ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

UNIVERSITY OF PIRAEUS

Complex Event Recognition for Maritime Surveillance

MASTER THESIS

Λάμπρος Καραγεώργος
MPSP 13042

Επιβλέπων Καθηγητής : κ. Ιωάννης Θεοδωρίδης

SEPTEMBER 2017

Acknowledgements

I would like to thank all my family and my supervisor Professors: Yannis Theodoridis, Nikos Pelekis and Alexander Artikis.

Contents

ABSTRACT

The main scope of this Master thesis is to analyze and design an innovative technological solution for Complex Event Recognition for Maritime Surveillance purposes, based entirely on the approach presented in the Paper "**Event Recognition for Maritime Surveillance**" by Kostas Patroumpas, Alexander Artikis, Nikos Katzouris, Marios Vodas, Yannis Theodoridis and Nikos Pelekis in the context of the AMINESS project. The master Thesis aims to tackle the challenge of processing and analyzing the available AIS Data sets in real time using Apache Flink. Apache Flink is a real time high-performance and accurate natural Stream Processing Engine from Apache Software Foundation. The ultimate goal is to inspire Maritime authorities to develop their digital culture and empower their ICT departments with a new big data innovative tool that uses the existing Paper's algorithms and semantics in an intelligent way, so as to detect vessel's Trajectories in the Aegean Sea while performing accurate Complex Event Recognition. The technical approach and the effective reasoning of complex events is totally based on the business logic of RTEC and the Event Calculus formal language semantics. We are going to map these semantics into Apache Flink DataStream and Dataset API and create an efficient alternative technical approach.

1.Introduction

1.1 Motivation - Real time analytics for Big Data

From the dawn of civilization until 2003 humankind generated five Exabyte's of data. Now we produce five Exabyte's every two days and the pace is accelerating. It is expected that by 2020, the amount of digital information in existence would have grown to 40 zettabytes. It is a growing need for new data management systems and distributed architectures not only to handle and store a wide variety of data coming from IOT sensors, Portals, Logs, Mobile apps but to analyze them and extract complex information in real time for the benefit of the government and the authorities. In our day's data scientists and ICT companies have already faced this growing need for improved and advanced analytical capabilities that extract information from the huge volumes of varied data in real time and create added-value services. These needs lead for optimized hardware appliances and software platforms ranging from multi-core processors to distributed computing and cloud storage infrastructures that will offer optimized performance of complex queries and will enable complex algorithms to run just in a few minutes.

1.2 Approach - Apache Flink

In December 2014, Apache Flink, a dedicated stream processing engine started at Technical University Berlin and was accepted from Apache Foundation as an Apache top-level project. Apache Flink introduced in the community as the high-throughput and low-latency natural stream processing engine which comes with very rich DataStream API and new Libraries. The presented technical approach of the Trajectory Detection and Complex Event Recognition concepts is based on Apache Flink API and its CEP Library using the event-at-a-time rather than batches of data –an important distinction from previous streaming approaches - which turns the current master thesis into an innovative service for the maritime community. Taking into consideration that Apache Flink ecosystem has been already deployed from Big IT companies in high performance Production environments we will aim to apply its API features and libraries for the benefit of Shipping industry and furthermore for the Maritime surveillance.

1.3 Contributions

The first micro-service that we are going to implement with Apache Flink is the Trajectory Detection Module. The second is the Complex Event Recognition Module. The third module is an approach for integrating Flink with Elastic and Kibana. We are going to visualize the given AIS data and share them through dashboards in order to make vessel's information easily accessible to maritime industry and create geodata reports and graphs so as the authorities can find their own answers to critical questions easily within a few seconds.

1.3.1 Trajectory Detection Module

The paper "How not to drown in a sea of information: An event recognition approach. Elias Alevizos, Alexander Artikis, Kostas Patroumpas, Mario's Vodas, Yannis Theodoridis, Nikos Pelekis" introduced the notion of critical vessel movement events which can be cleverly extracted from a stream of raw AIS elements. These vessel's critical movement events include the following types: *slow speed, speed changes, speed acceleration, communication gap, vessel turn,* and *vessel pause*, which are summarized in the following paper's **figure [1]** and we are going to implement with Apache Flink.

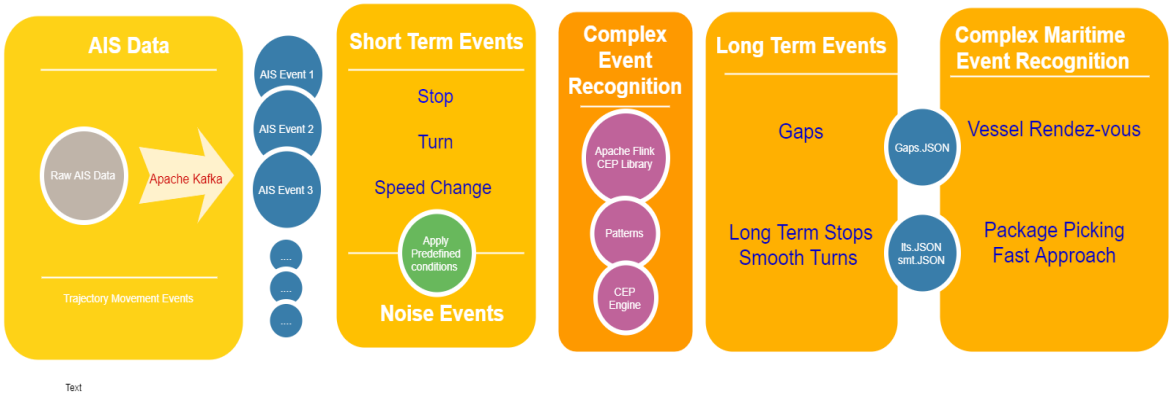

figure [1]

Trajectory Detection Module is the main module of the implementation since the Complex Event Recognition module is built on top of it. It is implemented as an Apache Flink stream processing pipeline which is totally based on the Event Calculus formalism, the business logic algorithms and semantics presented in the Paper "*How not to drown in a sea of information: An event recognition approach*". The Apache Flink pipeline is composed of a number of chained operators, each having multiple instances for parallel processing. Apache Flink composes the execution graph through which the data are processed in a streaming fashion.

As described in the Paper, a key problem in real time processing implementations is the detection of event patterns in data streams. By default, the streaming nature of Apache Flink make it suitable for handling this type of processing, which is facilitated by the DataStream API, and the Complex Event Pattern Library provided. The dataset, we are going to use is imis-1month dataset from **http://chorochronos.datastories.org** . We will model AISdata into AISEvents and tag them into two different categories of 'vessel trajectory events':
The category of the Short-term critical movement events includes the detected *vessel stops, turning points, slow motions, vessel accelerations, noise events.*
The category of Long-term Events includes sequentially processed critical movement events with the usage of the Flink CEP library in order to detect sequences of patterns, and extract information about Long Term Events like: *Long-term vessel stops, vessel Gaps and vessel smooth turns Events*.

1.3.2 Complex Maritime Events Recognition Module

The Complex Event Recognition module consumes the output of the Trajectory detection module so as to process the results and recognize in real time potentially complex maritime situations for preventing too complicated situations and for Maritime intelligence purposes also. According to the paper "*How not to drown in a sea of information: An event recognition approach*" complex events can be categorized into Instantaneous vessel complex events and Long lasting vessel complex events. Using the semantics and algorithms already defined, we aim to represent Vessel rendezvous, package pickings and fast approaches concepts designed with Apache Flink. We are going to implement this module with Apache Flink Dataset API. The reason of using the Dataset API is that the detection of those complex events depends on notions of proximity of the vessel's defined cells, and on complex reasoning about vessel's timestamps which are hard to implement using Flink DataStream API.

1.3.3 Visualizations and Maritime analytics

The third visualization module is based on ElasticSearch. Events are indexed into ElasticSearch can be then easily visualized using a graphic tool called Kibana that is connected to Elastic using the REST API. It aims to provide an easy to use interface to maritime authorities in order to perform real-time data analysis and visualizations on real time streaming data. In the presented Master thesis, we are going to represent the vessel's raw AIS data and easily understand them by taking advantage of their graphic representations on Kibana geo-map.

Keywords: Big Data, Maritime Intelligence, Real Time Analytics, Complex Event Recognition, Apache Flink.

2. Review of the References

The first contact to real time complex event reasoning was the paper "*How not to drown in a sea of information: An event recognition approach*" which motivates thinking about illustrating and implementing those practices in the top of many big-data open source frameworks from the Apache Foundation, like Apache Kafka and Apache Flink. Queries against the data should be performed continuously as the data coming in real time; What exactly Apache Flink is naturally doing. Also, my attendance in the premium conference of Apache Flink "Flink Forward at 11-13 September 2017 in Berlin" gave me the opportunity to learn many best practices in the technical Workshops, to understand how and where Flink is used in production environments and how it can be integrated with other Big Data frameworks. We are going to present the concepts of those references, and how we tried to develop a step further. First of all, we are going to describe concepts of RTEC.

2.1 RTEC

RTEC approach uses a formal syntax for reasoning as presented in this Paper "*How not to drown in a sea of information: An event recognition approach"*. RTEC is a Complex event recognition system using Event Calculus formal programming language for reasoning events. The idea of implementing a streaming real-time model that will extend the Implementation of the Event Calculus is motivated by the following reasons:

- RTEC sliding windows

The RTEC implementation is based on Sliding Windows approach for processing the incoming data, because data needs to be processed in batches, whereas with Apache Flink this constraint is not present because Flink naturally supports streams of events.

- RTEC transformations

The implementation of RTEC is using hard coded the Gap detection, elements buffering, the noise detection while Apache Flink provides a DataStream API which facilitates dealing with streams such as Map Functions, stateful operations and event time.

- Checkpoints

The advantage of Flink is that it comes also with a checkpointing mechanism, distributes processing, scalable state (with RocksDB), which are not provided in RTEC

implementation, and might be useful extension if a real time streaming application is going to be deployed in a production environment.

- CEP

In RTEC, Complex Event processing is implemented in prolog, while the usage of Apache Flink CEP library will help us to detect patterns without the need to translate them to a custom formal logic.

2.2 dRTEC - State of the art

In Big data ecosystem, real time processing frameworks are designed to ingest big volumes of data streams and provide analytics to the end users in real time. dRTEC event recognition engine is an enhanced version of RTEC which employs data partitioning techniques using dynamic grounding and indexing. dRTEC uses resources of distributed infrastructures very efficient and has introduced in the Paper “A Distributed Event Calculus for Event Recognition Alexandros Mavrommatis, Alexander Artikis, Anastasios Skarlatidis and Georgios Paliouras”. dRTEC intends to improve the abilities of RTEC by focusing on the event streaming analysis in order to detect patterns over streams in a more efficient way when the volumes of incoming data become very big and the velocity is increasing because it is a natural scalable, high-throughput and fault-tolerant stream processing engine.
Reasoning of statically determined and long term events is implemented with Apache Spark API by using the Apache Spark Streaming extension which offers all the capabilities of an in-memory application for efficient processing micro-batches of events. Complex event processing with dRTEC evaluated in the context of SYNAISTHISI project for real time human activity recognition and in datACRON project for the real time recognition of suspicious and illegal vessel maritime activities in the Aegean Sea.
For maritime surveillance monitoring, data analysts are interested in what happened for the last second of time in the vessels in the open sea and they want those statistics to refresh every minute. For this reason, dRTEC applies advanced windowing function based on the durations of the sliding windows, the sliding step and the maximum timestamp of each event so as to handle out of order events and compute in memory each time only the events that are inside the dynamic sliding window.

According to the empirical evaluations in the context of dataACRON project dRTEC is more efficient than RTEC when the incoming volumes of data become even bigger
The usage of Spark Streaming facilitates the integration of dRTEC with other modules (Kafka, Flume, Hadoop)

The Apache Flink implementation that we are going to introduce as we have already mentioned is using the Event Calculus formalism – like RTEC and dRTEC- including advanced techniques session windowing.

