Comparative Evaluation of Spark and Flink Stream Processing

MA-INF 4306 - Data Science and Big Data
Lab Report
Winter Semester 2016.17
University of Bonn

Ehab Qadah
Supervisor: PD Dr. Michael Mock.

March 13, 2017

Abstract

Recent years have witnessed advances in the development of Big Data frameworks like Apache Spark and Apache Flink. For instance, both frameworks support real-time data stream processing. The question of which platform is superior to the other still open. In this report, we provide a performance comparison of streaming data processing in Apache Spark and Apache Flink by measuring different performance metrics (latency and throughput). Furthermore, we cover some key aspects of streaming data applications and how they are handled in the two frameworks. The experiment is done using multiple streaming workloads over real-world datasets. Results show that Flink outperforms Spark streaming in term of latency. While Spark Streaming gives higher throughput rates than Flink.

1 Introduction

In recent years, Big Data is considered as one of the most appealing topics in Information Technology, it has gained increasing attention of professionals in industry and academia. This attention is a result of the fact that the amount of data we generate is increasing exponentially [1]. Different sources such as social media platforms, mobile phones, Internet of things (IoT), financial transactions, etc. create large volumes of data. Moreover, many Big Data frameworks have been recently developed such as Apache Spark [3] and Apache Flink [4] that provide efficient data processing functionalities.

Big Data is a term that does not only describe large datasets (volume), but also the data that arrives at a high rate (velocity), and it comes in a mix of different data format (variety) (i.e., 3Vs of Big Data (Volume, Velocity, Variety)) [2].

The processing models of Big Data are Batch processing and Stream processing. Batch processing involves handling large and finite volumes of data (e.g., processing of historical transaction records), and is more concerned with throughput. While Stream processing is a method to manage fast and continuously incoming data (real-time) as it arrives in the system (e.g., credit card

fraud detection and network monitoring applications). Stream processing systems usually provide low latency processing. Both processing models are capable of operating with structured or semi-structured data (i.e., variety).

There exist some publications that compare Apache Spark and Apache Flink such as the Yahoo Streaming Benchmark [5]. The benchmarking provides a performance comparison of Apache Flink , Apache Storm [6] and Spark Streaming in terms of latency and throughput over a single data pipeline. The results show that Flink and Storm have similar and lower latency, while Spark Streaming has higher latency and it gives higher throughput by increasing the batch duration. But we believe that we still need to carry out our evaluation for two reasons: (i) both frameworks are continuously improving in a short time (ii) we aim to study the performance of some complex operations such as `groupByKey` & `keyBy` in Apache Spark and Apache Flink, respectively.

The purpose of this report is to provide a comparative experimental evaluation of throughput and latency of stream processing in Apache Spark and Apache Flink. In order to simulate real-world use cases, we use datasets of airplane trajectories provided by the Datacron project[1] to develop multiple streaming data processing workloads. Besides the performance comparison of the stream processing of the two frameworks, we try to cover general aspects of stream processing tasks (see Section 3) and how they can be solved in each platform. Informally, this work aims to help the developers to determine which platform to use for different use cases.

The remainder of this report is organized as follows. In Section 2, we present the main characteristics, programming APIs, and main data abstraction of Apache Spark and Apache Flink. Section 3 presents the general aspects of stream processing tasks and the implementation of experimental workloads. Section 4 compares the performance results of the different workloads. And finally, Section 5 gives the overall conclusion and future work.

2 Technical Background

In this section, we present the architecture, key characteristics and the programming APIs of Apache Spark and Apache Flink. Moreover, a brief overview of the Apache Kafka platform is provided.

2.1 Apache Spark

Apache Spark is an open source project that provides a general framework for large-scale data processing [3]. It offers programming APIs in Java, Scala, Python and R. Its stack includes a set of built-in modules including Spark SQL, MLlib for machine learning, GraphX for graph processing, and Spark Streaming to process real-time data streams. It can access different data sources including Hadoop Distributed File System (HDFS), Apache Cassandra database, Apache HBase database etc.

