Diploma Thesis

Geometric Streams Monitoring on Apache Flink

by
Eduard Epure

Thesis committee:
Prof. Samoladas Vasilis
Prof. Deligiannakis Antonios
Prof. Garofalakis Minos

A dissertation submitted in partial fulfilment of the requirements
for the degree of
Electrical and Computer Engineering
December 14, 2020

Abstract

The amount of data generated every day by online applications is continuously growing, which results in a demand for capable real-time stream processing frameworks. Numerous monitoring algorithms have been proposed over the years, yielding exceptional results but with common shortcomings. The problem not yet addressed by previous work on monitoring algorithms is their integration into large data-stream processing frameworks. Part of the reason may be the lack of uniformity each monitoring algorithm presents and the requirements it imposes on the system architecture. That's where Apache Flink comes into play. It is an open-source framework and distributed processing engine for stateful computation over unbounded data-streams. Flink is a very versatile tool, designed to run in all common cluster environments and perform computations at any scale. This thesis describes the implementation of the Functional Geometric Monitoring algorithm on Apache Flink, while comparing the extracted results with these of previous simulated implementations of monitoring algorithms.

Keywords

data-streams monitoring, geometric monitoring, distributed streaming, continuous query, apache flink, stream processing

Acknowledgements

I would like to express my gratitude to my advisor Vasilis Samoladas for his guidance, patience and immense knowledge. Thank you for introducing me to the concept of distributed systems and the opportunity to engage with such a project. Furthermore, I would like to thank the rest of my thesis committee members: Prof. Minos Garofalakis and Prof. Deligiannakis Antonios, for their insight through their lectures and their time to evaluate this work. My sincere thanks to my dear friend Eftychia Seisaki for her relentless motivation and support. Last but not least I thank my beloved parents Remi and Violeta for their unbounded support, thoughtfulness and inspiration through this process.

Contents

List of Figures

1 Introduction

1.1 Objective

The amount of data generated every day by online applications is continuously growing while Internet of Things is set to further increase the load of network data that needs to be processed in real time. Consequently, the need for a highly scalable and distributed processing technique is crucial. Thankfully, numerous algorithms in the area of data steams monitoring have been proposed in the recent years, showing exceptional experimental results in communication cost reduction. Whereas this line of work, starting with the Geometric monitoring[10] and continuing with the Functional Geometric monitoring[9] over distributed streams have proved their applicability to continuous queries, they have not addressed their integration in real stream processing frameworks.

At the same time, many large scale stream processing engines have been developed over the recent years, making big steps in processing as efficiently as possible the high paced real-time data. Just to name a few, Apache's Flink[1], Spark and Kafka streams[2], are leading in the stream processing market at the moment. While these frameworks strive for highly scalable and optimized data processing they do not take advantage of the above-mentioned novel techniques of continuous query monitoring.

Flink is a very capable distributed processing engine, designed to process unbounded or bounded data streams and can perform computations at any scale. It is a powerful framework that provides many out of the box features needed for streams monitoring, to mention a few, various window mechanisms, event time handling, aggregation and groupBy operations. Furthermore it gives access to lower level API in order to satisfy the need for more complex functionality.

In hindsight, this thesis addresses this problem by connecting the two ideas. The Functional Geometric approach has been selected as the monitoring algorithm while Apache Flink is the choice for the processing framework. The objective of the thesis is to implement the monitoring algorithm on the chosen platform and furthermore to compare the extracted results to the results of previous implementations of monitoring algorithms on simulated distributed systems. In addition it introduces a new custom operator which can be used in Flink and facilitates the use of stream monitoring.

1.2 Thesis Outline

Related Work

Chapter 2 introduces the core concepts and frameworks used in this diploma, namely Apache's Flink and Kafka frameworks, the Functional Geometric Monitoring algorithm and the AGMS Sketches[5] which is used in conjunction with the basic continuous query monitoring technique. The 'Related Work' chapter aims to cover all necessary definitions and concepts before proceeding to the actual implementation.

Implementation

Chapter 3 begins by giving an overview of the whole pipeline and introducing it's components, the high level operators. Consequently it delves into the internals of the implementation while offering relevant explanations for the choices made through the process. Finally it addresses some issues of Flink's default functionality and how they were overcome.

Streams Monitoring API

Chapter 4 explains in depth the API of the custom monitoring operator. It showcases common use-cases and moreover it offers some demo implementations of them.

Evaluation

Chapter 5 introduces the real-world data set used in the final experiments. It describes the methodology used in validating the results and finally it shows the graphs yielded from this implementation while evaluating and commenting them.

Conclusion

The last chapter concludes the thesis, suggests alternative approaches and summarizes some ideas for future development.

2 Related Work

The following subsections are presented in order of importance for this thesis, with Flink and FGM being equally matched since the objective of the thesis is the merging of the two. Succeeding, is the reference to Fast-AGMS sketches[5] as an overall space optimization of the distributed state vectors and the introduction of sketch-related query functions[7]. Finally, Kafka message queue is mentioned because of its excellent connectivity with Flink and its ability to perform as an external feedback loop to a Flink pipeline.

2.1 Apache Flink

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. Flink has been designed to run in all common cluster environments, perform computation at in-memory speed and at any scale. Flink also provides multiple APIs at different levels of abstraction and offers dedicated libraries for common use cases.[1]

Overview

A Flink application may consume real-time data from streaming sources such as message queues like Apache Kafka[2], but can also consume bounded, historic data from a variety of data sources. Similarly, the streams of results being produced by a Flink application can be sent to a wide variety of systems that can be connected as sinks as shown in figure 2.1.

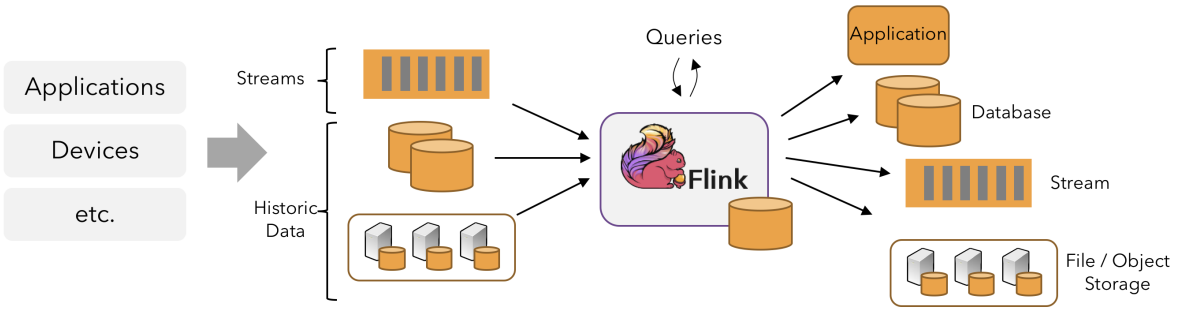

Figure 2.1: Flink's role as a data processing framework.

On top of that, users can implement their own business logic using a very intuitive API with multiple levels of abstraction and perform stateful operations on the assimilated data-streams. Queries can also be posed to the system while running and therefore act like a simple DBMS.

Key Concepts

A few key concepts to keep in mind before proceeding to the implementation are summarized below. The following terms are explained in detail in the official Flink documentation[1], but here are listed the most important notions for this work.

Pipeline

A graph representation of the transformations performed on data-streams. These transformations, commonly referred to as operators are the main building blocks of data pipelines. The pipeline comprises of all application logic and offers a great way to visualize the whole project at once. In Flink, a pipeline usually consists of 1 or more sources that produce data-streams, some operators that transform these data-streams and finally 1 or more sinks that can store the data long term or forward it to other frameworks for further processing.

DataStream API

The middle layer of Flink's API and provides many common stream processing operations such as windowing, map, aggregate functions, partitioning and keyBy. This is used most of the time along this work while sometimes extending to the lower-level process function. As an example, a source producing a stream of events of type CustomEvent will return an object of type DataStream<CustomEvent>. Transformations can be applied on this object with the java dot operator in the following manner: `input.keyBy().map().print();`.

State

A persistent data store used by stateful operators, iterative streams, check-pointing and other internal mechanisms. Basically any application that needs to remember information across multiple events must use some form of state. In particular, Keyed State is the preferred state for this work since it allows for key-specific grouping of data-streams and it is independent of job's parallelism.

Event time processing

One of the provided time notions, allows Flink to handle timestamped records as if they were streamed in real-time. As a result, all time-based operations can be performed naturally and provide accurate results.

Windowing

By definition is splitting an unbounded data-stream into finite sets. Essentially windows bring parts of a data-stream into scope in order to be processed. This can be done in numerous ways and Flink provides quite a few Window operators. In this case, an event-time Sliding Window with incremental aggregation is being used. The only prerequisite is that it must be applied on a keyed stream, meaning a keyBy operator is preceding in the pipeline.

Process Function

This low-level operator can be thought as a flatMap operator with access to state, context and timers. It can also make use of side-outputs because of their context.

Side Outputs

Additional outputs to a stateful operator that can handle different data-types than the main output. They are very useful for splitting the output of an operator or in this particular case for querying partial results while using an iterative stream.

Iterative Stream

A type of stream that contains a feedback loop. It is mandatory for the stream monitoring algorithm implemented in this thesis. Unfortunately, Flink's iterative stream is still evolving so it is still unreliable in some cases, therefore Kafka topics are used as a substitute.

2.2 Functional Geometric Monitoring

The contributions of Functional Geometric[9] approach to the distributed monitoring problem are quite a few. First of all, FGM comprises of a distributed algorithm which is independent of the monitoring problem. It can utilize many problem-specific functions termed 'safe functions' that have been previously proposed for monitoring problems. Additionally, this technique is ready to be employed by real distributed applications on the grounds that there is a strict separation of concerns between distributed systems issues and the monitoring problem. Another important mention is the resilience of FGM to high data variability and in presence of data skew, where in both of these adverse conditions it adapts seamlessly. This section covers some basic notions of FGM but reading the related article[9] is highly encouraged.

Approximate Query Monitoring

The adopted data model of this monitoring technique is the standard data model for distributed data streams, with k distributed sites that continuously collect streams of records and consequently update their state vectors. Every site communicates with a coordinator, where the user submits continuous queries on the global stream. In this type of query, the coordinator needs to maintain an estimate $Q(E(t))$ of the true value of the query $Q(S(t))$, within a predefined error margin ε, so that

$$Q(S(t)) \in (1 \pm \varepsilon)Q(E(t)) \tag{1}$$

This guarantee is maintained by the local sites flushing their drift vectors $X_i(t) = S_i(t) - S_i(t_0)$, where $S_i(t_0)$ are the local state vectors before the previous flush, to the coordinator and eventually the coordinator updating the global estimate (it is assumed that the global estimate is the average of the local stream states)

$$E = E + \frac{1}{k}\sum_{i=1}^{k} X_i. \tag{2}$$

Safe function

Functional Geometric Monitoring employs a real function $\phi : V \to \mathbb{R}$ whose $\phi(X_i)$ value is tracked by each local site as X_i is updated. System safety is ensured by tracking the sum $\psi = \sum_{i=1}^{k} \phi(X_i) > 0$ at the coordinator, meaning that sites don't need to flush their vectors as long as this condition holds, thus reducing the communication cost. For the sake of simplicity we will not dive into details about safe function design since this work aims to provide a generic monitoring API and not a strict implementation. Extensive discussion about safe zone composition and quality criteria is done in [7]. Instead we offer a simple example of

a safe zone function, which also served as a starting point in our implementation.