2.3 Open Source Stream processing solutions

To tackle the challenge of large scale stream processing, a number of open source frameworks were recently developed including - but not limited to Apache Flink - like Apache Spark and Apache Storm. We are going to present some key differences between Apache Flink and {Apache Spark - Apache Storm} as we extracted them from their characteristics and from many technical presentations in Flink Forward conference and their documentations:

2.3.1 Differences between Apache Spark and Apache Flink

Both support batch and stream capabilities and both are in-memory databases. The main difference is that Apache Flink is introduced to the community as a natural streaming framework that is built from scratch with a DataStream API logic, in contrast to Apache Spark which divides streaming data sets into micro batches in a continuous fashion to simulate real time processing. Apache Spark by default is not a real time Stream process engine. For this reason, it uses extensions like DStreams (Discretized Streams) a plugin for streaming data and RDD plugin (Resilient distributed dataset) for batch data. Apache Spark includes also the component Apache Spark Streaming, which can turn Apache Spark into a real-time stream processing engine.

2.3.2 Differences between Apache Storm and Apache Flink

In contrast to Apache Spark restrictions in real time streaming, Apache Storm is sharing a very similar logic with Apache Flink that means similar interfaces API and Libraries. Apache Storm is a data stream processor but with no batch capabilities - while Flink has both. When it comes to compare their streaming capabilities, Apache Flink offers a more high-level API and Libraries compared to Storm. Apache Flink DataStream API provides

built-in data transformations and aggregations such as Map, groupBy, Window, and Join, while in Storm we have to implement them from scratch.

2.4 Conclusion

Because accurate and real time data streaming analytics and metrics has vital business meaning in Maritime Industry, all the technical approach and implementation is based on Apache Flink API which turns the current master thesis into an innovative service for the maritime community. Taking into consideration that Apache Flink ecosystem has already been deployed for Big IT companies in high performance Production environments, like in the Banking Industry solving Fraud detection issues, we will take advantage and we will apply its API features operators and libraries for the benefit of Shipping industry and furthermore for the Maritime surveillance.

3. Apache Flink Ecosystem

The first use case that real time streaming technology applied, is the Twitter social media platform. Developers were able to use the Twitter API for querying all the tweets in real time as the content was generated from the users. As we told in the introduction, streaming data can be produced from wide variety of operational and transactional Source Systems every millisecond for example from IOT sensors in car/vessels/airplanes, traffic sensors, weather data sensors, social media applications mobile applications and logs that machines are producing every day. When we deal with streaming processing, the input data are supposed to be unbounded data sets that are continuously produced and we want to continuously process them in real time.

3.1 Dataflow Programming Model

A Flink program is defined by the notions of the data streams and their transformations. Conceptually a stream is a (potentially never-ending) flow of events, and a transformation is an operation that takes one or more streams as input, and produces one or more output streams as an output. When executed, Flink programs are mapped to directed acyclic graphs, consisting of streams and transformation operators. Each graphs starts with one or more sources and ends in one or more sinks.

Flink programs are executed in a distributed and parallel manner. Each data stream is divided into many partitions, which go through different instances of the operators defining the Flink program. The number of instances of each operator is each degree of parallelism. By defining parallelism over operators and partitions, Flink abstracts distributed execution from physical machines.

For example: A Flink cluster can be defined by 8 task managers each one installed in a different machine and having 1 task slot. This is equivalent to having only one Task manager on one machine having 8 task slots, and in the two cases the maximum parallelism of a Flink program on such cluster would be equal to 8.

3.2 API's and Libraries

Flink has natural DataStream API and Dataset API written in Java or Scala and Table / Sql API. Apache Flink supports data stream processing of events through its DataStream API and the windowing mechanism. The implementation of the vessel's Trajectory Detection Module is using DataStream API and many of its capabilities like the notion of event time and session window mechanism. DataStream API uses many operators from the Dataset API such as such as MapReduce, and joins written in Java or Scala to the streaming world. For the Implementation of the Complex Event Recognition Module, we used the Dataset API for performance reasons due to huge joins between the AIS datasets so as to detect critical information.

3.3 Time in Apache Flink

Flink supports different concepts in streaming time. Aggregations on data streams are different from the aggregations on the datasets. It is not possible to count for example all the elements of a DataStream, because the stream is infinite. Apache Flink is a stream processor with a flexible mechanism for building windows for evaluating real time data streams. In order to process infinite real time data streams, the stream is divided into finite slices / buckets with boundaries based on some criteria like the time passed or number of elements per window or other criteria like the period of inactivity. Flink offers explicit handling of time: and defines two types of time

3.3.1 Ingestion time

Ingestion time is the time that events enter Flink. At the source operator each record gets the source's current time as a timestamp attribute, and windows will based their computations on this timestamp. Internally, ingestion time is treated much like event time, but with automatic timestamp assignment and automatic watermark generation. Ingestion time can be used for example when developing a custom Flink source which directly ingests AIS events and therefore assigns timestamps at the source level.

3.3.2 Processing time

Processing time is the time that the event arrives in the system. Processing time refers to the system time (clock) of the machine that is executing the operations. When a streaming program runs on processing time, all time-based operations (like time windows) will use the system clock of the machines that run the respective operator. For example, an hourly

processing time window will include all records that arrived at a specific operator between the times when the system clock indicated the full hour.
Processing time is the simplest notion of time and requires no coordination between streams and machines. It provides the best performance and the lowest latency. However, in distributed and asynchronous environments processing time does not provide determinism, because it is susceptible to the speed at which records arrive in the system (for example from the message queue), and to the speed at which the records flow between operators inside the system.

3.3.3 Event time

Event time is the time that each individual event occurred on its producing device. This time is typically embedded within the elements before they enter Flink and that event timestamp can be extracted from the record. An hourly event time window will contain all records that carry an event timestamp that falls into that hour, regardless of when the records arrive, and in what order they arrive. For example, if a vessel suddenly stops at time t1, this timestamp is the event time of this element. Event time gives correct results even on out-of-order events, late events, In event time, the progress of time depends on the data, not on any machine clocks.

3.4 Watermarks

The Gap Detection, Windowing, State of previous/following events mechanisms depend on the event time. It's critical to define how Apache Flink handles run time. Generally, a streaming pipeline can depend either on the processing time or on the event time. Processing time as we state before is simply the time of the current machine clock. Event time on the other hand is the time event occurs, which is specified as a timestamp attribute. A time window of ten minutes in processing time lasts effectively ten minutes, but a time window of ten minutes in event time might be computed in few seconds, since event time is merely an attribute of the data which can be ingested instantly.
Apache Flink has to rely on its internal clock so as to be able to compute windows and handle event time correctly as if it was processing time. Watermarks are simply a way for defining such clock. A watermark of time T tells Apache Flink operators the all events of timestamp < T have passed. The advantage of using watermarks is that enables handling out of order events, by subtracting a delta from the watermarks. At event time T we inject a watermark of (T - delta), i.e. we specify that all AIS raw events of timestamps less than (T - delta) have passed. This allows handling types of events with a lateness of at most delta T. Watermarks are injected periodically in the beginning of the stream in a transparent way to the user. The user need just to specify how to extract timestamp

attributes and a strategy for watermarks injection. Either ascending, or by accounting late events.

3.5 Windows

Streaming applications are processing data in continuous fashion, and therefore we can't wait for the whole data to be streamed before starting the processing.
Of course, we can process each incoming event as it comes and move on to the next one, but in some cases we will need to do some kind of aggregation on the incoming data; e.g., how many vessels are in Piraeus port over the last 40 minutes. In such cases, we have to define a window and do the aggregations for the data within the window.

3.5.1 Tumbling Window

One kind of window is the tumbling window, where we don't have overlaps between the windows. Grouping the events in buckets (last five minutes, last five elements) and then apply aggregations on their elements is the concept. Actually it takes time equals to the window size, until the aggregation starts.

3.5.2 Sliding Window

Another type of windows are the sliding windows. Opposed to a tumbling window, the sliding slides over the stream of data. A sliding window can be overlapping and it gives a smoother aggregation over the incoming stream of data.

3.5.3 Session Window

Apache Flink is the first open source streaming engine that completely decouples windowing from fault tolerance, allowing for richer forms of windows, such as session windows. Session Windows in Apache Flink allows messages to be windowed into sessions based on vessel's activity. Flink allows us to define a time gap and all the messages that arrive within a "period of vessel's inactivity" less than the defined time - gap - can be considered to belong to the same session. Apache Flink is a stream processor with a flexible mechanism for session building windows and evaluating real time data

streams. In order to process infinite real time data streams, each logical stream is divided into finite buckets with boundaries based on some rules like timestamps of elements or other criteria. Window boundaries need to be adjusted as per incoming AIS raw data. A new session window starts at individual timestamp for each key and finishes when certain period of inactivity has passed. The configuration parameter is the session gap which is used to specify how long to wait for new AIS data before closing the session window. All the elements that arrive within a "period of inactivity" less than the defined session gap are considered to belong to the same session window. In our implementation we are going to use the concept of session windowing.

3.5.4 Triggers

The basic scope of the triggering policy in Apache Flink is to determine when a window is ready for data processing. While watermarks indicate the current state of the received data, triggers materialize the computations. Different kind of triggers are possible in Apache Flink, like on the processing time, every 5 minutes for example, on the event number, every 10 AISEvents for example, or at the end of processing a log file. Triggers are used to determine intermediate computations before the watermark reached the end of the window and it is possible in a window of 10 minutes to trigger or purge every minute for example.

3.6 State

Apache Flink is a stateful stream processing engine. Many operations in a dataflow simply look at an individual event at a time while other use a state in order to keep in memory data about the previous and current events. For example, a state can hold a counter which holds the number of seen event until the current one. Another example of stateful operators are windows which buffer events into an internal state until the window is triggered. In our use we use Flink's state to store the previous event for each vessel in order to be able to define the velocity of a vessel for example.

3.7 Consistency, fault tolerance, high availability

Apache Flink is using stateful functions and operators to store data while doing intermediate transformations and computations making state a critical component for a real time streaming application. When we are searching the stream for detecting certain event patterns, the state will store the sequence of events detected by each pattern. When we are grouping events, the state holds the pending aggregates.
In order to make state fault tolerant, Apache Flink needs to be aware of the state and checkpoint it. Apache Flink offers real time checkpointing functionality. The state of each computation can be checkpointed and guaranteed to have consistent data flow when a machine failure happens. Data are moving between source and sinks, after the machine recovers and the task managers is up and running again from the same point it stopped. Checkpointing mechanism is useful in production environments, when we want to ensure the consistent data movement between Kafka and HDFS after task managers failure. We are not using checkpoints, since it is not very critical in our case, because we can easily rerun the module, but the implementation can easily be extended with this functionality.

3.8 The DataStream API

Flink aims to support all types of input data. Apache Flink can handles many Java Primitives like the atomic data types of Arrays, Strings, Longs, Integers, Booleans and more complex Java data types like Tuples which Flink is using to create lists of elements. Tuples are more composite types because they can nest other types. In Flink we do not have to specify a schema file so as to read the data.

