







RMLStreamer-SISO: An RDF Stream Generator from Streaming Heterogeneous Data

Sitt Min Oo¹(✉) , Gerald Haesendonck¹ , Ben De Meester¹ ,
and Anastasia Dimou^{2,3} 

¹ IDLab, Department of Electronics and Information Systems,
Ghent University – imec, Ghent, Belgium

{x.sittminoo,gerald.haesendonck,ben.demeester}@ugent.be

² Department of Computer Science, KULeuven, Leuven, Belgium
anastasia.dimou@kuleuven.be

³ AI – Flanders Make, Lommel, Belgium

Abstract. Stream-reasoning query languages such as CQELS and C-SPARQL enable query answering over RDF streams. Unfortunately, there currently is a lack of efficient RDF stream generators to feed RDF stream reasoners. State-of-the-art RDF stream generators are limited with regard to the velocity and volume of streaming data they can handle. To efficiently generate RDF streams in a scalable way, we extended the RMLStreamer to also generate RDF streams from dynamic heterogeneous data streams. This paper introduces a scalable solution that relies on a dynamic window approach to generate RDF streams with low latency and high throughput from multiple heterogeneous data streams. Our evaluation shows that our solution outperforms the state-of-the-art by achieving millisecond latency (compared to seconds that state-of-the-art solutions need), constant memory usage for all workloads, and sustainable throughput of around 70,000 records/s (compared to 10,000 records/s that state-of-the-art solutions take). This opens up the access to numerous data streams for integration with the semantic web.

Resource type: Software

License: MIT License

URL: <https://github.com/RMLio/RMLStreamer/releases/tag/v2.3.0>

Keywords: RML · Stream processing · Window joins · Knowledge graph generation

1 Introduction

An increasing portion of data are continuous in nature, e.g., sensor events, user activities on a website, or financial trade events. This type of data is known as data streams; sequences of unbounded tuples generated continuously in different rates and volumes [3]. Due to the temporal nature of data streams, low latency

computation of analytical results is needed to timely react in different use cases, e.g., fraud detection [9]. Thus, stream processing engines must efficiently handle low latency computation of varying velocity and volume.

On the one hand, different frameworks were proposed to handle data streams, e.g., Flink, Spark or Storm [6, 19, 26]. On the other hand, RDF stream processing (RSP) engines, e.g., CQELS and C-SPARQL [1, 5, 16], were widely studied and perform high-throughput analysis of RDF streams with low memory footprints [16]. Yet, these stream processing frameworks are not substantially used in the domain of RDF graph generation from streaming data sources, despite the demand of these mature RSP engines for more RDF streams.

Between data processing frameworks and stream processing engines, there are tools to generate RDF streams from heterogeneous data streams (e.g. SPARQL-Generate [17], RDFGen [21], TripleWave [18], Cefriel’s Chimera [22]). However, some of these tools are inefficient when the data stream starts to scale in terms of volume and velocity, such as TripleWave, and SPARQL-Generate. While other tools are not open sourced nor suitable for the mapping of streaming data, such as RDFGen, and Cefriel’s Chimera respectively. Overall, there are no RDF stream generators that keep up with the needs of stream reasoning engines while taking advantage of data processing frameworks to efficiently produce RDF streams.

In this paper, we present the RMLStreamer-SISO, a parallel, vertically and horizontally scalable stream processing engine to generate RDF streams from heterogeneous data streams of any format (e.g. JSON, CSV, XML, etc.). We extended previous preliminary work [13] of heterogeneous data stream mapping solution: an open source implementation on top of Apache Flink [6], available under MIT license, which generates high volume RDF data from high volume heterogeneous data. RMLStreamer-SISO extends RMLStreamer to also support any input data streams and export RDF streams (Stream-In-Stream-Out (SISO)). RMLStreamer-SISO now supports a much larger part of the RML specification¹, including all features of RML but relational databases.

The RMLStreamer-SISO outperforms the state-of-the-art tools when handling high velocity data stream, increasing the throughput it could handle while maintaining low latency. The RMLStreamer-SISO achieves millisecond latency, as opposed to seconds that state-of-the-art solutions need, constant memory usage for all workloads, and sustainable throughput of around 70,000 records/s, compared to 10,000 records/s that state-of-the-art solutions take.

Through the utilization of a low-latency tool like RMLStreamer-SISO, legacy streaming systems could exploit the unique characteristics of real-life streaming data, while enabling analysts to exploit the semantic reasoning using knowledge graphs in real-time and have access to more reliable data.

The contributions presented in this paper are: (i) an algorithm to generate the RDF streams from heterogeneous streaming data; (ii) its implementation, the RMLStreamer-SISO, as an extension of RMLStreamer; and (iii) an evaluation demonstrating that the RMLStreamer-SISO outperform the state-of-the-art. The paper is structured as follows: Sect. 2 discusses related work, Sect. 3

¹ Implementation report of RML: <https://rml.io/implementation-report/>.

the approach and its implementation, Sect. 4 the evaluation of RMLStreamer-SISO against state-of-the-art, Sect. 5 the results of our evaluations, and Sect. 7 concludes our work with possible future works.

2 Related Works

Streaming RDF mapping engines transform heterogeneous data streams to RDF data streams. Several solutions exist in the literature for generating RDF from persistent data sources [2, 13, 14, 23], but only few generate RDF from data streams [17, 18, 21]. Although the implementations details are elaborated in these works, their evaluations are designed without considering the different data stream behaviours nor the resource contention between different evaluation components.

TripleWave [18] generates RDF streams from streaming or static data sources using R2RML mappings, and publishes them as RDF stream. However, the R2RML mappings of TripleWave are invalid according to the specifications of R2RML and it does not support joins. Although it is purported to support several input sources, the user has to write the code to process the input data and iterate over them before using the tool. This can result in poor performance from improper implementation. Last, it is not designed to support distributed parallel processing, resulting in limited scaling with data volume and velocity.

