:

- RDF streams cannot be found in the wild, yet. JSON is largely used in practice (e.g., Twitter Streaming APIs³ and W3C activity stream 2.0 working draft⁴).
- The results of C-SPARQL queries are often relational and forcing them into an RDF streams is not natural, i.e., a user would naturally use the REGISTER QUERY ... AS SELECT ... form instead of REGISTER STREAM ... AS CONSTRUCT ... one. It takes three triples to state how many times a hashtag appears in the micro-posts observed in 1 min, while the tuple (timestamp, hashtag, count) is more succinct.
- It is harder to express some computation using C-SPARQL over RDF streams and graphs than writing a path expression over JSON; or writing an SQL query over relations; or writing an EPL statement over events.
- SLD builds on the C-SPARQL engine and, thus, shares with it some shortcomings, i.e. it gives incorrect answers when it is overloaded engine [6, 7].

In this paper, we challenge the hypothesis that RDF streams should play such a central role in SLD. We investigate if *(i)* using time-stamped generic data items (instead of focusing only time-stamped RDF graphs) and *(ii)* processing them according to their event-, tuple-, tree- and graph-based nature, offer the opportunity to engineer a cheaper (uses less memory and CPU), faster (it reaches higher maximum input throughput) yet more accurate (i.t. with a smaller error in the results) version of SLD that we called SLD Revolution. We bring experimental evidence that supports the design decision of revolutionizing SLD in those two directions. Using our experience on social media monitoring, we design a set of experiments to evaluate our hypothesis: we chose the expected maximum rate of micro-posts per minute and the machine to deal with the worst-case scenario; by reducing the available memory and processor time requirement we push the overload status to an higher input rate.

The remainder of the paper is organized as follows. Section 2 introduces the state-of-the-art in stream processing with a focus on RSP. Section 3 presents SLD Revolution and its new processing model. Sections 4 and 5, respectively, describe

² <http://www.fluxedo.com/>.

³ <https://dev.twitter.com/streaming/overview>.

⁴ <http://www.w3.org/TR/activitystreams-core/>.

the settings and the results of the experiments we run. Finally, in Sect. 6, we conclude and present our future work.

2 State of the Art

Data model. RSP extends the RDF data model and the SPARQL query model in order to take into account the streaming nature of the data.

A **relational data stream** [9] S is defined as an unbounded sequence of time-stamped data items (d_i, t_i) : $S = (d_1, t_1), (d_2, t_2), \dots, (d_n, t_n), \dots$, where d_i is a relation and $t_i \in \mathbb{N}$ the associated time instant. Different approaches constrain t_i so that it holds either $t_i \leq t_{i+1}$, i.e. stream items are in a non-decreasing time order, or $t_i < t_{i+1}$, i.e. stream items are in strictly increasing time order.

An **RDF Stream** is defined in the same way, but d_i is either an RDF statement (as done in most of the RSP approaches [3, 10–12]) or an RDF graph (as done in SLD [2] and proposed by the RSP W3C community). An RDF statement is a triple $(s, p, o) \in (I \cup B) \times (I) \times (I \cup B \cup L)$, where I is the set of IRIs, B is the set of blank nodes and L is the set of literals. An RDF graph is a set of RDF statements.

Processing Model. Also the processing model of RSP [3, 10, 11] inherits from the work done in the database community. In particular, it is inspired by the CQL stream processing model (proposed by the DB group of the Stanford University [8]) which defines three classes of operators (Fig. 1.a):

- **stream-to-relation** operators are able to transform streams in relations. Since a stream is a potentially infinite bag of time-stamped data items, those operators extract finite bags of data enabling query answering. One of the most studied operator of this class is the *sliding window* that chunks the incoming streams into portions of length ω and slides of a length β
- **relation-to-relation** operators transform relations in other relations. Relational algebraic expressions are a well-known cases of this class of operators.
- **relation-to-stream** operators are optional and allow to output the results as a part of a stream. Alternatively, a time-varying relation is provided.

Figure 1.b presents the CQL model adapted to the RSP case. The stream and the relation concepts are mapped to RDF streams and to set of mappings (using

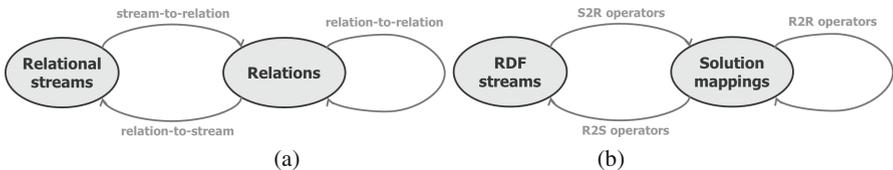


Fig. 1. The CQL model, cf. [8], and its adaptation for RDF stream processing

the SPARQL algebra terminology), respectively. To highlight the similarity of the RSP operators [12] to the CQL ones, similar names are used: **S2R**, **R2R** and **R2S** to indicate the operators respectively analogous to stream-to-relation, relation-to-relation and relation-to-stream operators.

RSP Middlewares. In order to ease the task of deploying the RSP Engine in real-world applications, three middleware were designed: the Linked Stream Middleware [13], a semantically enabled service architecture for mashups over streaming and stored data [14] and our SLD [2].

The three approaches fulfill similar requirements for the end user. They offer extensible means for real-time data collection, for publishing and querying collected information as Linked Data, and for visualizing data and query results. They differ in the approach. Our SLD and the Linked Stream Middleware take both a data driven approach, but they address in a different way the non-functional requirements; while SLD is an in-memory solution for stream processing of RDF streams with limited support for static information, the Linked Stream Middleware is a cloud-based infrastructure to integrate time-dependent data with other Linked Data sources. The middleware described in [14], instead, takes a service oriented approach, thus it also includes service discovery and service composition among its features.