3.8.1 Transformations

The basic transformations that can be done to the data stream are map, flatmap and the filter transformation. The Map defines a mapping between the input element to the output and is doing a transformation. When we have a DataStream of input elements and apply a map transformation, it will take as input all the elements of the input type one by one and emits one by one the same elements of the output Java type. For example, Apache Flink's map function is useful in cases we want to append all the input string elements with a static string or when we want to multiply all the numeric input elements with a number we will retrieve them as results in the output stream. Flatmap transformation is very similar to map, and is a computation that gets one by one the elements in, and it can give as output result zero or more elements of another data type.
Another transformation that we can apply in Apache Flink is the filter transformation. Filter evaluates a Boolean function for each element and retains those for which the function returns true values. Suppose we have a real time data stream of elements or events and we apply a filter condition to them one by one so we can exclude elements according to the specific logic from the result stream. Instead of specify a filter condition, we can apply a lambda function to the elements. The way we do partitioning on the data affects our computations. Suppose we have an input stream of Tuples. We use this transformation, when we want to compute a value based on a specific key-field of the Tuple so as to partition the Data Streams by the same key and emit the result to the next transformations. In the first module for example, many computations are done for each vessel individually so we use the key-by partitioning in order to partition the stream to logical streams according to the vessel id's.

3.9 The Dataset API

Flink is facing the Batch processing as a special case of Streams, as finite data sources are streams that have an end. Apache Flink offers the Dataset API which supports similar transformations as the DataStream Api, but with dedicated Libraries for graph processing and Machine learning. Apache Flink provides various optimizations like scheduling batches and query plan optimizations. We are going to use the Dataset API in the Complex Event Recognition module, for performance reasons due to huge joins between the AIS datasets so as to detect critical information. It is very hard to do self Joins so as to create

Cartesian products between huge datasets, so as to recognize a vessel rendezvous for example, as we will present in detail in the Part 3 Implementation - of this Master thesis.

3.10 The Deployment

Apache Flink can be deployed in a variety of Production environments, from a local Java Virtual Machine to a standalone cluster or a cloud provider managed by YARN. In the current implementation we are running Apache Flink in digitalocean.com cloud infrastructure using an Ubuntu 16.04.1 x64 Virtual Machine, with 8 cores, 16GB Memory and 40GB SSD Disk.

IMPLEMENTATION

4. Trajectory Detection Module

The Trajectory Detection Module is the main part of the current implementation. We are going to design a real time application by continuously read AIS data from a Kafka topic using Apache Flink DataStream API and Apache Kafka. We integrate Apache Flink with Apache Kafka because it solved us the low throughput issues due to backpressure while reading directly the data from the given static file. We created a Kafka topic from which contains all the AIS data set and we feed Apache Flink, fully simulating its operations in real time.

4.1 AISEvents class

The raw input elements are first modeled into an AISEvents class. An AISEvents class is designed to have the following fields and methods:

Fields	Methods
long previousTS; **long timestamp**; **int id**; **double lat**; **double lon**; **double instantSpeed**; **public** Velocity **velocity**; **boolean isNoisy**; **boolean isPause**; **boolean isTurn**; **boolean isSpeedChange**; **boolean isGapStart**; **boolean isGapEnd**;	**double** computeDistance(AisEvent other) **double** computeSpeed(AisEvent previous) **double** computeBearing(AisEvent previous)

The values of those fields are defined and computed progressively in the first, pre-processing, part of the pipeline. From raw AIS data, we create Data Streams of “Vessel Tagged Events” using Apache Flink DataStream API in order to apply computations over infinite data streams. So as to model input AIS data into an AISEvents java class we flat-map them with the InputParseOperator which takes as input the measurement lines and maps them to AISEvents objects so as to assign the vessel id, vessel latitude, vessel longitude and vessel timestamp which is the event time the AIS event happened. Events are already ordered, so we don't have to handle low latency issues and out of order events. We are using the ascending timestamp extractor to define timestamps and watermark policy.

4.2 Keyed Streams

A critical step in the pipeline is to detect the “Gaps Starts” and “Gap Ends” of the AIS Events as it is presented in the Paper: so as to include them in next calculations. This processing step is done for each vessel separately. We separate logically the vessels using DataStream Apache Flink API and the key by operator which selects the id of the vessel as the key. This way the subsequent operator is applied to each vessel separately. With this operator, we simply extract the vessel’s id, from an AISEvents object, so as to implement logical keyed streams based on the vessel’s id.

4.3 Session windowing policy

We used a session window assigner configured with session gap which defines how long is the required period of inactivity.

4.4 Triggering policy

We used EventTimeTrigger, that triggers the window based on the progress of event time as measured by the watermarks. This permits an incremental computation and purging of the session windows.

4.5 Window Process Policy

A Windowing function is used to process one by one the events that belong to the same window which fires and becomes ready for processing based on the triggering policy. For this scope, we implemented the SimpleGapOperator Operator which extends the functionality of the built-in WindowFunction (ProcessWindowFunction) and implements

the window processing policy. The SimpleGapOperator stores all the events of a session window inside a buffer in order to loop and define the first element of the window as the "Gap End" and the last one "Gap Start". We set boolean attributes isGapStart, isGapEnd in the subsequent operator which processes the session windows.

4.6 Computing Coordinates

Trajectory Movement Events like vessel Pause Events or vessel Speed Change Events, as well as Noise Filtering rules depends on the vessel's velocity vector and the vessel's acceleration. To facilitate computing the speed, bearing and distances, we use the MapFunction to assign each AISEvents its Cartesian coordinates using the Coordinates Operator Operator. The CoordinatesOperator Operator is applied to AISEvents one by one so we can extract and assign (x,y,z) coordinates to all AISEvents to indicate their points on the map. For this purpose, it is no need to treat each vessel separately. That's why we do not use the keyby operator so as to group by vessel's id's.

4.7 Short Term Events - Noise Filtering Process

4.7.1 State

To be able to compute Short-term Events we need to access the previous state of the same vessel. In other words, we need to keep a state which contains previous AisEvent values and to update correctly the state in real time. Apache Flink DataStream API offers stateful operators, which can be scoped by key, i.e. keep a state for each vessel separately. Keyby operator is used before the NoiseOperator Operator so as to group the stream by vessel's id.

4.7.2 NoiseOperator

Operator implements maritime rules, as they introduced in the Paper: "How not to drown in a sea of information: An event recognition approach" so as to detect and recognize with Apache Flink API both Short Term Events and Noise Events in order to keep only the critical instantaneous information in the pipeline and compress the Trajectory of the vessels.

4.7.2.1 Functionality

If the AisEvent is a GapEnd that is the first AISEvent potentially after a gap we cannot define its velocity, acceleration, or Bearing Differences and Speed Differences because we do not have its previous state. Otherwise, if the AisEvent is not a GapEnd - so we have a previous state that we retrieve the previous AISEvent state from Flink State API.
In the same logic we fetch its previous speed using the computeSpeed function. The way we compute the Speed of an AISEvent, is implemented inside the AISEvent class and is the distance from the previous event if it is not a GapEnd, divided by the difference between current and previous AISEvent timestamps. If there is no previous event the speed is undefined. The same goes for the bearing. We compute the Bearing of the previous AISEvent inside the AISEvent class. We define its Velocity vector using speed and bearing attributes. The implementation of the Velocity class based on angle and speed attributes is also part and implemented inside the AISEvent class. In order to fetch the instant acceleration of the vessel, we have to divide Speed Difference from the difference between the current and the previous timestamp of the same AISEvent. So, if the velocity of the previous status of the AISEvent is not null, we compute

a) The Bearing difference
b) The Speed difference
c) The acceleration

In any other case, if the velocity is null, we set all the previous attributes as not applicable.

Having done all the previous calculations, we are about to apply the rules for critical points along vessel trajectories as presented in the paper "**How not to drown in a sea of information: An event recognition approach**" so as to detect vessel's critical movement events such as stops, acceleration or turn events along their trajectory.

4.8 {Pause, Speed Change, Turn} Events

Based on the AISEvent class calculations we continue by analyzing instantaneous vessel Pauses, vessel speed changes and vessel turns and recognize them with Apache Flink Datastream API.

4.8.1 Pause Event – Speed Change Event – Turn Point Event – Noise Events

The minimum speed in km/h (=1 KNOT) so as to characterize an object as a moving one is following. If the value of the vessel's speed is less than 1.852 Km/h then, we recognize the AISEvent as a Pause Event. If the vessel's Speed difference is more than 0.25 KNOTS, the vessel is supposed to have a speed change and we recognize it as SpeedChange

AISEvent. If the maximum heading difference between successive positions is more than 15 degrees, this suggests that the vessel is changing its route direction, so we recognize that this is a Turn Point AISEvent. As presented in the paper: "**How not to drown in a sea of information: An event recognition approach**", the Trajectory Detection Module has to compress the Data Stream into a stream that contains only critical movement AISEvents. The following rules are used so as to define noise events and to exclude them from the pipeline.

a) If the maximum acceleration during a speed change AISEvent is greater than 10 measurement units, we consider this AISEvent a noise event, because this rate cannot happen in real conditions and may have been caused for example by high waves in the open sea or by AIS signal delays. The received position is considered an outlier and we recognize it as a Noise AISEvent.

b) If the maximum difference in vessel's heading between successive positions (in degrees) is greater than 60 degrees, then the AISEvent is considered as a noise event. Again, this is because this rate cannot happen in real conditions and may have been caused by high waves in the open sea or by AIS signal delays. The received position is considered an outlier and we recognize it as a Nose AISEvent.

c)If the speed in km/h is 55.56 measurement units (=30 KNOTs), we consider the AISEvent as a noise Event, because again this cannot happen in real conditions. The received position is considered an outlier and we recognize it as a Noise AISEvent.

By applying all the previous rules to the DataStream, we process the next incoming AISEvent sequentially and one by one which means that the current AisEvent becomes the previous AisEvent for the next one.

So, after the flatmap (**new** NoiseOperator ()) application, we are able to collect one by one all the rest AISEvents that are not Noise Events or Gaps:

The result of the process, is a recognized stream of only tagged - modified - instantaneous AISEvents in JSON format.

4.9 Long Lasting Movement Events

We are going to apply predefined Patterns to the DataStream of the tagged AISEvents so as to detect more complex Long Term AISEvents with Apache Flink CEP Library. The goal is to discover critical Long Term Events for the vessels in the open sea and to record critical events while making decisions so as to prevent complex situations. In this Master Thesis, the Trajectory Detection Module, introduce Complex event processing (CEP) with Apache Flink and gives a solution for this issue, by matching continuously incoming AISEvents against one or more patterns so as to detect these critical Events. All AIS data which do not match the patterns can be immediately discarded and all the rest are processed immediately once the system has detected all the events for a matching sequence. The results are emitted straight away in real time taking advantage of Apache Flink's streaming nature and its capabilities for low latency and high throughput stream processing as a natural fit for CEP workloads.

