Dynamic Windows Processing in RDF Mapping engines for data streams

Sitt Min Oo

Supervisors: Prof. Dr. Ruben Verborgh and Dr. Anastasia Dimou
Counsellor: Gerald Haesendonck

Abstract— **Current state-of-the-art approaches mapping non-RDF to RDF data in a streaming environment focus more on the efficiency of the mapping process with minimal support for multi-stream processing. The existing approaches for supporting simple multi-stream processing operators in mapping engines are very limited or apply fixed window size.**

Therefore, we implemented a dynamic window mechanism in RMLStreamer, which adapts its size according to the changing stream characteristics with negligible memory overhead, low latency, and high throughput. We evaluated the dynamic window under different workload with varying stream velocity. The results show that it achieves latency in the millisecond range, with higher throughput than fixed size windows in all workload situations.

Index Terms— **RDF, RMLStreamer, RML, Adaptive windows, Dynamic windows, Stream joins, Multi-stream processing.**

I. INTRODUCTION

Various data formats such as CSV or HTML are not designed to contain the semantics of their data which avoids machines to automatically interpret these data. To ensure that these heterogeneous data format are interpretable and processable by machines, data formats based on W3C standards, such as Resource Description Framework (RDF) triples [**intro_rdf**], are developed. There exist state-of-the-art approaches to consolidate heterogeneous data and transform them to an RDF serialization in a streaming environment.

These approaches support traditional stream operators like joins and aggregations. However, they do not consider the characteristics of streaming data sources such as velocity and time-correlations between the different input streams, either due to the fixed size windows or custom solutions which they apply.

Therefore, we propose a dynamic window as a solution to varying characteristics of streaming data. We evaluated our approach with the join operator applied inside the window. The dynamic window improves the performance of the join operator with higher throughput, lower latency and similar memory usage compared to a fixed size window.

The source code and the evaluation setup code can be found at `https://github.com/RMLio/RMLStreamer/tree/feature/window_joins`, and

https://github.com/Kortika/Thesis-test-scripts.git respectively.

II. RELATED WORKS

There exist state-of-the-arts approaches to map non-RDF data to RDF data in a streaming environment. These approaches implement diferent strategies when it comes to applying multi-stream processing. Furthermore, we will also outline the existing works on windowing to handle changing characteristics of the input data stream over time.

A. Mapping implementations

1) SPARQL-Generate: SPARQL-Generate [**sparql_generate**] is based on an extension of the SPARQL 1.1 query language, to leverage its expressiveness and extensibility. It can be implemented on top of any existing SPARQL query engine. M. Lefrançois [**sparql_generate**] clarified that when joining records from two different streams, SPARQL-Generate will fully consume the records from the "parent" stream and index it internally in memory first, before iteratively consuming the "child" stream for joining.

2) TripleWave: TripleWave [**triple_wave**] is based on an extension of R2RML to consume heterogeneous data publish it as RDF data on the Web. Therefore, it only focuses on the mapping of a non-RDF data stream to an RDF data stream. Although it supports a simple *join* operator, it does not have a dynamic window to handle changing characteristics of a data stream.

3) RDFGen: RDF-Gen [**rdf_gen**] is based on its own custom syntax which is a combination SPARQL and Turtle syntax. It processes the input streams individually and therefore employs windowing to perform multi stream operators. However, since the implementation is closed source, we could not confirm if the window is fixed size or dynamic.

4) RMLStreamer: RMLStreamer [**rml_streamer**] was developed to parallelize the ingestion and mapping process of RDF data generation pipeline. It is based on the work of RMLMapper [**rml**], an RDF mapping engine consuming bounded data and mapping them to RDF data with the use of RML. Hence, RMLStreamer can also process heterogeneous data and generate RDF data. It does not support multi stream operators at this moment, however, the ease of scalability provided by RMLStreamer is desirable as a stream mapping engine.

B. Windows

Several studies have been conducted to improve join algorithms in windows [**vctw_join**, **join_tracking**, **grubjoin**, **approximate_window_sem**, **approx_window**]. The approaches in [**grubjoin**, **approximate_window_sem**, **approx_window**] are based on dropping some of the records to be joined through *load shedding* if the records fail to meet a threshold. These threshold calculations require some memory overhead for storing a statistical model per window or assume the stream to be in-order for them to work [**grubjoin**, **approximate_window_sem**, **approx_window**].

