UNIVERSITEIT GENT

Faculty of Sciences

Dynamic Windows processing in RDF Mapping engines for data streams

by

Sitt Min Oo
Student number: 01503244

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

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science

Academic Year 2020–2021

Preface

Sitt Min Oo, May 23, 2021

Acknowledgement

First, I would like to thank Anastasia Dimou for offering the opportunity to work with IDLab for the past summers. This provided me a lot of experience in working with Semantic Web technologies which were vital for this thesis. Furthermore, I would like to extend my gratitude to Ruben Verborgh for inspiring me to further specialize in the field of Semantic Web and for giving amazing lectures in Web Development where it all started. I would also like to thank Gerald Haesendonck for his continuous guidance and great feedback on the thesis throughout the course of this semester. Clearly, it would have been a much more difficult journey for me, without the guidance from your respective area of expertise.

Nevertheless, I would also like to thank the great friends I have made throughout my education at Ghent University. Dries, without you coming up for the first conversation during my freshman's year, I would have missed making a great friend for life. You and Arne are the best friends I could have for every single event, moment, and project we could do together! Kobe, Tibo, Jonathan, and Arthur, without your company, my days would have been spent silently working alone. A huge thanks to all of you guys for being the amazing friends that you all are!

Lastly, all of this would have been impossible, if not for my parents dedicating a significant part of their life to make it here. Thank you mom and dad for not giving up even when life was really difficult for you. I thank you both deeply from my heart for being there when I was at my lowest and guiding me to become who I am today.

Sitt Min Oo, May 23, 2021

Summary

Current state-of-the-art approaches to convert 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 *join* operator is one such commonly used operator in multi-stream processing. The existing approaches in mapping engines for supporting simple multi-stream processing operators are very limited. They require either a *window* with fixed size, which performs badly with changing stream rate, or consume the data from one of the input streams fully before applying multi-stream processing operators. Furthermore, related works for improving the dynamicity of windows, require a memory overhead of keeping complex stream statistics to adapt to the varying stream characteristics.

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.

Keywords: RDF, RMLStreamer, RML, Adaptive windows, Dynamic windows, Stream joins, Multi-stream processing.

Samenvatting

Huidige state-of-the-art benaderingen om niet-RDF naar RDF data om te zetten in een streaming omgeving focussen zich meer op de efficiëntie van het mapping proces met minimale ondersteuning voor multi-stream verwerkingsoperatoren. De join operator is zo'n veelgebruikte operator in een multi-stream omgeving. De bestaande benaderingen in mapping engines voor de ondersteuning van eenvoudige multi-stream processing operatoren zijn zeer beperkt. Ze vereisen ofwel een window met een vaste grootte, dat slecht presteert bij veranderende streamsnelheid, of houden de data van één van de inputs volledig in geheugen alvorens multi-stream verwerkingsoperatoren toe te passen.

Daarom hebben wij in RMLStreamer een dynamisch window geïmplementeerd, waarvan de grootte zich aanpast aan de veranderende streamkarakteristieken met verwaarloosbare geheugenoverhead, lage latency en hoge doorvoer. We hebben het dynamische window geëvalueerd onder verschillende werklast situaties met variërende stream snelheid. De resultaten laten zien dat het dynamische window een latency **GH**: vertalen als "vertraging"? in de orde van milliseconden bereikt, met een hogere doorvoer dan de windows met vaste grootte voor verschillende werklast.

Keywords: RDF, RMLStreamer, RML, Adaptive windows, Dynamic windows, Stream joins, Multi-stream processing.

Vulgarising Summary

This thesis is situated in the field of Semantic Web, where the World Wide Web, *the internet* as we know it, is extended with standards and specifications to make the data on the internet machine-readable. Most of these data are in different formats or from different sources, which might not be compatible with each other. Furthermore, they are also difficult for our computers to understand and use the data since they do not inherently have semantics. How do we make these data, which are already widely available, compatible with each other? This is where a part of the Semantic Web technology comes into play — the transformation of heterogeneous data to semantically linked data.

Mapping of heterogeneous data to linked data

Presently, large amounts of data are generated at very high frequency. Think about the millions of tweets generated by Twitter users at a high frequency at any moment in a non-linked data format. For this work, we are interested in transforming these data straight into linked data format, the very moment they are generated. Two major approaches currently exist to transform these data; a query-based and a mapping-based approach. We shall denote the engines responsible for transformation as *mapping engines*.

The query-based approach allows the user to pose *questions* to the *mapping* engines which will then interpret the *questions* to transform heterogeneous data to linked data. This removes the need for the user to define exactly how the transformation of data should be.

In contrast, mapping-based approach requires user to explicitly state the transformation in the form of rules. The mapping engine will then follow these rules to transform the data.

There are implementations of both approaches supporting the transformation of the data *online* — the very moment the data is generated. Unlike processing data in large files on disks, these online data sources are *unbounded* — theoretically there is no upper limit to the amount of data generated by the source. Therefore, we cannot hope to process everything in memory on a machine which is limited. This is where *window* processing is applied to allow processing on a subset of the *unbounded* data.

Window processing in data stream

Processing *unbounded* data is usually done by splitting the incoming data into subsets of data with *windows*. You could view windows like a bus which collects passengers at a bus stop (processing step). All buses are of fixed size and passengers board the bus one by one. Now, imagine a scenario where the number of passengers boarding the bus changes over time. The bus' fixed size will not be able to accommodate the passenger's arrival rate and they will have to wait for the next bus. This is not desirable as the passengers will have to wait (high latency) and the rate of passengers leaving are also lowered (low throughput). **GH**: Perhaps say somewhere that this is the case for a rising arrival rate.

Solution

The aforementioned problem leads us to propose a dynamic approach to windowing. Windows will change their size to process the data records depending on the arrival rate of the data. To change the size of the *window*, we calculate a metric to adjust the size based on the arrival rate of the data and the current size of the window. Bringing it back to the bus analogy, this leads to buses of different sizes coming to the bus stop to pick up the passengers to lower the wait time (low latency) and improve the departure rate of the passengers (high throughput).

Dynamic Windows processing in RDF Mapping engines for data streams

by

Sitt Min Oo

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science

Academic year 2020–2021

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

Faculty of Sciences
University Ghent

Abstract

Here comes abstract.

Keywords

RDF, RMLStreamer, RML, Adaptive windows, Stream joins.

Contents

List of Listings

List of Figures

List of Tables

Chapter 1

Introduction

Large data volumes are generated daily on the Web in various domains. Data integration is difficult across the different applications since these data are often structured according to an organization's specific needs or formats. These generated data might have to be associated with heterogeneous archival data, to provide a coherent view for various applications such as data analysis. Heterogeneous data formats, such as CSV or HTML, are not designed to link related entities across various datasets.

Knowledge graphs are a solution to this particular problem by enriching the data with knowledge and association across different domains, through the use of common ontologies. Knowledge graphs are also extensively used in social networks like Facebook[**facebook˙linked˙data**], IoT devices[**graph˙of˙things**] and especially with Google's search engine[**google˙kg**], they enable machines to understand the data and perform complex automated processing using knowledge graphs. Based on W3C standards, semantic data formats such as Resource Description Framework (RDF) triples [**intro˙rdf**] are a possible serialization of knowledge graphs. RDF triples form the basic building blocks of knowledge graphs.

Considering the aforementioned scenarios, there is a need to transform these non-RDF data to RDF compliant formats on the fly while new data are being generated. Furthermore, we would also like to apply stream operators on the input records before transforming, to enhance the enrichment of the data.

There exists state-of-the-art approaches to consolidate heterogeneous data and transform them to an RDF serialization. These RDF transformation engines can be divided into two major categories based on the type of input they consume; *bounded* and *unbounded* data input. We are focusing on RDF transformation engines on unbounded data, as this thesis is centred around generation of RDF data in a streaming environment.

Some engines 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. This leads to a decline in the quality of the generated RDF triples. Because of the nature of the infinite, continuous and real-time changing data of the streaming environment, and the limited computing resources by the systems processing the streams, these operators have to be applied in the context of windows over a subset of the incoming data. These operators are usually applied in fixed size windows. Clearly, with these restrictions and characteristics of the streaming sources, there is a need for an adaptive approach to apply these operators in windows for the data transformation engines.

Therefore, we propose a dynamic approach to windowing for multi stream operators. The application of these operators in a dynamic window should maintain high memory efficiency and throughput, even when the velocity of the streaming sources varies greatly over time. The dynamic window would make fast and simple statistical calculations to be aware of the velocity of the input streams to react and adapt its size accordingly. For implementing this dynamic window, we (i) investigate the possible implementation sites which are near the *input* or the *output* of the engine, (ii) provide a reference implementation utilizing the dynamic window, and (iii) evaluate the implementation against widely used windowing schemes for memory efficiency, throughput, CPU usage, latency, and completeness.

This work improves the performance of multi-stream operators by dynamically adapting the window size according to the dynamic characteristics of the incoming records for the multi-stream operators. Evaluation of our approach is then intended to validate the following hypotheses:

Hypothesis 1. Dynamic windows consume less memory than fixed size windows even under fluctuating stream rate of incoming records.

Hypothesis 2. Dynamic windows on operators enable lower latency processing than fixed size windows.

Hypothesis 3. Dynamic windows enable the operators applied on unbounded data streams to generate results closer to the results of offline stream processing operations than fixed size windows.

Hypothesis 4. Higher throughput is achieved by multi-stream operators with dynamic windows even with a fluctuating rate of input records from the data streams.

The rest of this work will be structured as follows. Chapter 2 will introduce common Semantic Web terminologies and specifications. Readers familiar with RDF framework and Semantic Web in general, could skip this chapter. Chapter 3 elaborates on general data stream processing and how the state-of-the-art stream processing engines handle multi-stream operators. Chapter 4 discusses state-of-the-art in RDF data generation from non-RDF data sources in detail. To understand more about windows in stream processing, Chapter 5 will go into detail about the current fixed size windows and the other state-of-the-art dynamic window operators. Chapter 6 describes the concepts and the implementation of our dynamic window followed by an evaluation of our implementation in Chapter 7. Chapter 8 discusses the results obtained from our evaluation, and the difficulty of evaluating the dynamic window in detail. Finally, Chapter 9 concludes our work with possible extensions and improvements in the future.

Chapter 2

Semantic Web Technologies

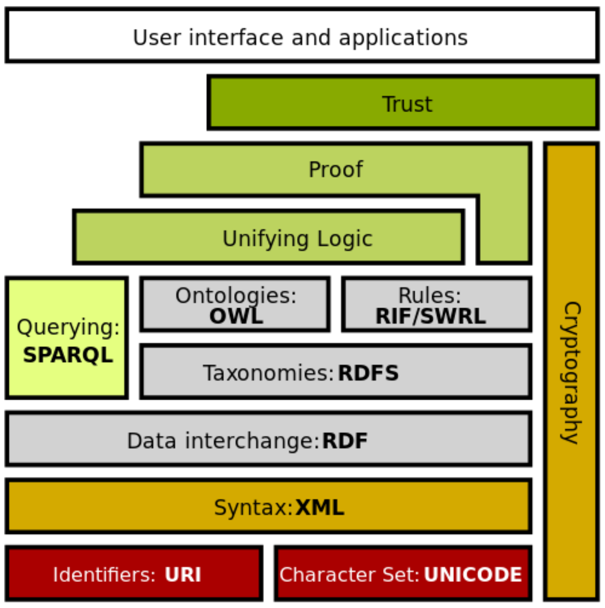


Figure 2.1: An overview of Semantic Web stack with core technologies[**sem˙web˙stack**].

The Semantic Web extends the existing *web of documents* with the ability to *link* different documents and *embed knowledge* in the document to transform it into a *web of data*. It is perceived by Tim Beners-Lee in 2001[**bernerslee2001semantic**] to be integrated into the existing architecture of World Wide Web. Embedding *knowledge* into documents enables machines to interpret the *meaning* of the document and interoperate with each other on a more complex level. This results in *knowledge graphs*.

The Semantic Web stack enables the development of a *web of data* using the core technologies shown in Figure 2.1. However, this is provided that the data is represented in the Resource Description Framework (RDF) model [**rdf˙concepts**]. Existing data on the web in non-RDF format must, therefore, be transformed into RDF to enable transition to a *web of data*. This work focuses on transforming non-RDF to RDF data, thus the RDF data model will be elaborated on in detail.

2.1 Resource Definition Framework

Resource Definition Framework (RDF) [**rdf˙concepts**] is a framework for representing data on the Web. It portrays the data as a directed graph with the resources as nodes in the graph and the edges as the relationship between the different resources. Figure 2.2 shows an example of an RDF triple statement describing the information "John has an apple". The triple statement consists of the subject *John*, the predicate *has* and the object *apple*.

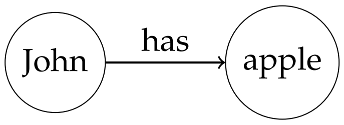


Figure 2.2: An RDF triple representing the information "John has an apple".

By composing these simple triple statements into a set of RDF triples, an RDF graph is created. In Figure 2.3, 4 triple statements are composed together to form a simple RDF graph describing *John* and *Mary* having the same *apple*. In contrast to the simplicity of figures, there are advantages to representing knowledge in RDF graphs. Data representation in a graph model allows machines to follow the *links* between the resources, and discover more unknown data in linked knowledge graphs. Following links is possible due to the nodes in the triples being classified as one of the 3 different term types.

2.1.1 Term types

Resources are classified into 3 different term types: IRI (Internationalized Resource Identifier), literals and blank nodes. An *IRI* is a unique string identifier in the global

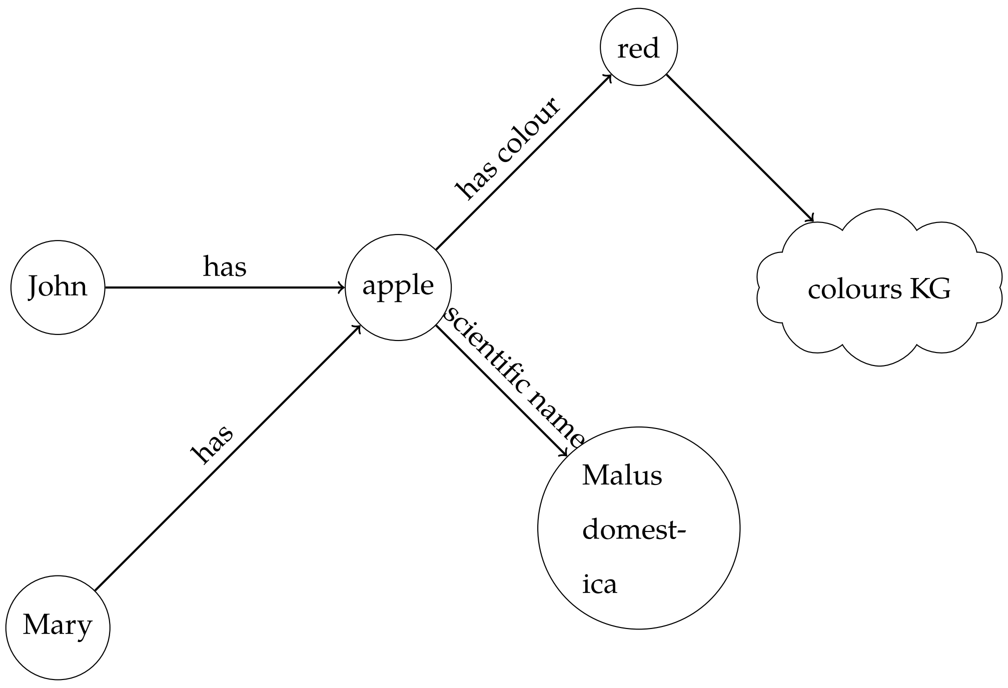


Figure 2.3: A simple RDF graph where the same "apple" is shared by both John and Mary.

scope to represent a resource. It is usually in the form of a web address, however, it can also be in other forms as long as it conforms to the syntax defined in RFC 3987 [1]. An IRI can represent a relationship, a concept or an object. Therefore, it could be used in the *subject*, the *predicate* and the *object* components of an RDF triple.