RDF-Gen [21] generates static or streaming RDF data from static or streaming data sources. A Data connector communicates with the data source, iterates over its data entries, and converts every entry to a record of values. These records are converted to RDF using a graph template: a listing of RDF-like statements with variables bound to the record values coming from data connectors. RDF-Gen generates RDF on a per record basis, theoretically allowing a distributed parallel processing set-up. However, the current implementation and documentation show no indication of a clustered setup nor how to run it.

SPARQL-Generate [17] extends SPARQL 1.1 syntax to support mapping of heterogeneous data to RDF data. SPARQL-Generate could be implemented on top of any SPARQL query engine, and knowledge engineers with SPARQL experience could use it with ease. The reference implementation of SPARQL-Generate² generates RDF streams from data streams, even though it is not reported in the original paper. Although joining data from multiple sources is supported, SPARQL-Generate waits for one of the data streams to end first before consuming other data sources to join the data. Thus, joins with unbounded streaming data sources are not supported. The implementation is based on single machine setup without scaling with data volume and velocity.

Cefriel's Chimera [22] is an integration framework based on Apache Camel³ split into four “blocks” of components to map heterogeneous data to RDF data: lifting block, data enricher, inference enricher, and lowering block.

² SPARQL-Generate: <https://github.com/sparql-generate/sparql-generate>.

³ Apache Camel: <https://camel.apache.org/>.

Chimera aims to be modular and allows each block to be replaced with custom implementations. The current implementation uses a modified version of RMLMapper⁴ in the lifting block for data stream processing. However, the whole RML mapping process is recreated with each incoming message which could lead to high performance overhead in a highly dynamic data stream environment.

3 Stream In - Stream Out (SISO)

We extend RMLStreamer’s architecture for generating RDF from persistent big data sources [13] to also generate RDF streams from heterogeneous data streams with high data velocity and volume, while keeping the latency low. The RDF mapping language (RML) [10], a superset of R2RML, expresses customized mapping from heterogeneous data sources to RDF datasets. We illustrate the concepts of RML with the example RML document in Listing 1.2.

We break the process of generating RDF from a data stream into tasks and subtasks (Fig. 1). Each task or subtask is a stream processing operator acting on an incoming data stream. They could be chained one after the other to form a pipeline of operators and result in one or more outgoing data streams. This approach introduces parallelism on both data and processing level, enabling each data stream and operator to be processed and executed respectively in parallel.

To illustrate RMLStreamer-SISO’s pipeline, we use the examples in Listing 1.1 and 1.2. The mapping document in Listing 1.2 is used to join and map JSON data (Listing 1.1) from websocket streams to RDF with dynamic window join.

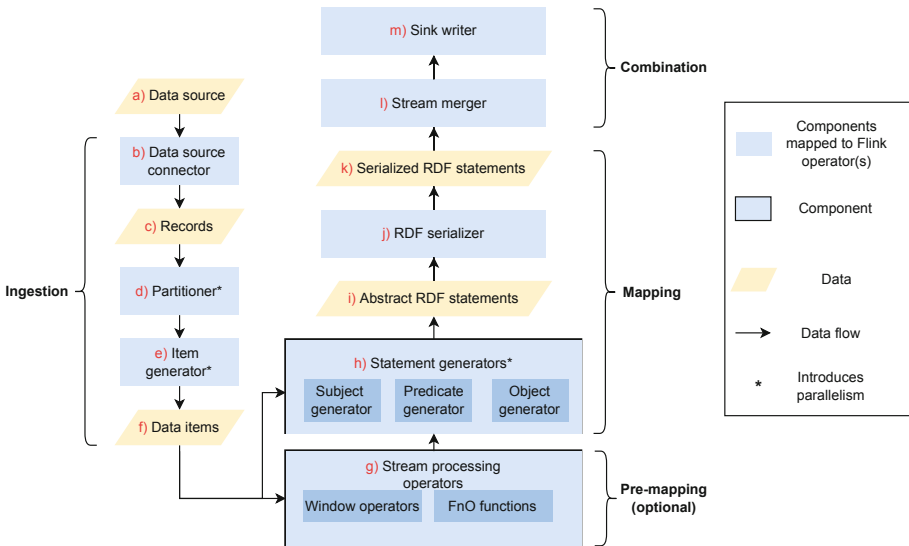


Fig. 1. Workflow of RMLStreamer. Data flows from the *Data Source* at the top through all the components pipeline to the *Sink writer* at the bottom.

⁴ RMLMapper: <https://github.com/RMLio/rmlmapper-java>.

Listing 1.1. Data records from 2 data streams “Flow” & “Speed”.

```

1 // data records from Speed stream
2 {"speed":123.0,"time":"14:42:00","id":"lane1"}
3 // data records from Flow stream
4 {"flow":1680,"time":"14:42:00","id":"lane1"}

```

Listing 1.2. Example RML Mapping file to generate streaming RDF from the streaming heterogeneous data of Listing 1.1.

```

1 # prefix definitions omitted
2 _:ws_source_ndwSpeed a td:Thing ;
3   td:hasPropertyAffordance [ td:hasForm [
4     hctl:hasTarget "ws://data-streamer:9001" ; # URL and content type
5     hctl:forContentType "application/json" ; # Data format
6     hctl:hasOperationType "readproperty" ] ] . # Read only
7 _:ws_source_ndwFlow a td:Thing;
8   td:hasPropertyAffordance [ td:hasForm [
9     hctl:hasTarget "ws://data-streamer:9000" ;
10    hctl:forContentType "application/json" ;
11    hctl:hasOperationType "readproperty" ] ] .
12 <JoinConfigMap> a rmls:JoinConfigMap ;
13   rmls:joinType rmls:TumblingJoin . # Trigger/eviction type
14 <NDWSpeedMap> a rr:TriplesMap ;
15   rml:logicalSource [ # Describes data source
16     rml:source _:ws_source_ndwSpeed ;
17     rml:referenceFormulation ql:JSONPath ; # JSONPath iterator
18     rml:iterator "$" ] ; # Iterates the data as JSON root object
19   rr:subjectMap [ # Generation of the subject IRI
20     rr:template "speed={speed}&time={time}" ] ;
21   rr:predicateObjectMap [ # Describes how predicate and object are generated
22     rr:predicate <http://example.com/laneFlow> ;
23     rr:objectMap [
24       rr:parentTriplesMap <NDWFlowMap> ; # TripleMap to be joined with
25       rmls:joinConfig <JoinConfigMap> ; # Configuration of join window
26       rmls>windowType rmls:TumblingWindow ; # Type of join window
27       rr:joinCondition [ # Attributes on which the data records are joined
28         rr:child "id" ; rr:parent "id" ; ] ] .
29 <NDWFlowMap> a rr:TriplesMap ;
30   rml:logicalSource [
31     rml:source _:ws_source_ndwFlow ;
32     rml:referenceFormulation ql:JSONPath ;
33     rml:iterator "$" ] ;
34   rr:subjectMap [ rr:template "flow={flow}&time={time}" ] .