[1] `http://www.datacron-project.eu/`

The main data abstraction of Spark core is the Resilient Distributed Datasets (RDDs), which is in-memory collection of elements partitioned across cluster of computers that can be processed in parallel [7]. RDDs support two categories of operations: transformations that drive new RDDs from the operated ones (e.g., `map`, `filter` and `groupByKey`), and actions which return a certain value (e.g., `count` and `collect`). Additionally, Discretized Stream (DStream) is the data abstraction in Spark Streaming, which is a series of RDDs that represent an input stream of data [8]. In other words, the Spark Streaming is a batch-based processing model, which processes the continuous stream of data by dividing it into micro-batches that are processed by the Spark engine. The operations of Spark Streaming are stateless due to the micro-batch design, the `updateStateByKey` transformation can be used to aggregate the state between stream's micro batches. Figure 1 illustrates the general process flow of Spark Streaming.

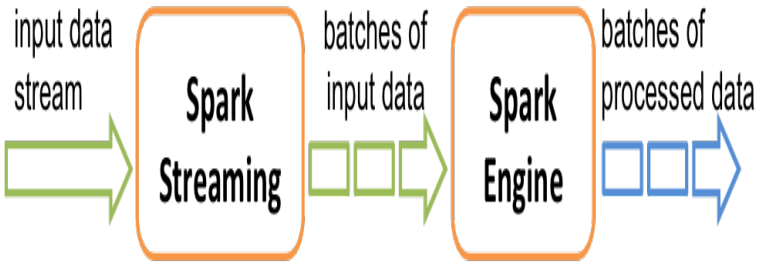


Figure 1: Process flow of Spark Streaming [8].

2.2 Apache Flink

Apache Flink is an open source project that provides a large-scale, distributed stream processing platform [4]. Flink treats the batch processing as a special case of streaming applications (i.e., finite stream). It offers programming APIs in Java and Scala. Its software stack includes the core DataStream and DataSet APIs with additional libraries such as Complex event processing for Flink (Flink-CEP), Machine Learning for Flink (FlinkML) and Flink Graph API (Gelly).

The main data abstractions of Flink are DataStream and DataSet that represent read-only collection of data elements. The list of elements is bounded (i.e., finite) in DataSet, while it is unbounded (i.e., infinite) in the case of DataStream.

The Flink's core is a distributed streaming dataflow engine, with each Flink program is represented by a dataflow graph (i.e., directed acyclic graph - DAG)

that executed by the Flink's engine [9]. The data flow graphs are composed of stateful (state is maintained per partition), parallel operations and intermediate data stream partitions. Figure 2 shows a data flow graph of a Flink program.

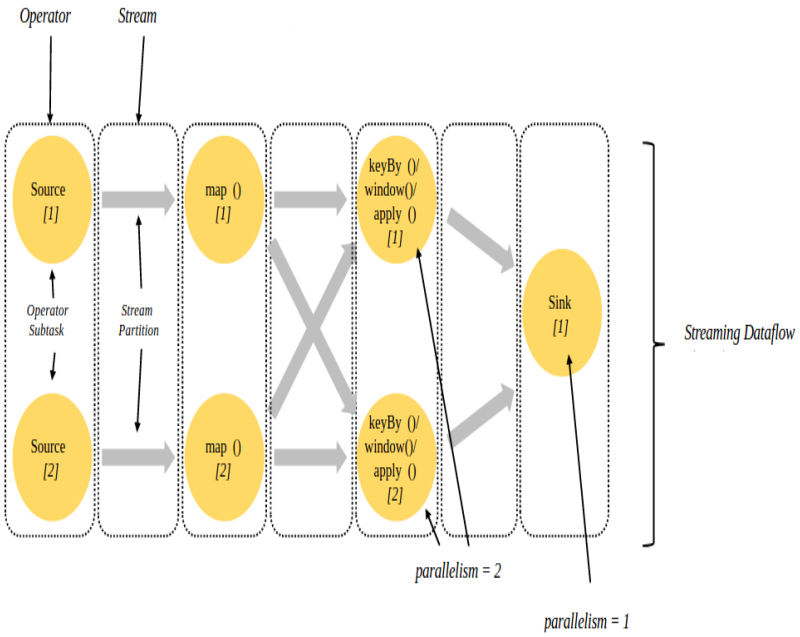


Figure 2: An example of data flow graph in Flink [4].