On the other hand, VC-TWJoin [**vctw_join**] considers only the velocity of the stream to adjust the window sizes. However, the approach is unstable when the data stream has periodic bursts of data. It either increases or decreases the size of the window when the metric triggers the thresholds of the algorithm. In the worst-case scenario it can lead to cyclic increase and decrease of window sizes if the metric borders around the threshold with every update of the metric.

III. DYNAMIC WINDOW

A dynamic window is a type of partitioned window [**generic_window_sem**]. We define it as a group of subwindows, which update their sizes dynamically according to the characteristics of the data stream. It groups the incoming streams into different partitions first, according to the *key* attribute value of the records. Subsequently, these grouped records are assigned to individual subwindows. These subwindows adjust their own size independently from each other at each update cycle. Furthermore, the subwindows will be independent from each other such that the stream rate is local to the attribute value for which each subwindow is responsible.

A. Dynamic window join

The join algorithm used, is a variant of Symmetric Hash Join [**symmetric_hash_join**]. The subwindows themselves work like a hash table, containing records with a specific *key* — since they are *partitioned*. The records are joined by probing the relevant stream and generating the joined result.

B. Dynamic window algorithm

For each subwindow, the following configuration parameters are provided:

- Δn: the **initial** interval of time before the next eviction trigger.
- ϵ_u: the upper limit for threshold ϵ. [GH: what kind of threshold? Specify somewhere that the ϵs define thresholds for cost.]
- ϵ_l: the lower limit for threshold ϵ.
- U: the upper limit for the window size.
- L: the lower limit for the window size.

Since we are implementing the join operator, the trigger event is fired when current record r_c arrives, and the symmetric hash join is executed. We denote the current window as W with the size as $|W|$. The streams are denoted as S_P and S_C with the corresponding states $List_P$ and $List_C$, for the parent and the child stream respectively. The states contain the records from their respective streams inside the subwindows.

The eviction trigger is fired every time when the current watermark $w \geq |W| + \Delta n$. At each eviction trigger we calculate the cost for each *list states* $List_P$ and $List_C$, containing the records from S_P and S_C respectively. The cost for $cost(List_P) = |S_P|/Size(List_P)$, idem for $cost(List_C)$. The total cost is $m = cost(List_P) + cost(List_C)$ and it is checked against thresholds ϵ_l and ϵ_u. If $\epsilon_l \leq m \leq \epsilon_u$, Δn stays the same, otherwise it will be adjusted accordingly. This provides some stability in window size by keeping the same size if the cost lies within the thresholds. The higher m, the higher the stream rate. Thus, the subwindow needs to be frequently evicted and lower Δn to reduce memory usage. There is also a limit on the minimum and maximum window sizes, L and U respectively, to ensure that the window size Δn does not keep growing or shrinking in size infinitely in a worst-case scenario. The sizes of the list states are also updated according to $Size(List_P) = Size(List_P) * cost(List_P) + 0.5$. Similarly for $Size(List_C)$. Hence the dynamic window maintains an ideal size by adjusting $\Delta n, Size(List_P)$ and $Size(List_C)$ according to the stream rate. The pseudo code for the eviction algorithm is presented in Algorithm 1.

This algorithm for the dynamic window is an adaptation of the VC-TWindow [**vctw_join**] with improvements for stability in window sizing, clarity in the metrics used for updates, and localization of window size update to each subwindow.

Algorithm 1: Dynamic window *onEviction* routine

```
Data: $\Delta n, \epsilon_u, \epsilon_l, U, L, List_P, List_C, S_P, S_C$
1: $cost(List_P) = |S_P|/Size(List_P)$
2: $cost(List_C) = |S_C|/Size(List_C)$
3: total cost $m = cost(List_P) + cost(List_C)$
4: if $m > \epsilon_u$ then
5:     $\Delta n = \Delta n/2$
6:     $Size(List_P) = Size(List_P) * (cost(List_P) + 0.5)$
7:     $Size(List_C) = Size(List_C) * (cost(List_C) + 0.5)$
8: else if $m < \epsilon_l$ then
9:     $\Delta n = \Delta n * 2$
10:    $Size(List_P) = Size(List_P) * (cost(List_P) + 0.5)$
11:    $Size(List_C) = Size(List_C) * (cost(List_C) + 0.5)$
12: clean both $List_C$ and $List_P$
```

IV. IMPLEMENTATION

The dynamic window is implemented for RMLStreamer to extend its capabilities to do simple stream processing by joining multiple streams. Furthermore, we extended RML with new ontologies to allow the configuration of the window.