The *literal* term type is used to represent values such as strings, numbers, boolean, and dates. Each literal can have a datatype IRI to specify the type of the data it represents. Moreover, the language in which the data is written, could also be explicitly stated with a language tag. However, a literal can only have either a datatype or a language tag.

Lastly, we have the term type *blank node*. Blank nodes identify resources in the local scope (i.e. to a local file or an RDF store). Blank nodes are only applicable as the *subject* and the *object* components of an RDF triple, since they are used to identify resources in nodes.

[1]RFC 3987: https://www.ietf.org/rfc/rfc3987.txt

2.2 RDF serialization formats

The aforementioned term types are defined in the abstract RDF model and require a concrete serialization to store and process RDF data. Several serialization formats have been developed alongside RDF specifications to represent RDF knowledge graphs. These formats usually have trade-offs for simplicity of parsing against human readability. The following sections will describe some widely used formats in detail: RDF/XML, JSON-LD, N-Triples and N-Quads, and Turtle.

2.2.1 RDF/XML

The earliest of the serialization formats is RDF/XML[2]. It was developed during the early days of RDF when parsers, serializers and storage systems for XML were widely used. RDF/XML can be quite verbose and difficult for humans to read and understand, due to the mixture of a tree-structure document with a triple-based graph. Moreover, XML is not suited to serialize unbounded RDF data.

2.2.2 N-Triples and N-Quads

N-Triples[**N-Triples**] is a subset of Turtle, which we will focus on in this thesis. In comparison to the aforementioned two formats, N-Triples is trivial to parse. It represents the RDF statements consisting of *subject*, *predicate* and *object* line by line. Each line consists of a *subject*, *predicate*, *object* and a full stop indicating the termination of an RDF statement. N-Quads is a superset of N-Triples by adding a fourth column to denote the *graph IRI*. *Graph IRIs* are used to group RDF statements into multiple graphs and associate them with an IRI.

N-Triples and N-Quards are great for streaming RDF as they represent RDF statements line by line. However, without the usage of *prefixes*, it is verbose and takes more bandwidth compared to Turtle format; which we will elaborate in detail in the following section.

[2]RDF/XML: https://www.w3.org/TR/rdf-syntax-grammar/

2.2.3 JSON-LD

JSON-LD[**JSON-LD**] was developed as the modern Linked Data format to represent RDF data in general. It provides a way to continue using existing infrastructures using JSON and upgrade from JSON to JSON-LD to utilize the power of semantic knowledge graphs. Although the flexibility of extending existing infrastructures to support Linked Data usage is favourable, it has one major drawback: a high performance cost for parsing due to the verbosity of the format[3].

2.2.4 Turtle syntax

Turtle (Terse RDF Triple Language) [**turtle˙syntax**] is the most important serialization of RDF data in this work. A simple triple statement is a sequence of *subject, predicate, object* terms, ending in a '.'. To reduce the repetition of writing the same subject and predicate combination with different objects, Turtle allows the use of ',' to separate the different objects. Additionally, one could also use ';' to separate the different predicates and objects sharing the same subject.

Listing 2.1: Usage of ';' where triples share the same subject.

```
<http://example.org/apple>  ex:hasColor "red";
                            ex:scientificName "Malus domestica".
```

Listing 2.2: Usage of ',' where triples differs only in the objects.

```
<http://example.org/apple>  ex:scientificName "Malus pumila",
                                              "Malus domestica".
```

IRIs are written between angle brackets like <http://example.org/#John>. Since blank nodes are locally scoped version of IRIs, the same syntax to write IRIs is also used. Turtle allows us to define *prefixes* at the head of the Turtle document. Users could then use prefixes, to write RDF triples in a more compact form. For example, <http://example.org/#John> could be shortened to <#John> using the relative *@base* path or written with the prefix *ex:* as ex:John.

[3]Performance of parsing JSON-LD: http://manu.sporny.org/2016/json-ld-context-caching/

Listing 2.3: Prefixes in Turtle syntax.

```
@base <http://example.org/> . # default base IRI
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rel: <http://www.perceive.net/schemas/relationship/> .
...
```

A literal is written between double quotes and has by default the *xsd:string* datatype. One could also *cast* the literal value to a specific datatype by appending '^^ [IRI of the datatype]'. For example, "12" is cast to an integer in "12"^^xsd:integer. The language of the literal value can also be specified using @ similarly to datatype casting.

Chapter 3

Data Streams

With the advent of the *Big Data* era, large amounts of data are being generated by companies from their infrastructure; from IoT sensors to events generated by customers using online services. These generated data are *unbounded* and infinite, since new data are generated continuously. For example, the healthcare industry has started to adopt the approach of data-driven diagnostics methods [**hospital˙diagnosis**]. This transition is needed to keep up with the increasing amount of physiological data generated by the monitoring sensors attached to patients [**hospital˙data˙monitoring**]. These physiological data are periodically generated in a continuous streaming manner. Consequently, there is a need for companies to be able to process these unbounded data streams.

Data Stream Management Systems (DSMS) are equipped to handle such unbounded data and are adopted widely in the industry today in the form of various frameworks such as Apache Flink [**flink**] and Apache Spark streaming [**spark˙streaming**]. The main features of these DSMSs are their ability to (i) scale horizontally, (ii) process records individually as they arrive, and (iii) process or analyse the data in real time.

In Chapter 3, we will discuss details on the characteristics of data streams, how a DSMS typically handles fault tolerance, and a general strategy to process streaming data. Finally, we will elaborate on various state-of-the-art frameworks in use by the industry.

3.1 Characteristics of Data Streams

What is a data stream? As defined by Golab and Ozsu [**golab˙data˙stream**]: "A *data stream* is a *real-time*, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items. It is impossible to control the order in which items arrive, nor is it feasible to *locally store* a stream in its entirety."

Big Data is usually characterized with the three "V's: *volume*, *velocity* and *variety* [**big˙data˙an** In the context of data streams processing, it is expected that the data *volume* is unbounded since the devices, sensors and applications are constantly generating data as long as they are alive. The *variety* aspect of the data streams is heterogeneous when different sources emit different types of data; structured and unstructured. Moreover, *variety* in data can also happen over time; a concept drift where the properties of data may change over time unlike Big Data.

Out of the three aforementioned characteristics, *velocity* is the most important factor to consider when it comes to data stream processing. Data in a streaming environment are rapidly changing, constantly arriving into the stream processing pipeline, and need to be processed within a limited time frame. As a result, only single pass algorithms can be applied since it is infeasible to have random memory access capability over the entire stream. Moreover, unlike traditional Big Data processing, data stream processing requires *real-time* responses. This requirement of low latency processing has the consequence that late decisions will result in missing informations. Related to *velocity*, the data streams can also have a sudden *'burst'*, a sudden increase in the arrival rate of records, which varies over time.

Monitoring applications of data stream processing are interested in the analysis of most recent data. Consequently, the order of the data sequence needs to be considered when processing data streams. Because data streams are typically transmitted over a network, the order of the data cannot be guaranteed. This happens if the network transmission protocols have no guarantee in the global ordering of data or when latency is introduced [**requirements˙dsp**]. In certain applications such as fraud detection in bank transactions and monitoring of patient's biometrics, the data streams emitted by the sensors need to be processed in real-time. This is because the data streams in such scenarios will rapidly degrade in value if they are not handled immediately. Thus, real-time processing of data streams is a necessary requirement for data stream processing engines.

In the context of this work, where we focus on the application of multi-stream operators, the *velocity* of a data stream will have a huge impact on the quality and efficiency of the generated results. As a consequence of the challenges, as mentioned in the example use cases, a Data Stream Management System (DSMS) needs to be able to deal with the incoming data appropriately by ensuring that the results are generated with low latency, and high throughput without ignoring the timeliness of the data stream.

3.2 Data Stream Management Systems (DSMS)

Data Stream Management Systems are designed to overcome the aforementioned challenges in data stream processing. Different from the common big data processing, DSMSs should respond in a limited time frame and process the incoming data with low latency [**data˙stream˙management**]. For the former requirement, continuous queries are evaluated repeatedly by the DSMS to react to incoming data. The latter can be tackled by reducing the inter-operator communications through optimizing operator scheduling and buffers to reduce network traffic [**low˙latency˙data˙stream**].

3.2.1 Fault tolerance

DSMSs should also scale horizontally by distributed stream processing over multiple computing nodes. This leads to the requirement that there needs to be some form of coordination and fault tolerance for distributed stream processing. The exact details of guaranteeing fault tolerance differs between frameworks, however, there are general strategies to provide some form of fault tolerance according to Gradvohl et al [**fault˙tolerance˙dsms**]. In the context of this thesis, **fault tolerance** is a crucial requirement when applying non-trivial operators that are stateful. An example of a non-trivial operator would be a recommender algorithm, where a user's parameters would need to be kept in the state to update them with each new interaction event. This state would then need to be restored in case of failure to ensure the proper working of the recommender algorithm. Another example would be joining streams where a time window is a form of state, containing a subset of the data stream. The following sections will describe the commonly used strategies to provide fault tolerance in DSMS.

Replication of components

Components are operators or computation nodes in the execution pipeline which contains some kind of state vulnerable to failure. Just as the name suggests, it involves duplicating the components of the stream processing engine to minimize the error in case of node failures. There are two kinds of replication; *active replication* and *passive replication*. In **active replication,** duplicated components are executed just like normal components and receive the same incoming data. Therefore, there are duplicates in the output — this allows the engine to check the output to identify errors in the normal components. The disadvantage of this method is the increase in the cost due to component duplication.

Passive replication has inactive duplicated components instead. These backup components are then on standby to replace their counterpart if there is a failure. This reduces the performance cost of executing the duplicated components at the same time as their counterpart. However, starting the backup component requires the input to be resubmitted for it to catch up and return to normal state. This results in a significant latency every time there is a failure and not acceptable for the requirement of real-time response.

Checkpoint

Checkpoints represent states the DSMS could be in at a certain point in time for each of the input streams, along with the corresponding state for each operator. Snapshots of records for each checkpoint are also kept by the DSMS to be replayed in case of failure. In this fault tolerance strategy, a DSMS is modelled by a sequence of checkpoints which are fault free [**fault˙tolerance˙dsms**].

In the case of a failure, the operators will be restarted and set back to the state of the last successful checkpoint. The records, which were part of the failure, will be replayed at the input using the snapshot of the state.

There are two approaches to implement checkpoints at the operator-level; *asynchronous* and *synchronous* checkpoints [**fault˙tolerance˙dsms**].

Uncoordinated checkpoints do not guarantee a global state consistency, since each operator performs a checkpoint on their state independently. This results in less overhead of saving the state of the whole system with the disadvantage of the higher difficulty to revert to a consistent global state. Figure 3.1 shows a scenario where uncoordinated checkpoint strategy results in inconsistency of global state. In this example, message *m* is not part of the checkpoint of operator A even though operator B includes the sending of *m* as part of its checkpoint. Therefore, rolling back to this checkpoint will result in an inconsistent global state, causing the DSMS to roll back to an earlier consistent checkpoint. In the worst case scenario, this could result in the DSMS trying to repeatedly roll back until a consistent checkpoint is met, resulting in a 'domino effect'.

On the other hand, in coordinated checkpointing, the operators will take a snapshot of their internal state at the same "time" and save them to provide a consistent global state, avoiding the 'domino effect' of asynchronous checkpointing. To ensure the global consistency, a global synchronization mechanism is required. A simple algorithm to achieve this mechanism, is to have a central coordinator to synchronize the checkpointing process amongst the different operators. This introduces an overhead while coordinating these operators, increasing the latency of the operators whenever a checkpoint has to be taken.

Combined approaches utilizing different operators, such as asynchronous, and non-blocking operators, exist for checkpointing with the consistency of global states. The Chandy-Lamport algorithm [**chandy˙lamport**] determines the global state of a distributed system which could be used to maintain the checkpoints.

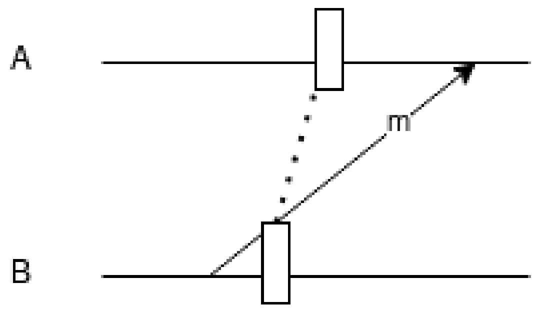


Figure 3.1: Inconsistent checkpoint scenario where operator A and B have different checkpointed states. The rectangle blocks indicate the moment when the operators commit a checkpoint of their internal state.

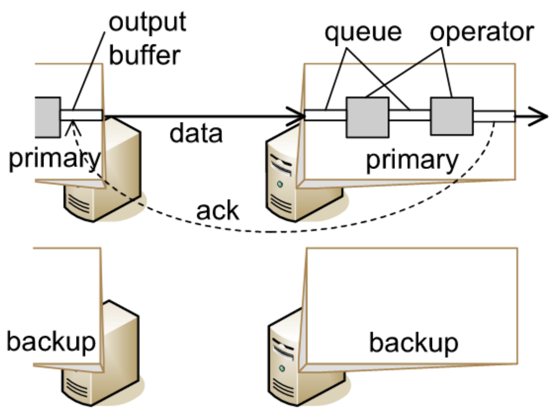


Figure 3.2: Upstream backup of the operators for fault tolerance [**upstream˙backup**].

Upstream backup

In this approach of fault tolerance, upstream operators will temporarily keep the records in their output buffer for downstream operators.The records stay in the buffer until they are completely processed by the downstream operator. The buffer will then be cleared when the upstream operator receives an acknowledgement from its direct downstream operators, as illustrated in the Figure 3.2.

The drawback of this approach is that the buffer might not be large enough to hold the records stored between the failure and the recovery event. Moreover, the whole system needs to be halted until the new operator catches up to the state of the failed operator from the replayed records.

3.3 Data Stream processing

Processing data streams requires a different approach than processing batch data due to the characteristics described in Section 3.1. Batch processing is defined as processing a small subset of the data simultaneously [**batch˙processing**]. Originally, it is meant to tackle a particular characteristic of Big Data; the **volume**. The job could therefore last from several hours to days, depending on the volume of the data and the complexity of the operations [**batch˙duration**]. Consequently, batch processing engines do not need to take the continuous and timely nature of the data into account [**flink**]. On the other hand, stream processing requires the notion of time and continuity of the data (Section 3.1). This brings several challenges that stream processing frameworks need to solve, resulting in a specialised architecture employed by the frameworks. In this section, we will summarise the general architecture and strategies to deal with the uncertainty of the order of items and the continuously processing of items in data streams. Different state-of-the-art frameworks will also be briefly mentioned, to get the reader familiar with the frameworks.

3.3.1 Lambda architecture

Batch processing introduces high latency — if a query is executed against a batch processing engine, it can take hours to return the results, which might be out of date once they are ready. A combination of the speed of stream processing with the accuracy of batch analysis allows one to process the unbounded data stream with low latency and maximal accuracy.

Lambda architecture [**lambda˙arch**, **lambda˙arch˙book**, **lambda˙arc˙bpost**] introduced by Nathan Marz, is a design pattern to combine the high accuracy of batch analysis and the low latency of stream processing engines. It consists of three separate layers; a **batch** layer, a **speed** layer and a **serving** layer. The three layers serve to enhance the result of the users' queries by taking advantage of both the low latency of the **speed** layer and the accurate analysis of the **batch** layer.