2.3 Apache Kafka

Apache Kafka is scalable, fault-tolerant and distributed streaming framework [10]. It allows to publish and subscribe to data streams. Kafka manages the stream records in different categories (i.e., topics) that are partitioned and distributed over the servers of the Kafka cluster. It allows the data producers to publish a stream of records to one or more Kafka topic. Additionally, consumer applications can subscribe to one or more topic to read data streams. The consumers within a group use the same group name, which allows Kafka to distribute and balance the stream partitions among the members of a certain group for the sake of scalability. Figure 3 shows how four partitions of a stream are distributed between two groups of consumers based on the number of members. Apache Kafka has been widely adopted, for example, Apache Spark and Apache Flink can ingest Kafka streams.

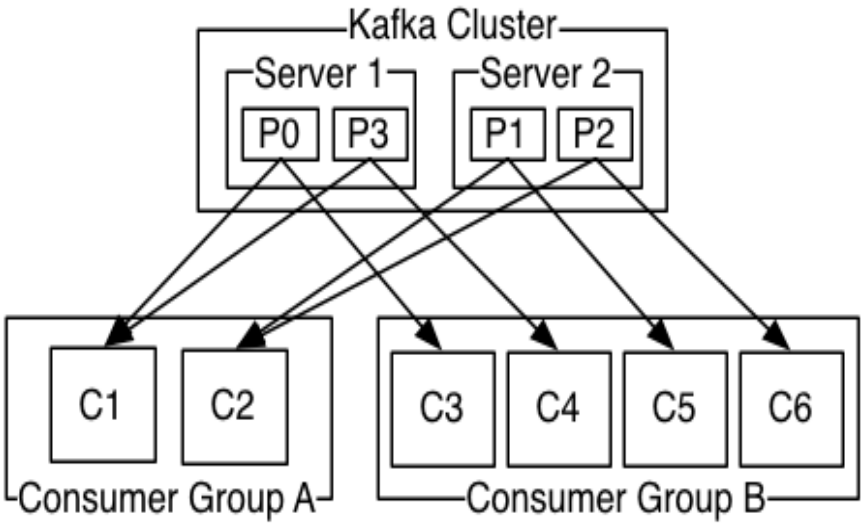


Figure 3: Distribution of a stream partitions for consumer groups in Kafka [10].

3 Experiment Setup and Implementation

This section describes the data stream setup, the design of evaluation streaming workloads and the implementation details of the proposed solution of Apache Spark and Apache Flink.

3.1 Data Stream Setup

The evaluation streaming workloads read an input data stream from Kafka. In order to simulate real-world use case we use datasets of Automatic Dependent Surveillance - Broadcast (ADS-B) messages that represent the aircraft's position over time. The message line comprises 22 fields of data such as aircraft ID, date message generated, longitude, latitude and altitude. And these datasets (2.4 GB) contain around `26 millions` messages.

In our experiment a data stream producer component reads the ADS-B message datasets, then publishes them to the Kafka cluster to be consumed by the workloads in Apache Spark and Apache Flink. Figure 4 illustrates the data producer and the Kafka cluster setup in all experiments. The data stream is divided into four portions over two servers. The data producer publishes the stream records randomly to a Kafka partition and all ADS-B messages are published to the same topic (`datacorn`). Moreover, the data producer attaches the time of publishing to the ADS-B message line before publishing it to the Kafka topic.

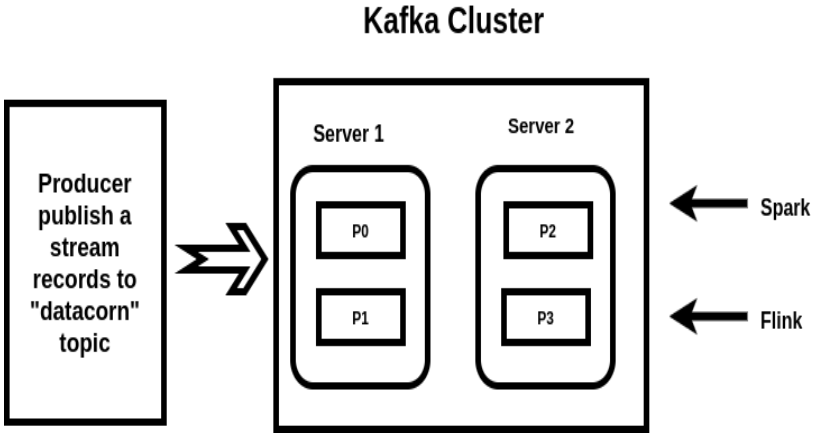


Figure 4: The setup of the Data Stream Producer and Kafka Cluster.

