Dynamically configured stream processing in Apache Flink - The use case of custom processing rules management and application

1st Stefan Andonov
Faculty of computer science and engineering
Ss. Cyril and Methodius University
Skopje, North Macedonia
stefan.andonov@finki.ukim.mk

2nd Gjorgji Madjarov
Elevate Global LLC
gjorgji@elevate-global.biz
Faculty of computer science and engineering
Ss. Cyril and Methodius University
Skopje, North Macedonia
gjorgji.madjarov@finki.ukim.mk

***Abstract*—This paper presents advanced Apache Flink application patterns for low latency distributed data stream processing. These patterns extend the concept of statically defined data flows and allow Flink jobs to dynamically change at runtime, without downtime. The introduced patterns allow dynamic configuration and change of the application logic and processing steps for implementing complex business scenarios. Using a real-life use case scenario and dynamic processing rules configuration, we present the patterns for dynamic data partitioning, dynamic window configuration, and dynamic data aggregation. They are implemented using the high-level APIs for windowing and aggregation and the low-level process function API. The patterns are implemented using the concept of control/configuration stream and broadcast stream and the carrier of the control information, control message. The real-life use case scenario tackles the problem of processing and analyzing air pollution data obtained from different sensors located in many different locations, as well as visualization of the data in third-party software.**

***Index Terms*—Apache Flink, stream processing, big data, stream analytics, distributed processing, visualization, software**

I. Introduction

A. Stream processing

Stream processing is a computer programming paradigm where the processing of the data is being done in motion, or in other words, the computing on the data is performed directly as it is produced or received. The stream processing is necessary because the majority of data are born as continuous streams: sensor events, user activity on a website, financial trades, etc.

The goal of stream processing is to create a system (so-called *stream processor*) that will react on a continuous data stream of events in the same moment when the system receives them. Usually, the processing logic of these systems includes triggering some actions, updating aggregations components, or remembering the data event in some window frame for future usage.

On the other hand, *batch processing* is a programming paradigm where a collection of multiple data events (batch of data), which was previously stored, is being processed in a particular moment. The batches of data could contain millions of records and therefore the querying of the data is a process that could take a lot of time and resources. The focus of this processing paradigm is the high throughput of processing the data which usually comes with the price of high latency and not even near to real-time results generation.

When it comes to applications with use-cases that are time-sensitive and demand a latency bounded or near real-time feedback, the stream processing paradigm is more preferable because it allows the systems to react to each event with low latency, collect small groups of events, process them, and generate a result. Examples of these use-cases are the IoT, smart ecosystems, mobile operators systems where an enormous amount of sensors are generating data every moment. This data carries information that needs to be processed, analyzed and accordingly to this analysis, corresponding feedback should be given.

For processing large volumes of generated streaming data, there are several proposed frameworks so far: Apache Flink [1,2,3], Apache Spark [5], Apache Storm [6], etc. Apache Flink [1,2,3] is an open-sourced framework for processing data in both streaming mode and batch mode. It was originally created as part of the Stratosphere [4] research project. Flink provides fault-tolerant and large-scaled computations. Additionally, Apache Flink provides stateful stream processing with easy high-level and user-friendly state management. However, Apache Flink only supports statically configuration of the operators through hard-coding into the source code or a configuration within the application initialization via setting program arguments when running the Flink job.

In this paper we propose software patterns for dynamic data partitioning, dynamic window configuration, and dynamic data aggregation, which overcomes the limitations that currently exist in Apache Flink, allowing more freedom and options when building stateful streaming applications.

B. Use case scenario

To explain the concept of dynamically configured stream processing we have developed a simple application with the following use case scenario:

We are gathering a huge amount of sensor data from more than 40 sensors that are set in different locations in Skopje, North Macedonia and all of them are collecting different types of measurements for the air pollution in the city. Each of the sensors records the measurements at different time intervals and sends them to a designated topic on the distributed streaming platform Apache Kafka, i.e the sensors are data stream producers. The sensor data is structured and it contains information about:

- the sensor ID (string)
- the location of the sensor (longitude and latitude concatenated in a string)
- timestamp (date and time of the measurement)
- type (string that represents the type of the measured value PM_{25}, PM_{10}, SO_2, etc)
- value (number, the measured value from the sensor)

Our application serves as a stream processor that firstly consumes some of the historic data from the sensors' topics, and then it continues with consuming the real-time sensors measurements in the same moment when they occurred. This is possible due to the offset configuration of the Kafka consumer.