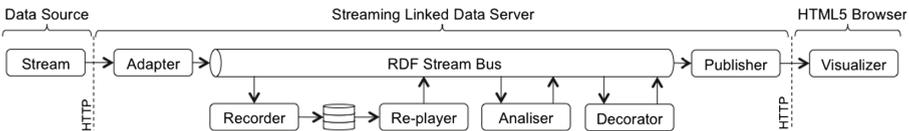


Fig. 2. The architecture of the streaming linked data framework [2].

Figure 2 illustrates the architecture of SLD that offers: (i) a set of adapters that transforms heterogeneous data streams in RDF streams attaching to each received element a time-stamp that identifies the ingestion time (e.g., a stream of micro-posts in JSON as an RDF stream using the SIOC vocabulary [5] or a stream of weather sensor observations in XML using the Semantic Sensor Network vocabulary [15]); (ii) a publish/subscribe bus to internally manage RDF streams, (iii) some facilities to record and replay RDF streams; (iv) a set of user defined components to decorate an RDF stream (e.g., adding sentiment annotations to micro-posts); (v) a wrapper for the C-SPARQL Engine [3] that allows to create networks of C-SPARQL queries, and (vi) a linked data server to publish results following the Streaming Linked Data Format [4].

3 SLD Revolution and Its Processing Model

SLD Revolution adopts *generic programming* [16] where continuous processing operators are expressed in terms of types *to-be-specified-later*. This idea - which was pioneered in [17] - can be adapted to stream processing by choosing to

model the element d_i in the stream S as time-stamped generic data items that are instantiated when needed for specific types provided as parameters.

Figure 3 illustrates the SLD Revolution processing model. The stream and the relation concepts of CQL are mapped to **generic data streams** $S\langle T \rangle$ and to **instantaneous generic data items** $I\langle T \rangle$.

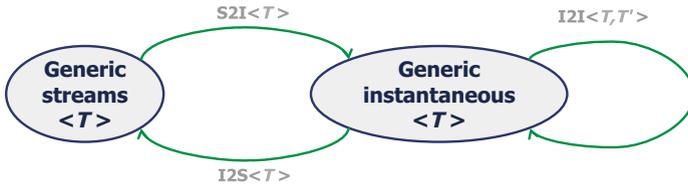


Fig. 3. The processing model of the SLD revolution framework.

In line with CQL and RSP-QL, SLD Revolution proposes three classes of operators:

- The **stream-to-instantaneous S2I** $\langle T \rangle$ operators transform the infinite **generic data stream** $S\langle T \rangle$ in to a finite bag of **instantaneous generic data items** $I\langle T \rangle$.
- The **instantaneous-to-instantaneous I2I** $\langle T, T' \rangle$ operators transform **instantaneous generic data items** $I\langle T \rangle$ into other **instantaneous generic data items** $I\langle T' \rangle$, where T and T' can be of the same type or of different types. For instance, a C-SPARQL query of the type REGISTER QUERY ... AS SELECT ... takes in input time-stamped RDF graphs and generates as output time-stamped relations.
- The **instantaneous-to-stream I2S** $\langle T \rangle$ operators transform **instantaneous generic data items** $I\langle T \rangle$ into a **generic data stream** $S\langle T \rangle$.

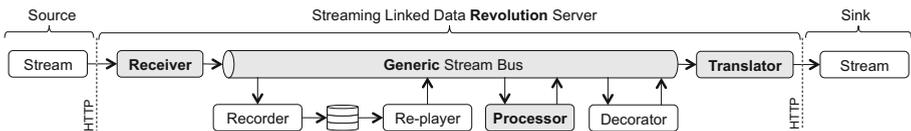


Fig. 4. The architecture of the SLD revolution framework. In gray the components that were redesign to adopt the generic programming approach.

SLD Revolution generalizes SLD architecture (cf. Fig. 4 with Fig. 2). The Generic Stream Bus replace of the RDF stream bus. The receivers replace the adapters. As the adapters they allow to ingest external data streams, but they no longer transform the received events in time-stamped RDF graphs. Data items remain in their original form, only the ingestion time is added, postponing

the transformation to the moment when it is required (we name this approach *lazy transformation*). The processors substitute the C-SPARQL-based analyzers. The C-SPARQL engine remains as one of the possible processors, but SLD Revolution can be extended with any continuous processing engine. The current implementation includes the Complex Event Processor Esper, the SPARQL Engine Jena-ARQ that operates on time-stamped RDF graphs one at a time, and a custom JSON path expression evaluation engine built on gson (<https://github.com/google/gson>). Translators generalize publishers, which are specific for the Streaming Linked Data format [4], allowing SLD Revolution to output in alternative formats.

4 Experimental Settings

In this Section, we present the experimental settings of this work. As domain we chose Social Network analysis as done by the Linked Data Benchmark Council (LDBC) in the SNBench⁵. We first explain the type of data we used for our experiments. Then, we explain how the data were sent to SLD and SLD Revolution. We describe the two continuous processing pipelines that we registered in SLD and in SLD Revolution. Finally, we state which key performance indicators (KPIs) we measure and how.

Input Data. SLD and SLD Revolution receive information in the same way, they both connect to a web socket server and handle JSON-LD files.

```

1 {"@context": { ... }, "@type": "Collection", "totalItems": 1,
2   "prov:wasAssociatedWith": "sr:Twitter",
3   "items": [{
4     "@type": "Post",
5     "published": "2016-04-26T15:40:03.054+02:00",
6     "actor": {"@type": "Account", "@id": "user:1", "sioc:name": "
7       @streamreasoning"},
8     "object": {
9       "@type": "Content", "@id": "post:2", "alias": "http://.../2",
10      "prov:wasAssociatedWith": "sr:Twitter",
11      "sioc:content": "You ARE the #socialmedia!",
12      "dct:language": "en",
13      "tag": [{"@type": "Tag", "@id": "tag:3", "displayName": "socialmedia"}]}
  ]}]

```

Listing 1. JSON representation of a Twitter micro-post. Due to the lack of space we omitted the context declaration that contains the namespace.