```

3.1 RDF Stream Generation Workflow

Our approach consists of a workflow with four tasks (see Fig. 1):

Ingestion. The ingestion task captures data streams and prepares the data records for the mapping task. Each data stream triggers one ingestion task that can run in parallel with the other ingestion tasks spawned by the other data streams. The ingestion task can be divided in three subtasks:

1. *Data source connector* (Fig. 1,(b)): This subtask is responsible for connecting to a (streaming) data source (a). It reads data records from the source and passes these records (c) on to the stream partitioner.

2. *Stream partitioner (d)*: The stream of data records is optionally partitioned in disjoint partitions to be fed to the next subtask. The partitioning depends on the order's maintenance. If the exact order of the incoming data records is not important to be maintained, then these records can be distributed evenly among multiple instances of the next subtask, increasing parallelism. If the order of generating RDF statements needs to correspond with the order of the incoming data records, then the stream is not distributed at this stage.
3. *Item generator (e)*: One data record can lead to zero or more RDF statements. This subtask splits a data record in zero or more items of internal representation called *data items (f)*, according to the logical iterators defined in the mapping document, before the actual mapping task takes place. Using the sample data and the mapping document from Listing 1.1 and 1.2 respectively, this subtask will use the logical iterator '\$', a JSONPath⁵, to generate data items from each data record shown in Listing 1.1. In this case, the logical iterator is the JSON root object, so the data item is the same as the incoming data records. Otherwise, if the data record contains a list of sub-records, and the logical iterator is specified over the list (e.g., \$.list[*]), each of these sub-records are turned into *data items*.

Pre-mapping (Optional). Before the data items are mapped to RDF, the data items may be processed with custom data transformations defined with FnO [8], or the window operators, such as joins, aggregates, and reduce. The FnO functions could be as simple as changing letters to uppercase or as complex as the window joins. This stage is optional and omitted if the RML document does not define pre-mapping functions. The pre-mapping task (g) is right before the mapping task since the data fields requiring preprocessing can be more than the data fields needed for mapping to RDF data. For example, with the given inputs and mapping document (Listing 1.2), the data items (Listing 1.1) from the two input streams, "Flow" and "Speed", are first buffered inside a window, and then joined based on their `internalId` value. Data records, having the same value for the "id", are joined pairwise. If windows joins were implemented after the mapping stage, the verbosity of RDF would substantially increase the network bandwidth. More, to fully map the data before joining, RMLStreamer-SISO needs to know all attributes present in the raw data records which would be infeasible.

To support joining with windows, RML was extended. New vocabulary terms were defined to support windowing operations with RML. We defined two new properties: `rmls:windowType` to provide the type of window to be used when joining and `rmls:joinConfig` when joining the *Child* and *Parent Triple Map* to define how the trigger, and eviction are fired inside the window.

Section 3.2 details the dynamic windowing algorithm and Sect. 3.3 elaborates on the design choice and windows' implementation for RMLStreamer-SISO.

Mapping. RDF statements are generated from data items coming from the ingestion task and the pre-mapping task.

⁵ JSONPath documentation: <https://goessner.net/articles/JsonPath/index.html>.

1. *Statement generator (h)*: Each data item leads to one or more RDF statements in this sub task. Each statement is generated in parallel as an abstract RDF statement (i) which could be fed to the next subtask for serialization.
2. *RDF serializer (j)*: The abstract RDF statements are serialized into various RDF serialisations based on the configuration given to the RMLStreamer.

Combination. This task brings back together all streams of RDF statements (l) into one final RDF stream which will be written out using the sink writer (m).

3.2 Heterogeneous Data Streams Join in RDF Streams

Supporting *joins* in RMLStreamer-SISO and any streaming RDF generator, is not trivial as windowing techniques are required for unbounded and unsynchronized streaming data. Unlike batch processing where data is bounded, processing whole data streams in memory is unsustainable due to the continuous and infinite characteristics of streaming data. Therefore, stream processing engines use buffers called *windows* to hold the most recent stream of records in memory. The windows’ lifetime is measured in terms of time interval, thus, the *window interval* determines the size of the window and their operation behaviour is defined by the trigger, and the eviction events [12]. A *trigger event* occurs when an operator is executed to process the data records inside the window interval. An *eviction event* occurs when the window evicts the data records inside its buffer.

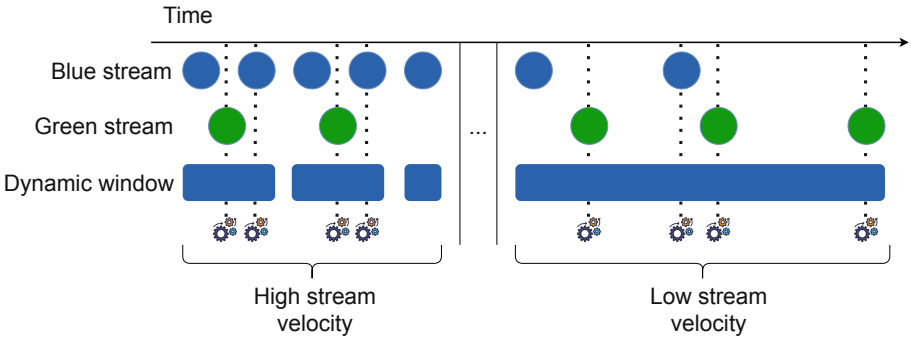


Fig. 2. Behaviour of the dynamic window under high, and low stream velocities. The cogwheels are the *trigger* events representing the moment when the data records are processed. In this figure, the *trigger* events are fired with every new data record, and only when there is at least one data record from each data stream.