4.9.1 General Approach - Flink CEP Library

To detect the Long Lasting Events like Long Term Stops, Smooth Turns and Gaps we use Flink CEP (Complex Event Pattern) library, which is implemented on top of Apache Flink and does custom pattern detection over an endless stream of tagged - modified AISEvents. First of all, to deal with Apache Flink CEP we have to define one or more custom pattern(s) and then apply them on the Data Stream so as to extract the subsequences of AISEvents matching those patterns. Based in the Paper "*How not to drown in a sea of information: An event recognition approach*" in order an AISEvent to be a candidate Long Term Stop, it should be an only a Pause AISEvent or a Turn AISEvent. The isCandidateForLts is a boolean indicating whether an AISEvent is a Turn or a Pause, since those two event types are used to define long term events. In the AISEvent class implementation, we have introduced the "Candidate Long Term Stop"

4.9.2 Long Term Stops and Smooth Turns

Pattern1 for identifying Long Term Events is specified by considering the first incoming element as not a candidate for Long Term Stop (non-Pause AISEvents) and assigning/marking a label “Start” to it. If the following one or more consecutively incoming AISEvents are considered as candidates for Long Term Stop (Pause AISEvents) then and we assign/mark a label “Middle” to them. If The following final event, is considered as a Candidate for Long Term Stop (Pause AISEvent) and a Gap Start at the same time, then we assign/mark the label “end” to it. Pattern1 and Pattern2 are presented below:

4.9.2.1 Pattern 1

For example, suppose we have the Sequence of AISEvents 000011111112 where 00000 is not candidates for LongTermStop (Non Pause Events), 1111111 are candidate LongTermStops (Pause Events) and 2 is candidate LongTermStop (e.g. Pause Event) and GapStart at the same time. Pattern1 claims that these Pause AISEvents (1111111) are surrounded by non-pause events (0000 and 2 as a gap Start). So 11112 is a Pattern1 for a LongTermStop Event. {“start”:0,” middle”: [1111111],” end”:2}

4.9.2.2 Pattern 2

Pattern2 for identifying Long Term Events is specified by considering the first element as not a candidate for Long Term Stop (non-Pause AISEvents) and assigns/marks a label “Start” to it. If the following one or more consecutively AISEvents are considered as candidates for Long Term Stop (Pause AISEvents) and not GapStart AISEvents at the same time, then we assign/mark a label “Middle” to them. If the following final event of this pattern is considered as a not Candidate for Long Term Stop (non-Pause AISEvent) and not a GapStart at the same time, then we assigning/marking label “end” to it.
For example, suppose we have the Sequence of AISEvents 0000111111110000000 where incoming 0000 is not candidate Long Term Stop (Non Pause Events), 1111111 are candidate LongTermStops (Pause Events) and not a GapStart and 0000000 are not candidate LongTermStop (e.g. non-Pause Event). In Pattern2 we state that Pause AISEvents (1111111) are surrounded by non-pause events (0000 and 0000000). So 011111110 is a Pattern2 for LongTermStops.

{“start”:0,” middle”: [1111111],” end”:0}. We use these Patterns and we apply them to the tagged Events Stream, so as to detect either Long Term Stops or Smooth Turns. The

separation between those two types of long term events will be done in next steps by the Trajectory Detection Module and the LongTermStop Operator.

4.9.2.3 Pattern 1 Union Pattern 2

The union of Pattern1 and Pattern2 is the general Pattern build for detecting LongTermStops and Smooth Turns. After applying flat map function, the output of the union is no more tagged events, but a stream of lists of events

- By applying Pattern1 - the first part of the Union - we will collect results according to this input e.g.: {“start”:0,”middle”:[1111111],”end”:2} a stream of lists of events like : 11111112

- By applying Pattern2 - the second part of the Union - we will collect results according to this input e.g.: {“start”:0,” middle”: [1111111],” end”:0} a stream of lists of events like: 1111111

4.9.2.4 Filtering

As we state before, the result of applying the pattern to the tagged events, is a stream of series of events matching the pattern. So each incoming individual element of the data stream is a list composed of one or more events. According to the Paper “*How not to drown in a sea of information: An event recognition approach*” to be considered a Long Term Event as valid it must contain at least 10 consecutive pause events or turn events. We process those lists with the LongTermStop Operator so as to separate long term stops from smooth turns and create two different lists of events.
The filtering process of the LongTermEvents based on the length of the list (e.g. 11111112, 1111111, 111111111111111).

4.9.3 The Long Term Stop Operator

The LongTermStop Operator takes as input the result of the subsequence data streams of AISEvents that matched the pattern. The pattern is built to recognize either a vessel’s smooth turn or a vessel’s long term stop. The role of the LongTermStop operator is to separate those two types of long term events. It works as follows: Given a list L of AIS Events matched by the pattern, we start looping through L to extract any events which lie

within a radius of 250 meters. If an event does not lie within 250 meters of the events already extracted, then it is part of a smooth turn. The number of events already extracted is either more than a threshold (for example 10), in which case they are considered a long term stop or less, in which case the extracted events are part of a smooth turn. Suppose we have the following list of long term events matched by the pattern: [e, e, e, e, e | e, e, e, e, e, e | e, e, e, e, e, e, e, e, e, e, e, e| e, e]
The comma "," delimiter between those events indicates that the right event lies within 250 meters of the previous ones, and the pipe "|", indicates the opposite.
The first 5 events lie within 250 meters of each other, so they should be collectively considered a long term stop. Except that their number is less than the threshold of long term events (= 10). The next 6 events lie within 250 meters of each other's but they cannot be considered a long term stop because they are less than 10. The first 11 events are then concatenated into one smooth turn.
The next 12 events lie within the predefined radius and pass the threshold criteria so they are concatenated on one long term event. The last 2 events are simply discarded.

Result: <" smt", 2222222222>, <" lts", 33333333333333>, <" smt", 44444444444444>

4.9.3.1 Filtering

According to the string label "smt" or "lts" we are going to filter the events so as to separate them into two different streams. The first stream will contain the Smooth Turn AISEvents. The second stream will contain the Long Term Stops AISEvents. We then map each element of those streams to a common model (class) "LongTermEvents" which has the following attributes:

- Start (type: AisEvent)
- End (type: AisEvent)
- Label (type: String)

For the long term stops stream, those attributes and the result of applying the pattern are serialized as a json with the following fields:

- Start: the starting AISEvent (of a long term stop)
- End: the ending AISEvent (of a long term stop)
- Label: "long term stops".

{label: "lts", start: First_AISEvent, end: Last_AISEvent}

The same goes for Smooth Turns.

The result of applying the patterns are serialized as a json with the following fields:

- Start: the starting AISEvent (of a smooth turn)
- End: the ending AISEvent (of a smooth turn)

- Label: “smooth turn”.

{label: “smt”, start: First_AISEvent, end: Last_AISEvent}

LongTermEvents either for LongTermStops or Smooth Turns AISEvents are finally converted to JSON representations, using the GSON serializer operator which implements a MapFunction in order to store the results as json files in the disk. Below are presented the results for LongTermStops and the results for Smooth Turns We finally output the results using Flink Filesystem connector.

4.10 Gaps

We specify the pattern to output consecutive Gap starts and Gap ends and to apply those patterns to the tagged events, so as to extract Gaps - Long Term Events.
The result of applying the pattern is serialized as a json with the following fields:

- Start: the starting AISEvent (of a gap)
- End: the ending AISEvent (of a gap)
- Label: “gap”

Gaps LongTermEvents are finally converted to JSON representations, using the GSON serializer so as to store the results as files on the disk. We finally output the results using Flink Filesystem connector.

5 Complex Event Recognition Module

5.1 Introduction

The second module is based on the output of **Trajectory Detection Module**, and its goal is to detect events which are more complex than the movement AISEvents. In this Master Thesis we recognize and implement Vessel Rendezvous, Package picking and Fast approaches. For this module we rely on Apache Flink Batch Dataset API, since many operations require performing joins between data sets, which are easier implemented using the batch API.

5.2 Grid partitioning

Some complex events, like vessels approach, depend on a notion of proximity of the vessels defined by the fact that two vessels lie in the same cell. That's why we need to be able to map each vessel coordinates to a cell in the map. For this we use *GeoHash* which is a geocoding system based on a hierarchical spatial data structure which subdivides space into buckets of grid shape.(github.com/davidmoten/geo)

Therefore, each cell is labeled using a geohash which is of user-definable precision:

- High precision geohash have a long string length and represent cells that cover only a small area.
- Low precision geohash have a short string length and represent cells that each cover a large area.

GeoHash can have a choice of precision between 1 and 12. As a consequence of the gradual precision degradation, nearby places will often present similar prefixes. The longer a shared prefix is, the closer the two places are.

5.2.1 Grid partitioning using GeoHash

Examples of Geohash mapping given latitude and longitude:

- 39.664148, 23.604166 :
  - Precision 5: **sx0c**9
  - Precision 6: sx0c9h
  - Precision 12: sx0c9hsh2vjg
- 39.56, 23.90:
  - Precision 5: **sx0c**p
  - Precision 6: sx0cpk
  - Precision 12: sx0cpks00000

As we can see, these two nearby points share a prefix of length 4 (**sx0c**).

The following table shows the cell dimension given the geohash precision:

GeoHash Length	Area width X height
1	5,009.4km x 4,992.6km
2	1,252.3km x 624.1km
3	156.5km x 156km
4	39.1km x 19.5km
5	4.9km x 4.9km
6	1.2km x 609.4m
7	152.9m x 152.4m
8	38.2m x 19m
9	4.8m x 4.8m
10	1.2m x 59.5cm
11	14.9cm x 14.9cm
12	3.7cm x 1.9cm

For GeoHash mapping we using the following open-source library github.com/davidmoten/geo, which provides convenient methods for geohash mapping In the Complex Event Recognition Module, input data are considered the JSON files that are produced from the Trajectory Detection Module.

1) Gaps
2) Long Term Stops
3) Smooth Turns

5.3 Vessel Rendezvous

We implement vessel Rendezvous and the other complex maritime events recognition using Apache Flink Batch API. The batch API is more suitable in this case, since multiple comparison between vessels locations and timestamps should be performed, and the batch API is more capable of such computation heavy workload.
Given the following logical rule that is representing and reasoning about vessel rendezvous with Event Calculus as presented in the Paper "**How not to drown in a sea of information: An event recognition approach**" we implement it as follows:

Event Calculus rule for Vessel Rendezvous:

1. holdsFor (possibleRendezvous(Vessel1 ;Vessel2)=true; I)
2. holdsFor (in(Vessel1)=Cell ; I1);
3. holdsFor (in(Vessel2)=Cell ; I2);
4. holdsFor (suspiciousDelay(Vessel1)=true; I3);
5. holdsFor (suspiciousDelay(Vessel2)=true; I4);
6. intersect all([I1 ; I2 ; I3 ; I4]; I)

Implementation with Apache Flink API:

1. We read the Gaps data which is output from the Trajectory Detection Module. The gaps data contains the following attributes: label (=" gap"), start Event, end Event.

2. Each gap event is then mapped using a Flink MapFunction to a tuple containing the following fields:
    a. GeoHash of the location of the GapEnd event
    b. Vessel ID
    c. gap Start timestamp
    d. GapEnd timestamp

    The result type is an Apache Flink Dataset of
    Tuple4<GeoHash, ID, gapStart_timestamp, gapEnd_timestamp>>

3. We perform a self-join operation on the dataset, where the join key is the GeoHash. This allows us to check vessel rendezvous for vessels which have been in the same vessel. For each couple events (first, second) having a matching key(=cell), we compute the gaps overlap as follows

intersection =
Range (first. GapStart_timestamp, first. GapEnd_timestamp)
. intersect (Range (second. gapStart_timestamp, second. gapEnd_timestamp))
. intersect (Range (first. gapEnd_timestamp - 60, first. gapEnd_timestamp + 60))
. intersect (Range (second. gapEnd_timestamp - 60, second. gapEnd_timestamp + 60));
If the intersection is not empty we output (vessel1_id, vessel2_id, intersection. Start, intersection. End) where vessel ids are sorted.

4. Then we perform a distinct operation on (vessel1_id, vessel2_id) to filter duplicates.
5. We finally output the results to text csv files using Flink Filesystem connector.