In Listing 1, we propose a JSON-LD serialization of the Activity Stream representation of a tweet as it was injected during the experiments in both systems. The JSON-LD representation of an Activity Stream is a *Collection* (specified by *@type* property) composed by one or more social media items. The *Collection* is described with two properties, i.e., *totalItems* and *prov:wasAssociatedWith*, which tell respectively the number of items and the provenance of the items. The collection in the example contains a *Post* created on 2016-04-26 (*published* property) by an *actor* (Line 6) that produce the *object* (Lines 7-13). The *Actor*

⁵ <http://www.ldbcouncil.org/benchmarks/snb>.

has a unique identifier *@id*, a *displayName*, a *sioc:name* and a *alias*. The *Object* has a *sioc:content*, a *dct:language*, zero or more *tags*, and optionally a *url* and a *to* to represent, respectively, links to web pages and mentions of other actors.

```

1 <post:2> a sma:Tweet ;
2   dcterms:created "2016-04-26T15:40:03.054+02:00"^^xsd:dateTime ;
3   dcterms:language "en"^^xsd:string ;
4   sioc:content "You ARE the #socialmedia!"^^xsd:string ;
5   sioc:has_container "Twitter"^^xsd:string ;
6   sioc:has_creator <user:1> ;
7   sioc:id "2"^^xsd:string ;
8   sioc:link "http://.../status/2"^^xsd:string ;
9   sioc:topic <tag:3> .
10 <tag:3> a sioc:Tag ;
11   rdfs:label "socialmedia"^^xsd:string .
12 <user:1> a sioc:UserAccount ;
13   sioc:account_of "StreamReasoning"^^xsd:string ;
14   sioc:creator_of <post:2> ;
15   sioc:id "1"^^xsd:string ;
16   sioc:name "@streamreasoning"^^xsd:string .

```

Listing 2. RDF N3 representation of a Twitter micro-post

Listing 2 shows the RDF produced by the SLD adapter in transforming the JSON-LD in Listing 1. The translation operation exploits well known vocabularies, in particular *sioc* to represent the online community information, *prov* to track the provenance of an item and *dcterms* to represents information about the *object*.

Sending data. A test consists of sending a constant amount of synthetic data using the JSON-LD serialization presented in Listing 1. The data is sent in chunks three times per minute (i.e. at the 10th, the 30th and the 50th seconds of the minute). Each chunk contains the same amount of posts. We tested the configuration for different rates: 1500 posts per minute (i.e., three chunks of 500 posts), 3000 posts per minute, 6000 posts per minute, 9000 posts per minute, 12000 posts per minute and 18000 posts per minute.

The rates and the input methodology were chosen based on our experiences on social monitoring (see Sect. 1). They test a normal situation for SLD (1500 and 3000 posts per minutes) as well as situations that we know to overload SLD (more than 6000 posts per minute).

Pipelines. We tested SLD and SLD revolution with different pipelines:

- the *area chart* pipeline computes the number of tweets observed over time. It uses a 15 min long window that slides every minute. The results can be continuously computed (*i*) using a generic sliding window operator, which works looking only to the time-stamps of the data items in the generic stream, and (*ii*) accessing with a path expression the *totalItems* property in the JSON-LD file, i.e., the number of items in the collection.
- the *bar chart* pipeline counts how often hashtags appear in the tweets received in the last 15 min. As the area chart pipeline, the window slides every minute. In this second pipeline, RDF streams are adequate and it is convenient to write a C-SPARQL query that counts the number of times each hashtag appears.

The two pipelines are coded in SLD and SLD Revolution in two different ways. SLD performs the transformations of JSON-LD in RDF by default, on all the input data, independently from the task to perform. SLD Revolution keeps the data in its original format as much as possible, i.e., it performs *lazy transformations*.

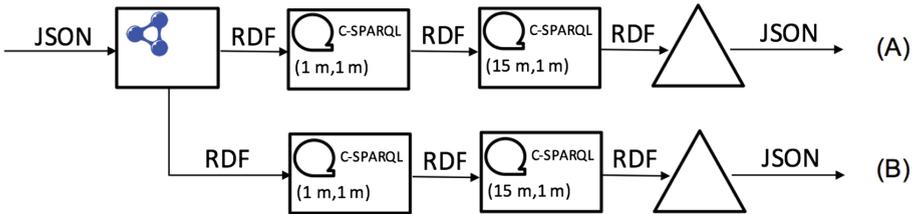


Fig. 5. SLD pipeline

SLD Pipelines. Figure 5 presents the two pipelines in SLD. The input data are translated in RDF as soon as they enter the pipelines. The computations for the area chart and for the bar chart (see the part marked with **A** and **B**) are composed by the same type of components and share the new RDF stream translated by the *Adapter*.

The pipeline **A** uses two C-SPARQL queries. The first (see Listing 3) applies a tumbling window of 1 min⁶ and counts the tweets.

The second aggregates the results from the first query using a 15 min time window that slides every minute (see Listing 4).

```

1 REGISTER STREAM presocialstr AS
2 CONSTRUCT { ?id sma:twitterCount ?twitterC }
3 FROM STREAM <http://.../socialstr> [RANGE 1m STEP 1m]
4 WHERE { SELECT (uuid() AS ?id) ?twitterC
5           WHERE { SELECT (COUNT (DISTINCT ?mp) AS ?twitterC)
6                     WHERE { ?mp a sma:Tweet } } }

```

Listing 3. C-SPARQL pre-query for the area chart that applies a tumbling window of 1 min and counts the tweets.

```

1 REGISTER STREAM ac AS
2 CONSTRUCT { ?uid sma:twitterCount ?totTwitter ; sma:created_during ?
3             unixTimeFrame }
3 FROM STREAM <http://.../presocialstr> [RANGE 15m STEP 1m]
4 WHERE {
5   SELECT (uuid() AS ?uid) ?unixTimeFrame (SUM(?twitter) AS ?totTwitter)
6   WHERE { ?id sma:twitterCount ?twitter ; sma:created_during ?timeFrame .
7           ?timeFrame a sma:15mTimeFrame ; sma:inUnixTime ?unixTimeFrame
8   }
8 GROUP BY ?unixTimeFrame }

```

Listing 4. C-SPARQL query for the area chart that aggregates the results from the query in Listing 3 using a 15 min time window that slides every minute.