3.2 Evaluation Streaming Workloads

In our experimental evaluation, we developed two real-time stream processing workloads that read a stream of ADS-B messages from Kafka in order to perform basic trajectories analysis methods. The workloads are designed to cover general key aspects of real-time stream processing tasks and evaluate the corresponding solutions in Spark and Flink. The following are the aspects of streaming data processing were covered by our workloads design:

- Handling parallel input streams (e.g., Kafka Stream).
- How to aggregate the state of an input stream.
- Manage the order of stream records.
- How to provide and update global data model in a stream processing task.
- Evaluate the performance by measuring the latency and throughput.

We give the description of workloads, the relation with these aspects and the implementation details of Spark's and Flink's solutions in Sections 3.2.1 and 3.2.2. Afterward, In Section 4 we discuss the performance evaluation and analysis of stream processing workloads in the two frameworks.

The two streaming workloads and the data stream producer are implemented in Java and the source code is available in a repository on Github[2].

3.2.1 Statistics Computation per Trajectory Workload

In the first stream processing workload, we construct a stream of trajectories by considering the position messages (i.e., ADS-B messages) that belong to the same aircraft as trajectory. Moreover, we continuously compute and aggregate statistics for each new position in a trajectory. As an example of the computed statistics quantities speed mean, mean of location coordinates, min and max altitude, etc. In this workload, we cover the parallel receiving of an input data stream, stateful aggregation over a data stream and preserving the correct order of the stream's records. The implementation details of this workload in Spark and Flink are presented in the following:

- **Implementation in Spark**

  The implementation of trajectory statistics computation in Spark Streaming, first, it reads the Kafka stream using KafkaUtils.createDirectStream that creates `DStream`, which represents the stream of position messages, in order to scale up and have parallel Kafka stream receivers, multiple instances of `DStream` must be created and combined together using a `union` operation. The output stream of the `union` is further processed using a `mapToPair` transformation that maps each position message to a tuple of

[2] `https://github.com/ehabqadah/Spark_vs_Flink`

aircraft ID and position message. Then, irrelevant tuples are filtered based on the position message type using a `filter` transformation.

Afterward, tuples that share the same aircraft ID are grouped together to construct a stream of trajectories (tuples of ID and list of positions) by applying the `groupByKey` operation. Since the processing model of Spark Streaming is micro-batch based and stateless an `updateStateByKey` operation must be used to maintain the state of the stream of trajectories. In the context of this workload, the state is the aggregated statistics of a trajectory, so the statistics computation is performed within the custom `updateStateByKey` function that is continuously applied for every batch.

The state update function uses the last position of a trajectory from previous batch to aggregate and compute statistics quantities to the received positions within a new batch and new positions are kept as the new state. Inside the state update function the new positions must be sorted to preserve the correct order, since the tuples after `groupByKey` operation are shuffled across the cluster's nodes and sent to the state update function.

- **Implementation in Flink**

  The details of Flink's implementation of statistics computation workload as follows. First, a `FlinkKafkaConsumer` is used to construct a `DataStream` instance that represents the Kafka stream of aircrafts positions. The `FlinkKafkaConsumer` implicitly runs multiple parallel Kafka stream consumers, which makes the Kafka stream's partitions balanced between the parallel instances. Second, the stream's records (i.e., ADS-B messages) are parsed and transformed to tuples of ID and position message using a configured `map` transformation. Third, a `filter` function is used to filter irrelevant tuples based on the message type. Fourth, the filtered tuples are grouped based on the tuple's ID using a `keyBy` operator that constructs a `KeyedStream` of trajectories.

  Finally, a `reduce` transformation on the `KeyedStream` is used to calculate the statistics for each new arriving trajectory's position by utilizing the aggregated statistics of the previous position. The old position is discarded and the new position with computed statistics is used to aggregate the statistics of a trajectory.