A. RML ontology extension

We extended RML to support configuration of window when joining the *reference object map* with its *parent triples map*. Currently, it only supports the rudimentary configuration of the type of join to be applied.

B. Dynamic window

The dynamic window is implemented using Apache Flink's *KeyedCoProcessFunction* API with the state management done using Flink's *ListStates*.

V. EVALUATION

To measure the effectiveness of dynamic windowing for multi-stream operators during the mapping of non-RDF heterogeneous data we need to measure the following metrics of our stream processing framework: *CPU usage*, *latency*, *throughput*, *memory usage*, and *completeness*.

A. Data

We use the same data as the benchmark in the paper by Van Dongen and Van den Poel [**evalution_of_spe**]. The data is provided by NDW (Nationale Databank Wegverkeersgegevens) from the Netherlands [1]. It consists of measurements of the number of cars and their average speed across the different lanes on a highway. The sensor data was replayed by a Kafka publisher into two topics *ndwflow* and *ndwspeed*, for the number of cars and the average speed respectively.

B. Evaluation setup

Docker[2] is set up with the containers as illustrated in Figure 1. The docker containers run on a standalone machine in order to mitigate the influence of network communication as much as possible. Apache Kafka is used as the message broker for our data. The setup for the Kafka broker is the same as described in [**evalution_of_spe**].

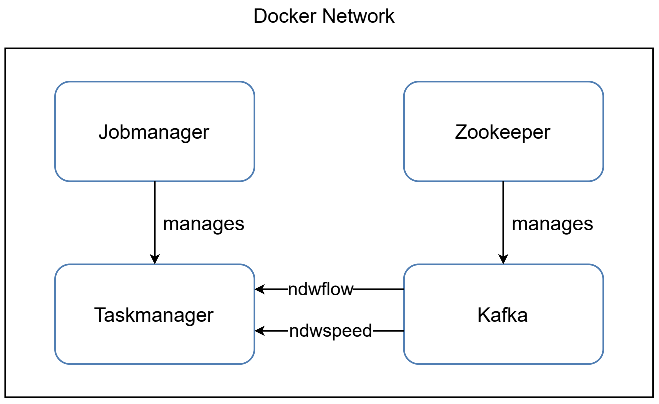


Fig. 1: Setup of application containers inside the same docker network. The kafka topics *ndwflow* and *ndwspeed* are consumed by the RMLStreamer job inside the Taskmanager.

[1]NDW data site: http://opendata.ndw.nu/
[2]Docker: `https://www.docker.com`

C. Metrics measurement

CPU usage, throughput, memory, and latency exposed by Flink's Rest API, are constantly polled every 100ms by a Python script scraper. CPU usage, throughput and memory usage are measured internally by Flink, while latency measurement is implemented manually using Flink's Metrics API to more accurately measure the amount of time spent in the window by a record before being processed.

The aforementioned metrics are averaged across the parallelized window operators. Intersection over union (IOU) is used as the metric to measure the completeness of the joined result generated by the windows.

1) CPU usage: For CPU usage, we measure Taskmanager's CPU usage, since it is the one responsible for executing the RMLStreamer code.

2) Throughput: Throughput measurement is defined as the number of records processed by the window operator per second. It is measured at the output of the window operator, since we want to measure output performance of the window operators.

3) Memory usage: Due to limitation in granularity of measurement in Flink, JVM heap memory of the job is used to estimate the memory usage of window operator. We expect the memory usage by other operators in the job to be consistent and low across the different evaluation runs since they are stateless operators.

4) Latency measurement: We measure the latency by attaching a processing timestamp to the records before they enter the window. Once the records are processed and emitted after the join, the difference between the current processing time and the attached old processing time is taken as the *latency* for the records.