⁶ A tumbling window is a sliding window that slides for its length.

It is worth to note that the first query is an important optimization in terms of memory consumption. It avoids the engine to keep 15 min of tweets to only count them. In SLD we often use this design pattern, we call this first query a *pre-query*.

Pipeline **B** also exploits this design; it applies a pre-query to reduce the amount of data and then a query to produce the final result.

It is also worth to note that all the C-SPARQL queries use the form REGISTER STREAM ... AS CONSTRUCT ..., because RDF streams are the only means of communication between SLD components.

The last components of both pipelines are publishers that make the results available to external process outside SLD. In this case, the publisher writes JSON files on disk.

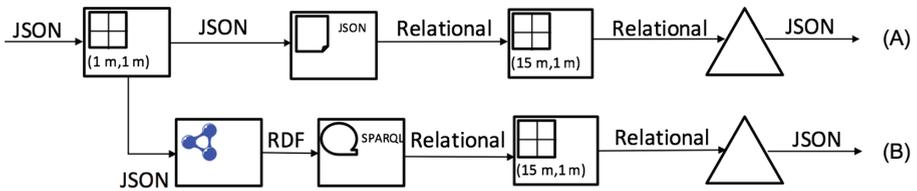


Fig. 6. SLD revolution pipeline

SLD Revolution Pipelines. Figure 6 presents the pipelines in SLD Revolution. As for SLD, the pipeline **A** is for the area chart, while **B** is for the bar chart. The first component is no longer an adapter. The data directly enter SLD revolution in JSON-LD format. The first query is a generic 1 min long tumbling window implemented with the EPL statement in Listing 5. `FORCE_UPDATE` and `START_EAGER` tells the stream processing engine, respectively, to emit also empty reports and to start processing the window as soon as the query is registered (i.e., without waiting for the first time-stamped data item to arrive). It is worth to note that this query exploits the event-based nature of the generic stream it is observing; it does not inspect the payload of the events, it only uses their time-stamps.

```
select * from GenericEvent.win:time_batch(1\,min, "FORCE_UPDATE,
START_EAGER")
```

Listing 5. The generic window query in common to both the pipelines in SLD Revolution

The `FORCE_UPDATE` flow control keyword instructs the view to post an empty result set to listeners if there is no data to post for an interval. When using this keyword the `irstream` keyword should be used in the select clause to ensure the remove stream is also output. Note that `FORCE_UPDATE` is for use with listeners to the same statement and not for use with named windows. Consider output rate limiting instead.

The `START_EAGER` flow control keyword instructs the view to post empty result sets even before the first event arrives, starting a time interval at statement creation time. As when using `FORCE.UPDATE`, the view also posts an empty result set to listeners if there is no data to post for an interval, however it starts doing so at time of statement creation rather than at the time of arrival of the first event.

As explained in Sect. 3, processors are the central components of SLD Revolution. They can listen to one or more generic streams, compute different operations and push out a generic streams. The type of the input and output streams can be different. The two pipelines uses different processors (e.g. RDF translator, windower and SPARQL).

SLD Revolution maintains the data format as long as possible in order to reduce the overhead of the translations. It can exploit the tree-based nature of JSON-LD. In pipeline **A**, it exploits a path expression data to extract *totalItems*, i.e., the number of items in each collection, from the time-stamped JSON-LD items in the generic stream it listens to. It outputs a tuple $\langle \text{timeframe}, \text{count} \rangle$ that is aggregated every minute over a window of 15 min using an EPL statement.

The Pipeline **B** of SLD Revolution translate JSON-LD in RDF in order to extract information about the hashtags. As for the pipeline **B** of SLD, we use a pre-query design pattern to reduce the amount of data. A SPARQL processor applies the `SELECT` query in Listing 6 to every data-item in the generic stream it listens to and pushes out a stream of tuples $\langle \text{hashtagLabel}, \text{count} \rangle$. The relational stream is then aggregated with an esper processor with a 15 min time window that slides every 1 min (see Listing 7).

```

1 SELECT ?htlabel (COUNT(DISTINCT(?mpTweet)) AS ?htTweetCount)
2 WHERE { ?mpTweet a sma:Tweet ; sioc:topic ?tweetTopic .
3         ?tweetTopic a siotypes:Tag ; rdfs:label ?htlabel }
4 GROUP BY ?htlabel
5 ORDER BY desc(?htTweetCount)

```

Listing 6. SPARQL pre-query for the bar chart

```

select htlabel, sum(count) as sumHt from HTCCountEvent.win:time(15\,min)
group by htlabel output snapshot every 1\,min

```

Listing 7. EPL query for the bar chart

KPIs. As key performance indicators (KPIs), we measure the resources consumption of the two systems and the correctness of the results. For the resource consumption we measure every 10 seconds: (i) the CPU load of the system thread in percent, (ii) the memory consumption of the thread in megabytes and (iii) the memory consumption of the Java Virtual Machine (JVM). For the correctness, we compared the computed results with the expected results. Being the input a constant flows of tweets that only differ for the ID, the area chart is expected to be flat and the bar chart is expected to count exactly the same number of hashtags every minute.