The goal of the application is to enable the end-users of the application to receive analysis and reports (statistics on the measured values filtered by user-created rules which are collected in different processing windows) on the data based on their requirements (rules).

C. Limitations from statically configuration data processing in Apache Flink

Apache Flink does not directly support dynamically configuration of the processors and their processing logic. It supports statically configuration through hard-coding or via program arguments when running the Flink job. This means that we are not able to apply different window and aggregation strategies to the different data streams that are processed in our system. Even more, the data streams are processed according to the same pre-defined windowing and aggregation strategies with no possibility to change them without downtime (Fig. 1).

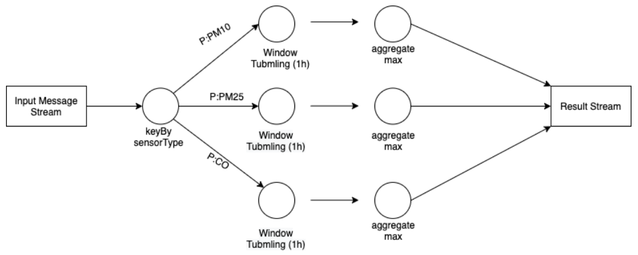

Fig. 1. Illustration of the static configuration of the operators graph imposed by Apache Flink defualt API

To overcome these limitations, we suggest using two data sources:

1) high throughput stream of the **input (sensor) messages** consumed from a data source
2) low throughput stream of **control messages** consumed from a different data source, that represents the stream of user-defined rules and the windowing and aggregation processing strategies.

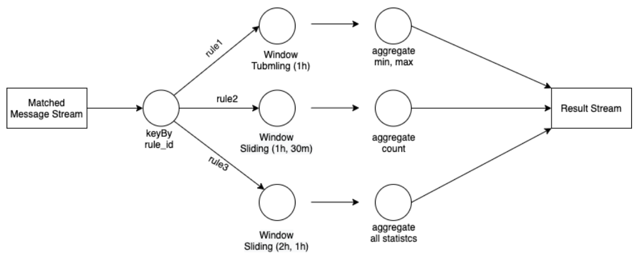

Fig. 2. Illustration of the goal dynamical configuration of the operators graph

The control messages, together with the high and low-level APIs of Apache Flink, will be used in the setting of the dynamical configuration of the tasks and operators in the stream processing as shown in Fig. 2.

II. Operators in Apache Flink

As shown in Figures 1 and 2, in our solution we are using the following Flink operators to achieve the goal of the application:

1) **keyBy (data partitioning)**: This is one of the most important operators in the Apache Flink framework. It enables both physical and logical partitioning of the data stream by a specified key. When using the keyBy operator, the number of created logical partitions of the data stream is the number of distinct keys. Keying a stream shuffles the data records and the data records with the same key are assigned to the same logical partition. This guarantees that all data records with the same key will be processed by the next operator on the same physical node.
2) **window (data windowing)**: This operator performs a split of the data stream into a smaller collection of data of finite size. Even though Flink supports a total of 4 types of windows, in this paper the main focus will be on the **timed windows**. There are two main types of time windows: *tumbling* and *sliding*. Windowing can be applied to both keyed and non-keyed data streams. By default, this is the most static operator and therefore it was the most complex one for dynamical configuration.
3) **aggregate (data aggregation)**: The main goal of data windowing is to perform some kind of aggregation of the data in the windows. Flink has build-in operators for specific aggregation of the data like sum, maximum, minimum, and reducing, but sometimes we need more than one type of aggregation of the data or we need some aggregation of the data that is not build-in.

Even though it is not shown in the figures above, we will be leveraging stateful computations in Flink. In a **state** of a Flink application, we can keep any kind of information like a collection of previous events, some aggregated information about those events, machine learning models parameters, etc. There are numbers of state types in Apache Flink, that can be used for keyed and non-keyed streams. [3]

III. Dynamical configuration of the operators

To dynamically configure the operators, two different approaches are used:

- The high-level API of Apache Flink with a combination of methods and interfaces that are available in the API
- Definition of new operators through the low-level Flink API (the process functions), that allows more freedom and options [7, 8].

