24th International Conference on Business Information Systems (BIS 2021)
Applications
https://doi.org/10.52825/bis.v1i.41
© Authors. This work is licensed under a Creative Commons Attribution 4.0 International License
Published: 02 July 2021

Stream Processing Tools for Analyzing Objects in Motion Sending High-Volume Location Data

Krzysztof Węcel[1[https://orcid.org/0000-0001-5641-3160]], Marcin Szmydt[1[https://orcid.org/0000-0003-4392-0205]], and Milena Stróżyna[1[https://orcid.org/0000-0001-7603-7369]]

[1] Poznań University of Economics, Poland

Abstract. Recently we observe a significant increase in the amount of easily accessible data on transport and mobility. This data is mostly massive streams of high velocity, magnitude, and heterogeneity, which represent a flow of goods, shipments and the movements of fleet. It is therefore necessary to develop a scalable framework and apply tools capable of handling these streams. In the paper we propose an approach for the selection of software for stream processing solutions that may be used in the transportation domain. We provide an overview of potential stream processing technologies, followed by the method for choosing the selected software for real-time analysis of data streams coming from objects in motion. We have selected two solutions: Apache Spark Streaming and Apache Flink, and benchmarked them on a real-world task. We identified the caveats and challenges when it comes to implementation of the solution in practice.

Keywords: stream processing, location data, transport, mobility, AIS

Introduction

The recent rapid development of communication and detection technologies, the emergence of low-cost and widespread smart sensors, and a significant drop in data storage costs have all contributed to a significant increase in the amount of easily accessible data on transport and mobility. The volume and speed at which sensor data is generated, processed, and stored is unprecedented [1].

The advent of Big Data, including massive streams of real-time data of high velocity, magnitude, and heterogeneity, have triggered changes in many fields. One of them is the transport sector that manages a massive flow of goods and at the same time creates large data sets. These data streams concern inter alia mil-lions of shipments and the movements of fleet that are tracked every day [2]. This data is then considered as a source for context-aware applications and intelligent services, aiming to improve traffic efficiency, safety, and security [3] as well as last-mile delivery optimization, route optimization, fleet management, and detection of anomalous behavior [2] These services are applied in various transport modes, such as road, maritime or public transport as well as in various forms of shared transport (e.g., carsharing, bike-sharing). For example, data extracted from computers embedded in vehicles, that concern an object in motion, i.e., its origin, destination, content, and location, can be used to better understand and predict the flow of goods and people in real time. These data streams can be further combined with other data gathered from navigational systems, mobile phones (e.g., location, activities), environmental sensors (e.g., pollution levels), or social net-works (e.g., people's preferences and relationships).

Nevertheless, the potential of this massive amount of data can be exploited only if there are tools and models able to efficiently extract, detect and analyze relevant information from the data streams [4]. However, most of the approaches or methods applied in the existing

transportation systems and applications do not fit the paradigm of Big Data analytics [1]. The challenge now is not only to collect the data but to draw conclusions from them. Therefore, it is required to develop modern system architectures that would allow to efficiently process large data streams. As indicated by [5] in their systematic over-view of big data stream analysis, a significant amount of previous research has been directed to real-time analysis of big data streams and not much attention has been given to the preprocessing stage. Moreover, only a few big data streaming technologies offer a possibility to do both batch, streaming and iterative jobs. Therefore [5] suggest that research effort should be directed towards developing scalable frameworks and architectures able to accommodate data stream computing mode, effective resource allocation strategy, and parallelization issues.

In order to design such a scalable and efficient architecture, a number of choices have to be made by designers, including the selection of proper software and hardware. In response to this challenge, we propose an approach for the se-lection of software for stream processing solutions that may be used in the transportation domain (analysis of data streams generated by objects in motion). The aim is to show steps that need to be followed while designing system architecture for data stream analysis. To achieve this goal, we provide an overview and characterization of some stream processing technologies, followed by proposing a method for comparing the selected software for real-time analysis of data streams coming from objects in motion along with identification of the caveats and challenges when it comes to implementation of the software in practice.

We designed and implemented in the selected technologies a Minimum Working Example (MWE) in which an exemplary data stream with location of moving objects is used. The stream is roughly 10 MB per minute, which gives around 15 GB for a 24h time window. The idea is to perform the real-time anomaly detection on a sliding window using the smallest time interval possible. This naturally depends on the size of the data, the complexity of the used algorithms, and avail-able processing power.