Consider the problem of monitoring the Euclidean norm of the global state vector $S \in V$, where S is a frequency vector containing the records arriving from all local streams. Assume also that the coordinator holds the estimate E which does not change as long as $S(t) \in A$, with A being the admissible region

$$A = \{x \in V \mid \|x\| \in (1 \pm \varepsilon)\|E\|\}. \tag{3}$$

The admissible region A can be visualized in a two-dimensional space as a ring with the inner radius $(1-\varepsilon)\|E\|$ and the outer radius $(1+\varepsilon)\|E\|$. Unfortunately that is not a convex set and therefore not a suitable safe zone. However we can obtain a convex subset $Z \subseteq A$ if we combine the Z_2 ball of radius $(1+\varepsilon)\|E\|$ with the half-space Z_1, as $Z = Z_1 \cap Z_2$. Geometrically this intersection can be characterized as a hyper-spherical cap. In order to utilize this concept in a practical way, we can describe the above sets by functions as follows:

$$\zeta_1(x) = x \cdot E - (1-\varepsilon)\|E\|^2 \tag{4}$$

$$\zeta_2(x) = (1+\varepsilon)\|E\| - \|x\| \tag{5}$$

The intersection $Z_1 \cap Z_2$ can be represented by taking the pointwise-minimum of the two functions, with the final safe zone function being

$$\phi(x) = min\{\zeta_1(x), \zeta_2(x)\}. \tag{6}$$

Rebalancing

The FGM rebalancing process is desirable since it can prolong the duration of a round without additional communication cost. The motivation for this optimization arises from the fact that condition $\psi > 0$ does not imply that the global state S has moved close to the boundary of the safe zone. Thus, the current safe zone might still be useful and the cost of shipping a new safe zone function to the site may be avoided. Specifically, it ships a scaling factor λ to the sites with which to scale their drift vectors. This only incurs insignificant upstream cost and achieves a positive communication gain over the duration of a round.

Adaptive safe zone shipping

The incentive behind the idea of shipping different safe zones among the sites is the issue of highly unequal stream rates. In the case that 98% of the sites process little to no stream updates, the cost of shipping a safe zone function is wasteful. Therefore, a practical way of avoiding this extra upstream cost is applied, along with a method to correctly identify the unproductive sites.

2.3 Fast-AGMS Sketches

The selected algorithm for sketching the data in the distributed nodes and estimating multi-join queries is the Fast-AGMS sketch[5] which is based on previous work[3].This section gives a quick overview of the algorithm but implementation-specific details are provided in Chapters 3 and 4.

A Fast-AGMS sketch for a stream $\boldsymbol{f}$ of records comprises of $D \times W$ counters C_{ij} arranged in D hash tables, each with W hash buckets (also denoted $sk(f)$). Each hash table $i = 1, \ldots, D$ is associated with (1) a *pairwise-independent hash function* $h_i()$ that maps incoming stream elements uniformly over the W hash buckets and (2) a family of *four-wise independent {-1, +1} random variables* (as in basic AGMS). To update the sketch in response to an insertion/deletion of element $e(key, value)$, we use the $h_i(key)$ hash functions to determine the appropriate buckets in the sketch, setting $C_{i,\, h_i(key)} \mathrel{+}= value \cdot \xi_i(key)$ for each $i = 1, \ldots, D$ as shown in figure 2.2 below.

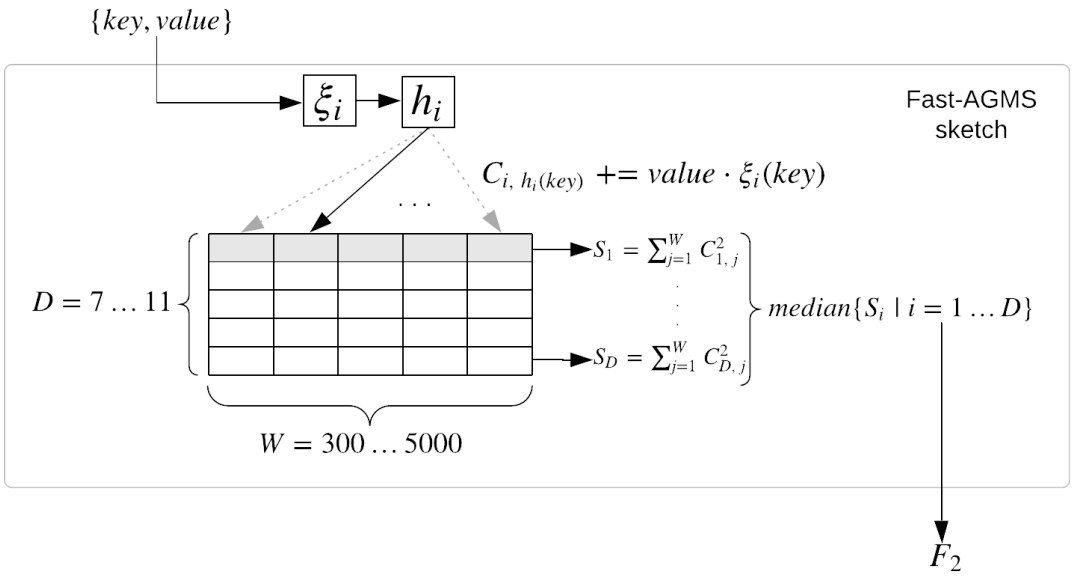

Figure 2.2: Overview of a Fast-AGMS sketch.

By selecting the median of the sums of squares S_i of each hash table, we can extract the estimated self-join size also known as 2nd frequency moment F_2.

$$Q(sk(f)) = median\{\sum_{j=1}^{W} C_{ij}^2 \mid i = 1, \ldots, D\} \tag{7}$$

The algorithm achieves this with error $O(\frac{1}{\sqrt{W}})$ and confidence at least $1 - O(2^{-D})$[1] while the required time per update is $O(log(\frac{1}{\delta}))$.

[1]The suggested values for W and D in figure 2.2 are empirical. E.g a sketch with W=3000 and D=7 has an estimation error of 1.8% with confidence of 99.2%

2.4 Apache Kafka

Apache Kafka is a distributed message queuing system designed to stream data to multiple consumers and at the same time store data arriving from multiple producers. It also has excellent connectivity with Apache Flink due to the immense work of the Flink community. This section will briefly present some of the abstractions that constitute this platform in addition to the peculiar use case for this thesis. Apache Kafka official documentation[2] and the practical guide from Ververica[8] are the main sources for this section.

Topics

The core abstraction Kafka provides for a data-stream is the topic. Topics are just handles for data-streams and internally they consist of one or more partitions uniquely identified. They have a log-like data structure, meaning that records from producers are continually appended to them. Kafka persists all published records to the topics regardless if they have been consumed or not, until the configurable retention period has passed.

Producers

Producers publish data to topics and they are responsible for selecting the right internal partition for the records. This can be achieved either in a random manner or by using a partitioning key.

Consumers

Consumers subscribe to topics and consume their data. They can be part of consumer groups which provides control over the distribution of the data. Furthermore, each consumer controls an offset value on every topic it subscribes to, which indicates the position of data it currently consumes.

Use case: Iterative stream via Kafka Topics

Using Kafka Topics as an external tool to perform a feedback loop in a Flink pipeline is a straightforward procedure. A single topic is needed which acts as a buffer between the producer and the consumer. The producer in this case would be the coordinator where the control messages are produced and the consumer would be the worker nodes. The incentive for this idea was the unreliability of Flink's default iterative stream and the inevitable use of a feedback loop due to the nature of the monitoring algorithm. Detailed explanation is provided in section 4 of Chapter 3.

3 Implementation

This chapter offers a detailed explanation of the operators and the classes used throughout this project. In the first section an overview of the working pipeline is discussed and the functionality and purpose of each operator is being described. The following sections focus on the core operators of this particular work which are the Worker and the Coordinator processes, along with the Sliding Window and the iteration mechanism. Last but not least, details about the entities connecting the operators and their structure are presented in the Internals section.

3.1 System Design

This implementation was designed to support continuous query monitoring on distributed streams while giving the user the flexibility to implement his own query function and safe zone. In addition it provides a custom sliding window operator which is much more suitable than the default window for this type of processing. The program was intended to work with Event-time notion but that choice along with the watermark generation and timestamp extraction remain the user's responsibility.

The chain of operators described in this chapter constitute a module that can be used independently or as part of a more complex setup. The minimum requirements for successful execution is a Stream Execution Environment, a Source function for the input streams, a Sink function for collecting the output and finally an implementation of the `BaseConfig` interface which is explained extensively in the next chapter.

3.2 Pipeline

The final pipeline after many revisions consists of source and sink functions, keyBy operators and of course flatMaps and processFunctions. Some of them are not part of the final product but at this point the intention is to give a complete explanation of this work. In figure 3.1 below, the sources and sinks are depicted with cylinder shapes, the simple operators are circles and the core operators are rectangles. Also note that some arrows and shapes are bolder than others, which means that those transitions and nodes have a flink parallelism greater than 1.

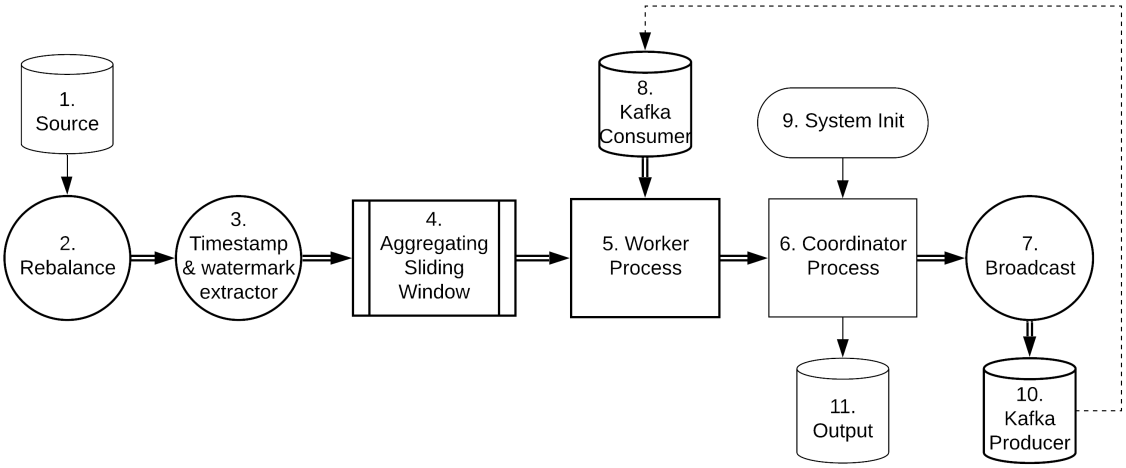

Figure 3.1: Overview of the implementation Pipeline.