We opted for an eager trigger implementation to lower the latency of RMLStreamer’s responses for the windowed joins’ implementation. The joined results are emitted as soon as possible without waiting for the eviction event to occur. We designed a dynamic window which adapts its window intervals according to the velocity of the incoming data streams. Adaptive windowing [27] was studied

in the context of batch stream processing with a positive impact on the stream processing job's performance: lower latency, and higher throughput. We opted for a simple cost metric based on the data records' number to keep the memory and latency low in a real-time stream processing environment where the time constraint is more stringent.

The algorithm is inspired by the additive-increase, and multiplicative decrease algorithm of TCP congestion control [7]. Figure 2 shows the high level behaviour of our dynamic window for the two different stream velocities. When the data stream velocity is high, the window size shrinks to process the data records as fast as possible, keeping the latency low and throughput high. When the data stream velocity is low, the size of the window grows to wait for more data records and process them. This ensures that the window do not miss the records due to short window size. We elaborate the details of the algorithm below. For each window, the following configuration parameters are provided:

1. $|W|$: The window interval
2. ϵ_u & ϵ_l : Upper and lower threshold limit for total cost metric
3. U & L : Upper and lower limit for the window interval
4. $Limit(List_P)$ & $Limit(List_C)$: Upper limit size for parent and child stream

Algorithm 1: Dynamic window *onEviction* routine

Data: $|W|, \epsilon_u, \epsilon_l, U, L, Limit(List_P), Limit(List_C), S_P, S_C$

```

1  $cost(List_P) = |S_P| / Limit(List_P)$ 
2  $cost(List_C) = |S_C| / Limit(List_C)$ 
3 total cost  $m = cost(List_P) + cost(List_C)$ 
  // adapts window size based on cost
4 if  $m > \epsilon_u$  then
5    $|W| = |W| / 2$ 
6    $Limit(List_P) = Limit(List_P) * cost(List_P) * 1.5$ 
7    $Limit(List_C) = Limit(List_C) * cost(List_C) * 1.5$ 
8 else if  $m < \epsilon_l$  then
9    $|W| = |W| * 1.1$ 
10   $Limit(List_P) = Limit(List_P) * cost(List_P) * 1.5$ 
11   $Limit(List_C) = Limit(List_C) * cost(List_C) * 1.5$ 
12 clean both  $List_C$  and  $List_P$ 
13 clip  $|W|$  in the range of  $[L, U]$ 
```

Since we implement the join operator with eager execution, the trigger event is fired when the current record r_c arrives inside the window. We denote the current window as W with interval size $|W|$. The streams are denoted as S_p and S_c with the corresponding states $List_p$ and $List_c$, for the parent and child stream respectively (the parent and child stream follows the RML specification for joining triples maps). The states contain the records from their respective

streams inside the window with for example $|S_p|$ denoting the number of records from S_p . $List_p$ and $List_c$ are only used in cost calculation to determine if the window interval needs to be changed; they do not limit the amount of records that could be buffered inside the window.

At each eviction trigger, we calculate the cost for each list states $List_p$ and $List_c$. For example, the cost for $cost(List_p) = |S_p|/Limit(List_p)$. The total cost is $m = cost(List_p) + cost(List_c)$ and it is checked against the thresholds ϵ_l and ϵ_u to adjust the window interval accordingly. We assume the stable zone to be achieved if the total cost fulfils the predicate: $\epsilon_l \leq m \leq \epsilon_u$. Algorithm 1 shows the pseudo-code for the eviction algorithm we just elaborated.

3.3 Implementation

RMLStreamer-SISO is released as version 2.3 of RMLStreamer to utilize Flink’s parallelism for horizontal scaling (via distributed processing in a network and vertical scaling (via multi-threaded execution of tasks). The update brings the windowing support for joining multiple data streams, the dynamic windowing algorithm, and FnO [8] as an extension point for joins execution. The code and usage instructions for RMLStreamer-SISO are available online at the Github repository: <https://github.com/RMLio/RMLStreamer>.

Windowing support is implemented through the use of Flink’s windowing API⁶ for common types of window, e.g., Tumbling Window. We implemented the *KeyedCoProcessFunction* provided by Flink’s low-level stream processing API to manage the different states required for the algorithm (Algorithm 1) of the dynamic window. We implemented the dynamic windowed join before the mapping stage, to group input streams and reduce network bandwidth usage. The generated RDF stream could be windowed by the RDF stream processing engines consuming the output.

Currently, FnO functions jar files have to be compiled together as part of the RMLStreamer-SISO jar. Examples on github⁷ show the working of RMLStreamer-SISO with TCP data stream. We also provide an extensive documentation on RMLStreamer-SISO in a containerized environment with docker⁸.

4 Evaluation

An extensive evaluation was conducted focused on variable data stream velocity, volume and variety of data formats to emulate the real-life workloads as close as possible. The code for the evaluation is available on github⁹. Since RMLStreamer-SISO is situated between traditional stream processing and RSP, state-of-the-art approaches for benchmarking in these domains are combined:

⁶ Window: <https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/dev/datastream/operators/windows/>.

⁷ RMLStreamer-SISO: <https://github.com/RMLio/RMLStreamer>.

⁸ Docker: <https://docker.com>.

⁹ Benchmark: <https://github.com/s-minoo/rmlstreamer-benchmark-rust>.

architectural design of RSPLab [25], workload design of Open Stream Processing Benchmark [11], and measurement strategies of Karimov et al. [15].