The patterns that we will define extend the concept of statically defined data flows and allow Flink jobs to dynamically change at run-time, without downtime. The introduced patterns allow dynamic configuration and change of the application logic and processing steps for implementing complex business scenarios.

A. Dynamic State Configuration

The control stream is introduced to bring the information for the changes of the processing logic that needs to be made on the data records, while the state should store and keep that information. To achieve this, the control stream is represented as a broadcast stream and it is connected to the stream of data records. That means that the stream of control messages is broadcasted to all physical nodes and a replica of that stream is connected to each parallel stream of data records. Every control message that arrives is stored into a broadcast state represented as a *MapState*. When a new control message is created, it is distributed and saved (updated) in all parallel instances of the operator using *processBroadcastElement*. When a new data record arrives, it is processed according to the processing logic specified in the broadcast state by the control message. Instead of hard-coded processing logic, the processing logic is performed by the dynamically configured broadcast state.

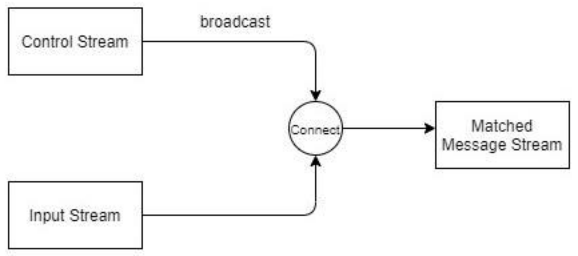

Fig. 3. Visualization of the connection of the control and input data streams

In our application, as shown in Fig. 3, the control stream will be connected as a broadcast stream to the input data stream (the sensor data). Every control message that arrives will be stored into a **broadcast state** from type *MapState*, where the key is the ID of the user who created that particular control message, and the value will be the corresponding control message. Having this broadcast state will help us in a broadcast process function, to check the match between every input message and every user-defined rule that is stored in the broadcast map state. For each pair of an input message (sensor data) and user-defined rule from the control messages, an object of class **MatchedMessage** will be created. The result from this connection will be a data stream of MatchedMessage objects where each matched message object contains:

- this input message that was matched
- the rule that was matched (extracted from the control message)
- the ID of the user who created the control message (extracted from the control message)

B. Dynamic Data Partitioning

The keyBy operator expects only one argument, a *KeySelector* object. The *KeySelector* is a generic interface with one function that extracts the key from a given object from any class. By default, a hard-coded KeySelector is used, which extracts specific fields from the data record. This means that, if the program logic should be changed, or the data records should be partitioned differently, the Flink job should be stopped, the changes in the program to be made offline and the code to be resubmitted for execution. However, to overcome these limitations and to support the desired flexibility, we have to extract the partitioning keys in a more dynamic fashion based on some specifications. We propose this to be done by using dynamic state configuration and the concept of control stream explained in the previous section.

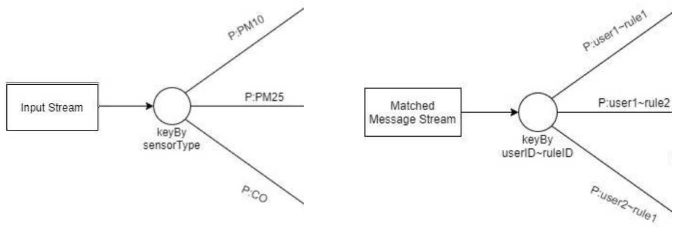

Fig. 4. Visualization of static (left) and dynamic (right) configuration of the keyBy operator

In our application, on the data stream of the matched messages, we will perform partitioning of the data by a key that represents the concatenation of the user ID and rule ID from the rule of the matched message. This was achieved with the implementation of our KeySelector and it represents a dynamical configuration with the usage of the high-level API. This kind of partitioning of the data will guarantee that all the sensor records (input messages) which satisfy the rule created by the user will be processed on a separate partition and the messages won't be mixed with the messages that don't satisfy the specific rule.

C. Dynamic Data Windowing

From all the operators that we are trying to configure dynamically, the window operator by its design is the most static one in the Apache Flink framework. This operator requires one of the four built-in window assigners. Each of those window assigners represents an assigner for one of the four types of windowing strategies that Apache Flink supports: tumbling, sliding, session, or global. As previously mentioned our focus will be on the timed windows (tumbling and sliding). For the dynamical configuration of the data windowing, we will use both approaches: definition using the high-level window API and the low-level *processFunction* API.