The flow of data streams starts at the main source of the pipeline. In our experiments we used a Kafka consumer which streamed line by line Strings from a predefined Kafka topic. Besides the main source there is also a Initialization source which pushes a single event to the coordinator in order to kick-start the system. This happens independently of the main source. We used the Initialization source to pass the `warmup` delay to the coordinator. In other words we instruct the coordinator to delay the first request of drift vectors by a predefined amount of time. Next in line is the Watermark assigner operator, which is responsible for extracting the timestamp values from the passing events and assigning Watermarks periodically. In turn these Watermarks will be used by the Window operator and the other timers of the system to progress their time. The Watermark assigner can be included in the source but we chose to have it as a separate operator for easier debugging and readability.

Following is the Window operator which is described in depth in the next section. This operator is optional in the sense that it can be disabled/omitted if chosen to by the user via the API implementation. Sequentially the input events reach the Worker operator where their aggregation to the state vectors takes place and where the FGM subroutines are executed. The Worker also handles events from the feedback stream and sends various FGM related messages downstream to the coordinator.

The coordinator processes the incoming messages from the Worker instances and produces output which then broadcasts via the Kafka producer to a predefined Feedback Topic. It is worth mentioning that Flink doesn't provide a way to broadcast one event to multiple instances, at least not in the fashion intended here, so a simple *for − loop* has been employed to ensure the even distribution of the feedback messages. The idea here is that the feedback stream that returns to the Workers needs to be keyed by the `streamID`, because the input stream is also keyed by that field. Therefore the coordinator must emit `k` events with different `streamID` each one, in order to be received by all of the workers. After the Feedback topic

has been populated with the newly broadcasted messages, the Kafka consumers read the messages and forward them to the Workers in order to close the feedback loop. Finally, when the FGM Protocol at the coordinator is in the end-of-round phase, it will also emit the value of the User-defined Query function to a side output.

3.3 Core Operators

In this implementation core operators are considered to be the worker and the coordinator processes because they contain most of the monitoring logic.

Worker Process

The $5th$ operator of the pipeline as shown in figure 3.1 contains the logic of the distributed sites in the monitoring problem. The operator has been chosen to extend the *KeyedCoProcessFunction* of the DataStream API for two reasons. Firstly, it must be a $Co - nnected$ processFunction because it processes two data-streams, (1) the input stream produced by the sliding window and (2) the feedback stream arriving from the coordinator via the Kafka iteration. Secondly, it must be a *Keyed* processFunction because it utilizes Flink's KeyedState in order to store vectors and values across multiple instances. In this case both input streams must be transformed to KeyedStreams using the *.keyBy*() operator right before the WorkerProcess (not shown in the pipeline diagram). The partitioning is done based on the *streamID* variable.

The responsibility of this operator is to read the events from the two data-streams and call the appropriate methods based on the FGM protocol. The pseudocode below shows the functionality of the operator.

Algorithm 1: Handling Input and Feedback streams via *KeyedCoProcessFunction*

```
Function processElement1(input):
    updateDrift(...);
    subRoundProcess(...);
Function processElement2(feedback):
    case GlobalEstimate do
        newRound(...);
        subRoundProcess(...);
    case Quantum do
        newSubRound(...);
        subRoundProcess(...);
    case RequestDrift do
        sendDrift(...);
    case RequestZeta do
        sendZeta(...);
    case Lambda do
        newRebalancedRound(...);
        sendZeta(...);
```

The methods of the FGM protocol have been implemented in a separate class, namely the *fgm.WorkerFunction*. It consists of several methods that are meant to handle the various phases of the algorithm and they are described in table 3.1 below. Note that these methods may not follow verbatim the protocol in §2.4 of the FGM paper[9], but have been constructed according to Flink's idiosyncrasies.

Table 3.1: fgm.WorkerFunction methods description

Name	Description
updateDrift	updates the drift vector with the newly arrived records
newRound	updates the safezone, initializes sub-round phase
newSubRound	updates quantum, restarts sub-round phase
newRebalancedRound	updates λ, initializes sub-round phase
sendDrift	flushes drift vector and clears it
sendZeta	flushes $\phi(X_i)$ and pauses sub-round phase
subRoundProcess	evaluates sub-round condition, sends increment

Another important aspect of this implementation that is not visible in the pseudo-code above nor in Table 3.1 is the *state.WorkerStateHandler* class. Its responsibility is to manage all the KeyedState variables used throughout the FGM methods and to provide simplified *get* and *set* methods for them.

Coordinator Process

The second core operator is the coordinator. It also extends the *KeyedCoProcessFunction* but but it is used in a slightly different way. The primary input is the union of the workers' outputs, while the secondary input is an initialization event which triggers the whole system. In the *.keyBy*() operation preceding the coordinator a special function has been employed in order to merge the multiple workers' outputs into a single one, namely the *InternalStream.unionKey*().

Algorithm 2: Handling Input and Initialization streams via *KeyedCoProcessFunction*

```
Function processElement1(input):
    case Drift do
        handleDrift(...);
    case Zeta do
        handleZeta(...);
    case Increment do
        handleIncrement(...);
Function processElement2(init):
    warmup(...);
```

The responsibility of this operator is similar to that of the worker, precisely to call the appropriate methods of the FGM protocol depending on the input type. The Coordinator Process also has two dependencies, the *fgm.CoordinatorFunction* class containing the FGM methods and the *state.CoordinatorStateHandler* class containing the state management functions. The methods descriptions are presented in table 3.2.

Table 3.2: fgm.CoordinatorFunction methods descriptions

Name	Description
handleDrift	aggregates drift vectors, starts a new round
newRound	updates safezone and globalEstimate
handleZeta	aggregates $\phi(X_i)$ values and evaluates $\psi > 0$ condition
handleIncrement	aggregates C_i values and evaluates sub-round condition

It is important to mention that using a ProcessFunction instead of a RichFlatMap for this operator is crucial since the main output is utilized for the feedback stream and the query output must be diverged to a side output. Emitting data to a side output is only possible from functions such as ProcessFunction.

3.4 Sliding Window

Windows play a major role in unbounded stream processing. They are responsible for splitting the data-streams into "batches" of finite size over which computations can be applied. There are many types of windows in stream processing and Flink provides a very intuitive API for many of them. Unfortunately, our scenario cannot be handled efficiently by any of the provided window operators.

Before diving into the actual implementations, there are two more things to be mentioned. The sliding window must be applied on a *KeyedStream* since it is using KeyedState and therefore the *.keyBy*(...) operator precedes the window operator. Another important step is the assignment of watermarks on the streaming data. This is the only way in which an event-time window can perceive the progress of time and trigger the evaluation function at the right moment.

Aggregating Sliding Window

The best implementation of sliding window we could come up with using Flink's window functions is the following. It consists of 3 major parts, namely (1) the WindowAssigner which handles the splitting of data-streams, (2) the AggregateFunction which accumulates the events arriving on every slide of the window and finally (3) the WindowFunction that fires when the whole window is filled and ready to be evaluated.

The underlying issue

After numerous efforts to implement a suitable window operator we concluded that as of Flink 1.10.x, the desired functionality cannot be achieved with the existing window-API. This outcome is based on several experiments and tests. Many issues have been also opened on Flink's Jira and an extended research has been made regarding efficient aggregation of sliding windows [11], but an actual implementation has not been yet made possible.

The default sliding-window cannot simply cope with the amount of data passed to it. Because of its architecture, it stores 1 copy of the whole window for every overlapping window in scope. In other words, if a sliding-window with size of 1 hour and slide interval of 1 second is selected, then 60 min * 60 sec = 3600 window instances will be created and stored in memory. Than cannot be supported of course because 1 single instance may contain thousands of events at any given time.

Even when coupled with the Aggregate function, this implementation still does not satisfy our requirements. That is because our scenario demands that the sliding window holds in scope events that can later update the state vector of the Worker. They must be stored in a List or Queue in order to preserve their unique keys and they cannot be reduced to a single

value. The percentage of events that can be actually aggregated is too small compared to the number of unique events in scope.

Another issue that emerged when experimenting with Flink's Window API was the inability to evaluate separately the head and tail of the window in scope. When the event time progressed so that a shift had to be performed, the whole window was evaluated and passed downstream to the next operator. This behaviour of course overloads the network because as we stated earlier our aggregated window holds a whole list of events and not a single value.

Custom Sliding Window

After many attempts and experimentation with various operators and java tools we managed to implement a solution that is both accurate and efficient. It consisted of a simple `ProcessFunction`, together with the `timer` functionality and a basic data structure. The implementation was inspired by Flink's blog series named Advanced Flink Application Patterns[6], but modifications had to be made in order to satisfy our requirements.

The main difference of this implementation to Flink's default is that it stores each incoming event only once. It utilizes a `MapState<Long, List<InternalStream>>` as the data structure to hold all the events in scope of the current window. The key of the MapState is the timestamp of the current slide and the value is a Java List containing all the events corresponding to the current slide.

As events arrive, a sequence of utility methods for collecting, managing and evicting events in the window are triggered. The first method to get called when the most current slide is ready to be collected is `collectPreviousSlideEvents()`. It simply collects all the events contained in that slide by iterating through the corresponding List in the MapState. Afterwards the `currentSlideTimestamp` is updated with the next value and a timer is registered to fire at the end of the window. When that timer fires, it means that the elements in that slide are out of scope and they should be evicted with the help of `evictOutOfScopeElementsFromWindow()`. Finally, after the slide management is concluded, the current event gets added to the current slide with `addToState()`.

Validation

A set of experiments have been executed in order to assert the correctness and efficiency of this custom operator compared to the previous solution. As input we used the first part of the WorldCup dataset containing 7 million events (ca. 8 hours). The maximum window size we could test on our local machine was 15 minutes (event-time) with a slide interval of 5 seconds. The test was passing the data through the two implementations described in section 4 of chapter 3, then aggregating the events in the window scope into a frequency vector and finally computing the Euclidean Norm of that vector.

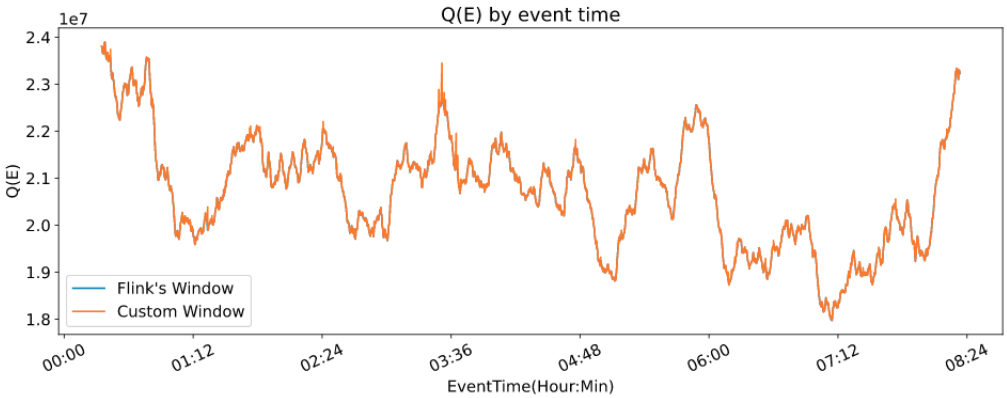

Figure 3.2: Flink's Window vs Custom Window

As we can observe in figure 3.2 the two outputs seem identical but in order to see that even clearer we also calculated their difference percentage and plotted it.