The paper is structured as follows: in section 2 a comparison of the batch and streaming processing is provided, followed by software selection options (section 3) and description of the method (section 4). Section 5 and 6 present the results of benchmarking of the selected tools and discussion of the result. The paper ends with conclusions and indication of future work.

Background

The focus of this paper is stream processing. It is then necessary to explain the characteristic features of this approach to processing compared to more traditional batch processing.

Batch processing is the processing of data in a group, referred to as batch. This means that data has to be collected first and then processed on-demand or based on a schedule. In this case we know in advance the volume of data to work on and the system we use for calculations can plan processing steps accordingly. Algorithms can also be designed in a way to allow optimization of the resources used. The batch processing is dedicated for large quantities of data that is usually not time-sensitive.

In the **stream processing** there is no collection of data for "future" use. Data is sent to analytical modules immediately. Such an approach allows addressing the real-time or near-real-time scenarios. This also entails additional requirements for the processing system. It is also more difficult to design algorithms that need to consider a constant update of the results when new records arrive in a stream.

There are then several features specific for the stream processing [6].

- Buffering past input as streaming state.

Data is not stored explicitly but this does not mean that it is forgotten. Arriving data changes the way we would interpret the overall situation or data to come. Thus, it changes the so-

called streaming state. The typical case in the studied domain would be to remember the last known position of a moving object, along with a timestamp. The state can also be remembered as values accumulated over time. Many algorithms can be implemented this way, where future input can be matched with past input.

- Joining streams with streams

It is very common in the SQL world to join tables to perform a query. The same can apply to the stream processing but here instead of static tables we have dynamic streams. Joining streams thus requires an additional temporal dimension to be considered. For example, we can detect if tracked moving objects were close to each other in a specific time frame.

- Streaming aggregations

Another very common query type in SQL is aggregation. A big number of rows is reduced to a small number of unique entities with assigned totals. The same can also apply to streams. One of the options is an accumulation of values over the whole history. The most demanding, however, is the accumulation of values in a specific time frame. For example, one can request the total number of moving objects observed in a monitored area in the past 24 hours. Results of the query will differ depending on the moment of its execution. In the stream processing, we need to have the answer at any moment, which represents the current state of the world characterized by the collected data. The totals in aggregations change when new data arrives.

- Handling late and out-of-order data

This issue is a consequence of how data is collected. We do not have any control over the objects, whether they send data immediately or need to cache because of connectivity issues. Data cannot be sorted either because it is not physically stored. Storing it locally would mean constant reprocessing because of out-of-order events and that the overall would be counterproductive. Therefore, additional measures were undertaken to handle it. First of all, the most distinctive for the stream processing is watermarking, which allows tracking of events (data accompanied by a timestamp) that still need to be processed although being late. The out-of-order events are not problematic as the stream processing considers them in specific time frames.

Software selection

The analysis of moving objects is a task that is best performed using stream processing. For this purpose, we need to choose appropriate software and this is what the paper is about. Currently, on the market, there are a couple of computation engines that support stream processing. The literature review reveals few studies focused on benchmarking streaming computation engines [7, 8, 9, 10]. However, those studies show different use cases and there is no clear winner. Therefore it is necessary to design our criteria for a specific case. Thus, let us discuss the selection of criteria for such software.

Criteria for Selection

There are many software solutions supporting this use case. We, therefore, de-fined criteria to support designers in making the choice. They are as follows:

- Horizontal scalability, that is the scalability of a cluster of many processing nodes. The horizontal scalability allows going beyond what a single machine, used in vertical scalability, may provide, as, depending on the requirements, one may distribute the processing to hundreds of thousands of nodes, which delivers resources many times greater than any vertical scaling may provide.
- Maturity of the solution and commercial as well as community support. While there are multiple big data stream processing solutions, we recommend picking the ones which are more widely adopted and available for not less than several years. This is

due to the fact that streaming applications running on clusters are notoriously difficult to debug. While some brand-new cutting-edge technologies may provide some benefits, without proper community support (on some web forums, etc.) it may be very difficult to find root causes for different issues that we may encounter along the way.

- Suitable licensing for a commercial project.
- Support by cloud providers. Some software solutions are well adopted on cloud platforms, which greatly increases ease of deployment and subsequent management of the system in production. Theoretically, we may use a cloud platform only in the Infrastructure as a Service (IaaS) model and install all the software manually on the hardware operated by the cloud platform. Then, we may use any software we like. Still, this does not allow us to utilize some of the most important advantages of the cloud. The cloud platforms provide more managed solutions, which deliver both the hardware and pre-configured soft-ware distributions, along with functions of monitoring, encryption, etc. Such managed solutions are nevertheless available only for limited software solutions and, for example, not all stream processing engines may be used this way.