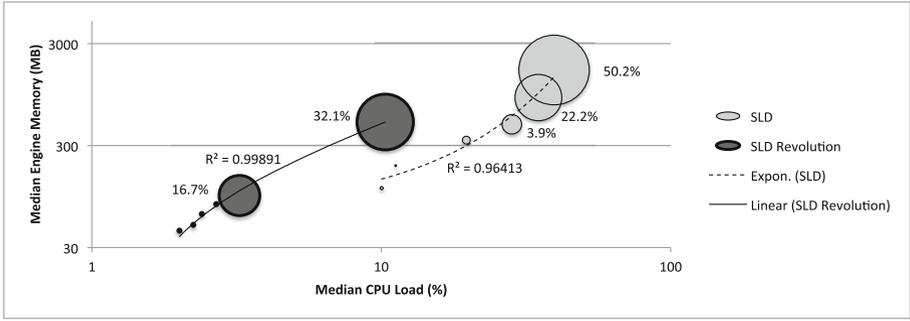


Fig. 7. An overview of the experimental results; larger bubbles means greater % errors.

5 Evaluation Results

Figure 7 offers an overview of the results of the experiments. The full results are reported at the end of this section in Fig. 10. On the X axis we plot the median of the CPU load in percent, while on the Y axis we plot the memory allocated by the engine thread. The size of the bubble maps the median of the error of the area chart. Bubbles in the lower left corner correspond to the experiment where we sent 1500 tweets per minute.

Increasing the throughput results in more memory consumption and CPU load for both systems. However, not SLD Revolution consumes less memory than SLD and occupies less CPU. Moreover, SLD Revolution presents a linear increment for both these KPIs, while the resource usage for SLD grows exponentially with the throughput. Also the error in the results increases with the throughput: SLD already shows an error greater than 3% in the bar chart at 3000 tweets per minutes and in the area chart at 9000 tweets per minute; SLD Revolution is faster - i.e. it reaches higher maximum input throughput - and more accurate - i.e. it reaches 3% error level only for 18000 tweets per minutes, providing more precise results than SLD.

Figure 8 presents the recorded time-series for CPU load and memory usage in both systems. The memory usage graphs contains two different time series. The blue one represents the memory usage of the system thread, while the orange one shows the total memory usage for the JVM.

The memory usage of the system thread accounts for all the components and data in the pipeline. Notably, when the system under testing is not overloaded, the memory usage is constant over time, while when the system is overloaded it grows until the system crashes. The total memory usage of the JVM shows, instead, the typical pattern of the garbage collector that lets the JVM memory grow before freeing it. Also in this case, when the system it is overloaded, the garbage collector fails to free the memory.

During the experiments the median of the memory used by SLD spans from 115 MB, when loaded with at 1500 posts/min, to 1.6 GB, when loaded with 18000 posts/min. For SLD Revolution, instead, it spans from 44 MB to 511.5 MB

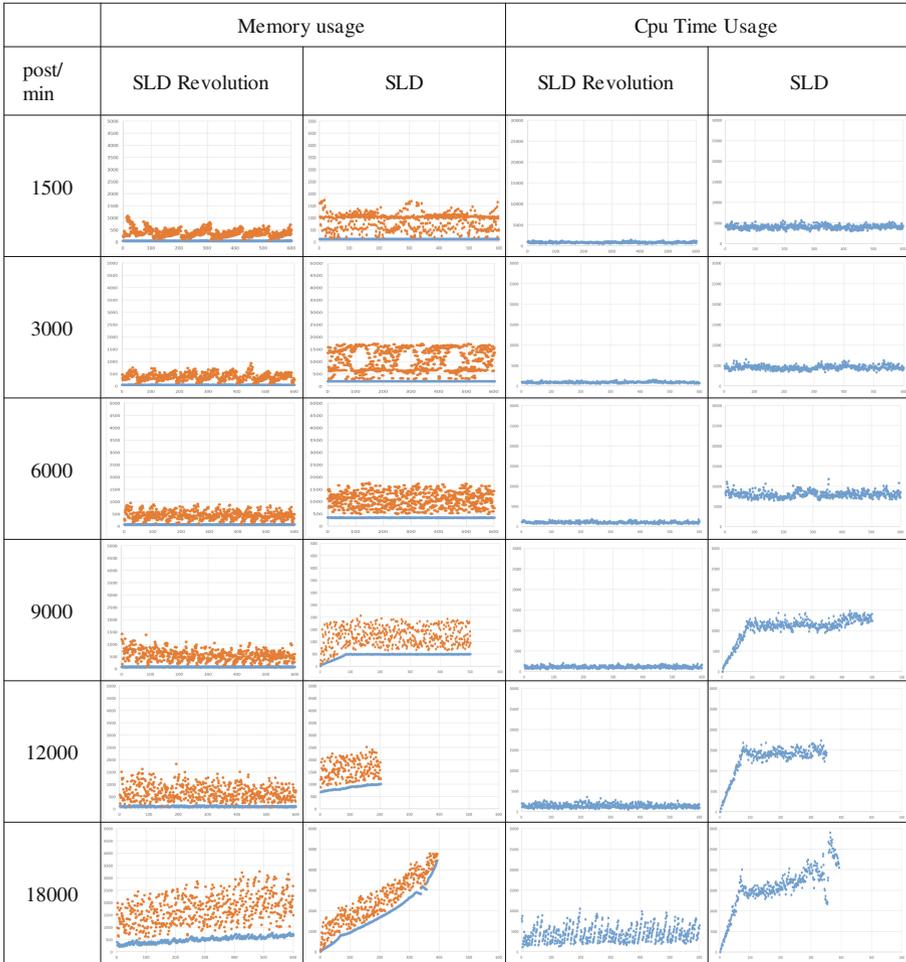


Fig. 8. Memory and CPU usage over time

in the same load conditions. The experimental results clearly shows that SLD Revolution consumes (in average) three times less memory than its ancestor.

The same considerations can be proposed for the CPU load. The median of the CPU load spans from 2% to 10% for SLD Revolution, while it spans from 10% to 39.5% for SLD. SLD Revolution consumes in average 4 time less CPU time than SLD. Moreover, offers higher level of stability for both the parameters in all the experiments. The memory usage and the CPU loads clearly explode at higher input rate and allow the machine to produce results for a higher load in input.