5.4 Package picking

According to the paper "**Online Event Recognition from Moving Vessel Trajectories**" is defined that in order to have a possible Package Picking complex event, the end of the stop of one vessel is the start of the stop of another vessel and this point of time the two vessels should be in the same cell in the grid, and the delta of the timestamps should be less than 1 hour, and also the distance between the two vessels should be less than 0.5 km. We will manage to deploy the above rules using Apache Flink API. Package Picking deals with couples of vessels when the first vessel does a Long Term Stop and drops the package when Long Term Stop Event finishes, and a second vessel stops in the same cell (start of stop) so as to pick the package. The process is doing by Joining two different Tuple2 Datasets. The first Dataset contains the Start of Stops and the second Dataset contains the End of Stops. We have to import in the Complex Event Recognition module the LongTermStops from ltsOut JSON file produced from the Trajectory Detection Module, so as to parse the Long Term Stops: DataSet<LongTermEvent> stops = env. readTextFile(ltsOut) and convert them back as LongTermEvents by mapping these Strings to LongTermEvent class, that has 3 attributes a) Start, b)End, c)Label:

Package picking in Event Calculus is characterized by the following rule:

Event Calculus rule for Package Picking:

1. happensAt(possiblePicking(Vessel1 ;Vessel2); Tpick)
2. happensAt(end(stopped(Vessel1)=true);Tdrop);
3. holdsAt(in(Vessel1)=Cell ; Tdrop);
4. happensAt(start(stopped(Vessel2)=true);Tpick);
5. holdsAt(in(Vessel2)=Cell ; Tpick);
6. Tpick - Tdrop < 1 hour;
7. holdsAt(coord(Vessel1) =(Lon1 ; Lat1); Tdrop);
8. holdsAt(coord(Vessel2) =(Lon2 ; Lat2); Tpick);
9. distance((Lon1 ; Lat1); (Lon2 ; Lat2); Dist);
10. Dist < 0,5 miles

It is therefore based on long-term stop events, which are output from the Trajectory Detection Module.

Implementation with Apache Flink API:

1. We first derive two datasets from the long-term stops dataset:
2. Dataset <Tuple2<GeoHash, AisEvent>> cell_startOfStops, and DataSet<Tuple2<GeoHash, AisEvent>> cell_endOfStops.
   This is done using Flink MapFunction which takes as input the long-term stops dataset and maps it to one of the previous datasets as follows:

```
Tuple2<GeoHash, AisEvent> cell_startOfStop =
map (LongTermEvent longTermEvent) {
  return Tuple2.of(
      GeoHash.encodeHash(
          longTermEvent.getStart().getLat(),
          longTermEvent.getStart().getLon(), accuracy),
      longTermEvent.getStart());
}
```

3. We then join the two datasets on the GeoHash key in order to find vessel couples such that the end of the long-term stop of one vessel happens in the same cell as the start of the long-term stop of the other vessel. The join is performed using the Flink batch API as follows:

```
cell_startOfStops
    . join(cell_endOfStops)
    . where(0)
    .equalTo(0)
```

4. Then we filter events which verify the distance and duration between T_drop and T_pick constraints.

5. The output result is a dataset of (vessel1_id, vessel2_id, pick_timestamp) that we output to a csv file using Flink Filesystem connector.

5.5 Fast Approach

Fast approach is defined by the following rule in paper "**Online Event Recognition from Moving Vessel Trajectories**" "*headingToVessels(Vessel) is a fluent that becomes true whenever a Vessel's direction of movement is towards at least one nearby vessel.*"

Event Calculus rule for Fast Approach :

1. happensAt(fastApproach(Vessel); T)
2. happensAt(speedChange(Vessel); T);
3. holdsAt(velocity(Vessel)=Speed; T);
4. Speed > 20 knots;
5. holdsAt(coord(Vessel) =(Lon; Lat); T);
6. not nearPorts(Lon; Lat);
7. holdsAt(headingToVessels(Vessel)=true; T)

Implementation with Apache Flink API:

To implement Fast Approach Complex Event, we are filtering all movement events which is output by the Trajectory Detection Module in order to keep only speed change events. This is done using Flink Filter function. The result is a DataSet that will contain Speed Change Events. We then assign to each event its GeoHash in the same way we did for Vessel Rendezvous and Package Picking. The scope is to create two different Datasets. The First Dataset is all the events that are SpeedChanges. The Second Dataset is all the events that detected from the Trajectory Detection Module. A join is performed between these 2 datasets, so as to get results of Fast Approaches events as couple of elements in the same cell on the grid.

5.5.1 SpeedChange DataSet:

Specifically, the input data are JSONs file that we convert back to AISEvent class, in order to have access to all its attributes as described in the Trajectory Detection Module. Using the is SpeedChange function that is implemented in the AISEvent class we are going to filter and keep only Speed Changes events whose speed is greater than 30 KNOTS.

Furthermore, we compute the cell of the events using the GeoHash function, and we create DataSet of Tuple2<String, AisEvent>, where the String is the result of the GeoHash function.

5.5.2 AllEvents DataSet:

The input data is a JSON file that we convert back to AISEvent, in order to have access to all its attributes as described in the Trajectory Detection Module. Same way as before, we compute the cell of the events using the GeoHash function, and we create DataSet of Tuple2<String, AisEvent> where the String is the result of the GeoHash function. Joining process is performed between the speed changes dataset and the dataset containing all events on the GeoHash key, to find couples of vessels that one of them is heading toward the other. This is implemented using Flink Join Function in the same way as package picking and vessel rendezvous.

5.5.3 Join between these two Datasets:

In order to extract the couples of vessels participating in the fast approach, we filter couple having the same id (same vessel). Next we check if the vessel with the speed change is heading towards the other event.

Given a vessel whose current position is at B, and its previous position is A, we can define its velocity at B by the speed and the bearing. The speed is given by the distance between A and B divided by the difference in timestamps. The bearing is defined by the angle with respect to a reference line (here 30 for example). The resulting velocity vector has the same direction as the line between A and B.
If a vessel is heading towards another vessel, but their timestamps are very distant, we have to exclude these from the results. The timestamp of the one Event should not be less than the delta duration from the timestamp of the other event: |event. f1. get Timestamp () - change. f1. get Timestamp ()| <= delta

We define the notion of “heading toward” by the the following: a vessel is heading toward another one if the direction of its velocity is within a given angle alpha, of the line joining the two vessels.
Given a velocity object V, we can compute the angle between its direction and the direction of another velocity object V1 using the method V .getBearingDiff(V1) that we implemented in the velocity class.

We want to compute: angle between: (my velocity) and (line connecting me and you)

(line connecting me and you) == velocity V
We can compute: angle between (my velocity) and (any velocity V2)

So all we need to do is to create a virtual Velocity vector which has the same direction as the line between the two vessels. This can be done by computing the velocity of the second vessel **as if** the first vessel was its previous position.
If the direction of the velocity vector of the first vessel is less than “within” degrees from the direction of the line connecting the two vessels, then the result is supposed to be a Fast Approach.

6 Visualizations with Kibana

Flink has an integrated ElasticSearch sink, which helps sending AISEvents to ElasticSearch and visualizing them in in real-time. One first we have to create an ElasticSearch index which will contain the events. Then one should specify fields types, to be able to visualize geopoints in maps in case or geographical data or to query data by time range in case of time series. In our case, we specify the following mapping for the ElasticSearch index we created:

```
curl -XPUT 'localhost:9200/aisevents/_mapping/all?pretty' -H 'Content-Type: application/json' -d
'{
  "properties": {
    "location": {
         "type": "geo_point"
       },
    "timestamp": {
         "type": "date"
       }
  }
}'
```

Then we launch another Flink pipeline which reads the raw AISEvents, pre-process them into json objects to be recognized by ElasticSearch and send them to ElasticSearch index.

The Flink Job has two operators:

The first one reads each line of the input and converts it to a json with the following schema:
{timestamp, id, location: {long, lat}}. This is achieved using a Flink MapFunction as follows:

```
.map(new MapFunction<String, Tuple2<String, JsonObject>>() {
   @Override
   public Tuple2<String, JsonObject> map (String s) throws Exception {
      String [] data = s.split(" ");
      JsonObject jsonObject = new JsonObject();
      jsonObject.addProperty("timestamp", 1000*Long.parseLong(data[0]));
      jsonObject.addProperty("id",data[1]);
      JsonObject location = new JsonObject();
      location.addProperty("lon", Double.parseDouble(data[2]));
```

```
    location.addProperty("lat", Double.parseDouble(data[3]));
    jsonObject.add("location", location);
    return Tuple2.of(data[1], jsonObject);
  }
})
```

The second one is the ElasticSearch sink which sends the resulting json objects to the specified ElasticSearch index (“aisevents” in our case):

```
stream.addSink(new ElasticsearchSink<>(config, transportAddresses, new ElasticsearchSinkFunction<Tuple2<String, String>>() {
  public Index Request createIndexRequest (Tuple2<String, String> element) {

    return Requests.indexRequest()
        . index("aisevents")
        . type("all")
        . source(element.f1);
  }
  @Override
  public void process (Tuple2<String, String> element, Runtime Context ctx, RequestIndexer indexer) {
    indexer.add(createIndexRequest(element));
  }
}));
```

This results in the events being indexed and shown in Kibana as soon as they are sent in Figure [2]:

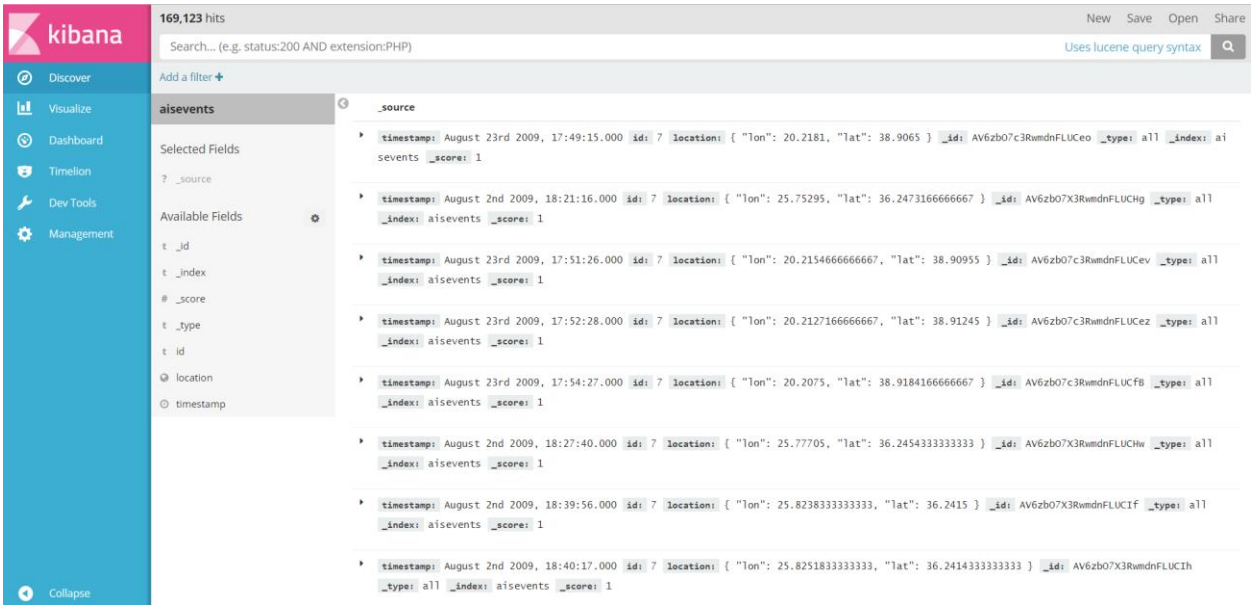

Figure [2] : Kibana indexing

One can then easily visualize trajectories using Kibana integrated Coordinate Maps as described below in Figure [3]:

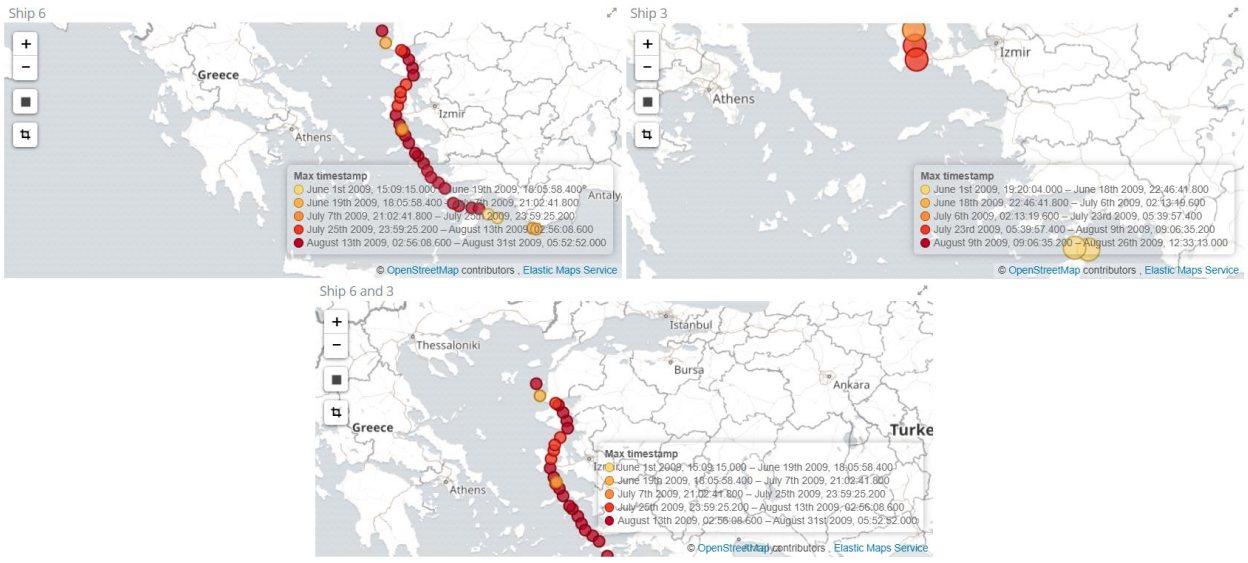

Figure [3]: Vessel Trajectory Visualization near Rhodes island

Using the coordinates map visualization, we can visually check for complex events like rendezvous for example. In the following Figure [3.1] we can see that the “Ship 3” was around Rhodes in June 1st between 18h and 22h:

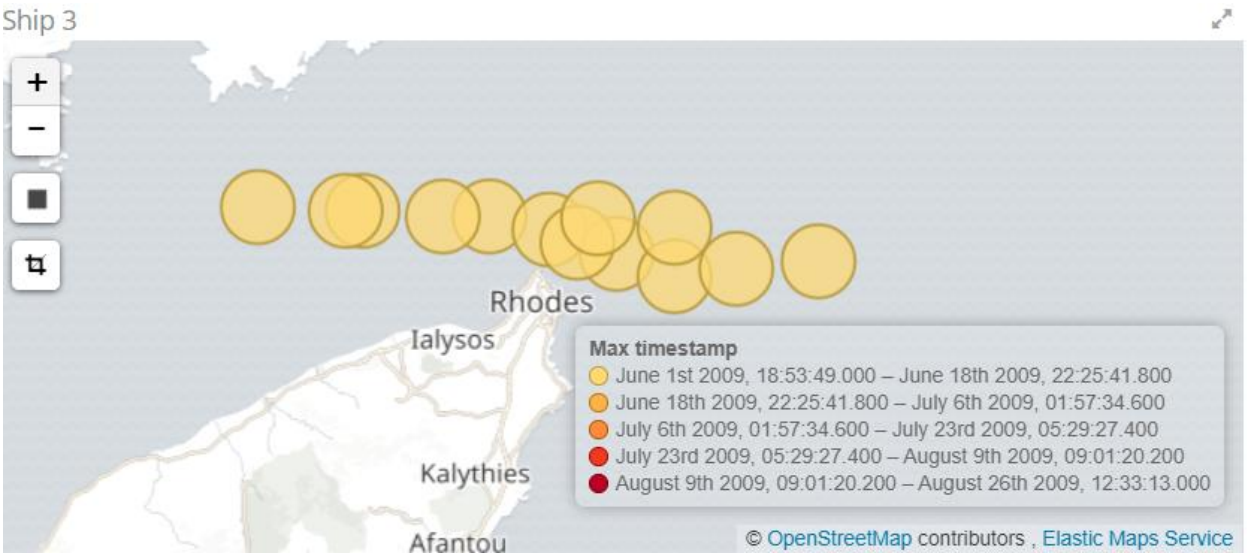

Figure [3.1]: Vessel Trajectory Visualization near Rhodes island

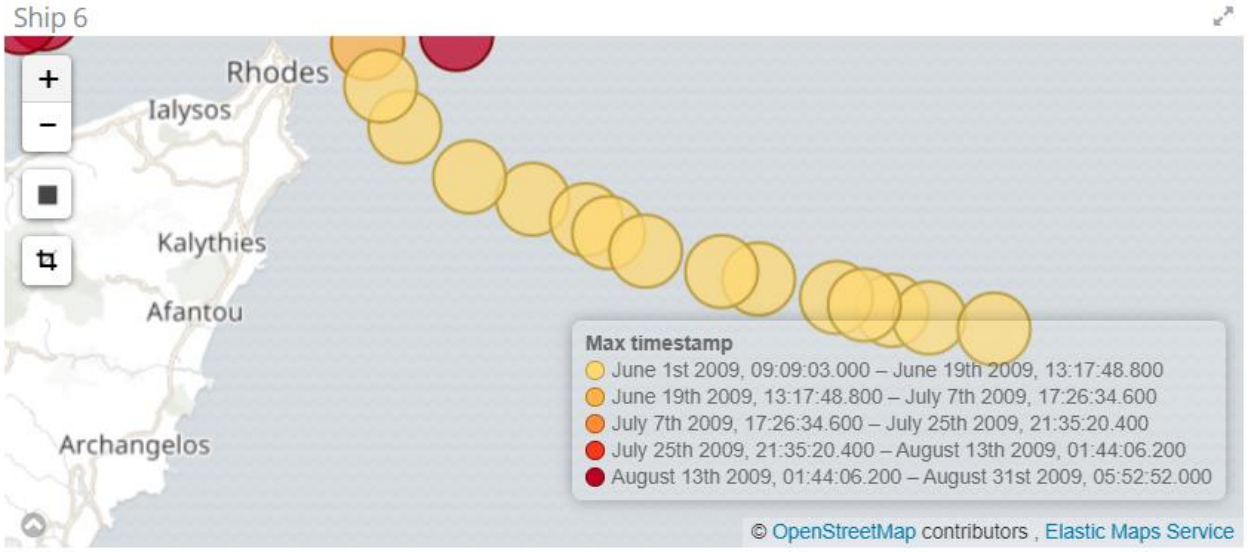

Figure [3.2]: Vessel Trajectory Visualization near Rhodes island

And that the “Ship 6” was in the same area around the same time which suggest a possible rendezvous which was successfully recognized by the complex event recognition module, as it is visualized in Figure [3.2]

7 Proposals and next steps

- The proposed service can be considered a proof of concept for treating maritime events and is far from being ready for production. To make it suitable for production, the full pipeline should be set up with Kafka a data source and also as a data sink given the streaming nature of the data, the whole process should be automated and tested, and input and outputs should be properly configured and possibly integrated to databases.

- One shortcoming of our implementation is the costly join performed between speed change events and all other events to detect fast approaches. This can maybe be optimized.

- Another track of improvement is the automated detection of noisy events, which is currently performed using hard wired rules, and could maybe be performed using machine learning techniques to predict vessel trajectories and filter noisy events as outliers.

- Another improvement might be to think of a streaming adaptation of the algorithms used in the complex event recognition module which is currently performed using the batch API due to the streaming API limitations.

- Developers who write big data programs (like MapReduce functions) with streaming data can take data in whatever format it is in, join different sets, reduce it to key-value pairs (map), and then run calculations on adjacent pairs to produce some final calculated value. They also can plug these data items into machine learning algorithms to make some projection (predictive models) or discover patterns (classification models).

8 Apache Flink Dashboard and metrics

8.1 Trajectory Detection Module

In the following screenshot we are presenting the Flink Dashboard while running the Trajectory Detection Module, as presented below in Figure [4].
The job can be executed in one of two modes: Either reading the whole input first and then processing data which is equivalent to batch processing, or reading the input incrementally and processing it on the fly which is more compatible with the streaming nature of Apache Flink. When reading the input as whole we can see the gap detecting operator which computes session windows is stuck and does not output any element:

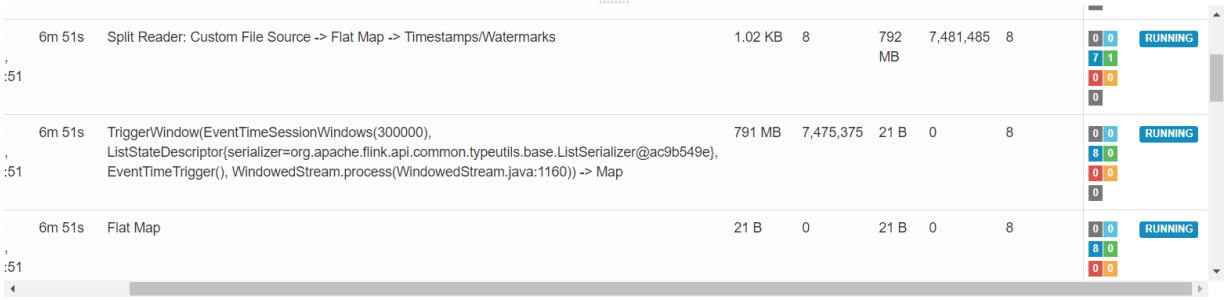

Figure [4]: Apache Flink execution dashboard

However, when reading the input incrementally, the operator starts sending elements as soon as it receives data, which we can see in the following Figure [5]:

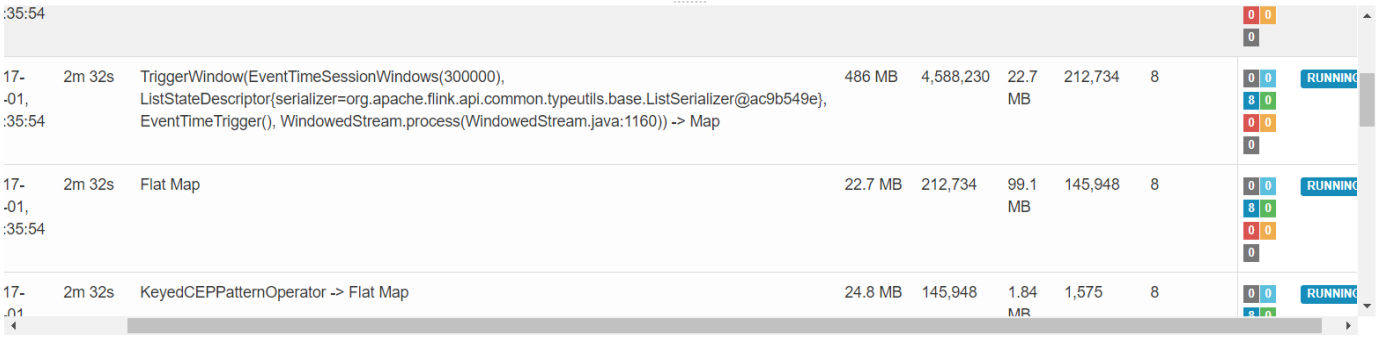

Figure [5]: Incrementally data ingestion

In the trajectory detection module, we are reading events from Kafka which are then processed by a windows in order to detect gaps then a subsequent Flatmap operator that assigns velocity information and filters noisy events and then patterns are applied to the

stream using Flink CEP to detect long term events (smooth turns and long term stops), as presented in Figure [6]. The results are output to files in the disk.