Our next step was the selection of software candidates based on a keyword search for "stream processing" and "stream analytics", along with "big data".

First Round Candidates

As one of our primary requirements was the stream processing, we have focused on software providing this capability. For each solution, we filled in details concerning an implementation language, a license, and stream features. We also con-fronted the features with our criteria for selection and presented the major pros and cons. Below we shortly characterize main solutions in this area.

Apache Spark[1] is implemented in Scala and can be used from Scala, Java, and Python. It is based on the Apache license. It supports the stream storage and the stream processing. As an advantage, we can mention tight integration with the batch processing framework and ML library and the capability for the structured streaming. It is also production-ready. On the negative side, it is a near-real time solution (up to a few seconds of delay). Continuous processing with low (~1 ms) end-to-end latency is still in the experimental phase, hence, not recommended for production environments.

Apache Kafka[2] is written in Scala and Java. It can be integrated with many languages, including C++, Haskell, Node.js, OCaml, PHP, Python, Ruby, C#/.NET. It is based on the Apache license. It supports the stream storage and the stream processing. On the positive side, it is very fast, mostly due to the simplicity – it is more a low-level publish-subscribe message broker than a data processing pipe-line.

Apache Flink[3] is also implemented in Java and Scala and additionally pro-vides interfaces to Python. It is based on the Apache license. It supports the stream storage and the stream processing. Its main advantage is speed – it is faster than Spark, and seems to be reliable and powerful for complex stream processing.

Apache Pulsar[4] is dubbed as a cloud-native, distributed messaging and streaming platform. It can be bound with Java, C++, Python, and Go. It supports the stream storage but not the stream processing. It is a publish-subscribe messaging system, and in this respect, it is very similar to Kafka. Still, Kafka is more efficient, supports more features and is easier to use.

[1] http://spark.apache.org
[2] https://kafka.apache.org
[3] https://flink.apache.org
[4] https://pulsar.apache.org

Apache Storm[5] can be used with JVM-based languages (predominantly Clojure), JavaScript, Python, and Ruby. It is based on the Apache license. It supports the stream processing but does not support the stream storage. Its advantage is speed. The drawback is that it is not so convenient for more complex tasks.

Apache Samza[6] is implemented in Java and Scala. It is based on the Apache license. It supports the stream processing but does not support the stream storage. There are not any advantages related to our task. It is not suitable for more complex tasks and depends on Kafka. It is not actively developed (the recent stable version was launched almost 2 years ago).

In **Table 1** we provide the comparison of the main features of the studied solutions.

Table 1. The comparison of the main features

Solution	Horizontal scalability	Maturity/ community	Suitable licensing	Cloud support
Apache Kafka	++	++	++	++
Apache Spark	++	++	++	++[7]
Apache Flink	++	+	++	++[8]
Apache Storm	++	-	++	-
Apache Samza	++	-	++	-

It is important to note that both Apache Flink and Apache Spark[9] are supported by cloud providers as a managed solution. An example of a good integration with the cloud is EMR[10]. It is the industry-leading cloud big data platform for processing vast amounts of data using open source tools such as Apache Spark and Apache Flink.

Apache Kafka is mentioned as an underlying solution both for Flink and Spark. It is not a fully-fledged stream processor but merely a streaming engine. It can be considered as a kind of database that needs higher-level tools to work with. Therefore, in the next sections, we will study the short-listed candidates, i.e., Apache Spark and Apache Flink.

Apache Spark

According to the documentation, Apache Spark is a unified analytics engine for large-scale data processing[11]. It was originally designed for the static datasets batch processing (Spark generic) and the streaming data processing (Spark Structured Streaming) was added later.[12]

Spark can be run using its standalone cluster mode or on the already existing clusters. Its engine can access diverse data sources. The important feature of Spark is being scalable using Hadoop YARN, Mesos, or K8s. Spark (generic) has also support for powerful data engineering and machine learning libraries (e.g. MLlib). Originally, Spark was written in Scala. However, there are also APIs available in Java and Python. It is an open-source solution and has a huge community. Spark Structured Streaming supports the stream processing using a micro-batch processing engine (100 milliseconds latencies) and continuous processing (still in an experimental phase). It ensures an end-to-end exactly-once fault-tolerance guarantee.