The correctness results are summarized in Fig. 9. As explained in Sect. 4, the percentage of errors is computed by comparing the results for each time interval

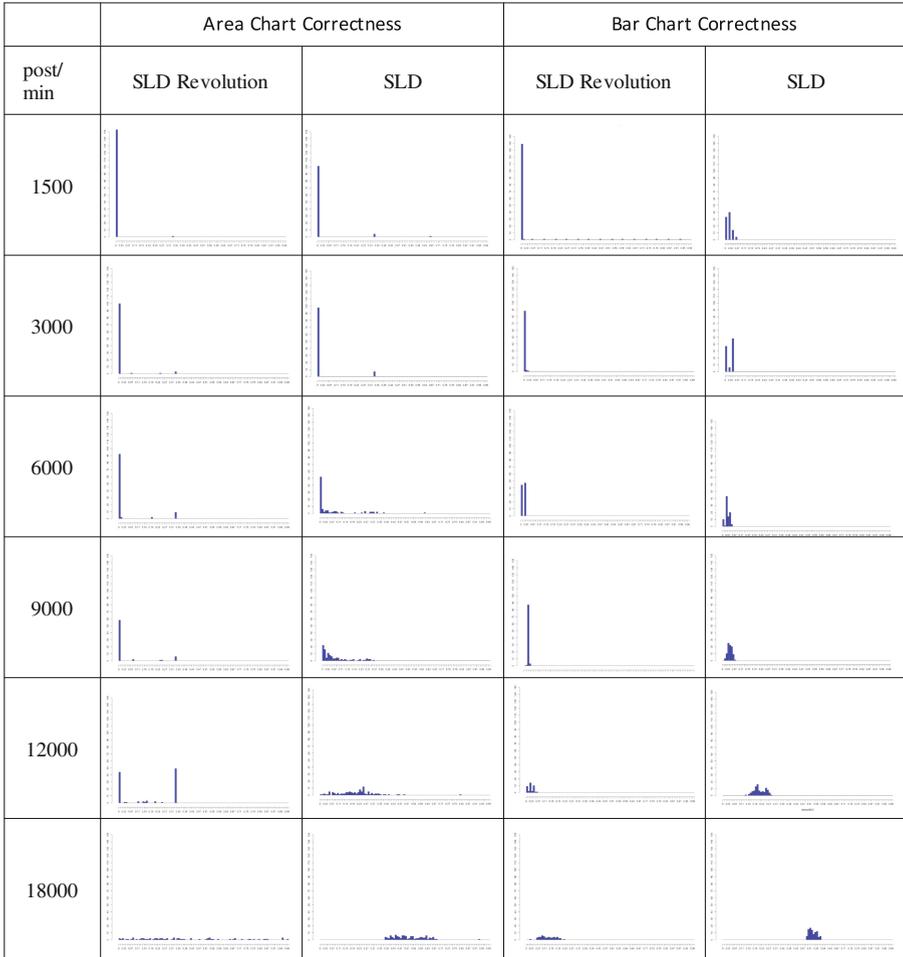


Fig. 9. Area chart and bar chart errors distributions

with the expected ones. The X axis of each plot shows the percentage of error; it ranges from 0% to 100%. The Y axis is the percentage of results with that error; it also ranges from 0% to 100%. A bar as tall as the Y axis in the left side of the graph means that all results were correct. The smaller that bar is and the greater the number of bars to the right is, the more errors were observed.

In general, the results show that SLD Revolution is more accurate (the result error is smaller) than SLD. For the area chart the distribution shows that SLD Revolution percentage of error is very low when the input throughput is between 1500 posts/min and 9000 posts/min. When it is higher (i.e., 12000 and 18000 posts/min) also SLD Revolution starts suffering and percentage of errors starts growing. For SLD, errors are present even at lower input rate, the graph

| post/min | KPI | SLD | | | | | SLD Revolution | | | | |
|----------|------------------|-------|---------|--------|---------|-------|----------------|---------|--------|---------|--------|
| | | Min. | 1st Qu. | Median | 3rd Qu. | Max. | Min. | 1st Qu. | Median | 3rd Qu. | Max. |
| 1500 | memory (MB) | 112 | 114 | 115 | 115 | 116 | 42 | 43 | 44 | 45 | 48 |
| 3000 | | 188 | 190 | 191 | 192 | 193 | 45 | 48 | 50 | 52 | 58 |
| 6000 | | 330 | 337 | 340 | 342 | 353 | 52 | 61 | 64 | 67 | 78 |
| 9000 | | n.a. | 481 | 485 | 488 | 504 | 59 | 73 | 80 | 84 | 102 |
| 12000 | | 684 | 783 | 888 | 956 | 1015 | 66 | 88 | 97 | 107 | 144 |
| 18000 | | n.a. | 919 | 1652 | 2565 | 4451 | 212 | 392 | 511,5 | 597,2 | 774 |
| 1500 | CPU load (%) | 7,3% | 9,3% | 10,0% | 10,8% | 14,1% | 1,3% | 1,8% | 2,0% | 2,2% | 3,4% |
| 3000 | | 8,3% | 10,5% | 11,2% | 12,1% | 16,2% | 1,3% | 2,0% | 2,2% | 2,5% | 3,8% |
| 6000 | | 16,1% | 18,6% | 19,7% | 21,0% | 29,5% | 1,4% | 2,1% | 2,4% | 2,8% | 4,5% |
| 9000 | | n.a. | 26,3% | 28,3% | 30,4% | 37,0% | 1,5% | 2,3% | 2,7% | 3,2% | 5,1% |
| 12000 | | n.a. | 32,6% | 34,9% | 36,9% | 43,3% | 1,3% | 2,6% | 3,2% | 4,2% | 8,9% |
| 18000 | | n.a. | 35,2% | 39,6% | 46,1% | 72,5% | 2,8% | 7,2% | 10,3% | 13,8% | 26,2% |
| 1500 | area chart error | 0,0% | 0,0% | 0,0% | 0,1% | 66,7% | 0,0% | 0,0% | 0,0% | 0,0% | 33,3% |
| 3000 | | 0,0% | 0,0% | 0,0% | 0,1% | 33,4% | 0,0% | 0,0% | 0,0% | 0,0% | 33,3% |
| 6000 | | 0,0% | 0,1% | 0,6% | 8,1% | 61,8% | 0,0% | 0,0% | 0,0% | 0,0% | 33,3% |
| 9000 | | 0,1% | 1,2% | 3,9% | 9,5% | 30,2% | 0,0% | 0,0% | 0,0% | 0,0% | 33,3% |
| 12000 | | 0,1% | 14,1% | 22,2% | 26,0% | 82,4% | 0,0% | 0,0% | 16,7% | 33,3% | 33,3% |
| 18000 | | 38,2% | 44,8% | 50,2% | 59,5% | 93,1% | 0,0% | 16,9% | 32,1% | 53,7% | 100,0% |
| 1500 | bar chart error | 0,0% | 0,0% | 2,2% | 2,3% | 6,7% | 0,0% | 0,0% | 0,0% | 0,0% | 93,3% |
| 3000 | | 0,0% | 0,1% | 4,5% | 4,5% | 4,5% | 0,0% | 0,0% | 0,0% | 0,1% | 2,2% |
| 6000 | | 0,1% | 2,3% | 2,3% | 4,1% | 5,7% | 0,1% | 0,1% | 2,3% | 2,3% | 2,3% |
| 9000 | | 1,8% | 3,4% | 4,2% | 5,4% | 7,5% | 1,6% | 2,4% | 2,4% | 2,4% | 3,2% |
| 12000 | | 13,6% | 19,4% | 21,0% | 25,0% | 28,0% | 0,2% | 1,5% | 2,5% | 4,7% | 6,1% |
| 18000 | | 49,5% | 51,1% | 52,6% | 54,7% | 57,9% | 2,9% | 9,4% | 12,6% | 16,3% | 22,4% |