We compare the RMLStreamer-SISO with the state-of-the-art streaming RDF generator, SPARQL-Generate, which is actively maintained, used and supports the same features as RMLStreamer-SISO. The other tools were not considered for different reasons: TripleWave requires a custom implementation to process each data stream and feed it in TripleWave which means it cannot be used as-it-is. More, TripleWave is meant purely for feeding RDF streams to RDF stream processing engines without performing joins, therefore it would have been an unfair comparison both in terms of features and scope. RDF-Gen’s source code is not available, but only a jar is available without any instructions to run it. Both TripleWave and RDF-Gen are also not actively maintained. Finally, Cefriel’s Chimera restarts the RDF mapping engine with every data record, which means that the processing of the input and mapping is not performed in a true streaming manner; the comparison would not be meaningful.

Data Source. The input data used in the evaluation comes from time annotated traffic sensor data from the Netherlands provided by NDW (Nationale Databank Wegverkeersgegevens)¹⁰, and also used by Van Dongen et al. [11]. It contains around 68,000 rows of CSV data with two different measurements across different lanes on a highway: number of cars (flow), and their average speed. The two measurements are streamed through a websocket data streaming server.

Metrics. Stream processing frameworks are typically evaluated using two main metrics: latency and throughput [15]. Latency can be further distinguished into two types: processing-time latency, and event-time latency. *Processing time latency* is the interval between the data record’s arrival time at the input and the emission time at the output of the streaming engine [15]. *Event-time latency* is the interval between the creation time, and the emission time at the streaming engine’s output, of the data record [15]. Latency measurement requires to consider the effect of coordinated occlusion, where the queueing time, a part of the event-time, is ignored [15]. Therefore, we consider event-time latency as our latency measurement to take the effect of coordinated occlusion in consideration.

For our evaluation, we considered the event-time latency of each record, the throughput as number of consumed records per second, the memory and CPU usage of the engine’s docker container. The measurements are captured on a machine separate from the host machine of the System Under Test (SUT), where memory and CPU usage are measured using cAdvisor¹¹. By treating the SUTs as a blackbox, we ensure that the measurement of the metrics incurs no performance penalty nor resource contention with the SUTs during the evaluation.

Evaluation Set Up. The architectural design is a modification of RSPLab with a custom data streaming component (Fig. 3). It consists of three components: a) the data streamer, b) the system under test, and c) the monitoring system.

¹⁰ NDW: <https://www.ndw.nu/>.

¹¹ cAdvisor: <https://github.com/google/cadvisor>.

With the proposed architecture where each component is isolated, we aim to reduce the influence of the benchmark components on the engine during the evaluation process. The modularity of the setup also increases the flexibility of configuring the evaluation environment with minimal changes for the engines.

Workload Design. To evaluate the performance of the engines under different data characteristics and processing scenarios, we devise three different workloads: (i) throughput measurement, (ii) periodic burst, and (iii) scalability measurement. As SPARQL-Generate is unable to join unbounded streaming data (it expects data streams with an end, Sect. 2), we evaluated the two workloads (throughput measurement and periodic burst) without joining functionality to compare.

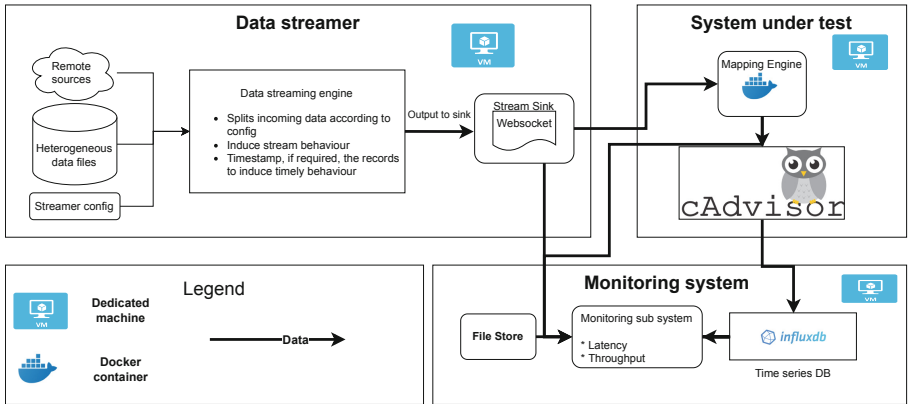


Fig. 3. Benchmark architecture to evaluate the different engines, inspired by RSPLab.

- throughput measurement: the data stream throughput is constant and steadily increases with each run to determine the engine’s *sustainable throughput* [15]. CPU, latency and memory usage are measured.
- periodic burst: a burst of data records is emitted periodically to mimic fluctuations in data streams; CPU, memory, latency and throughput are measured.
- scalability measurement: RMLStreamer-SISO is evaluated in two modes: centralised mode without parallelism and distributed mode with parallelizable data to measure the impact of parallelism on its scalability. In both modes, data from two input streams are joined and latency is measured.

System Specifications. We ran the evaluation on a single machine with multiple docker containers to emulate the communication between the data streaming source and the mapping engine in a streaming network environment. The machine has Intel i7 CPU with 8 cores at 4.8GHz, 16 GB RAM, and 200 GB hard disk space. The data streamer and the monitoring system docker containers (Fig. 3) have access to 4 of the cores, and the SUT docker container has access to the leftover 4 cores. This prevents CPU resource contention between the SUT and the other components used for running the evaluation.

To evaluate horizontal scaling, the data streamer component is replaced with Apache Kafka to support parallel ingestion of data streams by RMLStreamer-SISO. Apache Kafka is configured with default settings and the data (Sect. 4) is streamed into two topics¹²; “ndwFlow”, and “ndwSpeed” containing the records about the number, and the average speed of the cars respectively.

5 Results

In this section, we discuss the results of our evaluation using different workloads.

Workload for Throughput Measurement. For the throughput measurement workload, we ran the evaluation multiple times with increasing input data throughput for each run to evaluate the sustainable throughput of the SUTs.