[5] http://storm.apache.org
[6] https://samza.apache.org
[7] https://aws.amazon.com/emr/features/spark/
[8] https://aws.amazon.com/about-aws/whats-new/2019/11/you-can-now-run-fully-managed-apache-flink-applications-with-apache-kafka/
[9] https://aws.amazon.com/emr/features/spark/
[10] https://aws.amazon.com/emr/
[11] https://spark.apache.org/docs/latest/
[12] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

From a practical point of view, a stream processed by Apache Spark Structured Streaming is treated as an unbounded structured table and it is possible to perform the same operations as in static Spark SQL. Then, the streaming output can be saved on various sinks such as Kafka or file system files.

Apache Flink

According to the authors of the solution, Apache Flink is "a framework and distributed processing engine for stateful computations over unbounded and bounded data streams."[13] What is referred to as bounded stream processing is in fact equivalent to the batch processing. Unbounded streams have a start but no defined end and are equivalent to the stream processing.

It is important to emphasize that Apache Flink was designed with data streams in mind – it is referred to as a stream processor. The developers also paid attention to flexibility in programming, hence Flink allows for integration with SQL and Table API. Finally, in the context of the moving object analysis scenarios, one of the interesting features could be CEP (Complex Event Processing) for pattern discovery.

The main feature of the Flink approach is the abstraction of streams. They are perceived as temporary tables, also referred to as dynamic tables. The dynamic tables are changing over time. They can be queried like static batch tables but using a continuous query. Such a query never terminates and produces a dynamic table as a result. It is important to note that the result of the continuous query is always semantically equivalent to the result of the same query being executed in the batch mode on a snapshot of the input tables.

Summarizing, the following steps are performed in Flink to return the analysis results. 1. A stream is converted into a dynamic table. 2. A continuous query is evaluated on the dynamic table yielding a new dynamic table. 3. The resulting dynamic table is converted back into a stream.

Method

In order to choose the most appropriate tools for the analysis of objects in motion that send high-volume location data we have decided to get hands-on and test solutions on real data streams. This gave an extra opportunity for the identification of additional obstacles that are not clearly stated in the documentation. Also, we could assess the community involvement which is also very important as companies usually struggle with the implementation of more complex scenarios than the ones presented in tutorials.

Minimum Working Example

Based on data streams available for this study, we have defined a simple scenario. Two teams were implementing the same functionality: one in Apache Spark and the other in Apache Flink. They were supposed to come up with a so-called minimum working example (MWE). We then measured performance in terms of consumption of system resources (CPU, RAM). Please note that we did not consider the running time as we were operating on a data stream, so there is no such notion as the end of processing.

Implementation of MWE served several purposes. First, we checked the capabilities of the analyzed solutions. Second, it provided hints as to how difficult it is to implement a specific scenario, what are the challenges and obstacles. Third, we could compare the efficiency of solutions measured by CPU and consumption memory requirements.

The scenario for analysis consisted of counting the number of messages sent by moving objects, grouped by type of the object. There were two additional variants: counting all

[13] https://flink.apache.org/flink-architecture.html

messages for the whole history of observation and counting messages within the 5-minutes tumbling windows.

There were also additional non-functional requirements for the implementations. The streams to be used were served by Apache Kafka in CSV format. There were several streams and they had to be combined and deduplicated before any further processing. After the processing, the resulting streams should have been sent back in Avro format and stored in a database. Whereas the second variant could use “insert” queries (aggregation within window), the first required “upsert” queries (both insert and update), and message keys were necessary (global aggregation).

Getting System-wide Measurements

In order to compare both implementations, system-wide performance metrics were designed and then monitored. For the purpose of the monitoring, no other significant processes were running on the host machine at the time of those measurements. The main focus in the study was put on memory, processor load, and message processing lag. Therefore, the following metrics were used in the experiment:

- Average CPU (15 min, 12h) and Max CPU (1 min, 12h)

The server average CPU load during the 15 minutes sliding window was measured every minute for the period of 12 hours. The final value for this metric is a simple mean of those averages. Similarly, it was possible to calculate the maximum CPU usage. However, for the maximum CPU load, the one-minute sliding window was considered. The bash script used for recording the measurement is following:

```
while true; do uptime >> uptime.log; sleep 1; done
```

- Average RAM (1s, 12h) and Max RAM (1s, 12h)