Fig. 10. The experimental results.

shows that the error distribution starts moving to the right at 6000 posts/min. Similar consideration can be proposed for the bar chart error distribution. The degradation of performance of SLD starts a very low rate, a substantial presence of errors around 7% can be seen with 6000 posts/min in input.

Figures 8 and 9 show the deep correlation between resources usage and errors. Clearly, a growing input throughput drives the systems to be less reliable. For both the versions of the SLD framework (SLD Revolution and SLD) the correctness of the results decreases as soon as the machine is overloaded and the resources usage starts rising out of control.

6 Conclusions and Future Works

In our future work, we intend: (i) to empirically demonstrate the value of using SLD Revolution for all our deployments of Fluxedo, and (ii) to investigate if SLD Revolution can be the target platform for a new generation of Ontology Based Data Integration [18] system for Stream Reasoning [19]. This system could have the potential to tame the velocity and variety dimension of Big Data simultaneously.

As for the former, we first intend to stress test SLD Revolution using workloads that resemble reality. Then, we aim at putting it at work in parallel to

SLD in real-world deployments. Once we will have collected enough evidence that SLD Revolution is always cheaper faster yet more accurate than SLD, we will start using it for all our deployments.

As for the latter, we aim at further investigating the generic processing model presented in Sect. 3. We are defining an algebra able to capture the semantics of complex stream processing applications that need to integrate a variety of data sources. The current sketch of this algebra uses **S2I** and **I2S** operators from CQL [8] but keeping them generic w.r.t. the payloads. It uses SPARQL 1.1 algebra as **I2I** operators that take in input graph-based payloads and generate in output either graph-based or tuple-based payloads. The relational algebra will cover the **I2I** transformations of tuple-based payloads. We are studying the application of R2RML [20] for formulating mappings that works as **I2I** operators. Indeed, R2RML allows to write mapping from relational data to RDF; more generally, **I2I** operators takes in input tuple-based payloads and output graph-based ones. We still need to choose an algebra for transforming tree-based payloads.

In our opinion, the grand challenge is how to fit all those formal elements in a coherent framework that allows a system to automatically decide which is the latest moment for transforming data (i.e., introducing the concept of *lazy transformation*) and to perform optimization such as introducing the pre-query that we put in all the pipelines illustrated in Sect. 4.

When the work on this formally defined generic stream processing model will be completed, we will be able to start investigating how to extend mapping languages like R2RML⁷ and, potentially, also ontological languages in order to make them time-aware [21] while keeping the whole computational problem tractable.

Acknowledgement. We thank the reviewers of the 2nd RDF Stream Processing Workshop co-located with ESWC 2017 for their valuable comments. They allowed us to refine this version of [22] for the ESWC 2017 workshops post-proceedings.

References

1. Balduini, M., Celino, I., Dell’Aglia, D., Della Valle, E., Huang, Y., Lee, T.K., Kim, S., Tresp, V.: BOTTARI: an augmented reality mobile application to deliver personalized and location-based recommendations by continuous analysis of social media streams. *J. Web Sem.* **16**, 33–41 (2012)
2. Balduini, M., Della Valle, E., Dell’Aglia, D., Tsytsarau, M., Palpanas, T., Confalonieri, C.: Social listening of city scale events using the streaming linked data framework. In: [23], pp. 1-16
3. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Querying RDF streams with C-SPARQL. *SIGMOD Record* **39**(1), 20–26 (2010)
4. Barbieri, D.F., Della Valle, E.: A proposal for publishing data streams as linked data - a position paper. In: Bizer, C., Heath, T., Berners-Lee, T., Hausenblas, M. (eds.): *Proceedings of the WWW2010 Workshop on Linked Data on the Web, LDOW 2010*, Raleigh, 27 April 2010, Vol. 628 of CEUR Workshop Proceedings. CEUR-WS.org (2010)

⁷ <https://www.w3.org/TR/r2rml/>.