To further complement the three layers, the data stream is also split into two different paths:

- A **cold** path — data flowing in the cold path is not required to fulfil the low latency requirement. The **batch** layer is located along the cold path.
- A **hot** path — data along the hot path is constrained by the low latency requirement and needs to be processed at high velocity. Therefore, it flows through the **speed** layer.

Speed layer

The speed layer is responsible for processing the incoming data at high velocity and low latency. This layer focuses at low latency processing at the cost of a lower accuracy. The generated **real-time views** could also be used to incrementally update the **batch views** since they are more recent due to low latency processing. Stream processing engines are employed in this layer. **Windowing operators** are employed to restrict the stream processing to the elements contained within the **window**. This ensures that the analysis is done on a subset of the whole streaming data. Chapter 5 elaborates the different **windows** that are commonly used in stream processing.

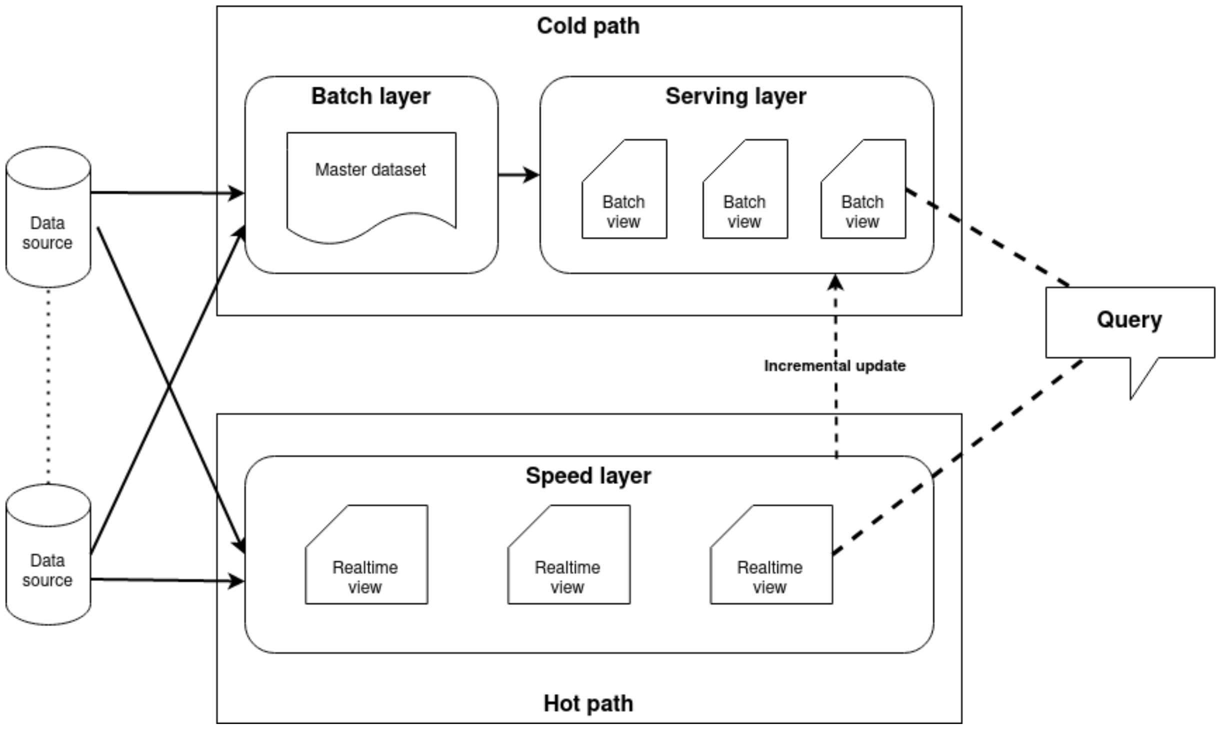


Figure 3.3: A general overview of the Lambda architecture. User queries are served from the real-time views until the batch views are ready. Incremental updates could also be applied to the batch views in the **serving** layer by using the results from the **speed** layer. [**lambda˙arch˙book**]

Batch layer

The batch layer stores the incoming stream data in a *master dataset*. This *master dataset* is an immutable and append-only dataset, to ensure that there is some notion of time or history of the data. Changes made to the previously stored data are re-appended to the dataset as a record with a new timestamp. This enables re-computation of the master dataset resulting in a new batch view as the dataset evolves over time. Batch processing engines, such as MapReduce [**mapreduce**]-based framework like Apache Hadoop [**hadoop**], are commonly deployed the batch layer for batch analysis.

Serving layer

The serving layer receives the batch views from the batch layer and updates them as necessary. When a user queries the application, the serving layer will redirect the user to the real-time views if the requested data needs to be timely. Otherwise, the batch views will be returned to the user which are more accurate but less timely.

Summary

Separating the whole processing system into two separate components along the two different data paths, the **batch** layer on the **cold** path, and the **speed** layer on the **hot** path, introduces complexity in maintaining the business logic of the system. If there is any change in the type of analysis done in one layer, the other layer needs to be updated with the corresponding batch analysis task. Therefore, there is unnecessary complexity in managing the system along the two dataflow paths.

3.3.2 Kappa architecture

Kappa architecture [**kappa˙architecture**], as proposed by Jay Kreps, combines batch and stream processing into one layer, to replace the two processing systems layout of the Lambda architecture. Intuitively, stream processing is only meant to handle a very small subset of the whole data. However, one could increase the parallelism of the streaming job and reprocess the whole historical data as a *subset*, by replaying the historical data at high velocity. This results in an output accuracy as batch analysis. The Kappa architecture is based on this intuition. Instead of consuming the streaming data sources directly with the stream processing engine, the input data is first stored in a distributed data log system like Apache Kafka [**kafka**]. The data logs can be stored indefinitely or temporarily depending on the use case. The stream processing layer has access to the data in two forms; a stream of events in real-time, or as a replay of historical data for batch analysis.

Summary

Even though the Kappa architecture solves the problem of code redundancy and complexity of system management introduced in the Lambda architecture, it is not meant as a full replacement for the Lambda architecture. The Lambda architecture is useful in cases where two different algorithms for batch analysis and stream processing are needed. For example, the batch layer could be responsible for training a machine learning model on the bounded dataset batches and improving the predictions, while the speed layer could deploy these models to provide real-time predictions.

In this thesis, we investigate the usage of dynamic windows in the context of stream processing heterogeneous data to RDF data. Therefore, the Kappa architecture is the best candidate to deploy and test our hypotheses since we have no need for batch analysis in this thesis.

3.3.3 Out-of-order processing

Stream events can arrive out-of-order due to a myriad of reasons; network latencies, bad connections or a delay at one of the multiple sensors generating the events for the stream processing engine. Stream processing engines should be aware of the *timeliness* of the data stream — it needs to be able to deal with out-of-order events.

To understand the strategy involved in the handling out of order events, two notions of *time* need to be introduced [**watermark˙flink**]:

Definition 3.1 (Processing time)**.** Processing time refers to the time of the system on which the stream processing engine is being executed. The system clock of the machine

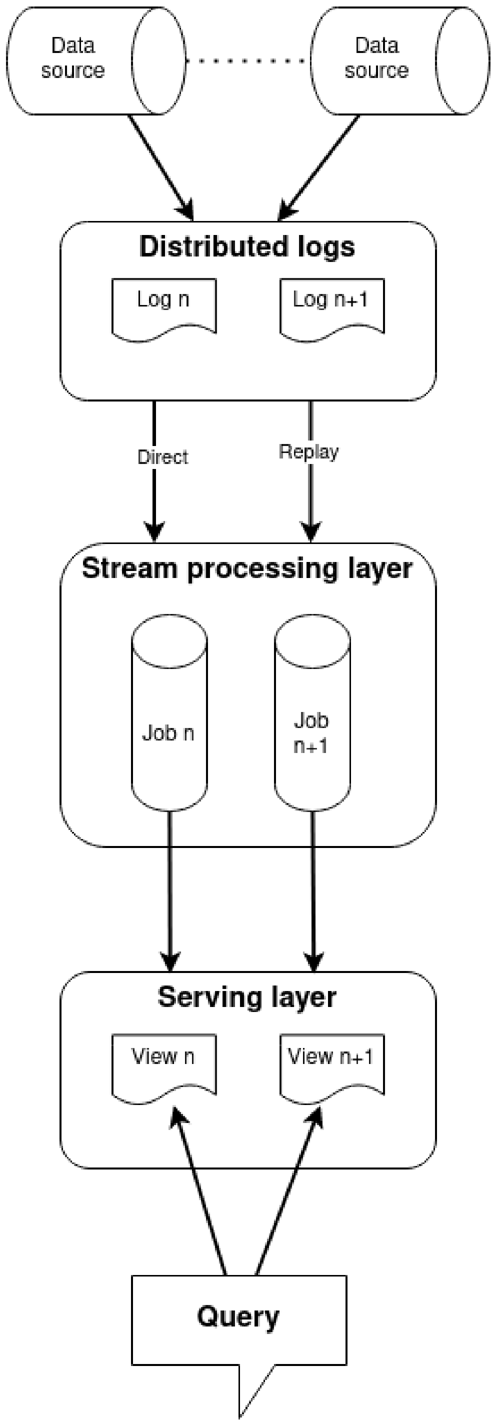


Figure 3.4: An overview of the Kappa architecture. Here, job *n+1* is re-processing using the replay data from the unified data logs in parallel. Once it has caught up to the output of job *n*, job *n* can be phased out and allow the latest job *n+1* to take over.

is therefore used to mark the *processing time*.

Marking events with processing time makes them susceptible to the speed and the order in which they arrive in the system. In a distributed environment, on which stream processing engines are often executed, processing time does not provide determinism. This is because reprocessing the same set of data stream will not lead to the same output, if the two execution are executed in sequence, since the processing time will be different, resulting in different watermarks.

Definition 3.2 (Event time)**.** Event time is the time at which the events are generated at the source.

For example, the *event time* of a user's click events, will be the exact time the user **clicks** on an object of interest. This timestamp is usually included in the record as they arrive in the stream processing engine. The engine could then extract these timestamp to use as *event time*.

Watermarking

The two notions of *time* are used in a strategy called **watermarking** [**watermark˙millwheel**]. Watermarks are timestamp thresholds used to deal with *very late* events in the data stream. It specifies how long the stream processing engine waits for *late events*. For example, if a watermark with timestamp 20 arrives in the dataflow, the system could assume that all events with timestamp earlier than 20 already have arrived in the system.

Notions of time and watermarks are insufficient to handle out-of-order events. The stream processing engine needs to be able to compare them to a subset of the incoming data. Windowing provides a grouped view of the infinite unbounded stream, enabling the engine to compare the subset of records for *timeliness* and out-of-order processing. A window can be *time driven*, every N seconds, or *data driven*, every N records. In Figure 3.5, a time window is applied on the stream that stretches from watermark $W(17)$ until $W(11)$. Once $W(17)$ arrives in the window, it knows that it contains or has processed all events earlier than timestamp 17 and later than timestamp of 11.

However, future events may arrive in the window with a timestamp earlier than 17. That event can be considered *late* as an outlier and be dealt with appropriately as defined by the developer. Different windows behave differently with the watermarking strategy. Chapter 5 will elaborate the workings of different windows and how windowing works in general.

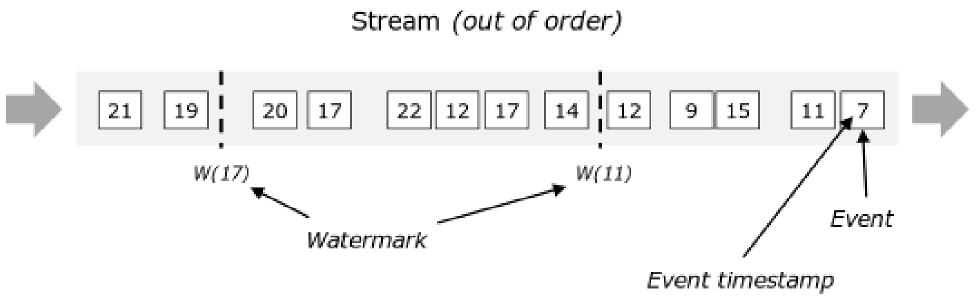


Figure 3.5: Out of order events in a stream with watermarks [**watermark˙flink**]. In this figure, all events on the right side of W(11) have a timestamp ≤ 11. If an event with a timestamp > 11 arrives in the future, it is considered *late*.

3.4 Frameworks

There are various state-of-the-art stream processing frameworks widely employed in the industry. These frameworks address the main challenges of Big Data stream processing, the three V's (Section 3.1), differently with their own advantages and disadvantages.

In this section, we elaborate on a few of these stream processing frameworks to identify the best characteristics of a stream processing engine and the different implementations to tackle the challenges by these frameworks.

3.4.1 Apache Spark Streaming

Apache Spark [1] is a batch processing framework developed to improve upon the existing implementations of the MapReduce paradigm. Spark processes data in RAM with

[1]Apache Spark: https://spark.apache.org/

a read-only data structure called *Resilient Distributed Dataset* (RDD) instead of working purely with a distributed file based system like Hadoop. RDD is distributed over different machines in the cluster allowing parallel operations. Fault tolerance is also enabled through incremental bookkeeping of how RDD is built over time, allowing it to be rebuilt in case of failure.

These RDDs are the crucial building block to implement the stream processing engine based on Spark; Apache Spark Streaming [**spark˙streaming**]. Spark Streaming structures stream processing as a series of *stateless, deterministic micro batch computations* on small interval times [**spark˙streaming**]. Discretizing the stream into micro batches frees up the stream processing engine from synchronization protocols and dependency of the data across time. Since the stream is modelled as *micro* batches, the powerful recovery mechanisms and exactly-once delivery semantics of the traditional batch processing engines could be applied. Furthermore, transparent integration of RDDs for both batch and stream processing in Spark, means that Spark Streaming utilizes the Kappa architecture to maintain low latency stream processing.

However, Apache Spark Streaming still has a higher latency compared to systems processing individual records in the stream, such as Apache Flink and Apache Storm, due to the discretization of data stream in *micro* batches.

3.4.2 Apache Storm

Apache Storm [2] is created by Nathan Marz. Storm's architecture consists of *streams of records* flowing through *topologies* [**storm˙twitter**], in comparison to the micro batches structure of the stream in Spark Streaming. This is a structure where operators are represented as *nodes* and the data flow between these operators as *edges* of a directed acyclic graph (DAG) as shown in Figure 3.6. The *nodes* are further distinguished between two main functions; *spouts* (data sources) and *bolts* (computations). By processing the stream events as individual records, the latency is kept to a minimum in the *bolts*.

Storm provides at-least-once delivery semantics through the use of record-level acknowledgements. A randomly generated "id" is attached to each new record and an

[2]Apache Storm: https://storm.apache.org/

extra *acker node* is attached to the execution graph. This *acker node* is responsible for tracking the records, enabling at-least once delivery semantics. Storm could be upgraded to enable exactly-once semantics with the utilization of an API called Trident [3]. However, Storm would then structure the stream events as mini batches just like Spark Streaming, forfeiting the low latency processing.

Implementing computation tasks in Storm is non-trivial, due to its low-level definition of the *bolts*, and requires the implementation of one or more readers and collectors. This results in non-intuitive composition of the execution topology. Furthermore, Storm does not support batch processing, offering no options to extend the functionality of the stream processing application if batch processing support is required in the future.

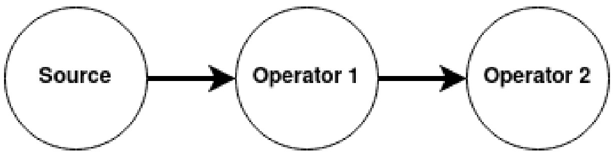


Figure 3.6: A common stream processing topology representation as a DAG where the dataflow is represented as the directional edges between de operators.