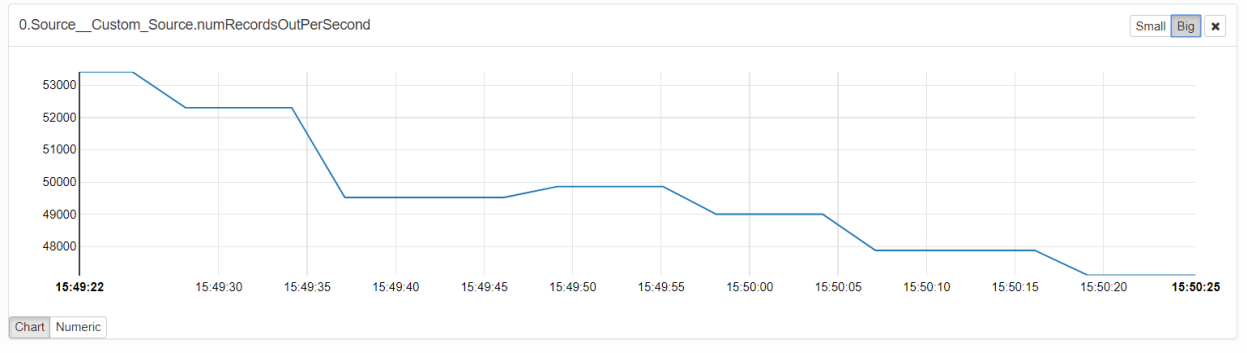

Figure [6] : Number of records emit by Kafka per second

The session window is used to detect gaps (by marking the first element of the window as a Gap End and the last one as a Gap Start). In the following graph we can see that the number of record out/s from the window operator (about 9000/s) is less than what is emit by Kafka (about 50000) due to the latency of the window operator as presented in Figure [7] :

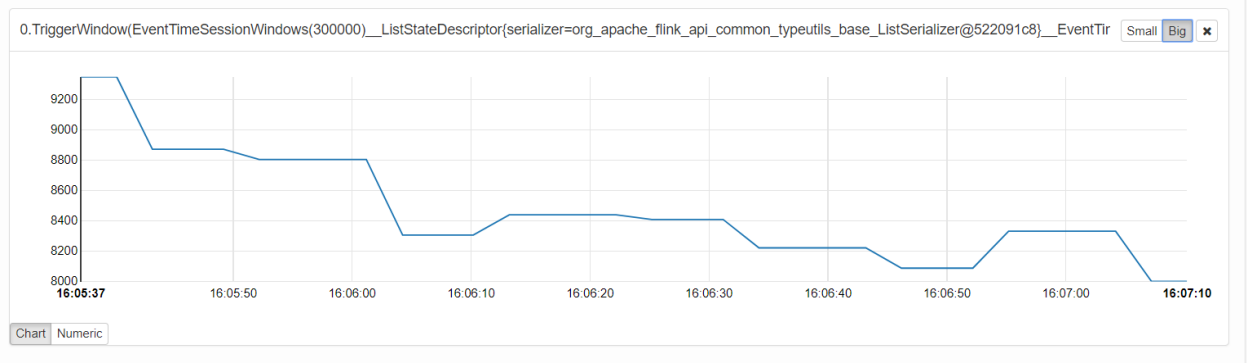

Figure [7]: Session Window Operator

In the following Figure [8] and Figure [9] we can see that The Flatmap operator emits less records (around 2 millions) than it receives (3.5 millions) since it filters noisy ones:

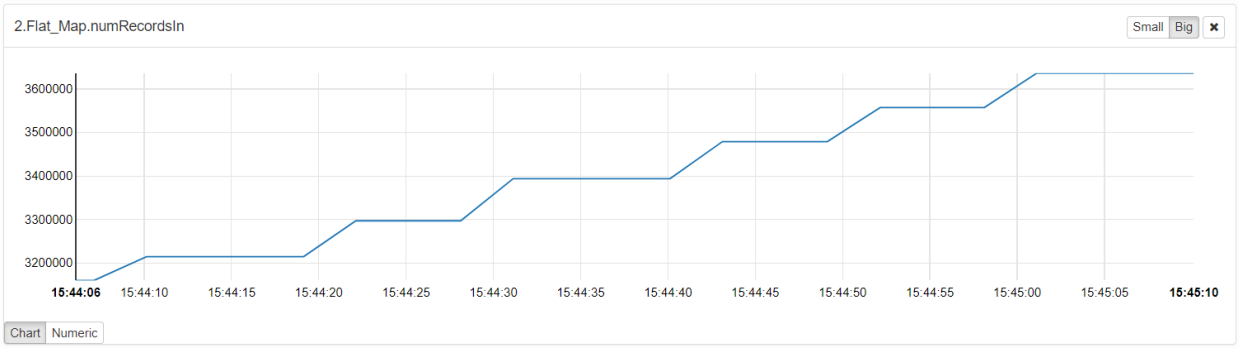

Figure [8]: Flatmap Operator, visualizing the number of Input rows

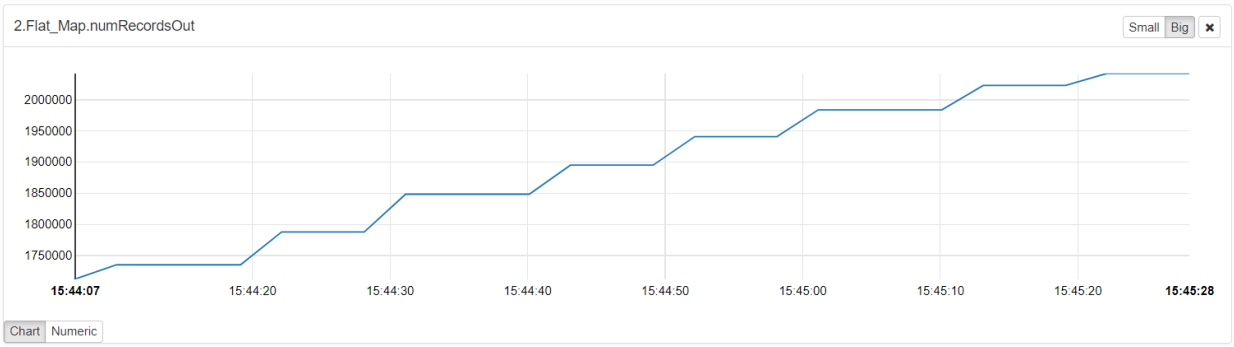

Figure [9]: Flatmap Operator, visualizing Number of Output rows

The patterns are applied to the stream and aggregate matching results into lists, which explain the difference between the number of input/output events in LongTermStop and Smooth Turn in the Pattern matching operator, as described in the next 2 figures, Figure [10] and Figure [11] for the LTS:

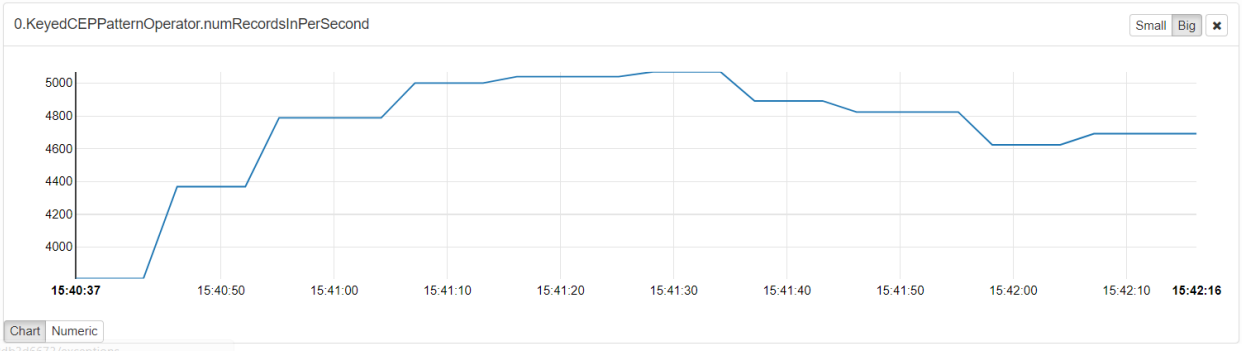

Figure [10]: Number of input records per second in LTS

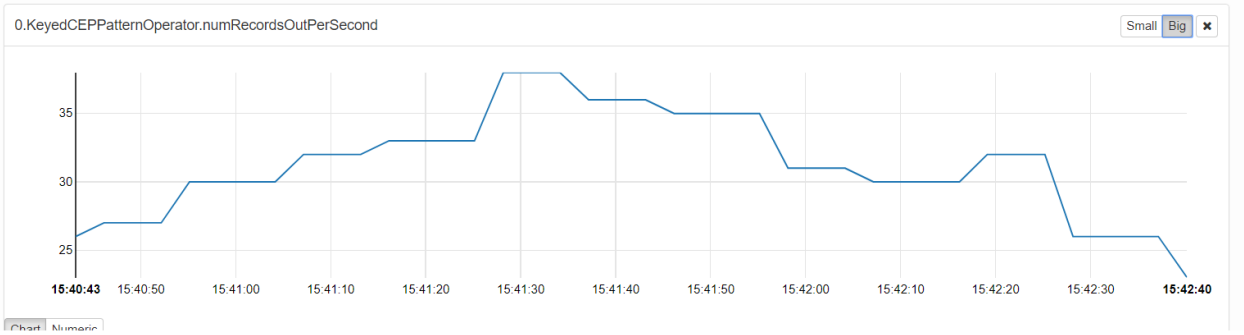

Figure [11]: Number of output records per second in LTS from CEP Pattern Operator

8.1.2 Metrics - Trajectory Detection Module

The “Imis 1month” file from http://chorochronos.datastories.org/?q=node/81 contains one month of data, 58691821 events and its size is 4.7Gb. Using the Ubuntu 16.04.1 x64 Virtual Machine, with 8 cores, 16GB Memory and 160GB SSD Disk, the experiment took 15 minutes.
As a result, we detect:

- 214 997 noisy events
- 101 909 long term stops
- 816 smooth turns
- 1 017 097 gaps (5 minutes gaps)
- 5 939 234 smooth turns

8.2 Complex event Recognition Module

Package Picking

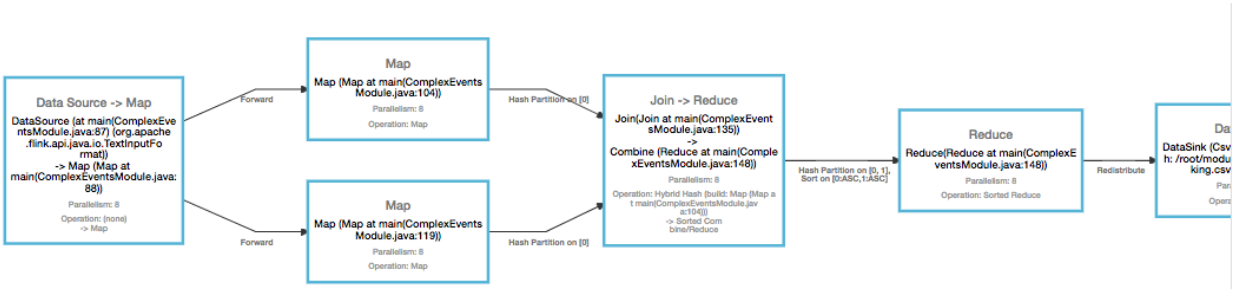

Figure [12]: Directed Acyclic Graph - Package picking Module

Here are visualizations about the DAG that is generated at Package Picking Jobs runtime- Figure [12] - and the number of records in and out/second from the join operator. After a transitional state we reach 50 input records/s and 90 records/s output.
The job took 50 minutes on a local machine of 16GB of Ram to process long term stops extracted from 1 month of data and detect package pickings. The number of long term stops used to recognize package pickings is 97077 events, (start of the stops & end of the stops) and they resulted in 7856 possible package pickings.