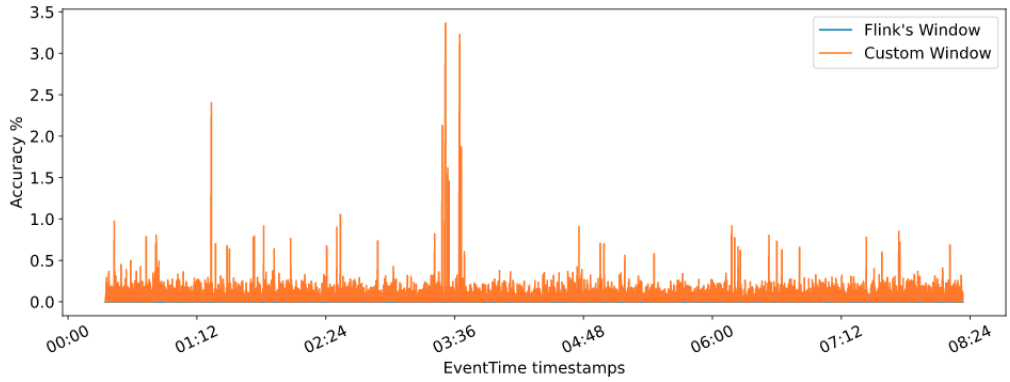

Figure 3.3: Accuracy of Custom Window

The average error is below 0.5% in this experiment and that is a formidable result if we correlate it with the speedup we gained because of this implementation. To be more specific, the experiments described in the evaluation chapter were executed with a window size of 4 hours and a sliding interval of 5 seconds over a total of 50 million events. In contrast, when we tested Flink's window with the same parameters, we got `OutOfMemory` errors after processing less than one million events. Even in the test case described here (15 minutes, 5 seconds), the execution time of the custom implementation was at least 20 times faster than the Flink's implementation.

In conclusion our approach of the sliding-window is a simple operator that manages a queue of events tailored to the needs of this work. It provides an easy and efficient alternative to the default window operator for the scenario of high frequency sliding and large window size.

3.5 Iteration

The FGM protocol requires an upstream communication channel which the coordinator uses to broadcast control messages to the distributed workers. This is a typical scenario in distributed algorithms and is commonly referred to as "iteration" or "feedback-loop". Flink provides the *.iterate*(...) operator for that matter and this implementation was based on it since the beginning. The iterative stream constitutes a major part in Flink's development and at this stage it has not stopped evolving. To avoid minor inconsistencies in our experiments we resorted to another solution, specifically an external iteration using Kafka.

Apache-Kafka and its excellent connectivity with Flink seemed like the place to go for a substitute solution. The iteration was implemented easily using the Kafka-Connector API. The steps followed in replacing the old iterative stream were:

- adding the flink-kafka-conector dependency to maven
- implementing Kafka Serialization and Deserialization schemas
- implementing FlinkKafkaProducer and FlinkKafkaConsumer
- integrating the above into the pipeline

Internally both the consumer and the producer are accessing the same Kafka topic, named "feedback" which is created with a number of partitions equal to the number of workers defined by the user, before the submission of the Flink job. Moreover, the serialization schema is utilizing the *ObjectMapper* class of the *org.apache.flink.shaded.jackson*2 package.

Performance comparison

We also run an experiment on our local setup and tested the two implementations of the iteration mentioned above. Because at IDE-level there wasn't a clear differentiation between the two methods, we also monitored the CPU Load and memory usage using JProfiler. The results as observed in figure 3.4 are quite similar with regard to the memory usage and the GC Activity. The distinction comes at the total execution time which is lower in Flink's implementation and at the CPU load which is evidently higher in the same implementation compared to the Kafka based. For that purpose we consider that these two remarks cancel each other out and therefore the two methods can be accepted as adequate.

In conclusion the two methods of iteration do not differ immensely, but we tend to believe that is also partly due to the Flink update to version 1.10.2. In the past, prior to this version, we came across many inconsistencies while working with the default iteration, hence we decided to run the evaluation of this work using the Kafka based iteration.

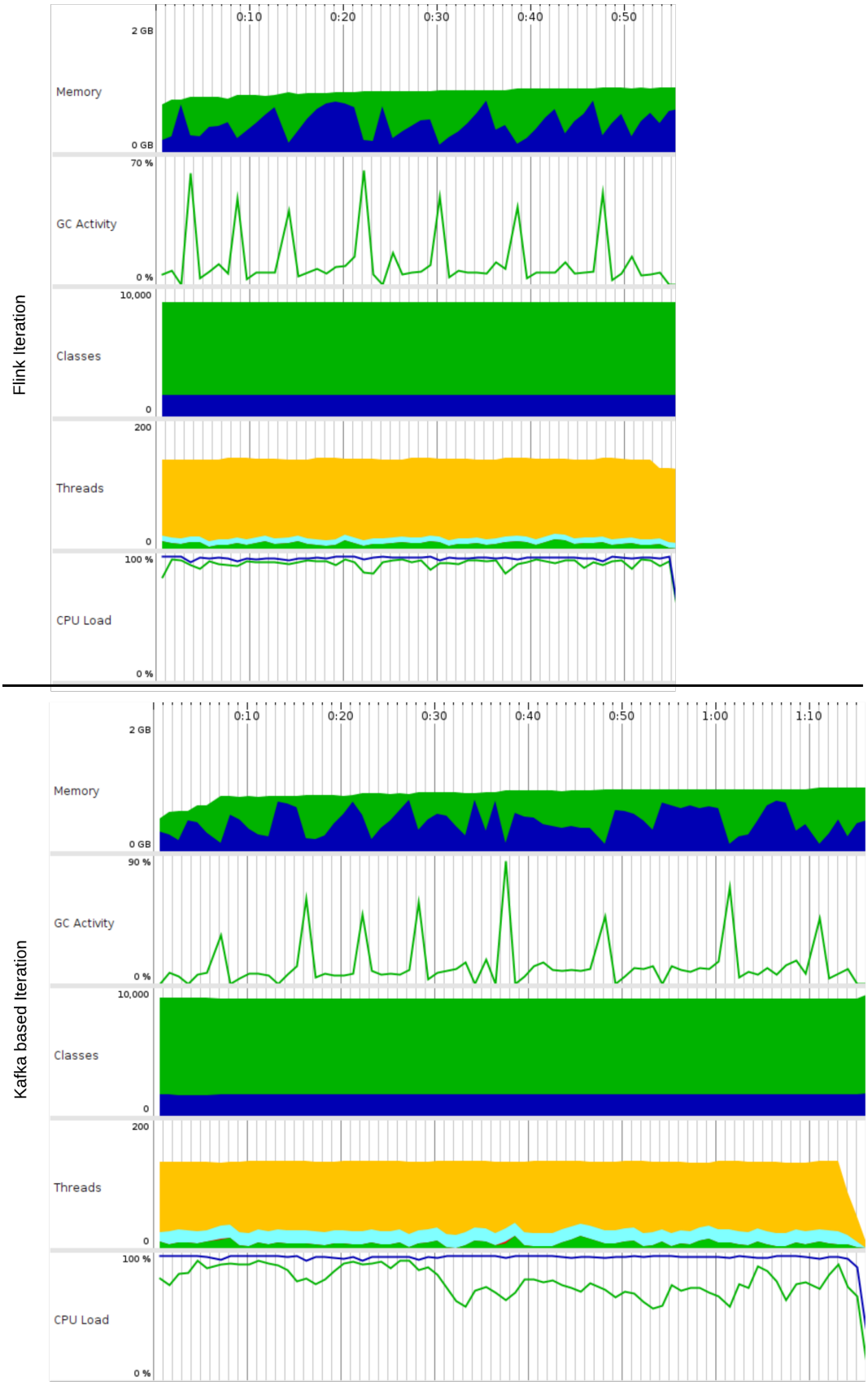

Figure 3.4: Performance comparison of the two iterative implementations. In the memory graphs, the green part is the maximum available heap memory while the blue part represents the currently used heap memory.

3.6 Internals

All the operators of the pipeline use some kind of Java Object to communicate with the other operators. To be specific the input and output of an operator is a DataStream< > object with a custom defined type. Because of the complexity of the monitoring problem and the requirement of minimizing communication between the distributed sites and the coordinator, an equally complex data-structure emerged. As shown in figure 3.5 below the Abstract class *datatypes.InternalStream* is the base class for the rest of the internal stream types. This is also the type that is passed to the high-level DataStream<> objects of the operators. Although, internally in the FGM functions, only certain sub-classes are instantiated depending on the method.

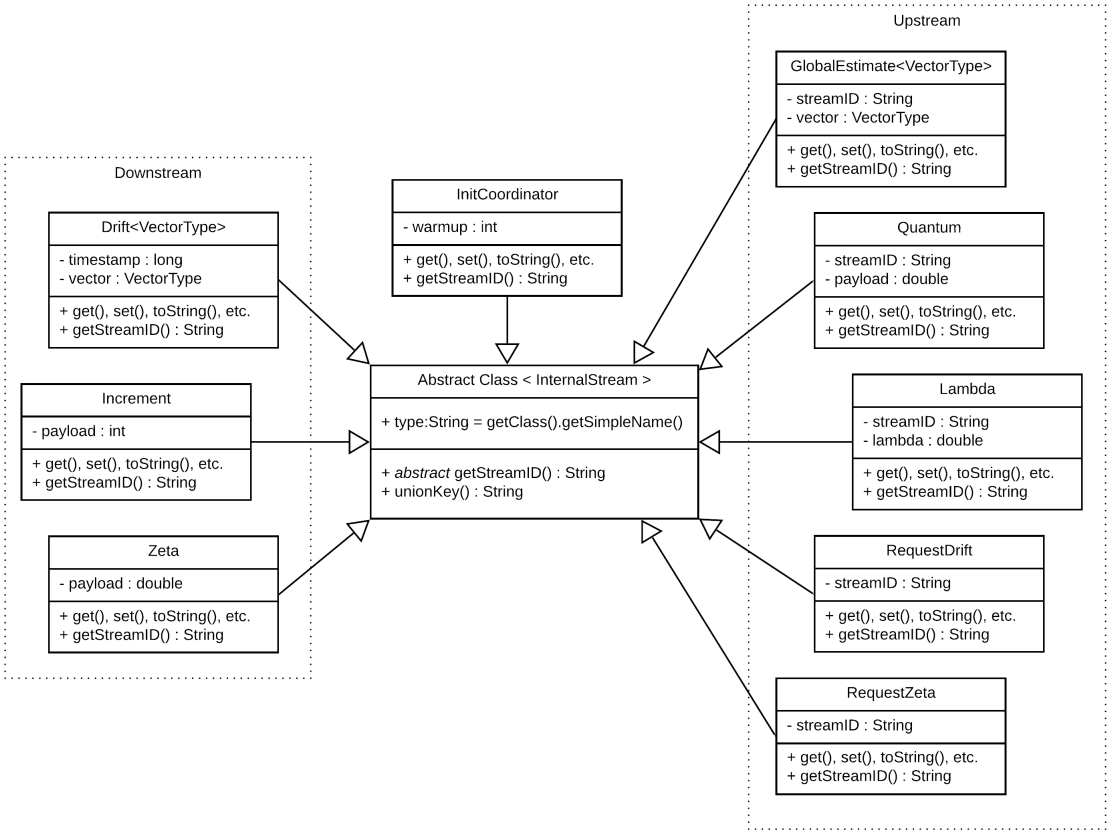

Figure 3.5: Class diagram of the InternalStream data structure

4 Streams Monitoring API

This chapter describes in detail the provided API for the configuration and execution of the Monitoring Job. The methods defined in this interface are called throughout the pipeline of operators in order to embed the user's desired functionality into the existing architecture. In the first section, an overview of the API is provided and its usage is clearly explained. In the following section, some examples of an existing implementation of the interface is shown as used during our experimentation phase.