3.4.3 Apache Flink

Apache Flink [**flink**] aims to bridge the gap between batch and stream processing aspects of Big Data and combine them into one unified framework. Similar to Storm, Flink's execution topology is also internally represented as a DAG (Figure 3.6). It also processes the incoming records as individual events to maintain low latency processing. Flink recognizes that dedicated batch processing is still needed for applications where no efficient algorithms exist for performing the same analysis on streaming data. Therefore, Flink implements a specialized API for batch processing on top of its streaming processing engine.

Due to the focus on both batch and stream processing, Flink fulfils the use cases for which Spark and Storm specializes in; batch processing and stream processing respectively. Furthermore, Flink provides exactly-once delivery semantics, overcoming

[3]Trident API: https://storm.apache.org/releases/2.1.0/Trident-API-Overview.html

the need to filter out duplicate results and calculation. This is achieved by implementing a variation of the *Chandy-Lamport* algorithm called *Asynchronous Barrier Snapshotting* (ABS) [**asynchronous˙barrier**]. It is much more lightweight than Storm's record-level acknowledgements for both fault tolerance and exactly-once delivery semantics due to the usage of in-stream barriers.

Finally, Flink provides high-level APIs that make development easier for developers in the style of functional programming; an intuitive style for the use cases in the data analysis community. Windows are supported in Flink as an integral part of the engine with easy to use high-level APIs for commonly used windows (elaborated more on in Chapter 5). It also provides access to low-level APIs to implement highly customised windowing logic, without burdening the developers with the complexity of the internals of Flink. With the ease of development, the aforementioned low latency, and exactly-once delivery semantics, Flink is the best candidate to move forward with the implementation of dynamic windows for this thesis.

Chapter 4

RDF data generation from non-RDF data

How do we generate RDF data from non-RDF data? A variety of specifications and languages exist which define the generation of RDF data from non-RDF data. These specifications and languages can be grouped into two major groups; query languages and mapping languages. Several state-of-the art implementations, utilizing these two groups of languages, exist to generate RDF data from non-RDF data.

The implementations can be categorized into two groups; *query-based* and *rule-based* implementations. These can be further categorized into two more subgroups based on the data that they consume; *bounded* and *unbounded* data. This work will focus on implementations working with unbounded data in a streaming environment (Chapter 1), thus implementations consuming bounded data will not be elaborated. In this section, we will first discuss the *languages* these implementations employ to transform non-RDF to RDF data. Finally, the details of the implementations are elaborated with simple examples.

4.1 SPARQL Query Language

Query languages such as SQL [**sql**] already exist in established relational database systems such as MySQL or PostgreSQL. It allows users to manipulate and retrieve data from relational databases using concise statements. Due to its widespread use in the industry for querying databases, it is important that a similar query language is used for RDF datasets to ease the transition for the users.

SPARQL [**sparql**] achieves this goal by having an SQL-like syntax to allow a seamless transition to RDF for existing data engineers. In terms of relational databases, RDF datasets can be viewed as a table consisting of three columns — the *subject* column, the *predicate* column, and the *object* column. Unlike relational databases, RDF datasets, in a table representation, allow the object column to be of heterogeneous datatype 2.2.4.

Moreover, different from SQL, SPARQL allows matching based on *basic graph patterns* composed of a set of *triple patterns*. Triple patterns are similar to the triple statements as clarified in Section 2.2.4 but extended with declared variables (i.e. *?variable_name*). The declared variables are then bound to the concrete value in the corresponding *triple statements*, matching the given *triple pattern*, from the RDF dataset. The result of a SPARQL query is returned as an RDF sub-graph of the queried RDF dataset.

How does this query language relate to transforming an unbounded dataset to RDF format in a streaming environment? There exist state-of-the-art engines utilizing SPARQL syntax for generating RDF data from heterogeneous streaming data sources, which will be elaborated on in Section 4.3.

Listing 4.1: Example of a SPARQL query of a medication on the given input.

```
SELECT ?medication
WHERE {
    #Basic graph pattern consisting of 2 triple patterns.
    ?diagnosisId example:name "Cancer" .
    ?medication example:canTreat ?diagnosisId .
}
...
——————
#Input RDF graph
<http://example/diagnosisID/1> example:name "Cancer".
<http://example/medication/RadiationTherapy>
        example:canTreat <http://example/diagnosisID/1>.
```

medication
http://example/medication/RadiationTherapy

Table 4.1: Result of executing the SPARQL query in Listing 4.1

4.2 RDF Mapping Language

RDF Mapping Language [**rml**] is a superset of W3C's R2RML [**r2rml**] which maps relational databases to RDF datasets. RML improves upon R2RML by expressing mapping rules from heterogeneous data sources and transforming them to RDF datasets whereas R2RML could only consume data from relational databases. An RML mapping document is composed of one or more *triples maps*, which in turn consist of *subject, predicate* and *object* term maps. As the names imply, the term maps are used to map elements of the data sources to their respective terms in an RDF triple. The definitions of these maps are similar to the specifications in R2RML [**rml˙tech**].

Logical sources could be defined by specifying the *source, logical iterator* and zero or one *reference formulation* property. Although the reference implementation RMLMapper can only process logical sources consisting of bounded data, the expressiveness and the flexibility of RML allows one to extend it to support unbounded data sources. For example, RMLStreamer [**rml˙streamer**] implementation in Section 4.4.2 extended the base RML vocabulary to support logical sources with unbounded data.

RML also supports defining relationships amongst the different triples maps through the use of *rr:parentTriplesMap, rr:joinCondition, rr:child and rr:parent* properties. Different triples maps might come from separate logical sources, therefore, referencing across triples maps might require applying the join operator across multiple logical sources.

Moreover, an ontology to semantically define functions, FnO [**fno˙ben**], has been supported in RML, allowing users to apply user defined functions during the mapping process. This extension provides RML implementations with the capabilities to transform the input data with lightweight computations before mapping them into RDF triples; providing some form of data processing.

Listing 4.2: An example of an RML mapping document [**rml˙tech**].

```
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix rml: <http://semweb.mmlab.be/ns/rml#>.
@prefix ex: <http://example.com/ns#>.
@prefix ql: <http://semweb.mmlab.be/ns/ql#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@base <http://example.com/ns#>.

<#TransportMapping>
  rml:logicalSource [
    rml:source "Transport.xml" ;
    rml:iterator "/transport/bus/route/stop";
    rml:referenceFormulation ql:XPath;
  ];

  rr:subjectMap [
    rr:template
      "http://trans.example.com/stop/{@id}";
    rr:class ex:Stop
  ];

  rr:predicateObjectMap [
    rr:predicate rdfs:label;
    rr:objectMap [
      rml:reference "."
    ]
  ].
```

4.3 Query based implementations

Intuitively, querying the data source with a query language is the common method to interact with a database. Allowing users to query the sources without explicitly defining the mapping semantics hides the mapping or data transformation details from the user. Moreover, it eases the transition to developing applications using RDF data since the syntax is similar to the existing query languages leading to lower learning curve for the developers. To the best of our knowledge, there exist two state-of-the-art template based implementations, using SPARQL query-like languages, for transforming unbounded non-RDF data to RDF data: SPARQL-Generate and RDF-Gen. These implementations and approaches will be elaborated more on in the following sections.

4.3.1 SPARQL-Generate

SPARQL-Generate [**sparql˙generate**] was proposed as an alternative to existing methods of transforming non-RDF data. The language is based on an extension of SPARQL 1.1 query language, to leverage its expressiveness and extensibility. Furthermore, SPARQL-Generate can be implemented on top of any existing SPARQL query engine. The reference implementation in the paper was based on Apache Jena's SPARQL 1.1 engine. Due to the use of the existing SPARQL 1.1 query language, experienced knowledge engineers could use SPARQL-Generate to improve the generation of RDF data in their existing workflow.

SPARQL-Generate supports the consumption of heterogeneous data sources by exposing the *binding* and *iterator functions API*. Therefore, covering a new data format or data source could be accomplished by implementing the corresponding *binding* and *iterator functions*. Currently, the reference implementation supports the consumption of data sources which are unbounded in a streaming environment or bounded datasets. For example, WebSocket and MQTT are currently supported in the latest version of SPARQL-Generate.

Although there is support for data stream processing, the paper did not go into details about the application of multi-stream operators like joins when involving multiple streaming data sources. 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. The "child" stream will then be iteratively consumed and joined with the records from the internal index. Hence, we could assume that SPARQL-Generate processes the "parent" stream in memory when there is a multi-stream operator. Furthermore, it is mentioned that SPARQL-Generate could use the SPARQL 1.1 operators such as *join operators*. From this, we could derive that SPARQL-Generate will apply *join* on the records by delegating it to the underlying SPARQL engine. Therefore, there is a lack of dynamic windowing schemes to efficiently exploit the characteristics of the streaming data sources.

4.3.2 RDF-Gen

RDF-Gen [**rdf˙gen**] is also based on a SPARQL-like syntax. However, instead of extending SPARQL engines, it provides its own architecture to meet the demands of real-time processing (Figure 4.1). Instead of adopting most of the syntax from SPARQL, RDF-Gen only keeps the basic graph pattern section of a SPARQL query to reduce the size of the transformation specification. Thus, it has the most compact mapping template compared to the other methods mentioned in this paper.

RDF-Gen consists of three main components: (a) Data Connector, (b) Triple Generator, and (c) Link Discovery.

Data Connectors allow close-to-source processing, which is not the case for SPARQL-Generate. This is due to the data consumption module being tightly coupled with the mapping process for SPARQL-Generate, unlike RDF-Gen as shown in Figure 4.1. Triple Generator handles the rapid generation of RDF triples by making use of template graphs and variable vectors. The Link Discovery component solves the problem of the link discovery problem which is defined as follows: Given two data sources S and T, the problem is to find the pairs of elements in $S \times T$ that are related to each other (e.g. following the predicate of *owl:sameAs* property). Link discovery remains out of scope of this thesis as we mainly focus on multi stream operators during RDF generation.

Data Connector

Data Connector has a similar functionality as the *iterator functions* from SPARQL-Generate. It consumes the data sources given a configuration setting. The configuration setting is used to specify the type of data sources, the *window* for processing the incoming data records and also apply functions on the incoming data elements. Data Connector can thus be defined as a mapping function as in Definition 4.1.

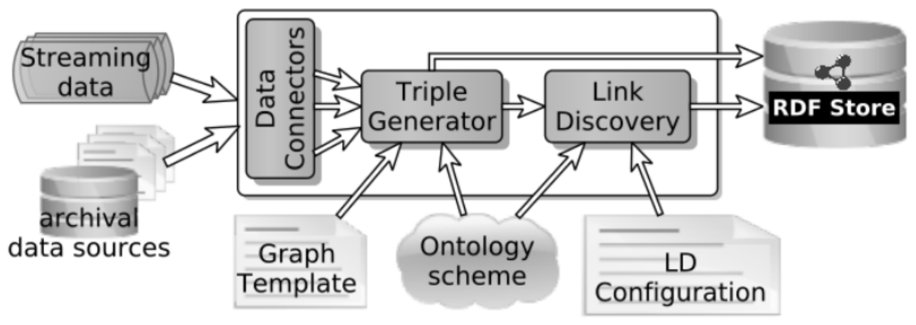


Figure 4.1: Architecture of RDF-Gen with dataflow from input sources, and components requiring their specific configuration files [**rdf˙gen**]

Definition 4.1 (Data Connector record [**rdf˙gen**]). Given a set of data sources $D = \{d_1, d_2, \ldots, d_n\}$ and a mapping function $F = \mu_f(d_i, e) \mid \forall d_i \in D$ with e, a data element of a data source d_i, and f, a filter function. Data Connector generates a record $R = \mu_f(d_i, e)$ $\iff$ all the attributes of e satisfy the filter function f. By default, the filter function f just returns true.

Using the Definition 4.1, we could now also apply an equi-join operator on the data sources. Formally, we could generate a new triple $R = \mu_{f_i}(d_i, e_i) \bowtie \mu_{f_j}(d_j, e_j)$ where e_i and e_j have common attributes under the filter functions f_i and f_j.

The processing is done on individual records, leading to RDF-Gen treating streaming and archival data sources the same way — as "streams" of records. Due to the record-by-record processing, the framework also has a very low memory usage.

Triple Generator

Triple Generator consumes the output records of the Data Connectors to convert them into RDF triples (Figure 4.1). A vector of variables V, an RDF graph template G, akin to the basic graph pattern from SPARQL 1.1, and a set of *functions* F can be used to configure the Triple Generators.

V consists of variables which corresponds to the attributes of the generated records from the Data Connector. These variables are referenced in the graph template G, and then used to bind to the attribute value of the record provided by the Data Connector.

In case these variables are the arguments of a function $f \in F$, the function will be evaluated and the result appended to the output. Therefore, this simple binding of values in a template graph enables Triple Generator to generate RDF triples efficiently and have a high scalability. The functions will have to be simple in complexity to keep the computation time as low as possible.

In this paragraph, we will illustrate how a Triple Generator works in practice through an example. Listing 4.3 shows an example of a graph template G provided to the Triple Generator to generate RDF triples. In this example, the provided vector of variables is $V = [\text{?diagnosis\_id}, \text{?name}]$. Using the record from Table 4.2, the specified variables, *diagnosis_id* and *name*, will be bound to the values *100* and *Cancer* respectively. Afterwards, the functions *makeUri* and *asString* will be called with the bounded values as arguments and the generated output will be used to generate the RDF triples specified by the graph template. The final generated set of RDF triples is (e.g. in Turtle format):

```
...
<http://example.com/100>  a example;
                          example:name  "Cancer".
...
```

diagnosis_id	**name**
100	Cancer

Table 4.2: A sample record generated by the Data Connector.

Listing 4.3: A simple graph template G with the functions *asString* and *makeUri*.

```
#BGP for the diagnosis data source
makeUri(?diagnosis_id)  a example:Diagnosis;
                        example:name asString(?name).
```

Consuming the input data on a record-by-record basis results in RDF-Gen having to handle multi-stream operators with the use of windows. However, the paper did not mention in detail about its implementation of windows to handle multi-stream operators since it was out of scope. Therefore, we still have no notion of how to handle data streams with dynamic characteristics where fixed windows size have negative consequences on the quality of the generated RDF data.

4.4 Mapping based implementations

Other than approaching the transformation of non-RDF to RDF from the viewpoint of queries, one could also employ mapping languages such as RML. Mapping languages are declarative and have a minimal cognitive burden composing it compared to the query based implementations; one does not need to deal with nested queries, due to defining the mapping relationship for each attribute in the input source. Despite being more verbose than query languages, composing complex mapping documents results in a much more human-readable format due to the direct mapping of the attributes. **GH**: Do you have a reference that can support these claims? The following subsections will elaborate more on the related state-of-the-art engines which utilize mapping languages in a streaming environment.

4.4.1 TripleWave

Despite the abundance of solutions combining semantic technologies with stream and event processing techniques, there was a lack of engines that disseminate and exchange RDF streams on the Web [**triple˙wave**]. TripleWave [**triple˙wave**] was conceived to play this role; to provide the mechanism to publish and spread RDF streams on the Web. It extends R2RML, which only allows ingestion of inputs from relational databases, to also consume other formats such as JSON or CSV and other data sources (just like RML, a superset of R2RML).

TripleWave generates an RDF stream in JSON-LD format, which could be consumed by existing RDF Stream Processing (RSP) engines as input, for the publication of RDF data. It also supports joining of multiple streams which could be inferred from its implementation of R2RML's *rr:parentTriplesMap* predicate [**triple˙wave**]. Since the goal of TripleWave is to publish RDF streams from RDF and non-RDF data sources, it does not support the application of arbitrary functions at its core unlike RDF-Gen and SPARQL-

Generate. However, as is the case with the aforementioned frameworks and engines, it does not support dynamic windowing to handle streams with dynamic characteristics.