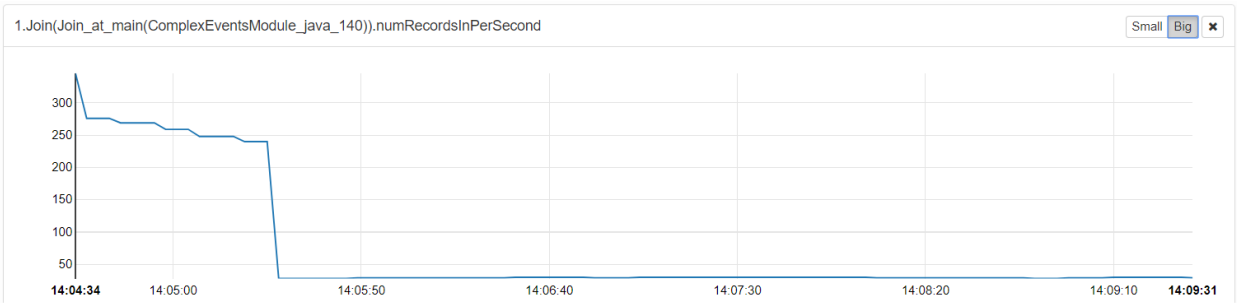

Figure [13]: Metrics in Join Operator - number of Records in per Second While Joining the two datasets.

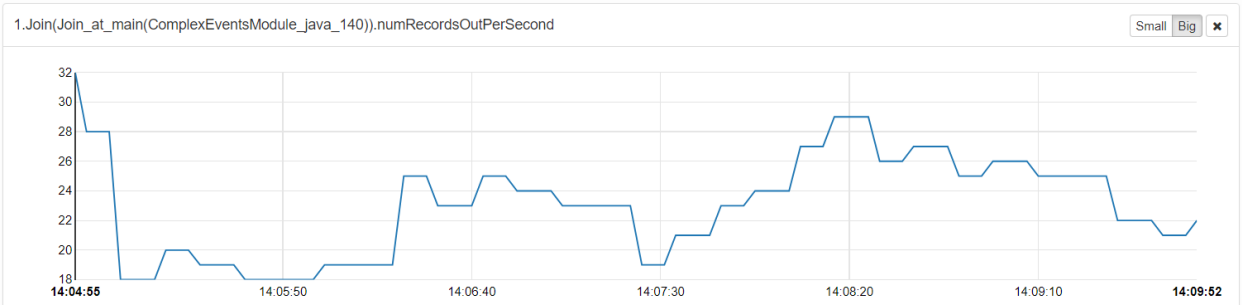

Figure [14]: Metrics in Join Operator - number of Records out per Second while Joining the two datasets.

Here is a sample of possible pickings detected by the module:

Vessel 1	Vessel 2	Picking timestamp
4	1261	1246585929
21	1953	1246858703
32	3020	1247950870
53	897	1246689547
64	1079	1246696652
64	4310	1248794609
92	4065	1247359127
124	1228	1247722378
133	1045	1247446542
133	1104	1246598392

Vessel Rendezvous

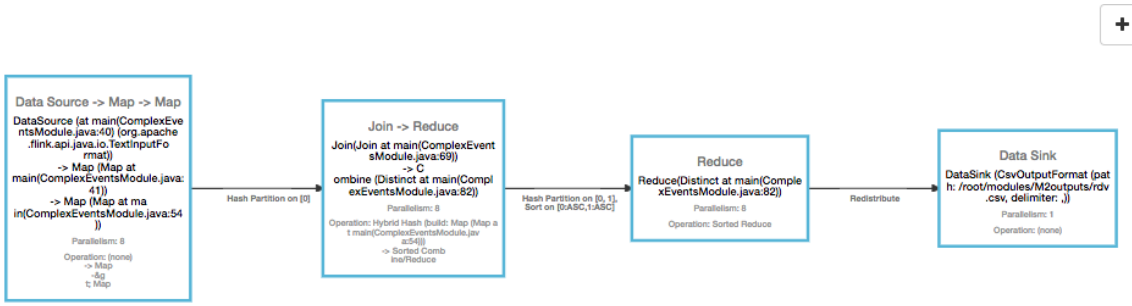

Figure [15]: Directed Acyclic Graph - Vessel Rendezvous.

In Figure [15] is presented the DAG of the Vessel Rendezvous Apache Flink Job and in Figure [16] and Figure [17] is presented the evolution of the number of records in / out of the join operator in the vessel rendezvous pipeline:

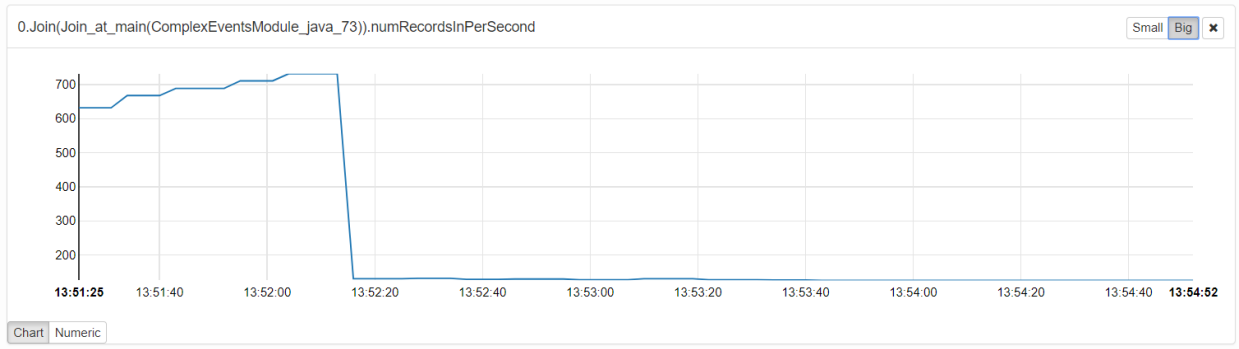

Figure [16] Number of input rows per second in Join Operator for Vessel Rendezvous

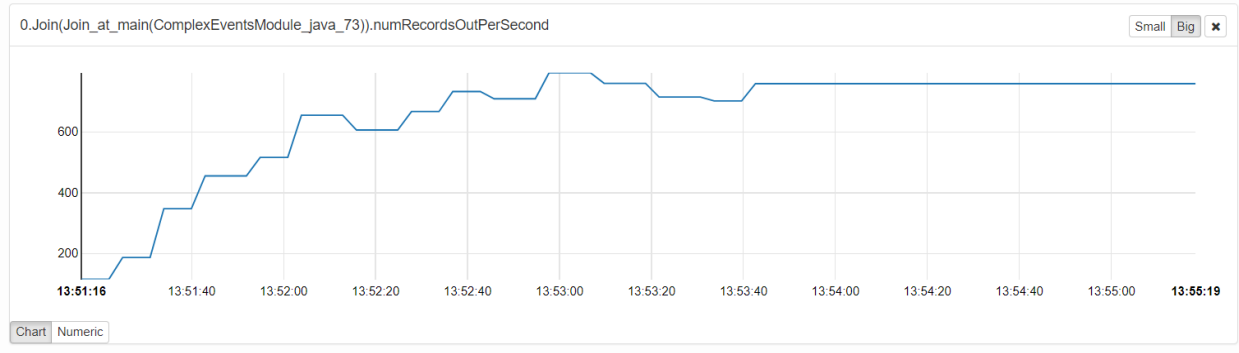

Figure [17] Number of input rows per second in Join Operator for Vessel Rendezvous

After a transitional state, the number of input records/s becomes steady around 120 record/s
And so does the number of out records /s which becomes constant at around 760 record/second.
The job takes 60 minutes to process 1 month of data on a local machine having 16GB of Ram.

Here are some possible vessel rendezvous detected.

Vessel 1	Vessel 2	RDV start	RDV end
1	88	1246443231	1246443273
1	1549	1246443251	1246443291
1	2676	1246431080	1246431112
1	2853	1246439991	1246440032
2	9	1246977284	1246977326
2	236	1246892513	1246892535
2	259	1246990826	1246990844
2	289	1247592663	1247592682
2	370	1246990843	1246990844
2	373	1246895054	1246895092

Fast Approach

The main operator in the fast approach pipeline is the join operator. We perform a join using a sort merge strategy provided by Flink, in which Flink sorts the inputs join keys before performing the join. An advantage of this strategy is that it is robust, in case the memory is limited with respect to the dataset size (which is our case) it spills the data to the disk to perform the sort.

The job takes 5 days to finish processing 1 month of data on a big machine of 128 GB memory.

Here is the evolution of the number of record in out of the join operator during the run:

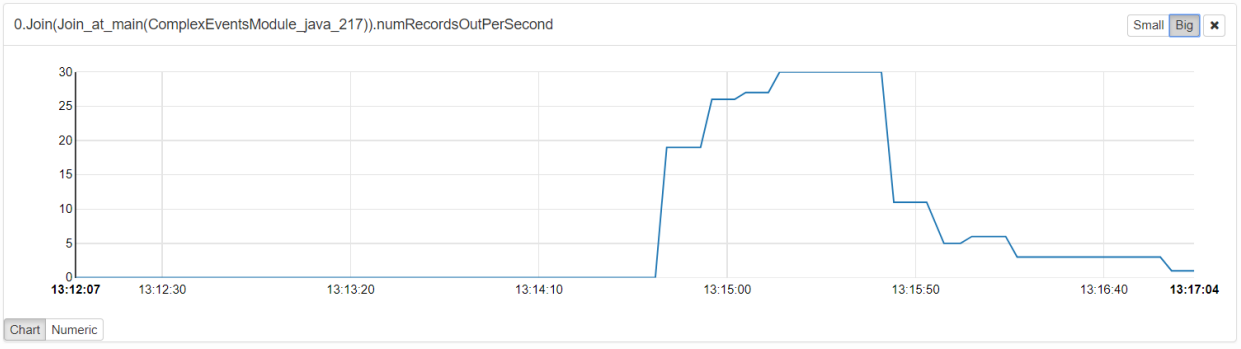

Figure [18] Number of output rows per second in Join Operator for Fast Approach Flink Job

8.2.2 Metrics - Complex Event Recognition

The experiment of Package Picking took 1h and 15 min, and we detected 7856 Package pickings.
The experiment of Vessel Rendez-Vous took 8h and 30 min and we detected 752551 Vessel Rendez-vous.
The experiment of Fast Approaches took 5 days (using 128Gb Ram) and we detected 10376275 Fast Approaches.

9 References

[1] Apache Flink: Scalable Stream and Batch Data Processing. Accessed on line at https://flink.apache.org/ - The Apache Software Foundation.

[2] Apache Flink Training. Accessed online at http://training.data-artisans.com/

[3] Chorochronos.org Datasets & Algorithms. Accessed on line at http://www.chorochronos.org/?q=node/9

[4] Event Recognition for Maritime Surveillance Kostas Patroumpas, Alexander Artikis, Nikos Katzouris, Marios Vodas, Yannis Theodoridis, Nikos Pelekis.

[5] Flink in Action by Sameer B. Wadkar, Hari Rajaram.

[6] How not to drown in a sea of information: An event recognition approach Elias Alevizos, Alexander Artikis, Kostas Patroumpas Marios Vodas, Yannis Theodoridis, Nikos Pelekis.

[7] Introduction to Apache Flink. Stream Processing for Real Time and Beyond by Ellen Friedman & Kostas Tzoumas.

[8] Kibana User Guide. Accessed on line at https://www.elastic.co/guide/en/kibana/current/index.html

[9] Logic-Based Event Recognition Alexander Artikis, Anastasios Skarlatidis, Françoις Portet and Georgios Paliouras.

[10] Mastering Apache Flink by Tanmay Deshpande.

[11] Online Event Recognition from Moving Vessel Trajectories Kostas Patroumpas · Elias Alevizos · Alexander Artikis · Marios Vodas · Nikos Pelekis · Yannis Theodoridis.

[12] Stream Processing with Apache Flink Fundamentals, Implementation, and Operation of Streaming Applications by Fabian Hueske, Vasiliki Kalavri.

[13] A Distributed Event Calculus for Event Recognition by Alexandros Mavrommatis, Alexander Artikis, Anastasios Skarlatidis and Georgios Paliouras