1) High-level window API: There are 2 available time windowing strategies (tumbling and sliding). Both window strategies are applied to the data streams for every partition. That is something that we wanted to change because every user-defined rule that exists should have its windowing configuration and that windowing configuration should be applied to each partition and for every rule.

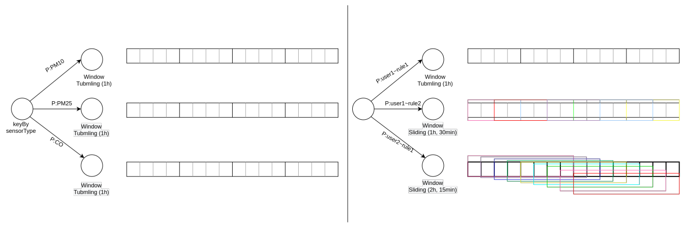

Fig. 5. Visualization of static (left) and dynamic (right) configuration of the key By operator

The window operator expects only one argument, a window assigner (object from the class WindowAssigner). We will introduce our window assigner (class **GenericWindowAssigner**) that will work with the matched messages which were created when we connected the control data stream with the input data stream. Our window assigner will be able to extract the windowing configuration from the user-defined rule that is located in every matched message. The rule has the following windowing configurations:

- the type of the windowing (textual field with valid options tumbling or sliding)
- the size of the window (milliseconds)
- the slide of the window (milliseconds, applicable only if the window type is sliding)

Also, our window assigner should be able to extract information about the timestamp of the input message which is contained in the matched message. To make the window assigner generic and reusable for other projects every message that enters in the window assigner should implement an interface IConfigurableWindow, including the matched message objects in our application. This interface has methods for getting the elements that need to be windowed (for ex. the sensor measurement in our use case) and all the windowing configurations listed above.

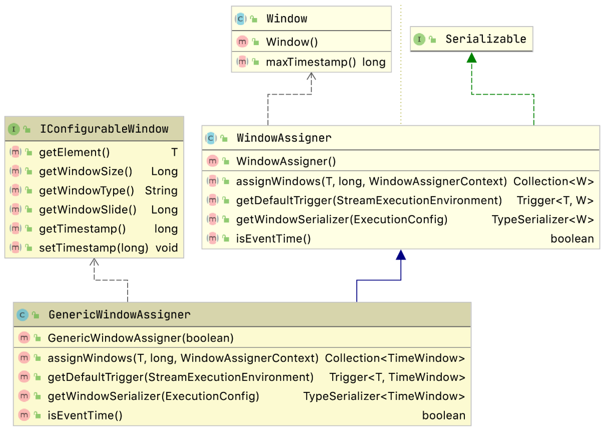
Window
Window()
maxTimestamp() long
Serializable
IConfigurableWindow
getElement() T
getWindowSize() Long
getWindowType() String
getWindowSlide() Long
getTimestamp() long
setTimestamp(long) void
WindowAssigner
WindowAssigner()
assignWindows(T, long, WindowAssignerContext) Collection<W>
getDefaultTrigger(StreamExecutionEnvironment) Trigger<T, W>
getWindowSerializer(ExecutionConfig) TypeSerializer<W>
isEventTime() boolean
GenericWindowAssigner
GenericWindowAssigner(boolean)
assignWindows(T, long, WindowAssignerContext) Collection<TimeWindow>
getDefaultTrigger(StreamExecutionEnvironment) Trigger<T, TimeWindow>
getWindowSerializer(ExecutionConfig) TypeSerializer<TimeWindow>
isEventTime() boolean

Fig. 6. UML class diagram of the window assigner class. *Note: The classes filled with light yellow color are part of the Flink source code*

As shown in the UML class diagram on Fig. 6 every window assigner in Apache Flink, both the default ones and our dynamical window assigner have 4 methods to implements.

- **assignWindows** - method that receives an element and its timestamp. Based on those information, it assigns *one window* (if the window type is *tumbling*) or *multiple windows* (if the window type is *sliding*) where the element belongs
- **getDefaultTrigger** - method that returns a default trigger that will activate the window i.e. send all the collected events/items in that window in the next operator or process function. In the timed windows, the trigger depends on that whether the window's end time has passed the current timestamp
- **getWindowsSerializer** - method that returns a serializer for the window type that is being used.
- A method that returns boolean value true if the windowing is using the event time notion of time [1] and otherwise false. In our case, the result is true because we're using event time in our case.