The average RAM usage across all server cores was measured every second during the period of 12 hours. The final metric was an average of the recorded measurements. Similarly, the max RAM metrics refer to the single maximal average RAM usage value across all cores during the period of 12 hours. The bash script used for recording the measurement is following:

```
while true; do
  echo "$(date)" `cat /proc/meminfo | grep Active: | sed
's/Active: //g'`/`cat /proc/meminfo | grep MemTotal: | sed
's/MemTotal: //g'` >> ram_monitoring.txt
  sleep 1
done
```

- Average 5 min lag (12h), Max 5 min lag (12h), and Percentiles 5 min lag (12h)

The experiment of counting and grouping received messages for the last 5 minutes (moving windows) with 5 minutes watermark involved inserting results to the relational database. The lag metrics refer to the time difference between the count result database insert timestamp and the timestamp of the sliding window period (end window timestamp). Those metrics can be considered as an average, maximum, and percentiles of event processing delay during the period of 12 hours. The metrics were calculated by creating appropriate SQL query for the relational database containing the results.

Results

We have conducted our benchmarking on the following hardware.

Processor	2 x Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz 16 cores
Memory	DRAM 256 GB
Disk Type	HDD, 4x 1.2TB 10K RPM SAS, transfer 6Gb/s, configured as RAID 5, interface SATA 6Gb/s / SAS 12Gb/s, PCIe 3.0
Network Adapter	4 x Broadcom Limited NetXtreme BCM5720 Gigabit Ethernet PCIe, 2x10Gb BT + 2x1Gb BT
Operating System	CentOS Linux (Version 7)

Below we present the results achieved with the two tools: Apache Spark Structured Streaming and Apache Flink. We compare them based on the system-wide measurements presented in the previous section.

Apache Spark Structured Streaming

This section presents the results obtained using single-machine (standalone mode) with Spark Structured Streaming (Scala) to deploy MWE (grouping and counting received messages in the 5 min window frames and global counts). From **Figure 1** and **Figure 2** we can infer that the results are stable throughout the whole measurement period. **Table 2** show average results.

Table 2. Spark Structured Streaming - MWE Performance Metrics

KPI	Value
Average CPU (15 min, 6h)	32.46%
Max CPU (1 min, 6h)	43.99%
Average RAM (6h)	66.96 [GB]
Max RAM (6h)	72.96 [GB]
Average 5 min lag (6h)	8.36 min
Max 5 min lag (6h)	12 min
95 percentile 5 min lag (6h)	10 min
Median 5 min lag (6h)	8 min
5 percentile 5 min lag (6h)	7 min

Source: Own work.