In the first few runs of the evaluation, the RMLStreamer fared a bit worse than SPARQL-Generate in all three measurements. This is due to the overhead of having a distributed task manager for executing, and managing the different tasks and subtasks of mapping heterogeneous data (Fig. 1). However, when the throughput starts increasing beyond 10,000 records per second, RMLStreamer-SISO outperforms SPARQL-Generate in terms of latency and memory usage.

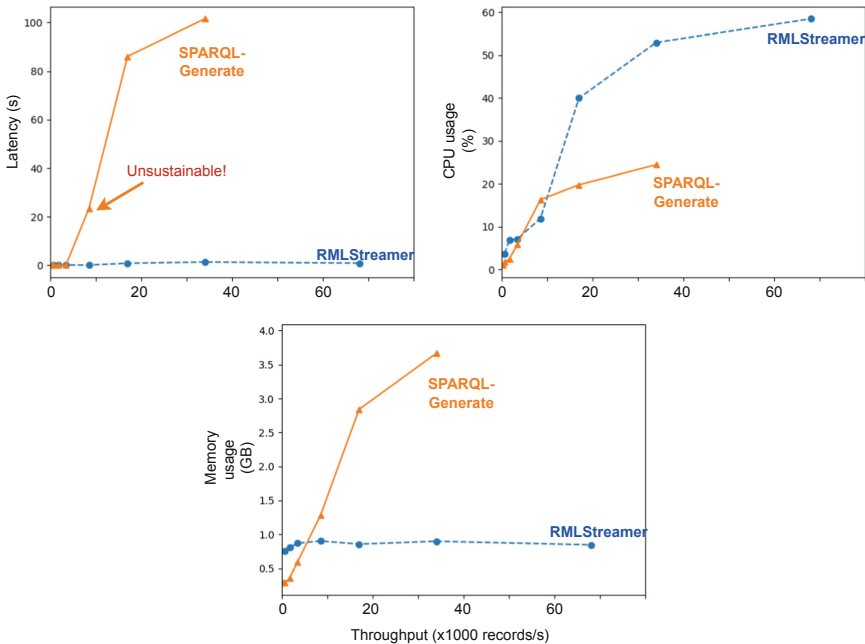


Fig. 4. SUTs performance under different data stream velocity for sustainable throughput measurement. The last run for SPARQL-Generate was omitted because it took more than 1 h instead of the expected 30 min to process the whole data stream.

¹² Kafka topics: <https://developer.confluent.io/learn-kafka/apache-kafka/topics/>.

Compared to RMLStreamer-SISO, SPARQL-Generate became unsustainable when the throughput of the input data streams passes 10,000 records per second with 20 s latency (Fig. 4). To the contrary, RMLStreamer-SISO has a consistent low latency of 1 s for all runs of the workload having 100x magnitude lower latency than SPARQL-Generate in later runs.

Regarding CPU usage, RMLStreamer-SISO has on average 20% more CPU usage for the overhead of Apache Flink managing the distributed tasks.

In terms of memory usage, RMLStreamer-SISO uses significantly lower memory than SPARQL-Generate at around 900 MB compared to 3 GB by SPARQL-Generate. Based on the previous observations, we conclude that RMLStreamer-SISO outperforms SPARQL-Generate at higher throughput with lower latency and memory usage. Even though, RMLStreamer-SISO's CPU usage is around 30% higher than SPARQL-Generate in the last run, it effectively copes with the increase in data stream velocity to maintain low latency processing.

Workload for Periodic Burst. The periodic burst workload studies the adaptability of the engine to the recurring sudden burst of data stream. We used the measurements from the last minute of the evaluation, when the engines are *stable* without warm-up overheads, to better visualize their performance during the periodic burst of data (Fig. 5). In Fig. 5, we see a periodic increase, and drop in the throughput metrics measurements, which is an expected behaviour in the engines when consuming a data stream input with periodic burst of data. Every 10 s we see a burst of around 35,000 messages. Both engines behave as expected for the throughput metrics measurement.

The spikes for latency measurement of SPARQL-Generate (Fig. 5) have a wider base than those of RMLStreamer-SISO. This indicates that SPARQL-Generate takes a longer time to recover from processing periodic workload than RMLStreamer-SISO by a few seconds. RMLStreamer-SISO's peak latency is around 500 ms whereas SPARQL-Generate has a peak latency of around 3.5 s. Although RMLStreamer-SISO uses more CPU than SPARQL-Generate to process data burst, it adapts to the sudden burst of data and recover more quickly than SPARQL-Generate. The latency of RMLStreamer-SISO is also 7 times lower than SPARQL-Generate due to the record-based processing capabilities. We conclude that RMLStreamer-SISO is better adapted to workloads with periodic burst of data with faster recovery period, lower latency and memory usage while maintaining the same throughput capabilities as SPARQL-Generate.

Workload for Scalability Measurement. We evaluated the RMLStreamer-SISO's capability to join two data streams with a constant throughput of around 17000 messages per second. CPU and memory usage of both modes of RMLStreamer-SISO are similar throughout the evaluation. However, despite the similar performance in terms of CPU and memory usage, *parallelized* mode fared significantly better in terms of the latency metric than *unparallelized* mode. Unparallelized mode has a median latency of around 50000 ms whereas *parallelized* mode has a median latency of around 57ms. This is around 1000x lower in terms of the median latency. Moreover, the minimum latency of *parallelized*