Our window assigner implements all of the above-listed methods in the following way:

- When creating it via a constructor, we share information (a boolean argument) that represents the notion of time (true means event time and false means processing time)
- For the first method (assignWindows) we have a Factory class that based on the windowing type (available from the IConfigurableWIndow interface), will calculate how many windows the element belongs to as well as the start and end time of those windows. The collection of windows returned from the Factory is the result of this method as well.
- Based on the boolean variable from the constructor of the assigner, we are creating an event time default trigger

[1]Event time is the time at which the event has occurred on its producing device. In our input messages, that is the timestamp field from the input message. The opposite notion of time is processing time that represents the time when the event was received in the application.

(which triggers the window based on the fact that the end of the window is before the pronounced watermark [2]) or a processing time default trigger (which is triggering the window based on the fact if the end of the window is before the current system time of the machine where the application is running). The created trigger is a result of the second method of the window assigner.
- In the third method we are returning an object of type TimeWindowSerializer, because our dynamical window assigner is meant to work only with time windows (tumbling or sliding)
- In the fourth method we are just returning the boolean value that we received through the constructor of the assigner.

2) Low-level process API: All high-level implementations of operators are based on low-level process function. Any functionality which is not defined could be implemented within a process operator. In this part, we introduce an implementation of a window operator over a partitioned data stream. In the proposed implementation as shown in Fig. 7, we introduced our process function which is a composition of the two main components that are used for data and windows management:

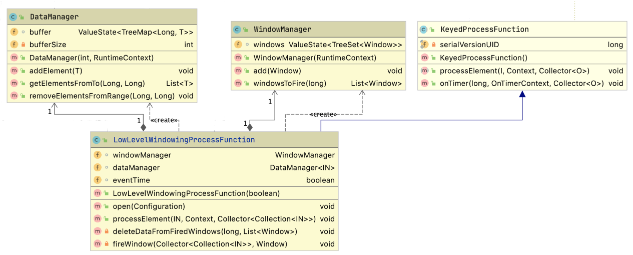
DataManager
buffer ValueState<TreeMap<Long, T>>
bufferSize int
DataManager(int, RuntimeContext)
addElement(T) void
getElementsFromTo(Long, Long) List<T>
removeElementsFromRange(Long, Long) void
WindowManager
windows ValueState<TreeSet<Window>>
WindowManager(RuntimeContext)
add(Window) void
windowsToFire(long) List<Window>
KeyedProcessFunction
serialVersionUID long
KeyedProcessFunction()
processElement(I, Context, Collector<O>) void
onTimer(long, OnTimerContext, Collector<O>) void
1
1
«create»
1
1
«create»
LowLevelWindowingProcessFunction
windowManager WindowManager
dataManager DataManager<IN>
eventTime boolean
LowLevelWindowingProcessFunction(boolean)
open(Configuration) void
processElement(IN, Context, Collector<Collection<IN>>) void
deleteDataFromFiredWindows(long, List<Window>) void
fireWindow(Collector<Collection<IN>>, Window) void

Fig. 7. UML class diagram of classes part of the low-level dynamical configuration of windowing

Window Manager. This component manages all windows that should be triggered. They are stored in a tree structure. This means that the windows are sorted in ascending order by the maximum timestamp of the window (the end of the window decreased by one). This tree structure is stored into a managed *ValueState* in Apache Flink. The Window Manager provides the creation and addition of the windows and obtaining the windows which should be triggered.

Data Manager. This component stores all the input data that should be windowed. The input data with assigned windows are stored in a tree structure (TreeMap in Java) so that the data will be sorted based on the key in the map which is the timestamp of the input messages. To be able to extract the timestamp the input data should implement the interface IConfigurableWindow as in the high-level configuration. The Data Manager provides the addition of new elements, the obtaining and the deletion of elements that belong in a certain window.

[2]When working in event time, the watermark is helping us to keep track of the progress in time while processing the events

Also, as in the high-level configuration, in the *processFunction* (LowLevelWindowingProcessFunction) implementation, we must keep information about the notion of time. In the keyed process function we have one DataManager object and one WindowManager object. The most important thing about these two objects is that they have to share the *runtime context* of the keyed process function because both of them have to store the windows and the input data into a *ValueState*.