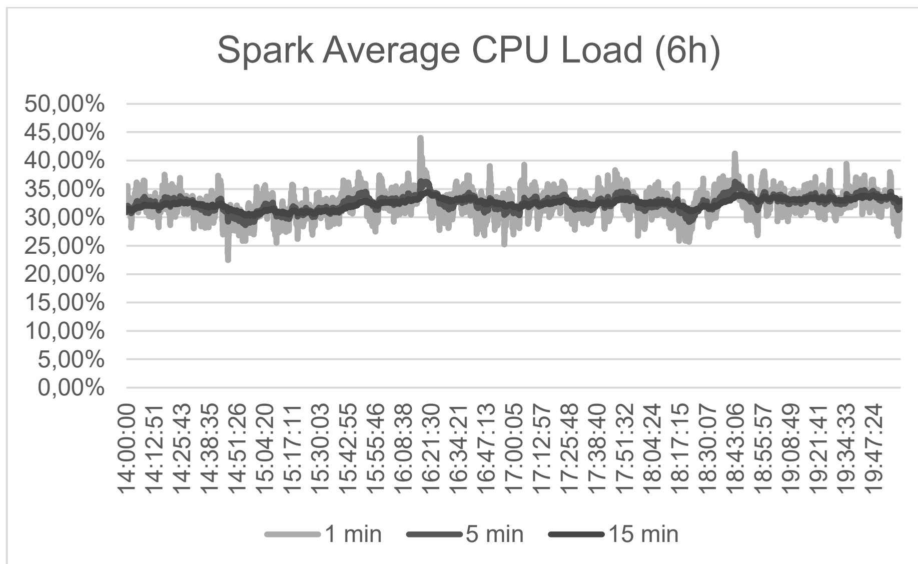

Figure 1. Average CPU load in Spark in various intervals

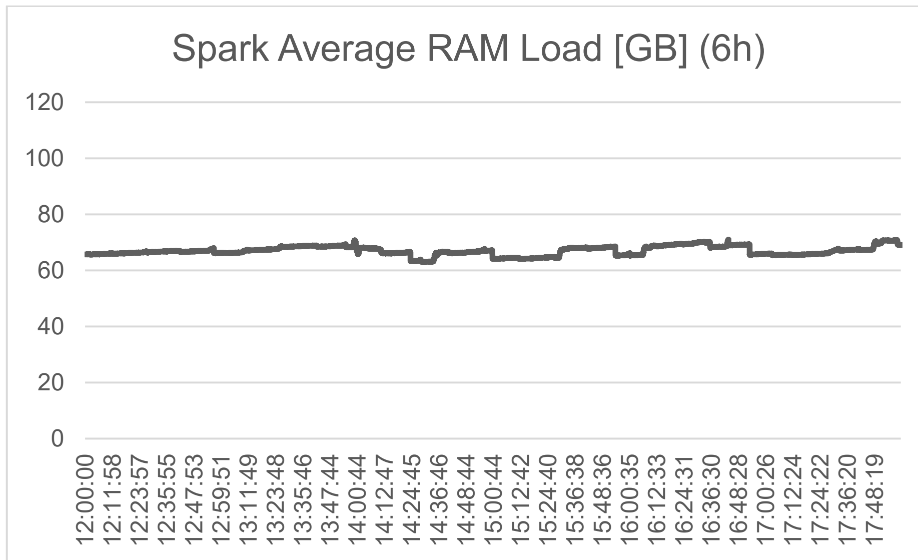

Figure 2. RAM usage in Spark

Apache Flink

The Flink job of making aggregations according to the described scenario was responsible for counting messages sent by objects in the 5 min window frames. The global counts were running during the measurement of the first metric. Below we present the summary statistics (**Table 3**), analogously to Apache Spark. **Figure 3** presents CPU load and **Figure 4** memory usage.

Table 3. Flink MWE Performance Metrics

KPI	Value
Average CPU (15 min, 6h)	4.20%
Max CPU (1 min, 6h)	7.50%
Average RAM (6h)	102.00 GB
Max RAM (6h)	108.69 GB
Average 5 min lag (6h)	11.10 min
Max 5 min lag (6h)	22 min
95 percentile 5 min lag (6h)	21 min
Median 5 min lag (6h)	9 min
5 percentile 5 min lag (6h)	6 min

Source: Own work.