5) Completeness measurement: To generate *bounded* input data for the static mapping engine, we write the stored data from the topics into a file on disk. This input data is processed by RMLStreamer in bounded data processing mode to generate the *complete* set of triples.

The generated output triples from the evaluation of the windows, and the bounded data processing are used to calculate the IOU metric to measure the similarity between the two outputs.

D. Workload scenarios

We evaluate our dynamic window implementation under different workload scenarios. These workloads are similar to the ones used in [**evalution_of_spe**] where relevant.

1) Workload for latency measurement: Measurement of the *latency* caused only by the window implementations, requires the stream processing framework not to be stressed by significant *throughput*. Thus, the Kafka broker publishes the records at a very low constant rate of around 400 messages per second for this workload. We denote this as *constant* stream rate in the paper.

2) Workload with periodic burst: We evaluate our implementation for its ability to cope with unstable streaming data sources with varying velocity. The workload has a constant low stream rate with an occasional burst of data. Therefore, 38 000 records will be published every 10 seconds which takes around 170ms to 180ms since the time taken to publish can deviate based on the load of the Kafka brokers. We denote this as *periodic* stream rate in the paper.

3) Workload for completeness: We will use two stream rates, constant and periodic, to measure the *completeness* of the results generated by the different windows.

VI. Results and Discussion

The workload evaluations are run multiple times for consistency of results. For each workload we give a brief overview of the performance gain achieved by the Dynamic window.

A. Workload for latency measurement

For latency, Tumbling window has a median value of 1915ms, with latency ranging from 1081ms to 2624ms (Figure 2c) more than 10× the lantency of Dynamic window, which has a median of 57 and ranging from 39 to 120ms (Figure 2d). Our improvement to fire the *trigger* event whenever a new record arrives inside the subwindow, allows Dynamic window to achieve sub second latency.

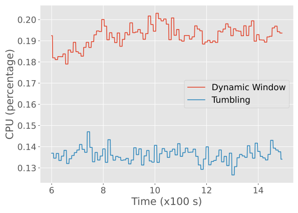
Dynamic window has a steady throughput of around 17200 records per second whereas Tumbling window fluctuates between 12500 and 12800 records per second before stabilizing at 12750 records per second (Figure 2b). This is due to the adjustment of window sizes at the subwindow level, allowing Dynamic window to wait for more records with infrequent *key* attribute in the stream, before evicting the subwindow. In contrast, Tumbling window always evicts the content of the window after 2 seconds without waiting for more records.

Relative memory usage of Dynamic window compared to Tumbling window is similar over the lifetime of the evaluation run (Figure 2e). Dynamic window causes infrequent *spikes* in memory usage of over 100 MB more than Tumbling window at a certain point in the lifetime of evaluation. This can be attributed to the subwindows of Dynamic window growing larger due to the low stream rate. However, Dynamic window stabilizes to a more optimal window size, where it uses less memory than Tumbling windows. At worst case, it uses as much memory as Tumbling window does over the course of the evaluation.

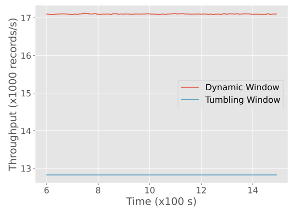
CPU usage is higher by around 7% for Dynamic window since it requires extra processing of the calculation of metrics and it increases the throughput causing RMLStreamer to process more joined records. (Figure 2a).

B. Workload for periodic burst

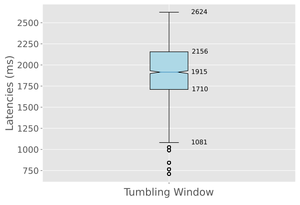
Dynamic window still handles the periodic burst of data with lower latency than Tumbling window with latency