4.1 API Overview

The Monitoring API is closely connected with the Operators Pipeline, in the sense that simply implementing the interface will not suffice for the execution of the monitoring job. The user also has to use the exact pipeline provided in this work, with the exception of source and sink functions. The interface implementation must be passed as argument to certain operators so that they can utilize the actual implementation functions. We now resume to the discussion about the interface itself.

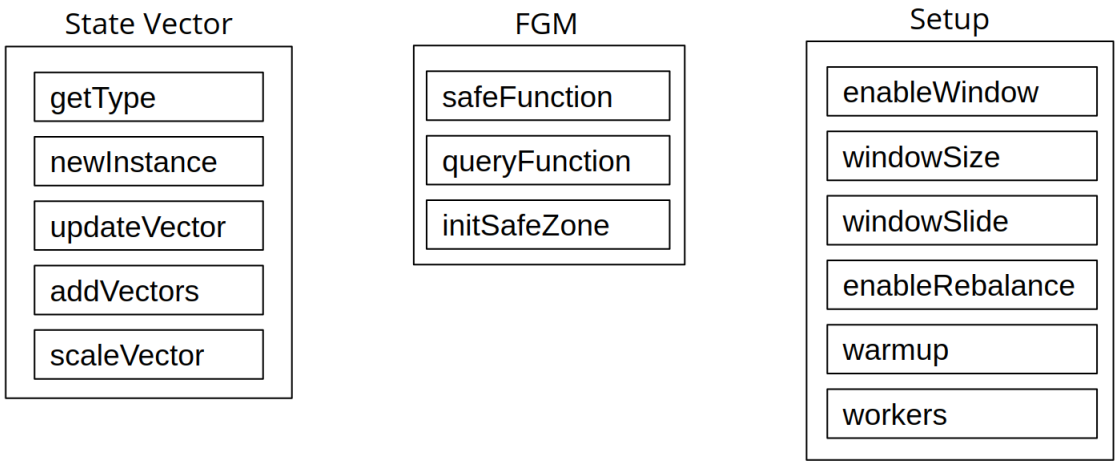

Figure 4.1: The BaseConfig interface is split in three groups of functions.

The user must define their own data structure which will be used as state vector at the Worker and the Coordinator and on which the safe function and the query function will be applied. Of course there is also a set of functions that define the monitoring behaviour.

The `safeFunction` is closely coupled with the `initSafeZone` and together they depend on the `queryFunction`. The example of the next section using AGMS Sketches Safezone should clarify this better. Finally, there is the setup part of the interface, which controls some discretionary features of system, such as data-stream model, startup delay and number of workers/parallelism. The complete documentation of the interface can be found in the appendix.

We now continue the discussion explaining how the interface is integrated in the subroutines of the monitoring protocol. Specifically, in figure 4.2 we describe the chain of function calls necessary for the simple operation of updating the Drift Vector of a Worker with an incoming event.

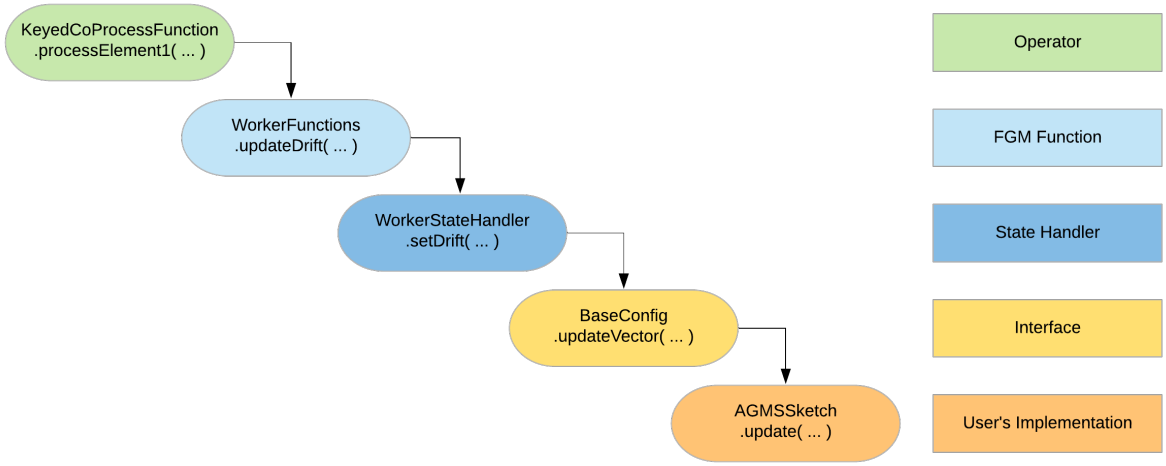

Figure 4.2: An example of the stack of functions needed for updating the drift vector at the Worker process.

As a new event arrives at the Worker's ProcessFunction, the appropriate FGM function will be called. Next, the FGM function `updateDrift` will retrieve the current state vector from the state handler and together with the incoming event will pass them as arguments to the interface's method `UpdateVector`. The interface's method is just a reference and therefore the user implemented function will be called, which is expected to update the given state vector with the event and then return the vector. Finally, all the methods will exit recursively and the `processFunction` will proceed its execution.

State Management

The state vector internally is stored in a ValueState object managed my Flink. Its type is generic so that the user can define their own type and methods for managing the vector. The choice of using ValueState instead of the MapState for the vector was made in order to provide more freedom to the user, despite being less optimal due to the CPU cost of the serialization and deserialization of the state object.

4.2 Examples

In order to test the implementation and later to experiment with it and extract results, we needed a complete implementation of the API. Therefore in this section the `AGMSConfig` is presented, an implementation of the `BaseConfig` interface built around the AGMS Sketch data structure. The AGMS Sketch data-type is also provided as part of this API and it contains many useful methods that can be used out of the box.

Some of the simpler methods will be omitted from this demonstration since their implementation is self-evident and the whole implementation can be found on the public GitHub repository. To begin with, the `updateVector` method is explained.

Vector update

The AGMS sketch is updated as described in the second chapter of this work. In the demo implementation, we simply extract the key and the value from the incoming event and then pass them as arguments to the update method of the given AGMSSketch vector. Internally, the relevant hash bucket is selected by the h hash function. The value of the event is scaled by the ξ hash function and the result is added to the existing value of the selected bucket. This process is repeated D times where D is the depth of the sketch.

Vector addition

The addition of two AGMSSketch vectors is very simply implemented. Since the data structure is a 2-dimensional array, we iterate through it with the help of two for-loops and add their elements one by one.

Safe Zone and Function

The safe zone we employed depends on the query function we selected for our experiment. In this case we monitor the self-join size of the estimate sketch E with depth D and width W which is

$$Q(E) = \underset{i=1,\ldots,D}{median}\{\sum_{j=1}^{W} E[i,j]^2\} = median\{E[i]^2\} \tag{1}$$

We split the safe zone into two parts, the lower $\phi_i^-(x)$ and the upper bound ϕ_i^+ respectively, where

$$\phi_i^-(x) = \frac{E[i]}{\|E[i]\|}(E[i] + x) - \sqrt{T^-} \tag{2}$$

$$\phi_i^+(x) = \sqrt{T^+} - \|x + E[i]\| \tag{3}$$

and their thresholds are $T^{\pm} = (1 \pm \varepsilon)|Q(E)|$. We then proceed to represent the $\phi^{\pm}$ functions of each sketch row as predicates of a Boolean function. This function is considered safe if only the majority of predicates are true. We calculate the majority of predicates by taking the logical conjunction of all the disjunctions of predicates. The clause which represents the disjunction of predicates is further reduced from D to (D+1)/2. This way each clause will be true if at least one predicates of the clause is true, since (D+1)/2 is the majority of D elements. Of course the admissible region is stricter by this reduction but it is allowed under the maximum-distance theorem 10 of [7].

The final safe function is the union of the lower and upper bounds

$$\phi(X) = min(\phi^{-}(X), \phi^{+}(X)) \tag{4}$$

where each of them is computed with the above methodology and can be described by

$$\phi^{\pm}(X) = \min_{I \in \binom{D}{(D+1)/2}} \frac{\sum \|\phi_i^{\pm}(0)\| \cdot \phi_i^{\pm}(X[i])}{\sqrt{\sum \|\phi_i^{\pm}(0)\|^2}} \tag{5}$$

like it's described in section 5. SafeZones and Boolean functions of [7].

The function `initSafeZone()` of the interface precedes the invocation of `safeFunction()`. At the beginning of an FGM round the safe zone is initialized and prepared with the newly updated global estimate. Firstly at the coordinator and secondly at the sites, after the shipping of the global vector to them is complete. In our example, the denominator of the safe function above is precomputed and cached by the `initSafeZone` so that it speeds up the computation of the `safeFunction`.

5 Evaluation

The objective of the experimental evaluation is to demonstrate that the implementation of FGM on a real-world framework like Apache Flink can produce exceptional results as far as scalability, throughput and communication cost gain are concerned. We also aim to show that our implementation has practical use for the problem of distributed query monitoring and for that reason we provide the Monitoring API for further experimentation.

5.1 Setup

In compliance with the initial implementation of the FGM[9] on a simulated distributed system, we used the same data set throughout our experiments. Namely the WorldCup '98 data set[4] which contains http requests logs of the World Cup's website. The logs are gathered from a total of 33 mirrors and span over 87 days. For this thesis, the 46th day of the dataset has been used, which contains ca. 50 million requests over 27 mirrors sites. The compressed logs were extracted according to the structure found in the dataset's documentation.

In particular only 4 of the extracted fields are of importance to this particular set of experiments. The *timestamp* was used to aid Flink in handling the event-time notion. The *server* which represents the mirror's id, was used as a key while partitioning the input streams. Finally, the *clientID* together with the request *type* formed the schema of the input records *R(clientID,type)*.