5. Breslin, J.G., Decker, S., Harth, A., Bojars, U.: Sioc: an approach to connect web-based communities. *IJWBC* **2**(2), 133–142 (2006)
6. Le-Phuoc, D., Dao-Tran, M., Pham, M.-D., Boncz, P., Eiter, T., Fink, M.: Linked stream data processing engines: facts and figures. In: Cudré-Mauroux, P., Heflin, J., Sirin, E., Tudorache, T., Euzenat, J., Hauswirth, M., Parreira, J.X., Hendler, J., Schreiber, G., Bernstein, A., Blomqvist, E. (eds.) *ISWC 2012*. LNCS, vol. 7650, pp. 300–312. Springer, Heidelberg (2012). doi:[10.1007/978-3-642-35173-0_20](https://doi.org/10.1007/978-3-642-35173-0_20)
7. Dell’Aglio, D., Calbimonte, J.-P., Balduini, M., Corcho, O., Della Valle, E.: On correctness in RDF stream processor benchmarking. In: [23], pp. 326–342
8. Arasu, A., Babu, S., Widom, J.: The cql continuous query language: semantic foundations and query execution. *VLDB J.* **15**(2), 121–142 (2006)
9. Garofalakis, M., Gehrke, J., Rastogi, R.: *Data Stream Management: Processing High-Speed Data Streams (Data-Centric Systems and Applications)*. Springer-Verlag, New York (2007)
10. Le-Phuoc, D., Dao-Tran, M., Xavier Parreira, J., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) *ISWC 2011*. LNCS, vol. 7031, pp. 370–388. Springer, Heidelberg (2011). doi:[10.1007/978-3-642-25073-6_24](https://doi.org/10.1007/978-3-642-25073-6_24)
11. Calbimonte, J.-P., Corcho, O., Gray, A.J.G.: Enabling ontology-based access to streaming data sources. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) *ISWC 2010*. LNCS, vol. 6496, pp. 96–111. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-17746-0_7](https://doi.org/10.1007/978-3-642-17746-0_7)
12. Dell’Aglio, D., Della Valle, E., Calbimonte, J., Corcho, Ó.: RSP-QL semantics: a unifying query model to explain heterogeneity of RDF stream processing systems. *Int. J. Semantic Web Inf. Syst.* **10**(4), 17–44 (2014)
13. Le Phuoc, D., Nguyen-Mau, H.Q., Parreira, J.X., Hauswirth, M.: A middleware framework for scalable management of linked streams. *J. Web Sem.* **16**, 42–51 (2012)
14. Gray, A.J.G., García-Castro, R., Kyzirakos, K., Karpathiotakis, M., Calbimonte, J.-P., Page, K., Sadler, J., Frazer, A., Galpin, I., Fernandes, A.A.A., Paton, N.W., Corcho, O., Koubarakis, M., Roure, D., Martinez, K., Gómez-Pérez, A.: A semantically enabled service architecture for mashups over streaming and stored data. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., Leenheer, P., Pan, J. (eds.) *ESWC 2011*. LNCS, vol. 6644, pp. 300–314. Springer, Heidelberg (2011). doi:[10.1007/978-3-642-21064-8_21](https://doi.org/10.1007/978-3-642-21064-8_21)
15. Compton, M., Barnaghi, P.M., Bermudez, L., Garcia-Castro, R., Corcho, Ó., Cox, S., Graybeal, J., Hauswirth, M., Henson, C.A., Herzog, A., Huang, V.A., Janowicz, K., Kelsey, W.D., Le Phuoc, D., Lefort, L., Leggieri, M., Neuhaus, H., Nikolov, A., Page, K.R., Passant, A., Sheth, A.P., Taylor, K.: The ssn ontology of the w3c semantic sensor network incubator group. *J. Web Sem.* **17**, 25–32 (2012)
16. Jazayeri, M., Loos, R., Musser, D.R. (eds.): *Generic Programming*. Springer, Heidelberg (2000). doi:[10.1007/3-540-39953-4](https://doi.org/10.1007/3-540-39953-4)
17. Milner, R., Morris, L., Newey, M.: *A logic for computable functions with reflexive and polymorphic types*. University of Edinburgh, Department of Computer Science (1975)
18. Lenzerini, M.: Data integration: a theoretical perspective. In: *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 233–246. Madison, 3–5 June 2002
19. Della Valle, E., Ceri, S., van Harmelen, F., Fensel, D.: It’s a streaming world! reasoning upon rapidly changing information. *IEEE Intell. Syst.* **24**(6), 83–89 (2009)

20. Priyatna, F., Corcho, Ó., Sequeda, J.: Formalisation and experiences of r2rml-based SPARQL to SQL query translation using morph. In: Chung, C., Broder, A.Z., Shim, K., Suel, T. (eds.) 23rd International World Wide Web Conference WWW 2014, pp. 479–490. Seoul, 7–11 April 2014. ACM (2014)
21. Artale, A., Kontchakov, R., Ryzhikov, V., Zakharyashev, M.: A cookbook for temporal conceptual data modelling with description logics. *ACM Trans. Comput. Log.* **15**(3), 25:1–25:50 (2014)
22. Balduini, M., Della Valle, E., Tommasini, R.: SLD revolution: a cheaper, faster yet more accurate streaming linked data framework. In: Joint Proceedings of the 2nd RDF Stream Processing (RSP 2017) and the Querying the Web of Data (QuWeDa 2017) Workshops Co-located with 14th ESWC 2017, pp. 1–15. ESWC (2017)
23. Alani, H., Kagal, L., Fokoue, A., Groth, P.T., Biemann, C., Parreira, J.X., Aroyo, L., Noy, N.F., Welty, C., Janowicz, K. (eds.): *The Semantic Web - ISWC 2013*. Springer, Heidelberg (2013). doi:[10.1007/978-3-642-41338-4](https://doi.org/10.1007/978-3-642-41338-4)