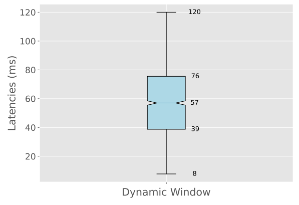
(a) CPU usage



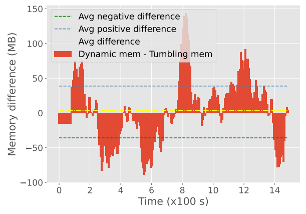
(b) Throughput



(c) Tumbling latency



(d) Dynamic latency

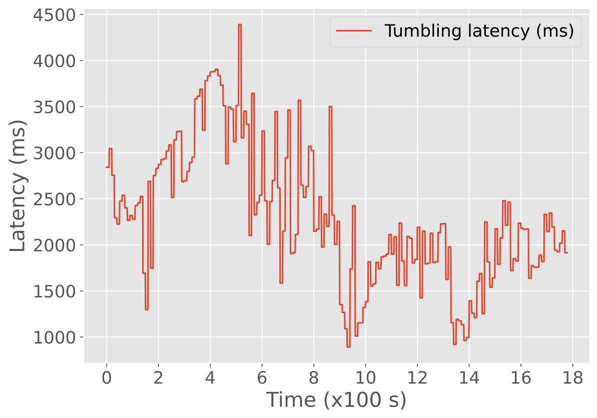


(e) Relative difference in memory usage from the perspective of dynamic window

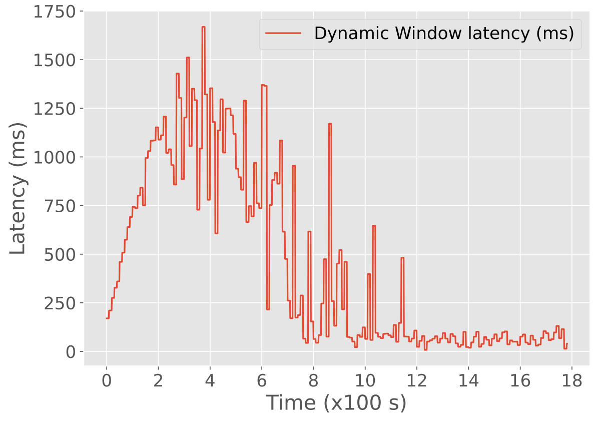
Fig. 2: Metrics measurements for latency workload. Dynamic window performs better in all measurements with the exception of CPU usage due to overhead in dynamic adjustment of subwindow sizes.

in the range from 8ms to 1669ms (Figure 4d) compared to Tumbling window's range from 891ms to 3904ms (Figure 4c) which is double the latency measured for Dynamic window. However, there is a temporary increase in latency at the beginning (Figure 3b), when the burst of data arrives at every 10th second. This is due to the initial size of 2s subwindows for the initial low stream rate of 400 records per second. The subwindow sizes start to grow **larger** than 2s because of the low stream rate. The increase in the subwindow size results in the window having more records to join; causing a back pressure to form and latency to increase. This results in a positive skew in the latency distribution for Dynamic window (Figure 4d). However, Dynamic window eventually manages to shorten the subwindow sizes as an adaptation to the periodic burst of data until it achieves sub second latency again.

The throughput of both windows increased as expected. Moreover, we observe a bigger difference in the throughput between the two windows of about 6000 records per second (Figure 4b). However, the constant and flat throughput measurement does not agree with the results of [**evalution_of_spe**] where there are clear "spikes" in the throughput measurement which correspond to the periodic burst of records processed by the windows. We

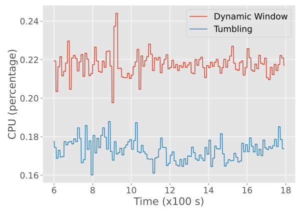


(a) Tumbling latency

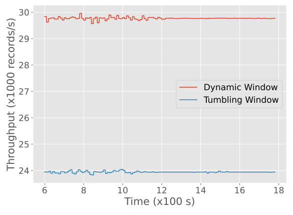


(b) Dynamic latency

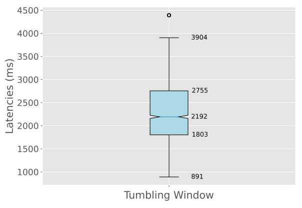
Fig. 3: Latency measurement of periodic workload over the lifetime of evaluation



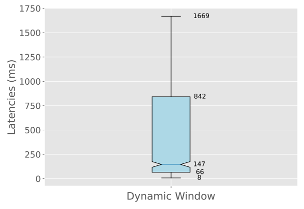
(a) CPU usage



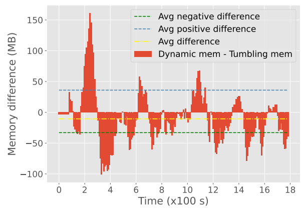
(b) Throughput



(c) Tumbling latency distribution



(d) Dynamic latency distribution



(e) Relative difference in memory usage from the perspective of dynamic window

Fig. 4: Metrics measurements for periodic workload. Dynamic window has lower memory usage on average than Tumbling window for periodic workload.

ran our evaluation as part of the whole RMLStreamer pipeline, causing a slight back pressure leading to a high and flat throughput measurement.