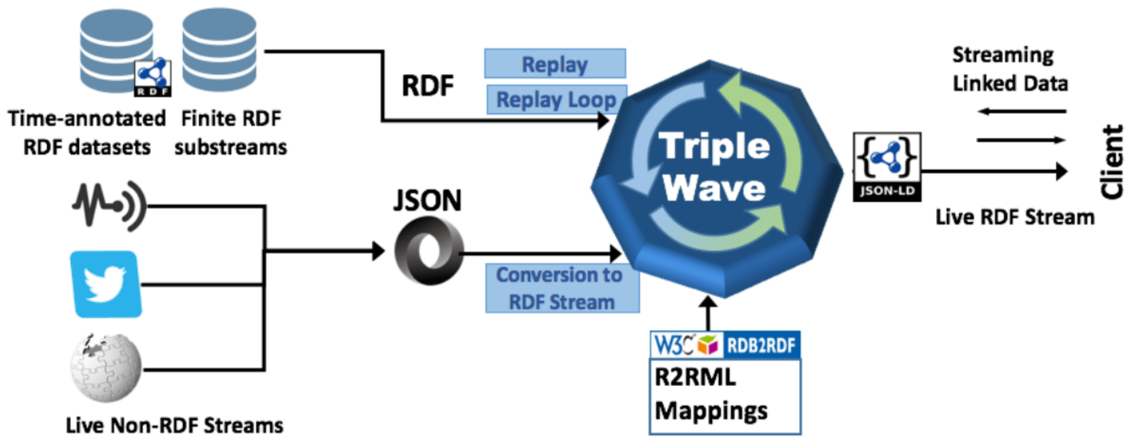


Figure 4.2: Architecture of TripleWave generating RDF streams from non-RDF and RDF data sources [**triple˙wave**].

The shortcomings The implementations elaborated on thus far scale insufficiently in terms of velocity and volume of data streams. Multiple instances of the engines will need to be started in order to scale with the higher volume and velocity of the input data streams. Since they are not built upon a distributed framework, there is also a need for a separate implementation to coordinate the different instances in a distributed environment.

4.4.2 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.

Due to the parallelization of the subtasks, the ingestion and the mapping process, these processes could be spread over different machines for distributed execution (Figure 4.3). However, a mechanism to coordinate the different machines is required in the environment of distributed computing. To fulfill this requirement, RMLStreamer is implemented on top of Apache Flink [**flink**] framework (Section 3.4.3). Not only does it allow the mapping of non-RDF to RDF data, it also guarantees fault-tolerance through the usage of *Asynchronous Barrier Snapshots* [**flink˙fault˙tolerance**] and *exactly-once* semantic, although these are disabled by default.

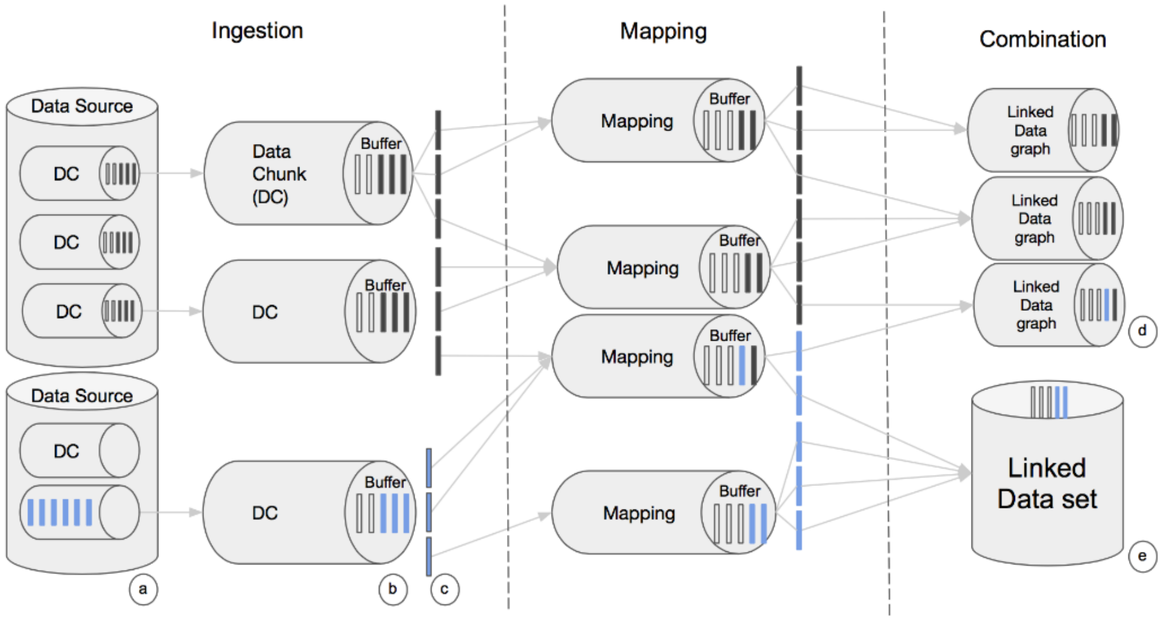


Figure 4.3: Parallelization architecture of RMLStreamer for RDF data generation [**rml˙streamer**]. RMLStreamer parallelizes the ingestion of the data sources and the mapping process of non-RDF to RDF data.

4.5 Summary

While SPARQL-Generate [**sparql˙generate**], RDF-Gen [**rdf˙gen**], and TripleWave [**triple˙wave**] support the join operator for executing a binary join, scalability is not built in those implementations to handle multiple streaming data sources with high velocity and volume. Therefore, a solution with built-in capabilities to handle high velocity and volume of data, is of interest to implement the dynamic windowing scheme. Hence, RMLStreamer is the best candidate for this work on dynamic window in RDF generation.

Chapter 5

Windows

Due to the continuous and infinite characteristics of streaming data, it might be impossible to process all data in memory. Therefore, stream processing engines utilize buffers to hold the most recent stream of records in memory. These buffers are called *windows*. Windows make it possible to apply streaming versions of traditional data processing operators such as joins [**grubjoin**], sorts and aggregations.

The behaviour of *windows* are configured by specifying *policies* based on different characteristics such as *count*, *attribute-delta*, *time*, and *punctuation* [**generic˙window˙sem**]. For this thesis, only a *time*-based policy is of interest to proof and evaluate the concept, specifically with the notion of *event time*, since we are dealing with streaming data with timeliness. **GH**: Here we need a good motivation why we limit the scope to the time characteristic. I can imagine a use case where we generate triples only if there are 5 particular records encountered (so count based, e.g. a thermometer signals a high temperature which might trigger an alert). A reason can be to proof the concept, and then later in the discussions / future work section say that the concept could also be investigated for other characteristics as well.

Moreover, the configurations of the policies will be based on the implementations in Apache Flink and will differ from those defined by Bugra Gedik [**generic˙window˙sem**].

To understand the different *windows* and *policies*, we will elaborate on them in the following sections. First, a few basic notations will be defined to help in the description of the semantic behaviour of the different *windows* and *policies*. Secondly, we will describe the mechanism of the different windows **with the semantic implication** imposed by the *time*-based *eviction* and *trigger* policies. Finally, a review of different state-of-the art custom window operations will be discussed to aid us in the development of our methodology for dynamic window processing.

5.1 Window Preliminaries

Before discussing different *window* types, we will introduce a few definitions and notations to describe *windows* based on the notations used in [**generic˙window˙sem**]. These notations and definitions will be used to aid in explaining different semantic behaviour of the *windows* under different *policies*.

Notations $W = \{r_i | i \in [0 \ldots |W| - 1]\}$ is a window, whose size is denoted by $|W|$ in terms of the number of records inside the window W, and r_i as the ith record inside the window. The records r_i are orderd from the newest to the oldest starting from $0 \ldots n$ with the oldest record being r_n with $n = |W| - 1$. The current record to be put into the window is denoted as r_c. The event timestamp associated with the record r_i is denoted as $\tau(r_i)$.

5.2 Window Types

According to Bugra Gedik [**generic˙window˙sem**], windows could be categorized into three different types; *tumbling* windows, *sliding* windows [**stream˙standford, spade˙stream**], and *partitioned* windows.

Tumbling and *sliding* windows are also commonly provided as default window types by stream processing engines. They offer the most flexibility in customization with *windows parameters* to fit most of the use cases. *Partitioned* windows are special variations of the *tumbling* and *sliding* windows, where a partitioned window consists of different sub-windows of same size. This enables *multiplexed processing* where independent computations are done across the different sub-windows at the same time, resulting in a higher throughput due to the parallelization of the processing [**generic˙window˙sem**].

5.2.1 Tumbling Window

Tumbling windows have a *fixed* window size and do not overlap. The size of the window influences the number of elements residing in the window. It stores the elements until the window is *full* and this is determined by the *eviction* policy. In the context of this thesis, the *size* of the window will be defined by a time period bounded by a lower and a higher event timestamp.

Processing of the elements within the window happens only when a *trigger* event is fired. This is determined by the *trigger* policy which fires the *trigger* event. For tumbling windows, the default *trigger* policy is fulfilled when the window is *full*.

Eviction Extending the notations used in Section 5.1, we define the event time based tumbling window $W = \{r_t | t \in \tau(r_0) \ldots \tau(r_n)\}$. Then, the size of the window is $|W| = [\tau(r_l), \tau(r_h)[$, where $\tau(r_l)$ and $\tau(r_h)$ represents the lower and the upper bound of the records timestamp respectively. The current record r_c must have a timestamp $\tau(r_l) \leq \tau(r_c) \leq \tau(r_h)$ to be accepted by the window. If $\tau(r_c) \geq \tau(r_h)$, the record will be considered *late* as described in watermarking mechanism in Section 3.3.3.

Since tumbling windows are processed only when the window is *full*, *trigger* events are fired at the same time as *eviction* events for tumbling windows.

5.2.2 Sliding Window

Similar to tumbling windows, sliding windows also have a *fixed* window size that is immutable throughout the lifetime of a stream processing job. In addition to the window size as a parameter, one could also specify the *window slide* for sliding window. This parameter specifies how frequently a sliding window is created. Therefore, if the *window slide* is smaller than the *size* of the window, an overlap will be formed and multiple windows could exists at that point in time. The records in the overlapped region will have to be copied across multiple overlapping windows. Taking the *slide* and the *size* into account, we could conclude that a tumbling window is a special case of a sliding window where the *slide* is equal to the *size* of the window.

Due to the overlapping of windows, extra checks have to be considered when evicting the elements in sliding windows, to ensure that the elements in overlapping windows are not removed from multiple windows. To overcome this overhead of checking when evicting the window, stream processing engines copy records across overlapping windows. This reduces processing time and latency at the cost of higher memory consumption.

The **two *policies*** for the sliding windows are the same as those for the *tumbling* windows.

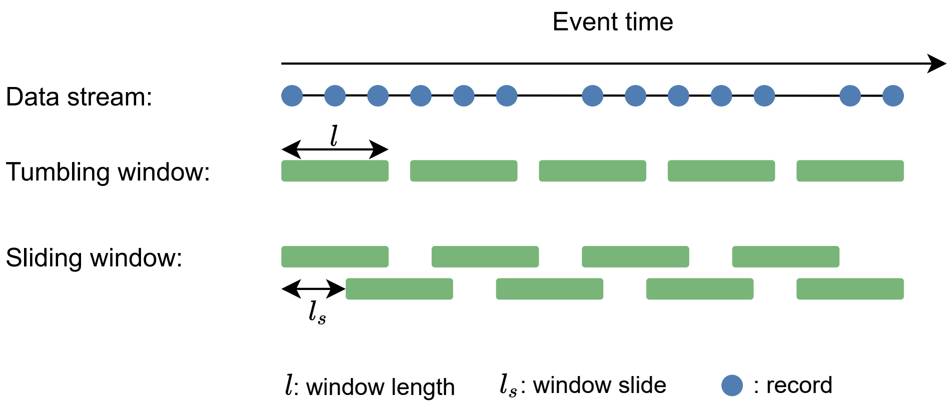


Figure 5.1: Sliding and tumbling window applied on a data stream [**jonas˙scotty**].

5.2.3 Partitioned Window

Unlike the aforementioned window types, where windowing is applied on the stream without any changes to the stream, partitioned windows split the data stream into different substreams based on the *key* of the partition. This enables techniques of *multiplexed processing* to be applied on the data stream, increasing the parallelization of stream processing engines.

Multiplexed processing Most data streams can be split up into multiple substreams consisting of records with the same *key* attributes. This is akin to the *groupBy* operations present in the traditional batch data processing where data records are grouped according to the values of a specific set of the records' *attributes*. For example, consider a stream containing Twitter tweets [1]. This stream of *tweets* can be partitioned into

[1]Twitter API: Twitter's tweet object model documentation.

substreams based on the user's id. Therefore, to support multiplex processing, any operator applied on the substreams must be local to that particular substream. The operators also need to keep the states for each substream independently.

Partitioned windows support *multiplexed processing*, since it consists of multiple subwindows; each corresponding to a particular substream identified by the *key* attributes of the records in the data stream. The subwindows are independent from each other and all computations on the records are in the local scope of the subwindow. The actual subwindow semantics can be of the aforementioned window types, *tumbling windows* or *sliding windows*. Therefore, the *eviction* and *trigger* policies of the subwindows are similar to those two types of windows.

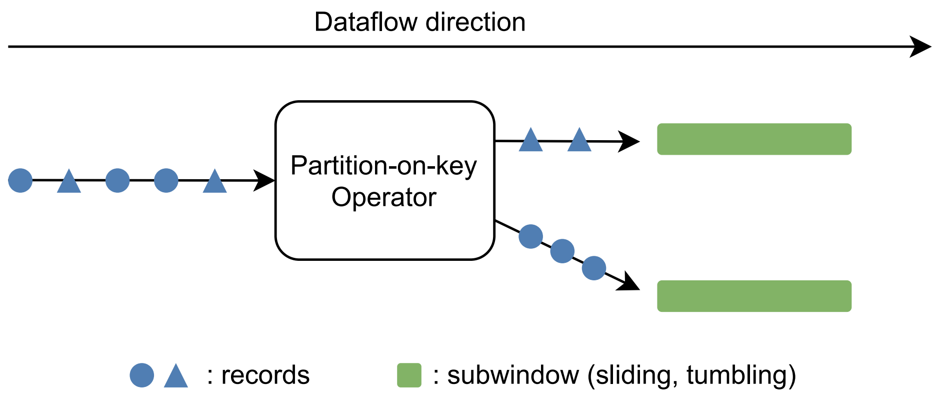


Figure 5.2: Partitioned window where the records are keyed by an attribute. In this case, records are keyed by their shape. After *keying* the records, circle records are sent to another subwindow than triangle records.

5.3 Windows processing

Streaming applications require some form of queries to be executed over the data stream. Joins and aggregations are some of the most common queries in stream processing which we will elaborate on in the upcoming sections.

5.3.1 Aggregation

Window aggregation is a class of operations where some form of computation is done on a group of records within a window. Operations like finding the *maximum value* or *average value* are some aggregations frequently done in data stream processing. However, aggregate operations can be expensive in some situations. For example, aggregation operations are expensive when there is an overlap of windows (sliding windows), since aggregations over the overlapping records will need to be recomputed for each overlapping windows. Given a sliding window of 1 minute *size* and 1 seconds *slide*, 60 windows will exist at any point in time and the 59 seconds worth of records aggregation will need to be recomputed whenever a window receives a *trigger* event for processing. This results in redundant computation.