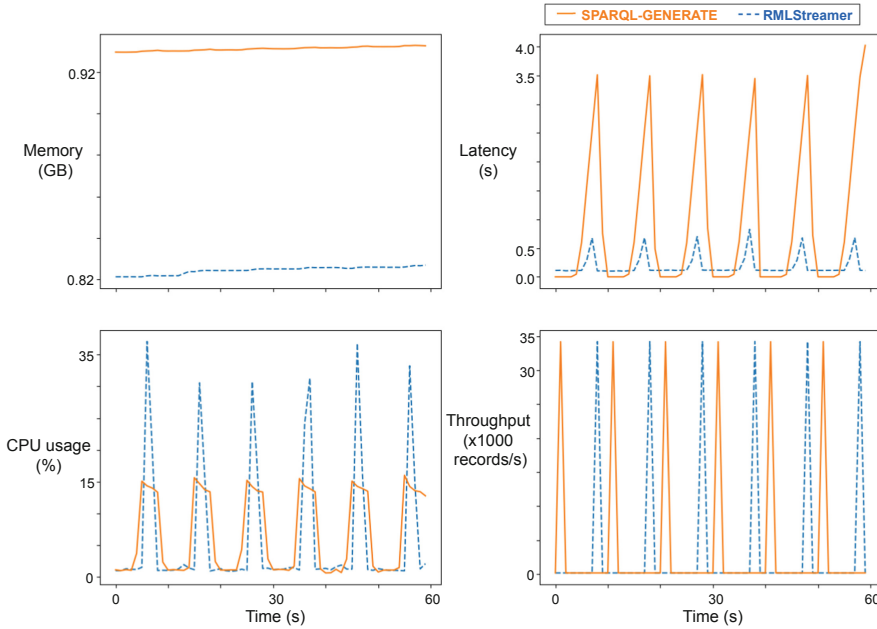


Fig. 5. Performance of SUTs in the last one minute of the periodic burst workload evaluation. A part of the *throughput* graph is blurred to give more clarity to the relationship between the trends in *latency* and *throughput* of the engines.

RMLStreamer-SISO at 8 ms is 10,000x lower than the minimum latency of *unparallelized* RMLStreamer-SISO at 13653 ms. The latency is kept low with high parallelization due to the effective distribution of the workload amongst the different parallelized tasks by the underlying DSP engine (Apache Flink). We conclude that RMLStreamer scales extremely well with significantly better performance in terms of latency if configured to be executed in a distributed mode.

6 Use Cases

RMLStreamer-SISO has seen uptake in multiple projects – covering different use cases in different architectures – to process streaming data and generate RDF streams. Largest validation was in research and development (R&D) projects between imec and Flemish companies such as DyVerSIFY on streaming data analysis and visualisation [20, 24], together with Televic Rail on IoT data, DAIQUIRI¹³ together with VRT on sport sensor data, and ESSENCE and H2020 project MOS2S¹⁴ on media data. Other projects include DiSSeCt¹⁵ on health data and transport data [4]. The variety of use cases shows that the resource is

¹³ <https://www.imec-int.com/en/what-we-offer/research-portfolio/daiquiri>.

¹⁴ <https://innovatie.vrt.be/project/essence>, <https://itea4.org/project/mos2s.html>.

¹⁵ <https://smit.vub.ac.be/project/dissect>.

suitable for solving the task at hand and also applicable to a multitude of use cases for society in general. Applications – within the knowledge graph construction problem domain – are varied, i.e., processing a large amount of low-frequency sensor data, a small amount of high-frequency sensor data, and large data sets combined with streaming data, processing Kafka streams, MQTT, Socket.io, and TCP streams. Beyond Belgium, RMLStreamer has received attention by the Institute of Data Science, proposed as part of RDF graph generation tutorials such as those by STIInnsbruck in Austria, and services such as Data2Services¹⁶ by the Institute of Data Science in Maastricht in the Netherlands.

7 Conclusion and Future Work

In this paper, we present RMLStreamer-SISO, a highly scalable solution to seamlessly generate RDF streams thanks to its dynamic window algorithm which adapts its window size to handle the dynamic characteristics of the data stream. This way, RMLStreamer-SISO enables low latency and high throughput mapping of heterogeneous data to RDF data. We showed that our solution scales better than the state-of-the-art in terms of latency, memory, and throughput. It is the only RDF stream generator which joins unbounded data streams and scale horizontally and vertically, enabling RDF streams generation from heterogeneous data streams which was not possible so far. Given it is open source and already widely used in different use cases involving not only academia but also industry, as shown in our use cases, it is expected that the community that grew around it will further grow and contribute at its maintenance, while its extensive documentation and tutorials allow for easy reuse¹⁷. The RML extensions will be further discussed with the W3C community group on knowledge graph construction and eventually will be incorporated to the revised RML specification.

RMLStreamer-SISO increases the availability of RDF streams following the high availability of data streams. Using a low-latency tool like RMLStreamer-SISO, legacy streaming systems could exploit the unique characteristics of real-life streaming data, while enabling analysts to exploit the semantic reasoning using knowledge graphs in real-time. This way, we enabled access to more data which should impact the further improvements of RSP engines and other semantic web technologies on top of RDF streams which were not possible so far.

Resource Availability Statement: Source code for RMLStreamer-SISO is available at <https://github.com/RMLio/RMLStreamer>. The source code for the benchmark is available at <https://github.com/s-minoo/rmlstreamer-benchmark-rust>. The dataset used for the benchmark is available at <https://github.com/Klarrio/open-stream-processing-benchmark/tree/master/data-stream-generator>.

¹⁶ <https://maastrichtu-ids.github.io/best-practices/blog/2021/03/18/build-a-kg/>, <https://stiinnsbruck.github.io/lkgt/>, <https://d2s.semanticscience.org/docs/d2s-rml/>.

¹⁷ Example of tutorial for use with docker technology, <https://github.com/RMLio/RMLStreamer/tree/development/docker>.