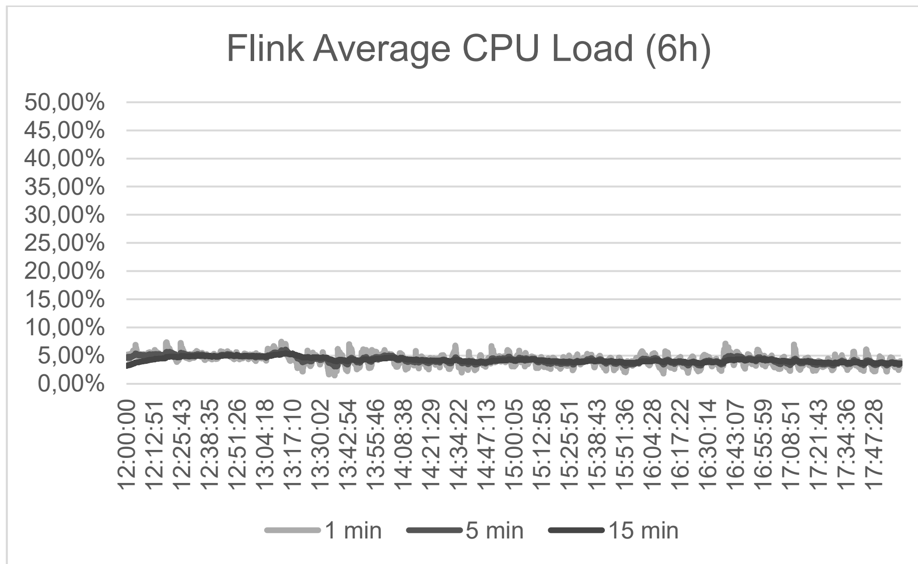

Figure 3. Average CPU load in Flink in various intervals

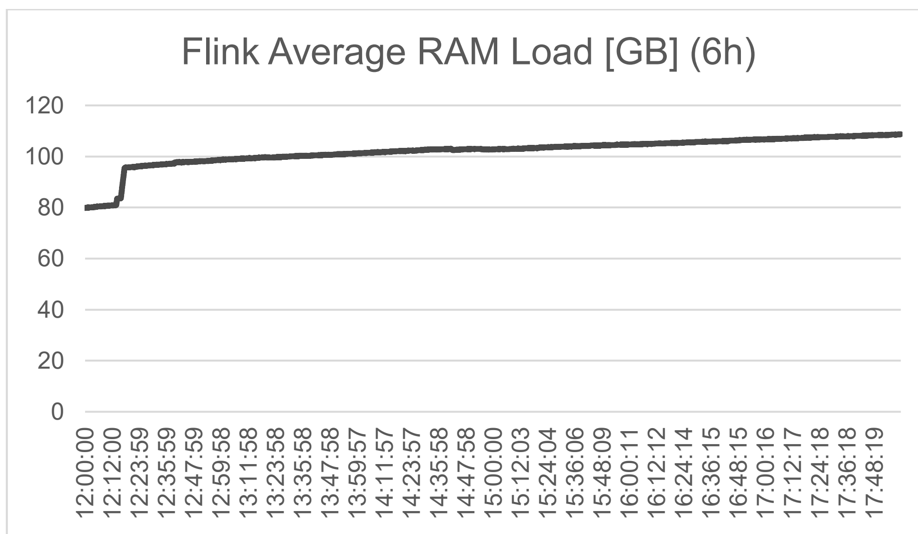

Figure 4. RAM usage in Flink

Discussion

In this section, we share our experience when it comes to the implementation of MWE in Spark and in Flink, as both provided a lot of technical challenges. The most cumbersome was understanding how the implementations work under the hood. Code examples available on the Internet were not always useful as they were fragmentary and did not mention any API version.

Initially, we assumed the transfer of data from streams to the stream processor via CSV files. Files were placed on RAID and mounted with NFS. Reading from files (FileSource) had a huge cold-start problem - around 60 seconds per query on data collected for only one day. Kafka was much faster to deal with this issue.

As Kafka was not planned in the first iteration of MWE implementation, initially we assumed that data will be stored directly in a database. Whereas Flink had no problems with such an approach, the JDBC driver for Spark did not provide update (UPSERT) mode. As a solution, we used Kafka topics to transfer data from Spark to a database. For comparability, Flink followed the same steps. Sending output to Kafka topics and creating a Confluent Kafka connector for saving data to the database worked with UPSERT mode as expected.

Concerning the output, saving Spark output to the Kafka topic in Avro format (instead of JSON) is the most popular way with the best community support. Message exchange between Flink and Kafka can be done in JSON and in Avro but the latter is more efficient.

There were also different approaches to submit a job. Scala does not go well with Jupyter Notebooks. In case of Spark, which was implemented in Scala, we executed jobs using either spark-shell or spark-submit with a prepared JAR file. For submission of job in Flink, which was implemented in Java, we used a compiled JAR file either.

The most important feature in the stream processing is time. Implementations in Python (pyspark and pyflink) were missing some important API calls referring to time windows. Moreover, documentation for Python was very often missing. Although we usually code in Python, we had to switch to Java for Flink and to Scala for Spark. Extending on time, it is very important to verify time zones in each environment (i.e., Docker containers, spark-shell, host machine, database) to prevent inconsistencies.

One of the specific challenges was the discovery of how to perform a specific step to achieve the required functionality. Although many tutorials were present for some solutions, there was always a “magic sauce” that was not revealed to the others. The discovery of an appropriate approach was mainly a trial-and-error process.

For example, in the case of Flink, there were drastic changes of API between 1.9 (when we started), 1.11 (our implementation target), and 1.12. (current stable); version 1.13 is in production. API is in constant reengineering and many methods were deprecated along with changes in classes. Although many examples were available on different fora, they were usually correct for older versions of Flink and were deprecated when we tried to run examples in version 1.11. The Flink project is developed by Alibaba, hence many discussions on problems are in Chinese.

Missing detailed documentation is also characteristic for Spark. In the documentation, there are only code snippets and not fully blown examples. Therefore, expertise in the selected tools is necessary to understand a missing context.