The evaluation of this implementation was made possible by running Flink as a standalone-cluster along with a Kafka single node cluster on a linux server. The setup was composed of 12 core Intel Xeon E5-2650v2 CPU @ 2.6 (up to 3.4 GHz) and 8 x 8 Gb of DDR3 regECC RAM. The linux distro was Ubuntu 18.04 LTS. More precisely, the Flink Cluster consisted of 1 JobManager and 5 TaskManagers with 4 Task slots each in order to submit jobs with parallelism up to 20.

We executed a few sets of experiments with different parameters to get a better idea of the accuracy and scalability of the system. In total 40 jobs were submitted to the flink cluster and a total of 160 log files were extracted containing various metrics. The logs were then further processed and plotted in Jupyter.

The parameters that were tweaked between experiments were:

e: the monitoring error [0.1, 0.5]
k: the number of workers [2, 4, 8, 12, 20]
rb: fgm rebalancing [true, false]
model: the datastream model [sliding-window, cash-register]

while the constants of the configuration were:

AGMS Sketch size: 7 x 3000
window size: 4 hours (event-time)
window slide: 5 seconds (event-time)
warmup: 1 minute (event-time)
safe function: AGMS Sketch Self-Join safezone (median quorum)
query function: AGMS Sketch Self-Join size

5.2 Results

Throughput

In the following figures we address the throughput of the system. Specifically we have plotted the total number of tuples processed by each worker against the processing time of the system. This particular figure is very similar amongst all the experiments with the sole difference being the distribution of data to the workers due to the hashing function at the source. As stated in a previous paragraph, the number of unique streams in this dataset is 27 but they are being hashed to $k \in [2 : 20]$ streams in order to study the effect of the distributed computation.

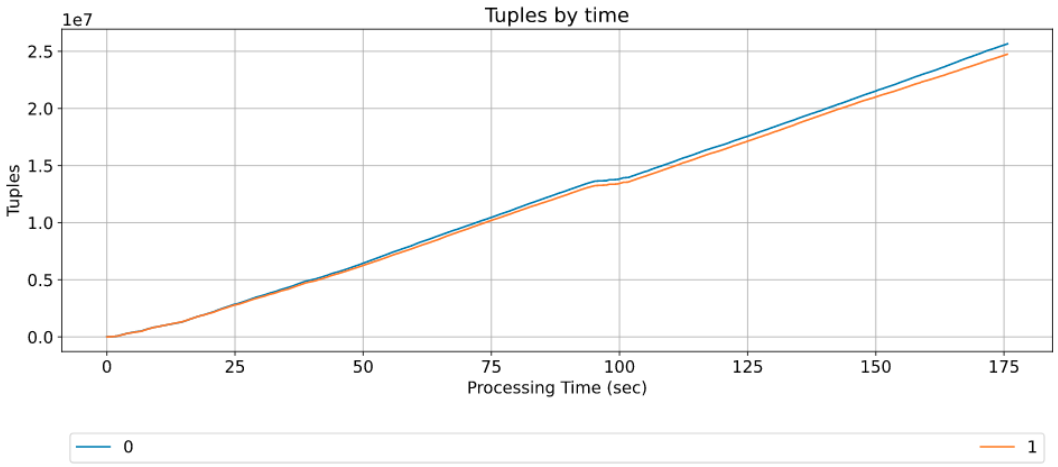

Figure 5.1: Parameters : e = 0.1, k = 2, w/o rebalancing, cash-register

Figure 5.1 shows that the load is split evenly between the two workers but as k increases there should be a clear imbalance among the tuples processed by the workers because we are experimenting on a real-world dataset. Figure 5.2 below, proves our assumption correct.

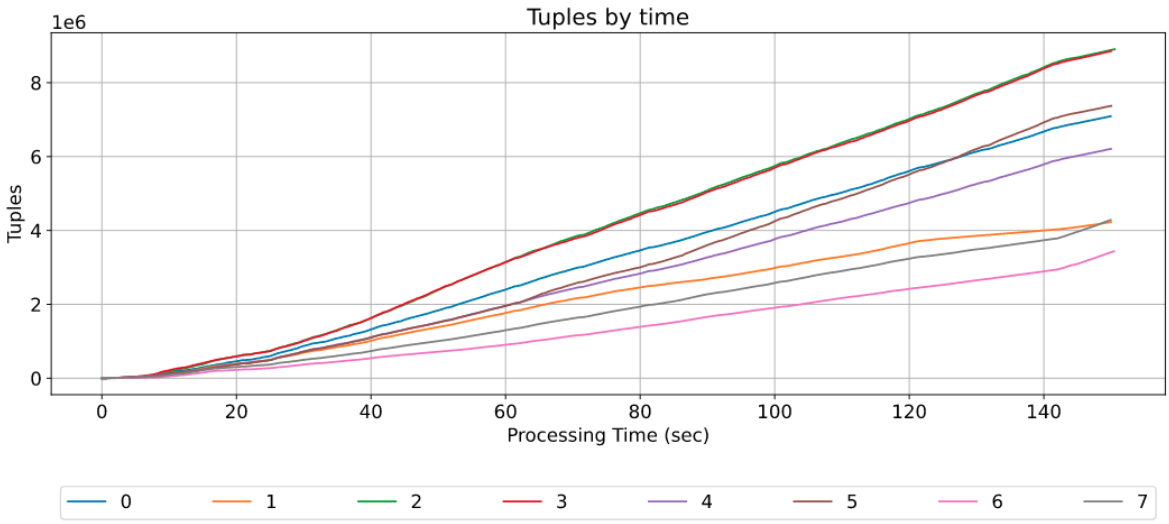

Figure 5.2: Parameters : e = 0.1, k = 8, w/o rebalancing, cash-register

Focusing on this particular experiment with k = 8 we analyse the behaviour of the system. In order to understand better what is happening we also plotted the number of tuples divided by the time. Now it is much clearer how each worker processes the incoming tuples. The gray vertical lines here represent the FGM rounds of the experiment. As expected, at the beginning of the experiment the system synchronises more often than later in time. That happens due to the variability of the data in the local state vectors in comparison with the global state.

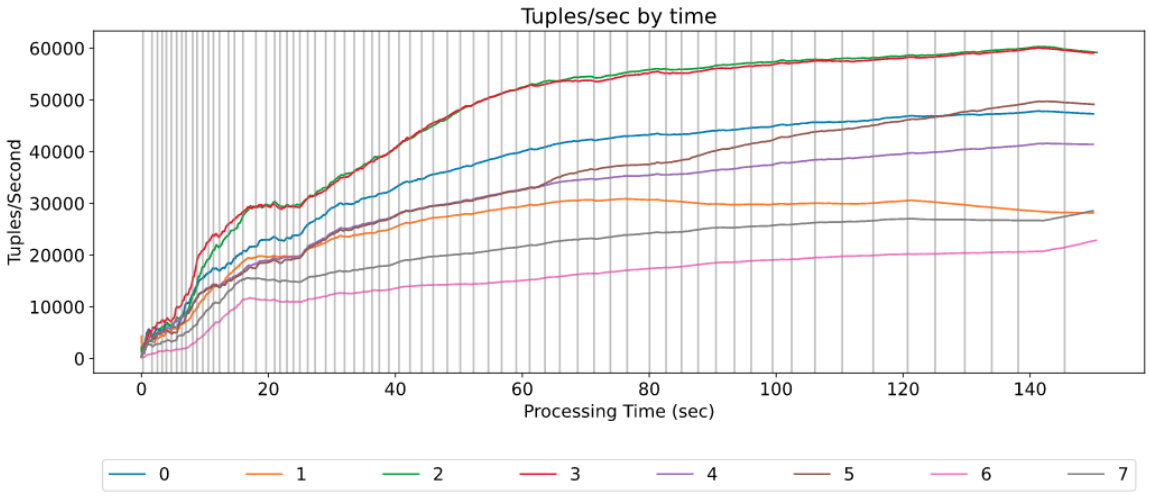

Figure 5.3: Tuples per Second and FGM Rounds

We also plotted the instantaneous throughput of the same experiment with a scope of 10 seconds. This shows that the system increases its throughput and maintains a high value even under high variability. Naturally there is a delay between the time when a round happens and the time of the update of events that caused that round sync. Precisely, at 60 seconds in the figure below we observe denser synchronisation but if we look a few moments back, almost all workers have an increasing throughput rate. This pattern can also be observed at 100 and 120 seconds.

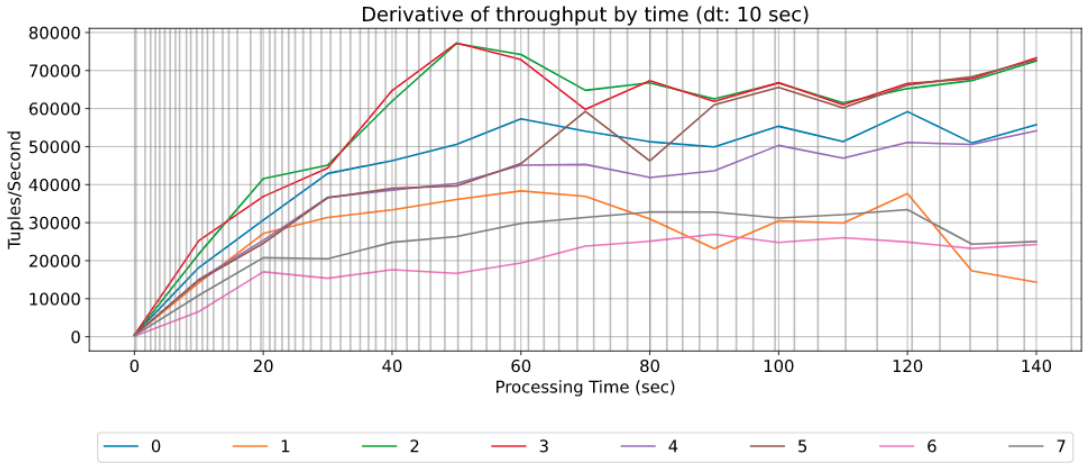

Figure 5.4: Derivative of throughput, slope analysis

Another interesting figure that reflects the resilience of FGM to the dispersion of data at certain times is the following. In Geometric monitoring if a local drift vector deviates from the global state too much, then logically a violation report happens and sequentially a synchronisation follows. FGM on the other hand, has an additional layer of protection against unnecessary synchronisation, namely the sub-round. In figure 5.5 it is shown that during rounds 30 to 45 there are a lot of subround violations (increments), but if we look at the corresponding period in figure 5.4 (approx. 30 to 60 seconds) the throughput is increasing, which proves this point.