The workflow of the keyed process function is described in the following steps which are applied on every new element that enters the process function:

1) Window or collection of windows where the new element should belong are created.
2) The created window or windows are stored in the WindowManager.
3) The element that arrived is stored in the DataManager.
4) Based on the specified notion of time, the current time of the stream processor should be determined.
5) All the windows which should be triggered i.e the current time is past their end time are obtained from the WindowManager.
6) All the data that belongs in the time intervals of the windows from the previous step is obtained from the DataManager and collected as a result into a collection of data. Later this collection of data will be used for the next operators (in our use case that is the aggregate operator).
7) The data from step 6 and the windows from step 5 are deleted correspondingly from the DataManager and the WindowManager.

D. Dynamic Data Aggregating

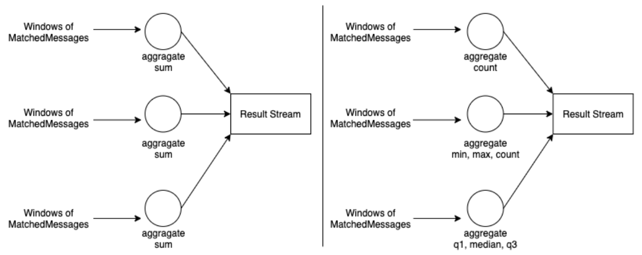
Windows of MatchedMessages
aggregate sum
Windows of MatchedMessages
aggregate sum
Windows of MatchedMessages
aggregate sum
Result Stream
Windows of MatchedMessages
aggregate count
Windows of MatchedMessages
aggregate min, max, count
Windows of MatchedMessages
aggregate q1, median, q3
Result Stream

Fig. 8. Visualization of static (left) and dynamic (right) configuration of the aggregate operator

The last step of the data stream processing and analytic scenario that we have presented in the introduction is aggregating the data collected into the windows from the windowing operator/process. Similar to the dynamical configuration of the window operator, high-level and low-level dynamical aggregation configuration is proposed.

The high-level dynamic configuration (Fig.9) is achieved with an implementation of the aggregate function interface (RuleAggregationFunction) from which an object is created and is sent as an argument to the aggregate operator.

The low-level dynamic configuration (Fig. 10) is achieved with the implementation of a low-level process function (Ag-

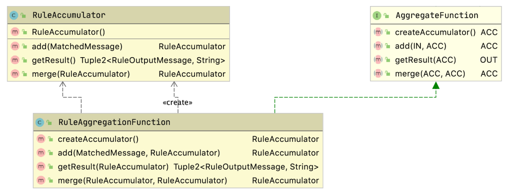
RuleAccumulator
RuleAccumulator()
add(MatchedMessage) RuleAccumulator
getResult() Tuple2<RuleOutputMessage, String>
merge(RuleAccumulator) RuleAccumulator
AggregateFunction
createAccumulator() ACC
add(IN, ACC) ACC
getResult(ACC) OUT
merge(ACC, ACC) ACC
«create»
RuleAggregationFunction
createAccumulator() RuleAccumulator
add(MatchedMessage, RuleAccumulator) RuleAccumulator
getResult(RuleAccumulator) Tuple2<RuleOutputMessage, String>
merge(RuleAccumulator, RuleAccumulator) RuleAccumulator

Fig. 9. UML class diagram of the classes part of the high level dynamical configuration of aggregation

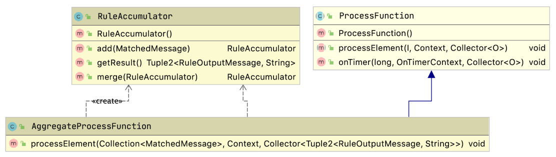
RuleAccumulator
RuleAccumulator()
add(MatchedMessage) RuleAccumulator
getResult() Tuple2<RuleOutputMessage, String>
merge(RuleAccumulator) RuleAccumulator
ProcessFunction
ProcessFunction()
processElement(I, Context, Collector<O>) void
onTimer(long, OnTimerContext, Collector<O>) void
«create»
AggregateProcessFunction
processElement(Collection<MatchedMessage>, Context, Collector<Tuple2<RuleOutputMessage, String>>) void

Fig. 10. UML class diagram of the classes part of the low level dynamical configuration of aggregation

gregateProcessFunction) that processes the elements that belong to a particular window (collections of matched messages) collected in the previous operator.