CPU usage difference of the windows, is similar to the workload for latency measurement with relative increase for burst data procesing (Figure 4a).

Surprisingly, Dynamic window uses about 10 MB less memory on average than Tumbling window during the lifetime of the evaluation (Figure 4e). The initial memory usage for Dynamic window is about 100 MB higher than for Tumbling window due to the long window growth caused by the low stream rate. However, once the dynamic adjustment of subwindow kicks in, reducing the memory usage.

C. *Workload for completeness measure*

From the results in Table I, and Table II, we conclude that Dynamic window outperforms Tumbling window in terms of generating a more *complete* output. Dynamic window has an IOU score of **1** for constant low stream rate due to the subwindow sizes growing large enough to accommodate all the required records, to generate the *complete* output set. In contrast, Tumbling window scores only **0.749**, leading to the conclusion that a window size of 2s is not enough to process the low stream rate of the evaluation data.

Similarly for periodic burst input, Dynamic window outperforms Tumbling window with a score of **0.982** whereas Tumbling window only scores **0.780**. The high IOU score of Dynamic window is due to its ability to adapt to the changing stream rate to hold enough records for maximal joined output generation.

Stream rate	**IOU score**
Constant rate	**1**
Periodic burst	**0.982**

TABLE I: Dynamic window's completeness measurement.

Stream rate	**IOU score**
Constant rate	**0.749**
Periodic burst	**0.780**

TABLE II: Tumbling window's completeness measurement.

D. Summary

In summary, these results show that our implementation of Dynamic window provides lower latency, higher throughput, and a more complete output than Tumbling window for both workloads of constant stream rate, and unstable periodic burst stream rate.

Although memory usage dropped in the workload with periodic burst rate, we could not confidently conclude that the Dynamic window effectively used less memory on average than Tumbling window. The measurement was based on the heap memory of the whole evaluation job, not just the window operator. Therefore, there is a need for a more precise measurement of memory usage.

The results for throughput also deviate from those reported by Van Dongen and Van den Poel(2020) [**evalution_of_spe**]. This is due to the fact that we ran the evaluation using the whole pipeline of RMLStreamer. This incurs some back pressure from the mapping stage, causing the throughput of the join stage to stay constant and flat.

Overall, we could conclude that Dynamic windowing is viable to replace the fixed size windows. Especially in use cases, where windows are not required to be of fixed size with variable stream rate.

VII. Conclusion and Future Works

In this paper, we have presented an approach for Dynamic window which adapts its window size according to the stream rate of the input data sources. We introduced a simple heuristic to adapt window sizes dynamically without huge memory or computation overhead.

We implemented our Dynamic window on top of the existing RMLStreamer, to evaluate its performance under a realistic processing environment. We adapted the benchmark framework as stated in [**evalution_of_spe**] to accurately evaluate the performance of our implementation against the standard fixed size Tumbling window.

The results show that our implementation of Dynamic window performs better than Tumbling window in terms of latency, throughput, and completeness with only a slight increase in CPU usage. Even though we could not confidently conclude that memory usage is lower in Dynamic window, our preliminary results indicate that it performs the same as Tumbling window in the worst-case scenario.

Therefore, there are still areas of improvement to be made. On the evaluation side, we could further increase the precision of our memory measurement by only counting the number of records residing in the windows at any moment instead of the whole JVM heap of the RMLStreamer job. Furthermore, the evaluation could be done in the same benchmark pipelines as in [**evalution_of_spe**] to further evaluate the Dynamic window performance in a general stream processing case. Improvements on our dynamic approach could be achieved by allowing users to define other statistical approaches to better calculate the threshold for adapting the window sizes.