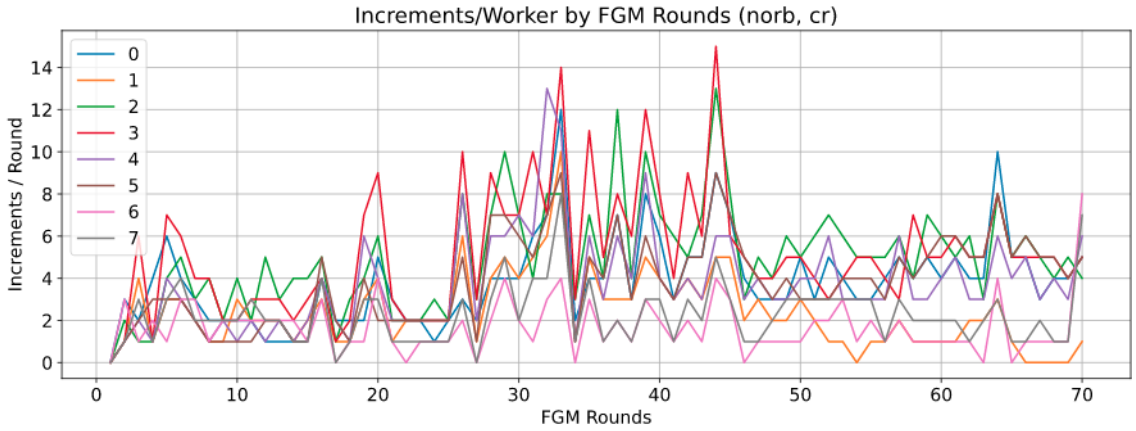

Figure 5.5: Increments/Worker by FGM Rounds

Scalability

As for the scalability of the system, we expected that the total execution time of the job would decrease as k increased. Unfortunately the results were not as good as expected.

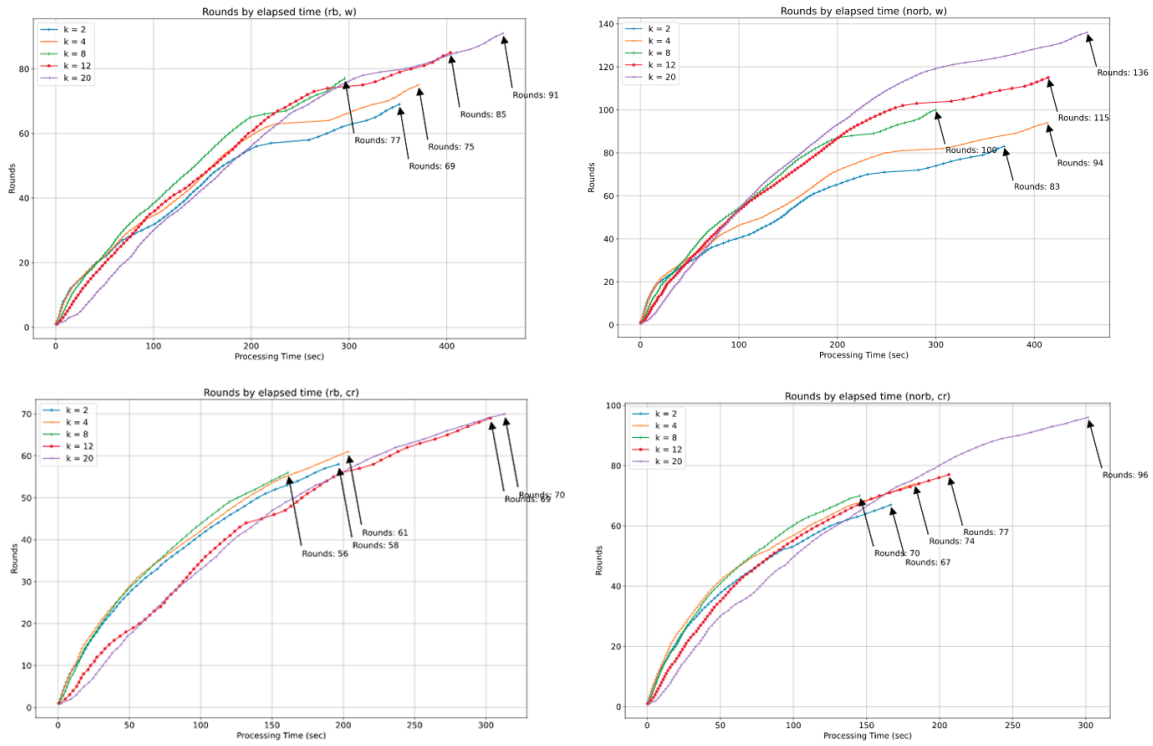

Figure 5.6: FGM Rounds/Worker by time. Parameters : e = 0.1

If we take a closer look at the figures above, we can observe a pattern as far as the total execution time is concerned. In all cases the shortest time is achieved by the experiments with $k = 8$, following $k = 2$ and $k = 4$, then $k = 12$ and finally $k = 20$. For simplicity we will refer to these experiments as $k2, k4, k8$ etc..

It is also a known fact that the Flink Job Manager distributes the workload in a certain manner. For example $k2$ and $k4$ can be executed only using the 4 slots of a single Task Manager. Furthermore, our evaluation setup consisted of 12 CPU cores and the fact that $k12$ and $k20$ have much bigger execution times indicates in our opinion that the overhead of processing and network communication is quite substantial. Under these circumstances, the cause of this pattern it is apparent, and therefore we conclude that $k8$ experiments were the most efficient with this particular configuration.

On the bright side, with rebalancing enabled the number of rounds decrease in both the sliding-window and the cash-register scenario. This behaviour is also expected since the rebalancing mechanism instructs the coordinator to delay the Round finalization by sending a scaling factor λ to the workers instead of the whole updated safe zone.

By bumping the monitoring error to 0.5 we expect the safezone to be more tolerant to the high variability of the incoming data and therefore reduce the total number of rounds.

In figure 5.7 below, this comportment is depicted clearly as the number of rounds of each experiment is amortized by a factor of 3 compared to the previous experiments with $e = 0.1$.

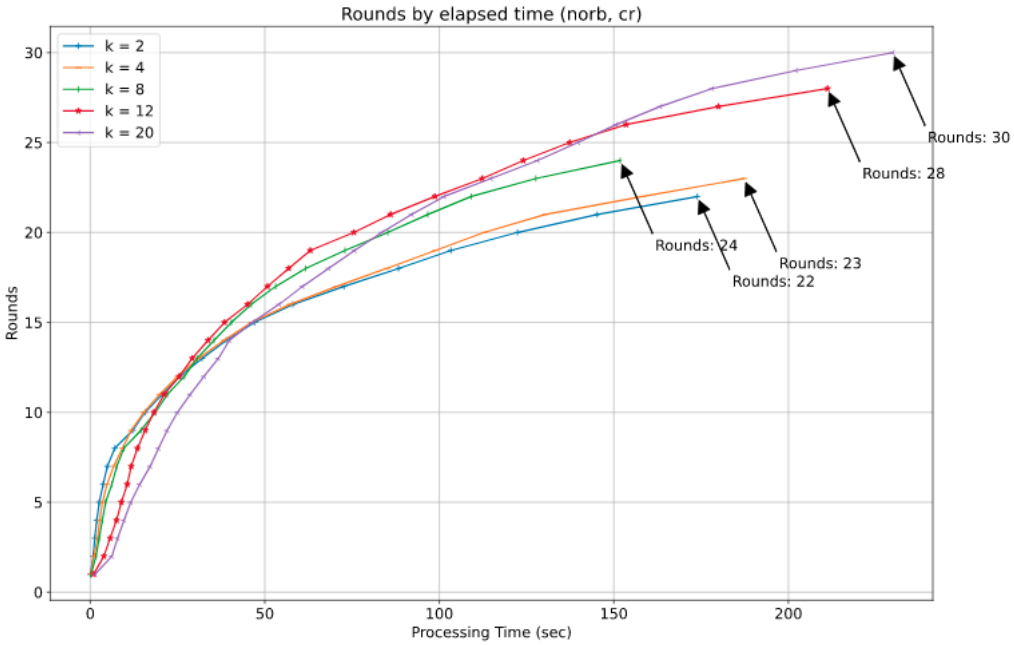

Figure 5.7: FGM Rounds/Worker by time. Parameters : e = 0.5

In conclusion the system did not scale appropriately when increasing the parallelism of the job, but we believe that is because of the default load balancing of Flink. We did not opt for a solution with stream reshuffling or uniform data distribution since our goal was the reduction of communication cost. This idea might have offered better scalability overall but it would definitely overload the network buffers of the Flink cluster.

Another advantage of our simplistic approach is that with the communication cost gain and the fairly good scalability results, there are more resources left to the cluster for allocation to other jobs running in parallel.

Communication Cost

Part of the objective of this experimental evaluation was to provide evidence about the communication cost gain of FGM on a real world framework. We calculated the traffic(MB) of various phases/entities of the experiments, namely the volume of Input events and the traffic size of the rounds and the subrounds. We have plotted below on linear and logarithmic scale the Traffic plots of several experiments with the cash-register and the sliding-window models.

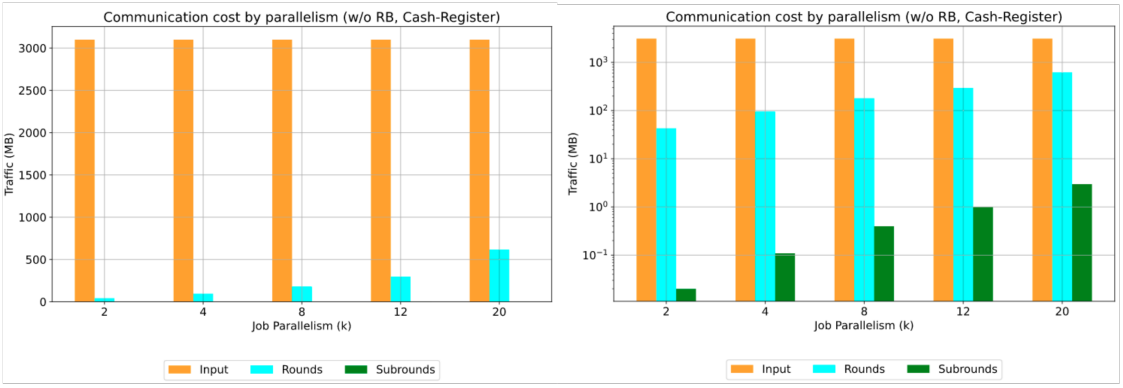

Figure 5.8: Communication cost with Cash-Register. Linear scale on the left, logarithmic scale on the right.

We observe in figure 5.8 that the traffic size of the FGM protocol compared to the volume of Input events is much smaller even with the highest parallelism. The traffic of FGM consists of all the messages exchanged between the workers and the coordinator, with the largest percentage of traffic being held by the downstream Drifts and the upstream Global Estimates which together compose the Rounds.