The mutual part of the low- and high level-dynamical configuration is the RuleAccumulator class (Fig. 9 and 10). In both scenarios, an object from this class is used to accumulate the extracted numeric values from the matched message. With the help of this class, we can successfully calculate the descriptive statistics: minimum, maximum, average, count, first quartile (Q1), median and third quartile (Q3).

IV. Real-life example

The dynamically configured stream processor that we have designed can be used as a software component in other software. For example, we have integrated the stream processor with a web application that enables the user to send their rules through a user-friendly interface. After the rules are sent the user gets a link to his personalized real-time dashboard where the events and event's aggregations are visualized in the way the user required that in the rule.

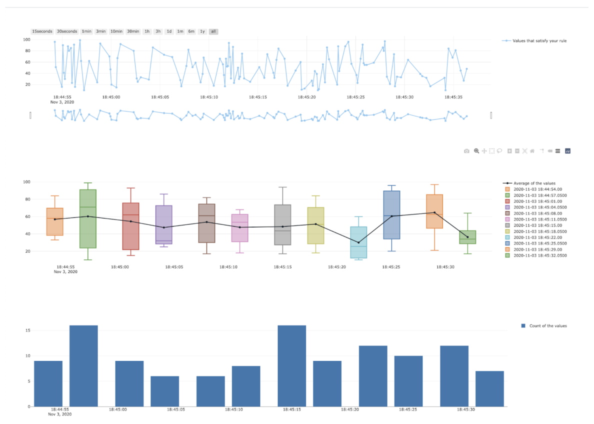

Fig. 11. Example of a dashboard created with the utilization of the stream processor

The first plot in Fig. 11 is a time-series plot of the extracted numeric values from all the matched messages that were created from the user's rule. The second one is a plot of boxplots where each boxplot is generated from the statistics aggregated for each time widows. The third plot is a bar plot where each value is the count of messages that were part of the corresponding window whose statistics are shown right above the bar.

The benefit of using a dynamically configured stream processor in this application is that the application can have an unlimited number of users which will send an unlimited number of rules and for each rule, a new dashboard will be created while using **the same stream processor**.

V. Conclusion

In this paper, software patterns for dynamic data partitioning, dynamic window configuration, and dynamic data aggregation in Apache Flink are presented. These patterns extend the concept of statically defined data flows and allow Flink jobs to dynamically change at runtime, without downtime, allowing implementation of complex business scenarios. The introduced software patterns are applied on real-life use case scenario that tackles the problem of processing and analyzing air pollution data obtained from different sensors located in different locations, as well as visualization of the data in third-party software.

Acknowledgements

This research was partially funded by the Faculty of computer science and engineering, Ss. Cyril and Methodius University in Skopje.

References

[1] Apache Flink. Scalable batch and stream data processing, 2016

[2] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S. et al. (2015) Apache Flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 36(4)

[3] Carbone, P., Ewen, S., Fora, G., Haridi, S., Richter, S., Tzoumas, K.. (2017) State management in Apache Flink®: consistent stateful distributed stream processing. Proc. VLDB Endow. 10, 12 (August 2017), 1718–1729. DOI: https://dl.acm.org/doi/abs/10.14778/3137765.3137777

[4] A. Alexandrov, R. Bergmann, S. Ewen, J. C. Freytag, F. Hueske, A. Heise, and D. Warneke. The stratosphere platform for big data analytics. The VLDB Journal—The International Journal on Very Large Data Bases, 23(6):939– 964, 2014

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, USENIX Association, Berkeley, USA, 2012, pp. 2–2.

[6] "Storm Homepage." [Online]. Available: https://storm.apache.org/ [Accessed: 06-April-2021].

[7] "Advanced Flink Application Patterns Vol.1: Case Study of a Fraud Detection System" [Online]. Available: https://flink.apache.org/news/2020/01/15/demo-fraud-detection.html [Accessed: 08-April-2021]

[8] "Advanced Flink Application Patterns Vol.2: Dynamic Updates of Application Logic" [Online]. Available: https://flink.apache.org/news/2020/03/24/demo-fraud-detection-2.html [Accessed: 08-April-2021]