Optimizing aggregation queries on windows has been extensively studied. Jonas et al [**scotty**, **jonas˙scotty**] have come up with a method to apply generic window *slicing* for optimizing aggregation operators in windows. The data stream is modelled as *slices*, where records are assigned to exactly one *slice* resulting in a $O(1)$ complexity for memory. The *slices* are created every time a window begins or ends. Partial aggregations are then applied on the records within the slices which will eventually be consolidated once a window ends for a final aggregation result eliminating redundant computations. According to the authors [**jonas˙scotty**], this maintains the high throughput processing capabilities of Apache Flink even if the number of overlapping concurrent windows increases.

5.3.2 Join

Join operations are different from aggregations. We do not need to wait for the window to be full, before processing the records inside the window. A *trigger* event could be fired at the very moment a new record arrives in the window. Furthermore, multiple statistical approaches could be applied for window optimization without disturbing the results of the join operations significantly. Several studies have been conducted to improve the join algorithms in windows [**vctw˙join**, **join˙tracking**, **grubjoin**, **approximate˙window˙sem**, **approx˙window**]. The approaches in [**grubjoin**, **approximate˙win**

approx˙window] are based on dropping some of the records to be joined through *load shedding*. Load shedding is a technique to drop the record if it fails to meet the conditions specified by a metric specific to the implementations. Approaches in [**grubjoin**, **approximate˙window˙sem**, **approx˙window**] assume that the stream is not out-of-order with centralized execution for windows processing. Additionally, histograms of the attributes of the records are used, either per window or per stream, to calculate the threshold for dropping records, inflicting an overhead on memory. This leads to higher latency and computation overhead when joining records from multiple streams. Furthermore, these approaches only deal with *join* operators and could not be applied to other types of stream processing operators such as *aggregations* and *reductions*.

VC-TWJoin [**vctw˙join**] approaches the problem differently by only considering the behaviour of the data stream. The approach adjusts the size of the window after every *eviction* trigger based on the stream rate. This simple approach of dynamically adjusting window sizes, enables load shedding without the overhead of keeping statistical data. Since the metric used by the approach does not use record attributes, it has no impact on calculations done by other operators such as *aggregation* and *reduction*. 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 threshold 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.

5.4 Summary

The aforementioned window types are fixed and not resilient to changing data stream characteristics over time. Once initialized, the window sizes are fixed throughout the runtime of the application. This results in inefficient window sizes if the data stream rate diverges greatly over time; a window size might become too small to process the incoming data. Thus, a dynamic window that adjust its size throughout the runtime of the application will adapt better to the changing characteristics of the data stream.

Chapter 6

Methodology

In this chapter, we introduce an approach to dynamic windowing in data stream processing with the following improvements to fixed size windows; (a) stream-characteristic-aware processing (b) adaptive window sizes, and (c) high throughput with low latency processing.

Existing fixed size windows are inflexible when the characteristics of a data stream, such as the stream rate, change over time. Either the window size is too small to produce any output or too large which leads to high memory usage when the stream rate is high. Furthermore, fixed size windows only process the records inside the window at fixed interval (usually equal to the size of the window). This leads to high latency proportional to the window size. As a consequence, it causes unnecessarily high latency in some operations, where records could be processed once they arrive inside the window. One example of such operation is the join operator, where the joined result can be emitted the moment a new record arrives inside the window. Solving these disadvantages leads us to our approach of a dynamic window, which can adjust its size during runtime, resulting in a low latency and high throughput solution.

This chapter will be outlined as follows. First, we analyse the possible sites across the different stages of RMLStreamer to implement our dynamic window. Next, we define which operator (join, aggregation, etc ...) to implement as the reference processing operator inside our dynamic window. Subsequently, we elaborate our dynamic window algorithm to allow dynamic adjustment of window sizes depending on the stream rate of the incoming records. Finally, we give a brief overview of our reference implementation.

6.1 Analysis for the implementation site

There are two possible sites to implement our dynamic window; *before* or *after* the mapping of non-RDF to RDF data. There are different trade-offs for choosing one over the other.

After the mapping process Implementing the windowing operations after the mapping process allows the possibility of applying graph processing operators in the window; the input data for the windows will be RDF triples from the mapping process.

However, it comes with a major disadvantage. All the attributes required in the operator will have to be mapped in the mapping stage to their corresponding triple statements. This could lead to an explosion in the number of triple statements being generated, increasing the network cost of sending the data to the windowing stage. High latency processing in the mapping stage will also occur if the amount of attributes required by the operator is sufficiently large enough.

Before the mapping process To mitigate the explosion of triple statements after the mapping process, we could implement the windowing right before the mapping process. A reduction in the amount of data needed to be processed in the mapping stage could be achieved by operators such as *filter* or *joins* in the windowing stage. Furthermore, direct access to the raw data allows more arbitrary transformations to be applied such as those specified in FnO [**fno˙ben**], without the loss of information due to mapping process.

If graph processing operations are required, we could delegate the processing computations to existing state-of-the-art RDF stream processing engines by attaching them at the output of RMLStreamer. In conclusion, it is ideal to implement the windowing before the mapping process.

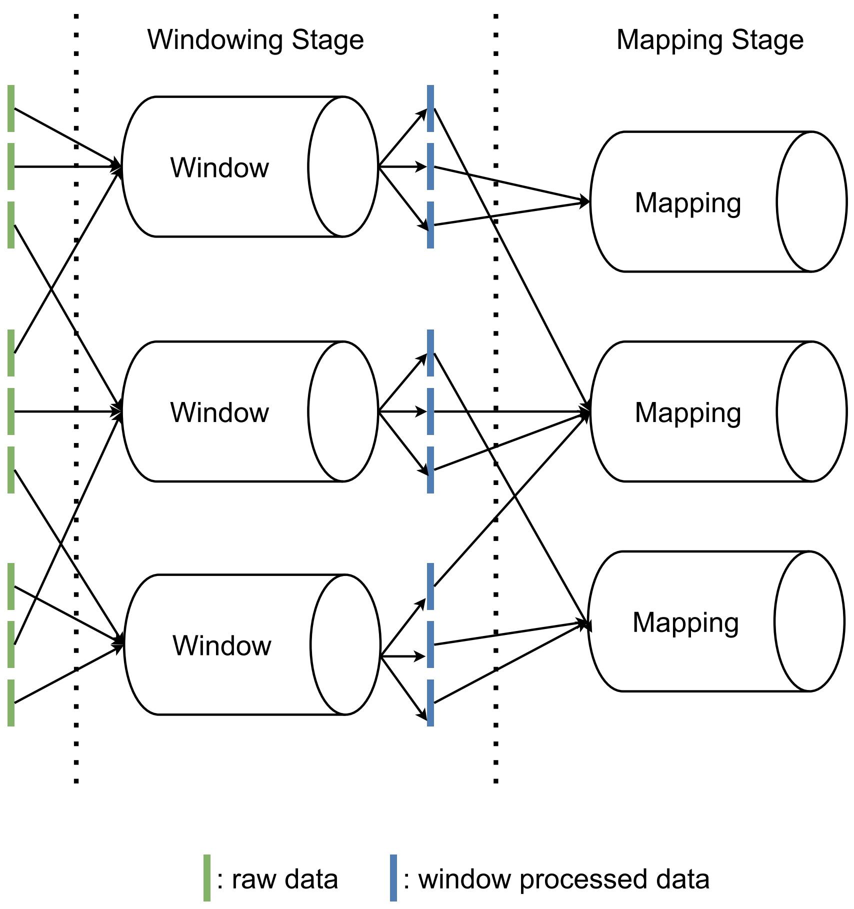


Figure 6.1: Implementation of windowing stage **before** the mapping stage. The blue blocks are chunks of processed records of an *operator* (join, aggregation, etc...) in the windows. The green blocks are raw data chunks from the ingestion stage (Figure 4.3).

6.2 Operator inside window

Which type of operation is suitable for our dynamic window? In the context of this work with RMLStreamer, we will have to consider optimizing the *join* operation.

RDF mapping engines focus on just mapping the input to RDF compliant formats without applying complex stream processing queries like *aggregations* and *reductions*. These stream processing queries, if required, could be delegated to RDF stream processing frameworks processing RDF data streams. Therefore, we focus on a reference implementation of dynamic windows with stream joins as part of the mapping process of heterogeneous data streams to RDF data streams.

6.3 Dynamic Window

A dynamic window is a type of partitioned window (Chapter 5.2.3). 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.

We shall first extend the notations in Chapter 5.1 to aid in the elaboration of our dynamic window. Finally, the full algorithm for dynamic windowing is described in detail in pseudo code.

Notations Similar to the notations in Chapter 5.1, we extend the notations used in [**generic˙wi**
$W^a = \{W^{a=v} | v \in \{r.a | r \text{ is a record in } W^a\}\}$ is a dynamic window, where a represents a *key* attribute. $|W^a|$ is the number of subwindows in W^a, and $W^{a=v}$ represents a subwindow associated with a *key* attribute value v. Each subwindow is defined as $W^{a=v} = \{r_i^{a=v} | i \in [0 \ldots |W^{a=v}| - 1]\}$, where $r_i^{a=v}$ are ordered from the newest to the oldest starting from $0 \ldots n$ with the oldest record being $r_n^{a=v}$ with $n = |W^{a=v}| - 1$. The cur-

rent record to be put into the subwindow $W^{a=v}$ is denoted as $r_c^{a=v}$. The **event** timestamp associated with the record $r_i^{a=v}$ is denoted as $\tau(r_i^{a=v})$.

The two streams to be joined are denoted as S_P and S_C, for streams representing the parent triple map and child triple map respectively [**rml**]. The number of records in the subwindow $W^{a=v}$ from S_P, and S_C are respectively denoted as $|S_P^{a=v}|$, and $|S_C^{a=v}|$. The list states, a data structure to hold the records from $S_P^{a=v}$ and $S_C^{a=v}$, are respectively denoted as $List_P^{a=v}$ and $List_C^{a=v}$. We also denote the pseudo maximum size of the list states $List_P^{a=v}$ and $List_C^{a=v}$ as $Size(List_P^{a=v})$ and $Size(List_C^{a=v})$ respectively. A record from $S_P^{a=v}$ or $S_C^{a=v}$ is denoted as $r_i^{a=v,P}$ or $r_i^{a=v,C}$ respectively.

We also define a threshold ϵ used to check if the window size needs to be adjusted based on the join cost m. Generally, if $\epsilon = 1.5$, it represents the threshold where the number of records inside a subwindow, $W^{a=v}$, is 1.5 times the size of $Size(List_P^{a=v}) + Size(List_C^{a=v})$.

6.3.1 Dynamic window join

The join algorithm used, is a variant of Symmetric Hash Join [**symmetric˙hash˙join**]. The subwindow themselves work like a hash table, containing records with a specific *key* — since they are *partitioned* (Chapter 5.2.3). The records are joined by probing the relevant stream and generating the joined result.

We will work through a simple example when a record r_c arrives inside a subwindow $W^{a=v}$. For ease of writing, we omit $a = v$ in our notations.

Example 6.1. Provided the incoming record is r_c^P, we first insert it into $List_P$. Subsequently, we probe for $\forall i, r_i^C \in List_C$, and join all r_i^C found, with r_c^P as tuples of (r_c^P, r_i^C) with $\forall i, r_i \in List_C$. Similar steps are executed for r_c^C.

6.3.2 Dynamic window sizing

For each subwindow $W^{a=v}$, the following configuration parameters are provided:

- Δn: the **initial** interval of time before the next eviction trigger (Chapter 5).
- ϵ_u: the upper limit for threshold ϵ.

- ϵ_l: the lower limit for threshold ϵ.
- U: the upper limit for the window size.
- L: the lower limit for the window size.

For ease of writing, we will omit $a = v$ to specify the *key* attribute associated with the subwindow. Therefore, the following events happen for every subwindow $W^{a=v}$ independently from each other.

Since we are implementing the join operator, the trigger event is fired when r_c arrives, and the join operation in Chapter 6.3.1 is executed.

The eviction trigger is fired every time when the current watermark (Section 3.3.3) $w \geq |W| + \Delta n$. At each eviction trigger we calculate the 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 sizing 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 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: $\Delta n, \epsilon_u, \epsilon_l, U, L, List_P, List_C, S_P, S_C$
   // Note: cost(..) is ratio for how "full" a list-state is
1  $cost(List_P) = |S_P|/Size(List_P)$   // Calculate the cost (ratio) of list-state P
2  $cost(List_C) = |S_C|/Size(List_C)$   // Calculate the cost (ratio) of list-state C
3  total cost $m = cost(List_P) + cost(List_C)$
4  if $m > \epsilon_u$ then
       // the cost is too high, shrink the window size and adjust list-states
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
       // the cost is too low, increase the window size and adjust list-states
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 $List_P$
```

6.4 Implementation

The dynamic window is implemented for RMLStreamer to extend its capabilities to do simple stream processing by joining multiple streams. RMLStreamer is implemented in Scala on top of the Apache Flink framework and with Flink's easy to access, low-level APIs, we could easily implement our custom dynamic window.

Furthermore, new *properties* and *classes* are defined to extend RML with support for window configurations. RML's extensibility allows us to define new RML vocabularies without changing the existing ones. The ontology definitions is rudimentary at this moment and it is used only to configure our window type. However, it could easily be extended to also include windo specific parameter configurations if required. An example of this RML mapping file for dynamic window can be found in Appendix A.1. For example, one could adjust the window type by changing the literal value for the property *rmls:windowType* with either *rmls:Tumbling* or *rmls:DynamicWindow*.

Chapter 7

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: *latency*, *throughput*, *memory usage*, and *completeness*. In this paper *Throughput* refers to the number of processed input records per second. *Latency, throughput,* and *memory usage* are measured following the recent benchmark studies by Van Dongen and Van den Poel(2020) [**evalution˙of˙spe**]. However, to evaluate for the *completeness* of the generated output, a reference output needs to be generated for comparison. Since a data stream is unbounded and cannot be processed completely to check for *completeness*, we use a bounded dataset of relatively large volume to measure *completeness*. To ensure consistency in the complexity of the multi-stream operator being used, we apply the join operator in the windows — a common operator used in data enrichment scenarios. The *key* attributes used to join are used as much as possible to reduce the output size and keep the completeness measurement as concise as possible. Using less *key* attributes would just lead to a significant increase in the number of redundant duplicate triples generated for completeness measure.

The following sections describe the workloads and the data used to evaluate the different metrics. Setups required to reproduce the evaluation are elaborated on in their respective sections.

7.1 Data

Assessment of our dynamic windowing scheme for multi-stream operator requires the data streams to have some common attributes to enrich the data. Furthermore, to mirror a real-world scenario, we employ data gathered from IoT sensors. 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 will be replayed by a Kafka publisher into two topics *ndwflow* and *ndwspeed*, for the number of cars and the average speed respectively. The json structure for the records from each stream are shown in Listing 7.1.

Listing 7.1: JSON data structure of the records.