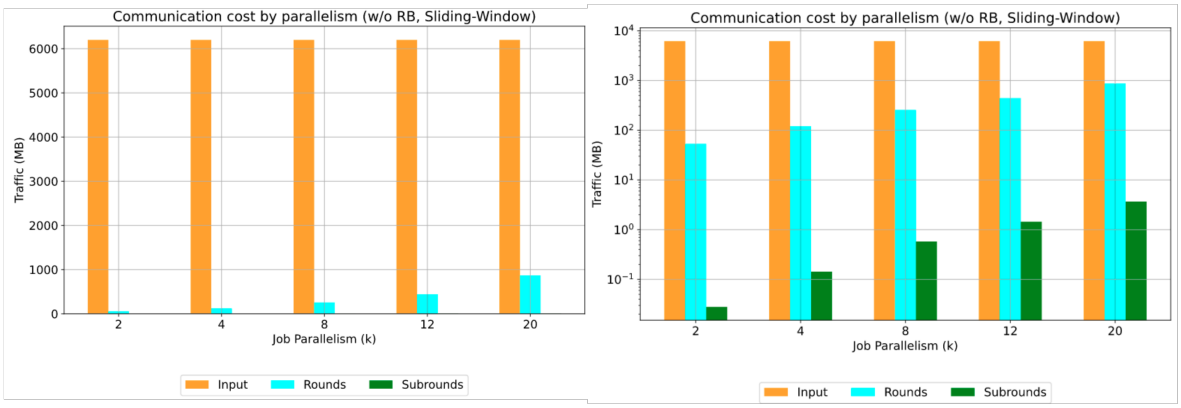

Figure 5.9: Communication cost with Sliding-Window. Linear scale on the left, logarithmic scale on the right.

The difference is even greater if we take a look at the experiments with Sliding-Window. That is because the sliding-window operator produces twice as many events in order to maintain a constant size. Despite the size of the Input, we notice that the FGM Traffic remains almost unchanged.

In conclusion the evaluation shows that the desired communication cost gain has been achieved.

6 Conclusion

In this work we implemented the Functional Geometric Monitoring protocol on a real-world stream processing framework, Apache Flink. Furthermore we provided a generic implementation and an interface that makes this work suitable for future experimentation and extension in a simple manner. Apart from the core monitoring algorithm, other features are also supported such as the Fast-AGMS Sketch data-type and its related functions, a custom implementation of the sliding-window operator, as well as the use of Kafka-based iteration. The performance of this program satisfies the theoretical assumptions as far as throughput, accuracy and communication cost are concerned but unfortunately it does not fully satisfy the scalability aspect.

An important aspect that has not been addressed by this work is the integration and testing of Flink's checkpointing mechanism. Its ability to maintain the only-once semantic and its fault tolerance capabilities would make this an excellent objective for future work.

The optimization of safe zone shipping in the case of uneven distribution of streams is introduced in the FGM Protocol[9]. Unfortunately this idea did not materialize in the course of this project but it is undoubtedly an important goal for further work.

The last objective and easiest maybe to address is the expansion of the reduce mechanism at the coordinator. At the moment only vector averaging is possible because the support for other reduce functions had not been one of the requirements in the initial design phase. Nevertheless, many other *reduce* techniques are used in this kind of distributed processing (max, sum, or, etc.) and therefore this feature would be a significant addition.

Bibliography

[1] Apache flink — stateful computations over data streams, 2020. https://flink.apache.org/.

[2] Apache kafka — a distributed streaming platform, 2020. https://kafka.apache.org/.

[3] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. Journal of Computer and System Sciences, 64(3):719–747, 2002.

[4] M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup web site. IEEE network, 14(3):30–37, 2000.

[5] G. Cormode and M. Garofalakis. Sketching streams through the net: Distributed approximate query tracking. In Proceedings of the 31st international conference on Very large data bases, pages 13–24, 2005.

[6] A. Fedulov. Advanced flink application patterns vol.3: Custom window processing, July 2020. https://flink.apache.org/news/2020/07/30/demo-fraud-detection-3.html.

[7] M. Garofalakis and V. Samoladas. Distributed query monitoring through convex analysis: Towards composable safe zones. In 20th International Conference on Database Theory (ICDT 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[8] R. Metzger. Kafka + flink: A practical, how-to guide, 2015. https://www.ververica.com/blog/kafka-flink-a-practical-how-to.

[9] V. Samoladas and M. N. Garofalakis. Functional geometric monitoring for distributed streams. In EDBT, pages 85–96, 2019.

[10] I. Sharfman, A. Schuster, and D. Keren. A geometric approach to monitoring threshold functions over distributed data streams. ACM Transactions on Database Systems (TODS), 32(4):23–es, 2007.

[11] J. Traub, P. M. Grulich, A. R. Cuellar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl. Scotty: Efficient window aggregation for out-of-order stream processing. In 2018 IEEE

34th International Conference on Data Engineering (ICDE), pages 1300–1303. IEEE, 2018.

Appendix

BaseConfig interface

The interface's signature contains a generic parameter type called *VectorType* which is used as parameter or return type in various abstract methods. This type represents the data-type of the state vector. This section describes 3 groups of abstract methods, namely *State Vector, FGM and Setup*.

State Vector

The first group of abstract methods is allocated to state-vector specific operations. In the distributed monitoring problem the state vector represents the data structure that accumulates incoming stream events at the distributed sites. An example for such a data structure is a `HashMap<K,V>` which holds unique keys of events and a counter for the times that each key has appeared in the stream. At the coordinator, the state vector has the role of holding the reduced global state.

signature: `getVectorType`
parameters: -
return type: `TypeInformation<VectorType>`
description: A type hint used by Flink when defining ValueState objects of type VectorType. This is basically the state vector mentioned earlier and it is a crucial component of the distributed monitoring architecture.

signature: `getTypeReference`
parameters: -
return type: `TypeReference<GlobalEstimate<VectorType>>`
description: A type hint used by Kafka-Connector when de-serializing feedback messages of type `GlobalEstimate<VectorType>>`. Only called when using Kafka-based iteration.

signature: `newVectorInstance`
parameters: -
return type: `VectorType`
description: A new instance of `VectorType`. It is used as initial value for the state vector. To be more specific, during the initialization phase of Flink's `ValueState<T>` object, a `null` value is assigned to that object. Therefore it is the user's responsibility to check the object before using it. This method will be called each time the state vector's object has a `null` value and it will replace it with an empty state vector. e.g. `return new HashMap<K,V>();`

signature: `addVectors`
parameters: `VectorType` vector1, `VectorType` vector2
return type: `VectorType`
description: Point-wise vector addition. The resulting vector should be a new instance of `VectorType` with elements the sum of each of the elements of vector1 and vector2. The arguments of this method should be treated as **immutable**.

signature: `scaleVector`
parameters: `VectorType` vector, `Double` scalar
return type: `VectorType`
description: Multiplies every element of the given vector by a scalar. The arguments of this method should be treated as **immutable**.

signature: `updateVector`
parameters: `InternalStream` inputRecord, `VectorType` vector
return type: `VectorType`
description: It updates the given vector with an input-stream record. The vector in this case should be treated as **mutable** and should be returned after the update. The user's input-event class should extend InternalStream class and then use casting in order to access its variables in this method.

Functional Geometric Monitoring

The second group of abstract methods are FGM related operations. These are called by the protocol's functions and their role is to allow the user to tweak various components of the monitoring algorithm.

signature: `safeFunction`
parameters: `VectorType` drift, `VectorType` estimate, `SafeZone` safeZone
return type: `double`
description: The safe function is found at the core of the FGM algorithm. It is used both in the distributed sites and the coordinator as a way to ensure system's safe configuration. It is called by the workers' FGM functions every few updates of the Drift vector and by the coordinator's FGM functions when checking ψ for violation. The arguments should be treated as **immutable** objects. They are also expected to be `non-null` even when empty. The third argument can be **ignored** if the implementation is simple enough not to require an initialization of the safe zone or for any other reason. Otherwise, `initializeSafeZone()` should be implemented properly before calling `safeFunction()` with a working safe zone argument.

signature: `initializeSafeZone`
parameters: `VectorType` global
return type: `SafeZone`
description: As mentioned in the description of `safeFunction()`, this method is used when initializing complex safe zones. For example the safe zone for the self-join size query of the AGMS Sketch must be prepared beforehand in order to speed-up the execution of the safe function call. This method is only called once per round in contrast to the safe function which is called every few drift updates. The only parameter of this method is the global state vector because that is usually needed during initialization. The user's custom safe zone should extend the `SafeZone abstract` class.

signature: `queryFunction`
parameters: `VectorType` estimate, `long` timestamp
return type: `String`
description: The query function is applied on the global state vector at the coordinator. The function provides the state vector and the timestamp that corresponds to that state. After computing the query function it is expected to return a `String` value which will then be directed to a side-output. This method is called at the end of an FGM Round.

Setup

The models supported by this implementation are *Cash-Register* and *Sliding-Window*, but the interface only defines methods for the second model. That is because the first one is the default behaviour of the pipeline. It is worth mentioning that the 3 methods below all have default implementations and can be left not overridden.

signature: `slidingWindowEnabled`
parameters: -
return type: `boolean`
description: If the returned value is `false` the sliding window operator will be omitted from the execution plan and the streaming model will be of type `Cash-Register`. If the returned value is `true` then the window operator will also call `windowSize()` and `windowSlide()` internally in order to function properly. The default return value is `false`.

signature: `windowSize`
parameters: -
return type: `org.apache.flink.streaming.api.windowing.time.Time`
description: Should return the event-time *size* of the window. It will only be called in the sliding-window scenario. The default return value is `Time.hours(1)`.

signature: `windowSlide`
parameters: -
return type: `org.apache.flink.streaming.api.windowing.time.Time`
description: Should return the event-time *slide* interval of the window. In contrast to the `windowSize()` method, this one is also called by the worker function when using the cash-register scenario. The reason behind this is that the `subRoundProcess()` is too expensive to be called after each drift update. Therefore, a delay of `windowSlide` seconds is introduced allowing a few events to accumulate before computing the safe function. The defualt return value is `Time.seconds(5)`.

signature: `workers`
parameters: -
return type: `Integer`
description: This method indicates the number of worker instances the distributed system will have. It is used as a hashing factor when partitioning the input stream, as operator-level parallelism in the main pipeline and also as a number of distributed sites in the FGM protocol. For example, if `workers()` returns 10, then the input streams will be partitioned based on keys in the range 0 to 9. The instances of all the distributed components of the pipeline will also be 10, one for each of the keys. The coordinator will also broadcast its messages to 10 sites.

signature: `warmup`
parameters: -
return type: `long`
description: Should return a timestamp at which the coordinator will request the first Drift vectors. One common usage for this method is to delay the start of the system until the sliding-window is full. In order to use it properly the user should know the first timestamp of the data-set and add to that the desired event-time warmup delay in milliseconds.

signature: `rebalancingEnabled`
parameters: -
return type: `boolean`
description: It enables/disables the rebalancing mechanism of FGM. The only supported rebalancing method is the bimodal which means that λ factor and sequentially μ will be set to $\frac{1}{2}$ during the rebalancing period. The period is also defined implicitly so that only 1 rebalanced round is allowed every 1 normal FGM round.

signature: `getMQF`
parameters: -
return type: `double`
description: The Monitoring Quantization Factor is scalar by which the safe function is multiplied at the coordinator when checking ψ condition. By default it will return 0.01 as the interface provides a default implementation. For that reason it will not trigger any compile-time errors if not overridden.