Conclusions and Future Work

Looking for a technology capable of processing streams for analyzing objects in motion that send high-volume location data we came up with a list of potential solutions, which was later restricted to Kafka, Flink, and Spark. The objective of the paper was to narrow the selection and identify the caveats and challenges to be verified in the second phase of our benchmarking effort. So far we have implemented a simple Minimum Working Example to make sure we know the effort necessary to implement a fully-fledged solution. We found that a learning curve is rather steep, especially when we consider more-than-standard requirements, i.e., integration with Confluent’s version of Kafka, storage of intermediate results in a database with key-based updates, exchange of messages in Avro serialization format. To meet these requirements Spark required an external library and for Flink we needed to provided dedicated implementations of some interfaces.

Concerning the efficiency of the solutions, Spark seems to be more memory efficient but at the cost of a higher CPU. Spark job made the machine quite busy with 30% CPU usage; allocated RAM was at the range of 70 GB. Flink was much more efficient – the same task caused a load of only 5%, but RAM consumption reached 110 GB. As we cannot tell which solution is preferred in the mentioned task, for future work we plan to extend MWE with additional scenarios. These scenarios will cover more tasks typical for the analysis of moving objects. In Scenario “Time Difference” we will measure the time passed since the last message was received; we can then alert if the time difference exceeds a certain threshold. In Scenario “Area Count” we will count the number of objects in a given area in a given moment or entering the area. In Scenario “Value Difference” we will measure the difference

between values of a parameter in two consecutive messages, e.g. speed of an object to detect if it is accelerating or slowing down.

Another direction of extension is the improvement of the measurement methodology. So far we have used the system-level measures which are just aggregations and do not help much in the identification of weak points of each solution. In the future, we plan to use the built-in solutions available via additional monitoring API. Both Flink and Spark offer their own interface for monitoring various additional parameters, like for example heap size or separation of CPU usage between manager and workers. We will look for a common denominator to compare Spark and Flink in greater detail. The ultimate goal of the next paper will be confronting the capabilities of the tools with specific algorithms according to requirements to determine architecture choice.

References

[1] Amini S, Gerostathopoulos I, Prehofer C. Big data analytics architecture for real-time traffic control. *2017 5th IEEE International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS)*. 2017 5th IEEE International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS). 2017 06. https://doi.org/10.1109/mtits.2017.8005605

[2] Lekić M, Rogić K, Boldizsár A, Zöldy M, Török Á. Big Data in Logistics. *Periodica Polytechnica Transportation Engineering*. 2019 Dec 17;49(1):60-65. https://doi.org/10.3311/pptr.14589

[3] Xu H, Lin J, Yu W. Smart Transportation Systems: Architecture, Enabling Technologies, and Open Issues. In: Sun Y, Song H, eds. *Secure and Trustworthy Transportation Cyber-Physical Systems*. Vol SpringerBriefs in Computer Science. SpringerBriefs in Computer Science. Singapore: Springer; 2017. https://doi.org/https://doi.org/10.1007/978-981-10-3892-1_2

[4] Nguyen D, Vadaine R, Hajduch G, Garello R, Fablet R. A Multi-Task Deep Learning Architecture for Maritime Surveillance Using AIS Data Streams. *2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA)*. 2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA). 2018 Oct. https://doi.org/10.1109/dsaa.2018.00044

[5] Kolajo T, Daramola O, Adebiyi A. Big data stream analysis: a systematic literature review. *Journal of Big Data*. 2019 06 06;6(1). https://doi.org/10.1186/s40537-019-0210-7

[6] Hueske F, Kalavri V. *Stream processing with Apache Flink*. O'Reilly; 2019.

[7] Chintapalli S, Dagit D, Evans B, Farivar R, Graves T, Holderbaugh M, Liu Z, Nusbaum K, Patil K, Peng BJ, Poulosky P. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). 2016 05. https://doi.org/10.1109/ipdpsw.2016.138

[8] Marcu O, Costan A, Antoniu G, Perez-Hernandez MS. Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks. *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. 2016 IEEE International Conference on Cluster Computing (CLUSTER). 2016 09. https://doi.org/10.1109/cluster.2016.22

[9] Quoc D, Chen R, Bhatotia P, Fetze C, Hilt V, Strufe T. Approximate stream analytics in apache flink and apache spark streaming. *arXiv preprint arXiv:1709.02946*. 2017;

[10] van Dongen G, Van den Poel D. Evaluation of Stream Processing Frameworks. *IEEE Transactions on Parallel and Distributed Systems*. 2020 08 01;31(8):1845-1858. https://doi.org/10.1109/tpds.2020.2978480