References

1. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: SPARQL for continuous querying. In: Proceedings of the 18th International Conference on World Wide Web. WWW 2009, pp. 1061–1062. Association for Computing Machinery, New York (2009). <https://doi.org/10.1145/1526709.1526856>
2. Belcao, M., Falzone, E., Bionda, E., Valle, E.D.: Chimera: a bridge between big data analytics and semantic technologies. In: Hotho, A., et al. (eds.) ISWC 2021. LNCS, vol. 12922, pp. 463–479. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88361-4_27
3. Botan, I., Derakhshan, R., Dindar, N., Haas, L., Miller, R.J., Tatbul, N.: Secret: a model for analysis of the execution semantics of stream processing systems. Proc. VLDB Endow. **3**(1–2), 232–243 (2010). <https://doi.org/10.14778/1920841.1920874>
4. Brouwer, M.D., et al.: Distributed continuous home care provisioning through personalized monitoring & treatment planning. In: Companion Proceedings of the Web Conference 2020. ACM, April 2020. <https://doi.org/10.1145/3366424.3383528>
5. Calbimonte, J.-P., Corcho, O., Gray, A.J.G.: Enabling ontology-based access to streaming data sources. In: Patel-Schneider, P.F., et al. (eds.) ISWC 2010. LNCS, vol. 6496, pp. 96–111. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17746-0_7
6. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flinkTM: stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**, 28–38 (2015)
7. Chiu, D.M., Jain, R.: Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. Comput. Netw. ISDN Syst. **17**(1), 1–14 (1989)
8. De Meester, B., Dimou, A., Verborgh, R., Mannens, E.: An ontology to semantically declare and describe functions. In: Sack, H., Rizzo, G., Steinmetz, N., Mladenicić, D., Auer, S., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9989, pp. 46–49. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47602-5_10
9. Dias de Assunção, M., da Silva Veith, A., Buyya, R.: Distributed data stream processing and edge computing: a survey on resource elasticity and future directions. J. Netw. Comput. Appl. **103**, 1–17 (2018). <https://doi.org/10.1016/j.jnca.2017.12.001>, <https://www.sciencedirect.com/science/article/pii/S1084804517303971>
10. Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: RML: a generic language for integrated RDF mappings of heterogeneous data, vol. 1184 (2014)
11. van Dongen, G., Van den Poel, D.: Evaluation of stream processing frameworks. IEEE Trans. Parallel Distrib. Syst. **31**(8), 1845–1858 (2020). <https://doi.org/10.1109/TPDS.2020.2978480>
12. Gedik, B.: Generic windowing support for extensible stream processing systems. Softw. Pract. Exper. **44**(9), 1105–1128 (2014). <https://doi.org/10.1002/spe.2194>
13. Haesendonck, G., Maroy, W., Heyvaert, P., Verborgh, R., Dimou, A.: Parallel RDF generation from heterogeneous big data. In: Proceedings of the International Workshop on Semantic Big Data. SBD 2019. Association for Computing Machinery, New York (2019). <https://doi.org/10.1145/3323878.3325802>
14. Iglesias, E., Jozashoori, S., Chaves-Fraga, D., Collarana, D., Vidal, M.E.: SDM-RDFIZER. In: Proceedings of the 29th ACM International Conference on Information and Knowledge Management, October 2020. <https://doi.org/10.1145/3340531.3412881>

15. Karimov, J., Rabl, T., Katsifodimos, A., Samarev, R., Heiskanen, H., Markl, V.: Benchmarking distributed stream data processing systems. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE), April 2018. <https://doi.org/10.1109/icde.2018.00169>
16. Le Phuoc, D., Dao-Tran, M., Le Tuan, A., Duc, M.N., Hauswirth, M.: RDF stream processing with CQELS framework for real-time analysis. In: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems. DEBS 2015, pp. 285–292. Association for Computing Machinery, New York (2015). <https://doi.org/10.1145/2675743.2772586>
17. Lefrançois, M., Zimmermann, A., Bakerally, N.: A SPARQL extension for generating RDF from heterogeneous formats. In: Blomqvist, E., Maynard, D., Gangemi, A., Hoekstra, R., Hitzler, P., Hartig, O. (eds.) ESWC 2017. LNCS, vol. 10249, pp. 35–50. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58068-5_3
18. Mauri, A., et al.: TripleWave: spreading RDF streams on the web. In: Groth, P., et al. (eds.) ISWC 2016. LNCS, vol. 9982, pp. 140–149. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46547-0_15
19. N.A: Apache storm. <https://storm.apache.org/>
20. Paepe, D.D., et al.: A complete software stack for IoT time-series analysis that combines semantics and machine learning—lessons learned from the diversify project. *Appl. Sci.* **11**(24), 11932 (2021). <https://doi.org/10.3390/app112411932>
21. Santipantakis, G.M., Kotis, K.I., Vouros, G.A., Doukeridis, C.: RDF-GEN: generating RDF from streaming and archival data. In: Proceedings of the 8th International Conference on Web Intelligence, Mining and Semantics. WIMS 2018. Association for Computing Machinery, New York (2018). <https://doi.org/10.1145/3227609.3227658>
22. Scrocca, M., Comerio, M., Carenini, A., Celino, I.: Turning transport data to comply with EU standards while enabling a multimodal transport knowledge graph. *Semant. Web - ISWC* **2020**, 411–429 (2020). https://doi.org/10.1007/978-3-030-62466-8_26
23. Simsek, U., Kärle, E., Fensel, D.A.: RocketRML - a NodeJS implementation of a use case specific RML mapper. arXiv abs/1903.04969 (2019). <https://doi.org/10.48550/ARXIV.1903.04969>
24. Steenwinkel, B., et al.: FLAGS: a methodology for adaptive anomaly detection and root cause analysis on sensor data streams by fusing expert knowledge with machine learning. *Futur. Gener. Comput. Syst.* **116**, 30–48 (2021). <https://doi.org/10.1016/j.future.2020.10.015>
25. Tommasini, R., Della Valle, E., Mauri, A., Brambilla, M.: RSPLab: RDF stream processing benchmarking made easy. In: d’Amato, C., et al. (eds.) ISWC 2017. LNCS, vol. 10588, pp. 202–209. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68204-4_21
26. Zaharia, M., et al.: Apache spark: a unified engine for big data processing. *Commun. ACM* **59**(11), 56–65 (2016). <https://doi.org/10.1145/2934664>
27. Zhang, Q., Song, Y., Routray, R.R., Shi, W.: Adaptive block and batch sizing for batched stream processing system. In: 2016 IEEE International Conference on Autonomic Computing (ICAC), pp. 35–44 (2016). <https://doi.org/10.1109/ICAC.2016.27>