```

//NDWSpeed topic's record structure
{
    "internalId": "lane1",
    "lat":51.41268,
    "long":5.42929,
    "speed":92,
    "accuracy":100,
    "timestamp":"2021-04-01 17:58:10",
    "num_lanes":1
}

// NDWFlow topic's record structure
{
    "internalId": "lane1",
    "lat":51.41268,
```

[1]NDW data site: http://opendata.ndw.nu/

```
    "long":5.42929,
    "flow":360,
    "period":60,
    "accuracy":100,
    "timestamp":"2021-04-01 17:52:22",
    "num_lanes":1
}
```

7.2 Evaluation setup

Docker[2] is used to package our applications in separate containers. The docker setup with the containers is illustrated in Figure 7.1. The docker containers are 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**]. Message replication is disabled to prevent heavy load on the Kafka broker.

In order to measure the metrics without significant influence from Apache Flink's configuration, we kept the default values as much as possible. Table 7.1 describes the configuration for the Apache Flink to run our RMLStreamer with window implementations. We disabled checkpointing since we are only evaluating the performance of the window on processing the stream, not fault tolerance. Thus, the default InMemoryStateBackend is enough for this evaluation.

Watermarks are emitted every 50ms and object reuse is enabled to mitigate the performance impact incurred by redundant object creations just like in [**evalution˙of˙spe**].

7.2.1 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

[2]Docker: https://www.docker.com

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 accuracy of the measurements depends on the default internal metric update interval by Flink. The aforementioned metrics are averaged across the parallelized window operators since Flink spreads the operations across the available task slots of Taskmanager. Intersection over union (IOU) is used as the metric to measure the completeness of the joined result generated by the windows.

For *CPU usage*, *throughput*, and *memory*, we drop the first 10 min of measurements because of the overhead of the warm up period of the JVM for RMLStreamer. However, it might be interesting to observe the stabilization of latency pattern by the different windows implementation. Therefore, latency measurement is done throughout the lifetime of the evaluation workload.

CPU usage

For CPU usage, we measure Taskmanager's CPU usage, since it is the one responsible for executing the RMLStreamer code. The CPU usage is monitored at the Taskmanager level.

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.

Memory usage

Due to limitation in granularity of measurement in Flink, JVM heap memory used by 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 since they are stateless operators. Therefore, measurement of JVM heap memory is a good enough estimate for memory usage of window operators for storing the state.

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. This enables accurate measurement of the amount of time spent in the window since the measurement is done on the same machine with the same clock. However, there is a subtle difference in the *latency* calculation between the Tumbling and Dynamic window.

This is due to the difference in the *trigger* policy between the two windows. Dynamic window fires the *trigger* event immediately, once it receives a record, to process it and emit the joined results as a batch. This means that we need to measure the **minimum** latency amongst the records being emitted in the joined records batch. Tumbling window fires the *trigger* event only when it has to *evict* the window. Therefore, there is no joined results generated before the eviction of the records inside the window; we have to measure the **average** latency of the records in the joined records batch.

Completeness measurement

Evaluating completeness requires a dataset which could be mapped completely by a static mapping engine. For our evaluation, this dataset will be the same set of data used to evaluate the window implementation in a streaming environment. These streamed data are stored in the Kafka topics as long as the Kafka broker is alive. To generate *bounded* input data for the static mapping engine, we could 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. This set of output triples will be generated according to the RML mappings defined for bounded data (Appendix A.3).

Streaming data is also usually ordered according to the event time (Chapter 3) and the same entity could appear in the stream at a different timestamp. This can lead to duplicates in the generated output by the bounded data processing. We need to preserve these duplicates in the output since they are valid records produced by the data sources and duplicates are expected in streaming data.

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.

7.3 Window Configurations

The windows are configured according to Table 7.2. Dynamic window is configured with the same initial window size as the Tumbling window to ensure that Dynamic window is not processing too little or too much records in the first few minutes.

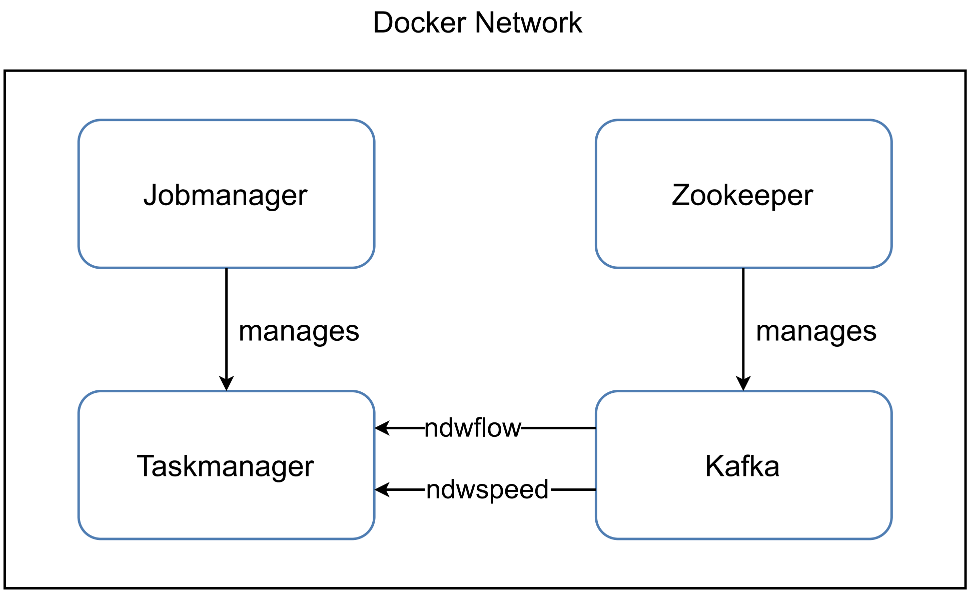


Figure 7.1: Setup of application containers inside the same docker network. The kafka topics *ndwflow* and *ndwspeed* are consumed by the RMLStreamer job inside the Taskmanager. **GH**: Maybe change "Jobmanager" to "Flink JobManager" and "Taskmanager" to "Flink TaskManager running RMLStreamer job". Also, I don't think Zookeeper is very relevant, so maybe leave it out of the picture.

Parameter	**Default**	**Value**
JobManager count	/	1
TaskManager count	/	1
JobManager CPU cores	/	2
TaskManager CPU cores	/	5
TaskManager heap / memory	512 MB / 1.7 GB	512 MB / 1.7 GB
Number of task slots	1	4
Default parallelism	1	4
StateBackend	InMemory	InMemory
Checkpoint Interval	None	None
Time characteristics	processing time	event time
Watermark interval	/	50ms
Object reuse	disabled	enabled

Table 7.1: Apache Flink's configuration.

Parameter	Value
Window size	2s (Event time)

(a) Tumbling Window's configuration

Parameter	Value
ϵ_l	1.2
ϵ_u	0.8
Initial window size	2s (Event time)
Upper window size limit	5s (Event time)
Lower window size limit	50ms (Event time)

(b) Dynamic Window's configuration

Table 7.2: Configuration of the windows used for evaluation.

7.4 Evaluation pipeline

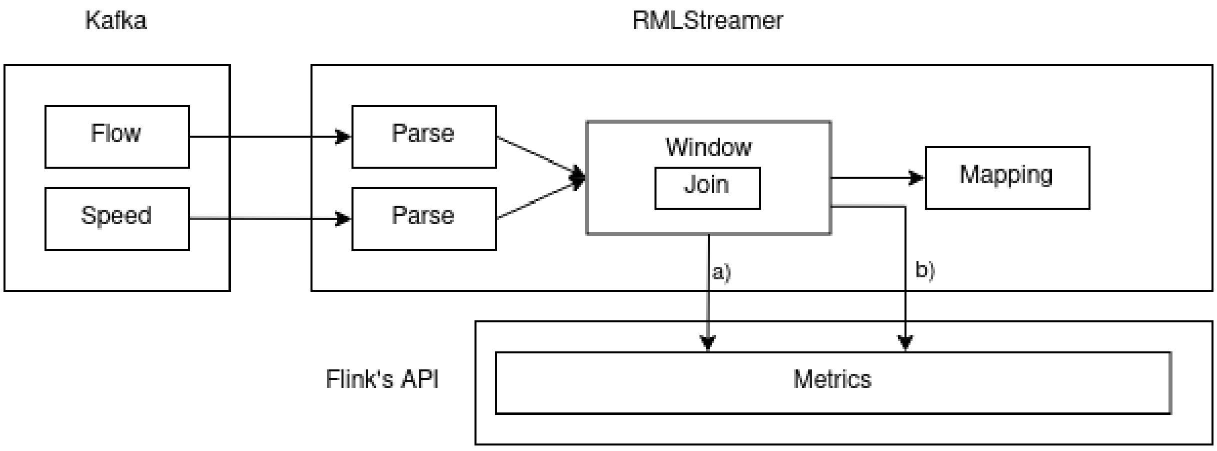


Figure 7.2: Evaluation flow based on [**evalution˙of˙spe**] with some changes to only take metrics at the "Join" stage. a) *latency, memory*, and *cpu usage* of evaluating the join operator in windows. b) *throughput* of the generated output triple.

Similar to the workflow process in [**evalution˙of˙spe**, **benchmark˙sce**], we set up our evaluation pipeline as illustrated in Figure 7.2. Setting it up according to the illustrated pipeline allows us to have fine control over the workload scenarios and to narrow the scope of the metrics to the windows only.

7.5 Workload scenarios

Data streams have characteristics which determine the performance of stream processing frameworks and its operators (Chapter 3). Therefore, there is a need to evaluate our dynamic window implementation under different workload scenarios. These workloads are similar to the ones used in [**evalution˙of˙spe**] where relevant.

7.5.1 Workload for latency measurement

As described in the paper by Van Dongen and Van den Poel [**evalution˙of˙spe**], *throughput* and *latency* can affect each other if the workload is improperly configured. Therefore, 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.

7.5.2 Workload with periodic burst

Fluctuation in the streaming sources are normal with unstable network connections. The windowing operator must therefore cope with sudden changes in the fluctuation of the velocity of the data stream. This workload attempts to emulate the scenario where multiple IoT devices send data in bursts with a periodic time interval. 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.

7.5.3 Workload for completeness

The implementation of the dynamic window should also ensure that the generated output is as *complete* as possible. Depending on the type of window being used, *completeness* of the output will be affected by the window size. If the window size is not sufficiently large enough to handle the input velocity, the next incoming record will be processed in the new window instead of the old window where it is more relevant. Thus, we will use two stream rates, constant and periodic, to measure the *completeness* of the results generated by the different windows. The stream rates for constant and periodic are the same as their workloads previously mentioned; 400 messages per second for *constant rate* and *periodic burst* of 38 000 messages at every 10th second with 400 messages per second between each burst.

As described in Section 7.2.1, the input data for this workload will be derived from the respective workloads with constant and periodic stream rate. We will write out the input data to a file, from the relevant Kafka topics *ndwflow* and *ndwspeed*, after each evaluation run of the two windows implementation.

Chapter 8

Results and Discussion

In this chapter, the results gathered from our evaluation of the dynamic window against the tumbling window will be discussed by the order of the workloads. The workload evaluations are run multiple times for consistency of results. We will first discuss the workload for latency measurement, followed by periodic workload and finally the completeness measurement. For each workload we elaborate on the results and discuss in detail the shortcomings of the evaluation methods and the possible outlier situations for which the evaluation result would not agree with.

8.1 Workload for latency measurement

This workload is run with a constant low stream rate **GH**: Perhaps repeat *how* low (400 msg/s?) because that's specified in the previous chapter... to ensure that the RML-Streamer is not overloaded for a more accurate latency measurement. We also measured CPU, throughput, and relative memory usage to determine the improvement brought by Dynamic window in an ideal streaming environment.

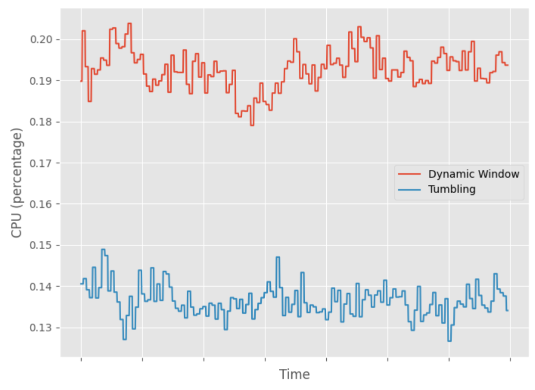
For latency, Tumbling window has a median value of 1915ms, with latency ranging from 1081ms to 2624ms. This is as expected since the window is measured using the *average* latency of all records used in the joined result at every *trigger* event, which is fired every 2 seconds (the size of tumbling window). In contrast, Dynamic window has sub second latency with median 57ms and ranging from 39ms to 120ms. Clearly, our improvement to fire the *trigger* event whenever a new record arrives inside the subwindow, allows Dynamic window to achieve sub second latency.

Just like latency, the throughput difference between the two windows is also significant. Dynamic window has a steady throughput of around 17200 records per second whereas Tumbling window fluctuates around 12500 and 12800 records per second before stabilizing at 12750 records per second. This is expected as Dynamic window eventually stabilizes to a window size with the capability of processing more records than Tumbling window. 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, even if it means that there might be more eligible records to be joined with the records in the soon to be evicted window, leading to lower throughput.

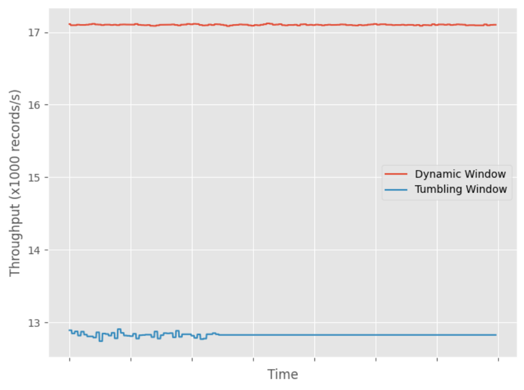
Relative memory usage of Dynamic window compared to Tumbling window is similar over the lifetime of the evaluation run (Figure 8.1e). Dynamic window causes infrequent *spikes* in memory usage of over 100 MB more than Tumbling window at a certain point in the lifetime of evaluation. This can be attributed to the subwindows of Dynamic window growing larger than Tumbling window due to not enough records of the same *key* attributes arriving inside the subwindows. However, Dynamic window stabilizes to a more optimal window size, where it uses less memory than Tumbling windows, over longer stretches of the evaluation run. 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. However, this increase in CPU usage can also be attributed to the increase in throughput, where the RMLStreamer has more joined results to process and map to RDF data. **GH**: It's most probably both. I would dare to put it like this: "CPU usage for Dynamic window is approximately 7% higher than for Tumbling window since it requires extra calculations on the metrics and it increases the throughput causing the RMLStreamer to process more joined records."

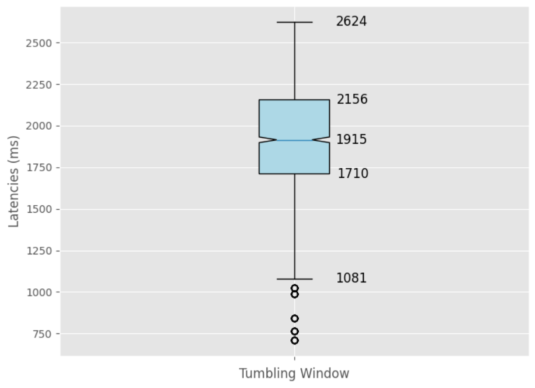
GH: Figure 8.1: can you add units to the time axes? Then people looking at the



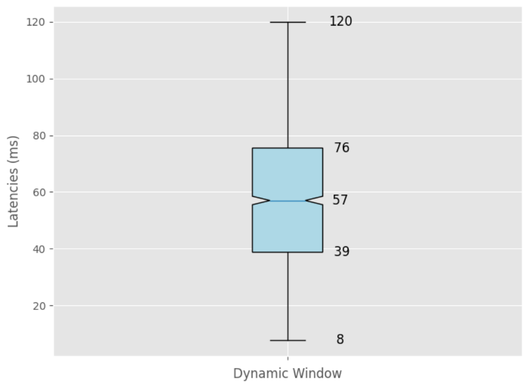
(a) CPU usage



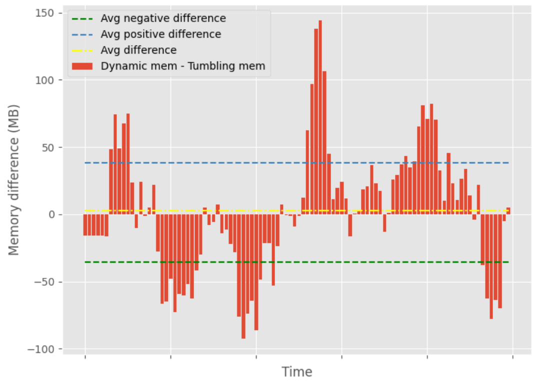
(b) Throughput of joined records



(c) Tumbling latency



(d) Dynamic latency



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

Figure 8.1: Metrics measurements for latency workload.

charts have an idea how long the evaluation ran. Figure (e): somewhere make clear in the legenda or at least the caption that the ”difference” means ”Dynamic - Tumbling”. Now it is stated for the red bars only, but it also counts for everything else on that chart.

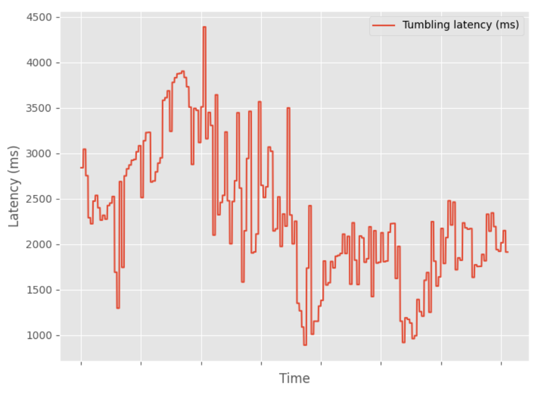
8.2 Workload for periodic burst

Dynamic window still handles the periodic burst of data with lower latency than Tumbling window with latency in the range from 8ms to 1669ms compared to Tumbling window's range from 891ms to 3904ms. However, there is a temporary increase in latency at the beginning (Figure 8.2b), 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 8.3d). However, Dynamic window eventually manages to shorten the subwindow sizes for adaptation to the periodic burst of data. The shorter subwindow sizes lower the latency until it is comparable to the one achieved under the constant stream rate from the previous workload.

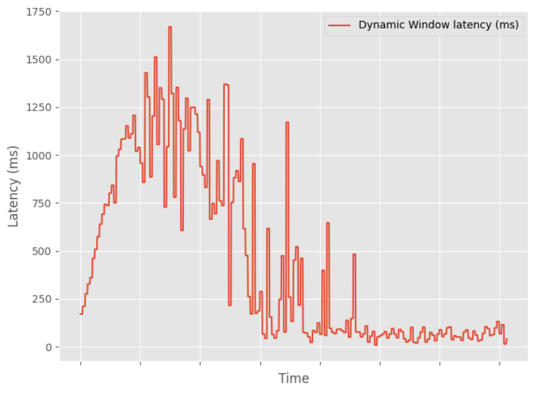
As expected, the throughput of both window mechanisms is higher for the periodic burst stream rates compared to the throughput for the lower constant stream rate for latency measurement. Moreover, we observe an even bigger difference in the throughput between both window mechanisms of about 7000 records per second. 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. In contrast to the separate evaluation of stages in [**evalution˙of˙spe**], we ran our evaluation as part of the whole RMLStreamer pipeline, from the ingestion of data and window joins until the generation of mapped RDF data. This causes a slight back pressure leading to a high and flat throughput measurement.

CPU usage difference of both window mechanisms is similar to that of the workload for latency measurement with relative increase in the usage to account for the processing of periodic burst of data. Dynamic window uses more CPU resources for calculation of metrics and dynamic adjustment of subwindow sizes.

Surprisingly, compared with the memory usage under the workload for latency measurement, Dynamic window uses about 10 MB less memory on average than Tum-



(a) Tumbling latency

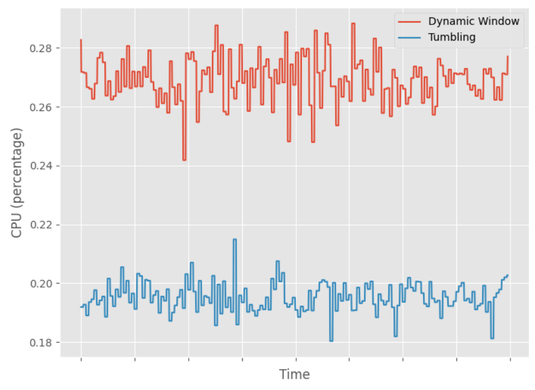


(b) Dynamic latency

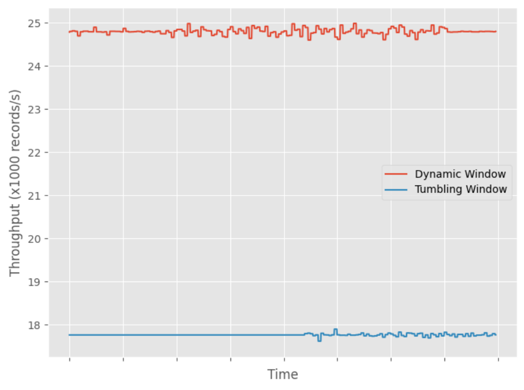
Figure 8.2: Latency measurement of periodic workload over the lifetime of evaluation

bling window during the lifetime of the evaluation (Figure 8.3e). The initial memory usage for Dynamic window about 100 MB higer than for Tumbling window due to the initial long window growth caused by the low stream rate. However, once the dynamic adjustment of subwindow kicks in, it reduces the memory usage even when there is a burst of data. The subwindow size is adjusted to be small so that records are frequently evicted.

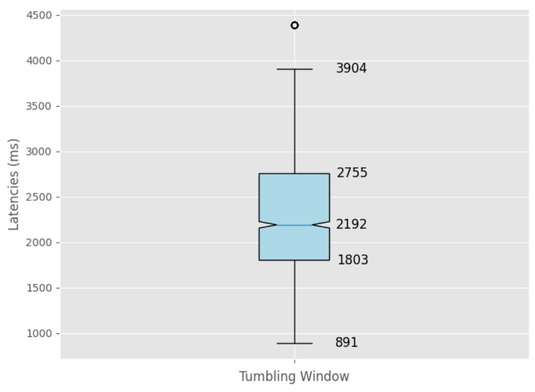
GH: Same comments for figures 8.2 and 8.3 as for 8.1



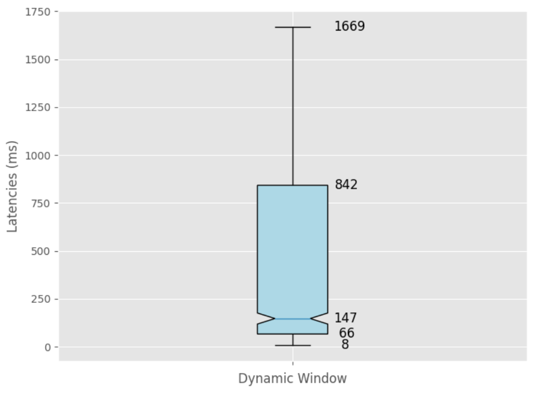
(a) CPU usage



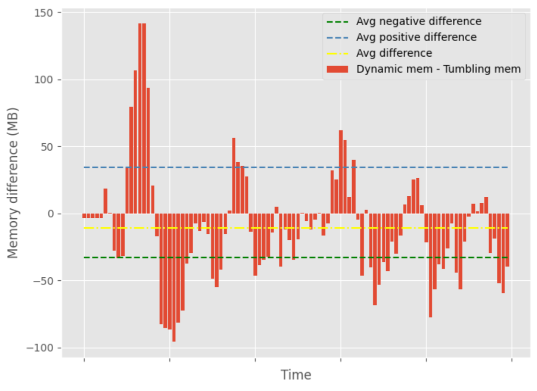
(b) Throughput of joined records



(c) Tumbling latency distribution



(d) Dynamic latency distribution



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

Figure 8.3: Metrics measurements for periodic workload.

8.3 Workload for completeness measure

From the results in Table 8.1, and Table 8.2, 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* set of output. In contrast, Tumbling window scores only **0.749**, leading to a conclusion that a window size of 2s is not enough to process the low stream rate of the evaluation data.

Similary 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 could be explained by its ability to adapt the window sizes according to the changing stream rate to hold enough records for maximal joined output generation; moreso than Tumbling window.

GH: Table 8.1 and 8.2: maybe it would make sense to replace "Common triples" with "Missing triples"? It highlights the difference...

Stream rate	Generated (triples)	Expected (triples)	Common (triples)	**IOU score**
Constant rate	30,771,450	30,771,450	30,771,450	**1**
Periodic burst	31,753,420	32,319,110	31,753,420	**0.982**

Table 8.1: Dynamic window's completeness measurement. The *Expected (triples)* are the number of triples generated by the bounded data processing RMLStreamer.

Stream rate	Generated (triples)	Expected (triples)	Common (triples)	**IOU score**
Constant rate	23,059,350	30,771,450	23,059,350	**0.749**
Periodic burst	24,412,150	31,287,300	24,412,150	**0.780**

Table 8.2: Tumbling window's completeness measurement. The *Expected (triples)* are the number of triples generated by the bounded data processing RMLStreamer.

8.4 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. This validates three of our hypotheses that Dynamic window would outperform windows of fixed sizes in those three areas.

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. For example, by counting the number of records that effectively reside in the windows at any moment. This approach is currently limited since Flink does not expose API to count the number elements residing in its default implementation of Tumbling window.

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, including the stage where the joined results are mapped to RDF data. 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.

Chapter 9

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. Our approach addresses the shortcomings of the existing state-of-the-art dynamic windowing approaches for join operator. 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 under different stream characteristics.

The results gathered from our evaluation concludes 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 in terms of the evaluation setups and the dynamic algorithm. 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 use case instead of in the context of RDF mapping engines. Improvements on our dynamic approach could be achieved by allowing user to define other statistical approach to better calculate the threshold for adapting the window sizes. For example, one could provide RMLStreamer with an external function to calculate the metrics in Dynamic window similar to the approach in FnO [**fno˙ben**]. Furthermore, utilizing mapping file to configure our window might also allow further optimization in the mapping file for a more efficient mapping job as proposed in FunMap [**funmap**].

Appendix A

RML mapping specifications

A.1 Dynamic window join's RML mapping file

```
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix ex: <http://example.com/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rml: <http://semweb.mmlab.be/ns/rml#> .
@prefix rmls: <http://semweb.mmlab.be/ns/rmls#> .
@prefix ql: <http://semweb.mmlab.be/ns/ql#> .
@prefix activity: <http://example.com/activity/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix schema: <https://schema.org/>.
@base <http://example.com/base/> .

_:kafka_source_ndwSpeed a rmls:KafkaStream ;
              rmls:broker "192.168.0.237:9092";
              rmls:groupId "1";
              rmls:topic "ndwspeed".

_:kafka_source_ndwFlow a rmls:KafkaStream ;
             rmls:broker "192.168.0.237:9092";
             rmls:groupId "1";
             rmls:topic "ndwflow".

```

```


<JoinConfigMap> a rmls:JoinConfigMap;
            rmls:joinType rmls:DynamicWindowJoin.


<NDWSpeedMap>
   a rr:TriplesMap;

   rml:logicalSource [
      rml:source _:kafka_source_ndwSpeed;
      rml:referenceFormulation ql:JSONPath;
      rml:iterator "$"
   ];

   rr:subjectMap [
      rr:template "http://example.com/resource/{internalId}?lat={lat}&long={
           long}&speed={speed}&accuracy={accuracy}&timestamp={timestamp}"
   ];


   rr:predicateObjectMap [
      rr:predicate <http://example.com/ontology/laneFlow> ;
      rr:objectMap [
         rr:parentTriplesMap <NDWFlowMap>;
         rmls:joinConfig <JoinConfigMap>;
         rmls:windowType   rmls:DynamicWindow;
         rr:joinCondition [
            rr:child "internalId,lat,long,timestamp" ;
            rr:parent "internalId,lat,long,timestamp" ;
         ]
      ]
   ] .

<NDWFlowMap>
   a rr:TriplesMap;
   rml:logicalSource [
    rml:source _:kafka_source_ndwFlow;
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$"
  ];



  rr:subjectMap [
    rr:template "http://example.com/resource/{internalId}?lat={lat}&long={
        long}&flow={flow}&period={period}&accuracy={accuracy}&timestamp={
        timestamp}"
  ].
```

A.2 Tumbling window join's RML mapping file

```
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix ex: <http://example.com/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rml: <http://semweb.mmlab.be/ns/rml#> .
@prefix rmls: <http://semweb.mmlab.be/ns/rmls#> .
@prefix ql: <http://semweb.mmlab.be/ns/ql#> .
@prefix activity: <http://example.com/activity/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix schema: <https://schema.org/>.
@base <http://example.com/base/> .

_:kafka_source_ndwSpeed a rmls:KafkaStream ;
              rmls:broker "192.168.0.237:9092";
              rmls:groupId "1";
              rmls:topic "ndwspeed".

_:kafka_source_ndwFlow a rmls:KafkaStream ;
             rmls:broker "192.168.0.237:9092";
             rmls:groupId "1";
             rmls:topic "ndwflow".

```

```



<JoinConfigMap> a rmls:JoinConfigMap;
            rmls:joinType rmls:TumblingWindowJoin.


<NDWSpeedMap>
  a rr:TriplesMap;

  rml:logicalSource [
    rml:source _:kafka_source_ndwSpeed;
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$"
  ];

  rr:subjectMap [
    rr:template "http://example.com/resource/{internalId}?lat={lat}&long={
        long}&speed={speed}&accuracy={accuracy}&timestamp={timestamp}"
  ];

  rr:predicateObjectMap [
    rr:predicate <http://example.com/ontology/laneFlow> ;
    rr:objectMap [
      rr:parentTriplesMap <NDWFlowMap>;
      rmls:joinConfig <JoinConfigMap>;
      rmls:windowType  rmls:Tumbling;
      rr:joinCondition [
        rr:child "internalId,lat,long,timestamp" ;
        rr:parent "internalId,lat,long,timestamp" ;
      ]
    ]
  ] .

<NDWFlowMap>
  a rr:TriplesMap;
  rml:logicalSource [
    rml:source _:kafka_source_ndwFlow;
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$"
  ];

  rr:subjectMap [
    rr:template "http://example.com/resource/{internalId}?lat={lat}&long={
        long}&flow={flow}&period={period}&accuracy={accuracy}&timestamp={
        timestamp}"
  ].
```

A.3 Bounded data join RML mapping file

```
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix ex: <http://example.com/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rml: <http://semweb.mmlab.be/ns/rml#> .
@prefix rmls: <http://semweb.mmlab.be/ns/rmls#> .
@prefix ql: <http://semweb.mmlab.be/ns/ql#> .
@prefix activity: <http://example.com/activity/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix schema: <https://schema.org/>.
@base <http://example.com/base/> .


<NDWSpeedMap>
  a rr:TriplesMap;

  rml:logicalSource [
    rml:source "ndwspeed.json" ;
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.*"
  ];

  rr:subjectMap [
    rr:template "http://example.com/resource/{internalId}?lat={lat}&long={
        long}&speed={speed}&accuracy={accuracy}&timestamp={timestamp}"
  ];


  rr:predicateObjectMap [
    rr:predicate <http://example.com/ontology/laneFlow> ;
    rr:objectMap [
      rr:parentTriplesMap <NDWFlowMap>;
      rr:joinCondition [
        rr:child "internalId,lat,long,timestamp" ;
        rr:parent "internalId,lat,long,timestamp" ;
      ]
    ]
  ] .

<NDWFlowMap>
  a rr:TriplesMap;
  rml:logicalSource [
    rml:source "ndwflow.json";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.*"
  ];



  rr:subjectMap [
    rr:template "http://example.com/resource/{internalId}?lat={lat}&long={
        long}&flow={flow}&period={period}&accuracy={accuracy}&timestamp={
        timestamp}"
  ].
```