In summary, The following are the main key differences between the Flink and Spark Streaming implementation of the statistics computation workload:

- Flink handles the parallel consumers of Kafka stream implicitly, while multiple `DStream` must be created in Spark Streaming and union them to have parallel Kafka receivers.
- The Flink's operations of the `KeyedStream` are stateful, so the management of the stream's state is not required as Spark Streaming that requires the usage of `updateStateByKey` to manage the state of the micro batches.

- A sort action is required in Spark Streaming to preserve the correct order of the position messages inside the state update function. while in Flink the sort is not required by using a `reduce` transformation over the `KeyedStream` that processes the stream records item by item.

3.2.2 Air Sector Change Detection Workload

The goal of sector change detection workload is to detect the entering or leaving of an aircraft from one air sector to another one, by processing the real-time stream of aircraft positions. Given the fact that a dataset of 20,000 sectors (i.e., polygons) is available as a reference to assign the corresponding sector for certain aircraft's position. In this workload, we test the aspect of providing a global data model (sectors dataset) in stream processing workload. Moreover, the preserving of the stream's records order and consuming of parallel Kafka streams are covered.

The implementation details of this workload in Spark and Flink are presented in the following:

- **Implementation in Spark**

  We implemented the sector change detection workload in Spark Streaming by constructing the stream of trajectories and assigning a sector for each position of a trajectory. A change between the assigned sectors of two consecutive aircraft's positions is marked as a sector's transition for a certain trajectory.

  First, multiple instances of `DStream` are created using the KafkaUtils.create `DirectStream` and combined using a `union` to read in parallel from Kafka. Second, the records (i.e., ADS-B messages) of the resulting stream are parsed and mapped to tuples of (aircraft ID, position message) using `mapToPair` transformation. Third, the unrelated tuples are filtered based on the message type using a `filter` operation. Fourth, the stream of trajectories is constructed by combining all tuples with the same ID using the `groupByKey` function. Then, to manage and aggregate the state of trajectories stream between the mini-batches a configured `updateStateByKey` function is applied. The state update function assigns the corresponding sector for each new position, the old positions of a certain trajectory from the previous batch are discarded, but only the last position of the old batch is kept to be used alongside with the new positions. In addition to that the new positions are sorted by the streaming time to maintain the correct order.

  The sector data set is provided using the built-in `Broadcast` feature that distributes a shared value for all cluster nodes in an efficient matter, and it can be update at the driver program by using the `unpersist` function and then update the sectors. Finally, the detection of a sector change is done by an extra filtering transformation using a `filter` operator, which checks the sectors of consecutive positions in a trajectory to find the ones with a change in the sector assignment.

- **Implementation in Flink**

  The Flink's implementation of sector change detection workload uses the `FlinkKafkaConsumer09` that reads the position messages (i.e., ADS-B) stream from Kafka, which is used to construct a `DataStream` instance. The `DataStream`'s records are mapped to tuples of (aircraft ID and aircraft's position) by a configured `map` transformation. Irrelevant tuples are filtered based on the message type using a `filter` operation. Afterward, the tuples that share a common ID are grouped together by the `keyBy` operator. Since the Flink processes the stream records as they come, a `reduce` operator assigns the corresponding sector of position on the new tuple, while the old tuple with the same ID is used to retrieve the previous sector to attach it to the new tuple then the old tuple is discarded. Finally, the tuples (i.e., tuples of (aircraft ID, position message)) with different previous and current sectors are filtered using a `filter` transformation.

To sum up, The following are the main key differences between the Flink and Spark Streaming implementation of the sector change detection workload:

- In this workload we have the same differences between Flink and Spark Streaming regarding the state management, parallel consumers of Kafka stream and preserving the correct order of stream records as was discussed in Section 3.2.1.
- Another key difference between them is how to provide a global data model (e.g., sectors) in a streaming workload. Spark Streaming offers the `Broadcast` feature to handle this issue in an efficient way, while the global data model is manually provided to the stream operations in Flink.
- The broadcast variable of sectors in Spark Streaming can be updated in the driver program by using the `unpersist` function and then update it. While in Flink the update of the sectors is not possible at run time (the program must be reloaded), since they are provided manually to the transformation.

4 Performance Results

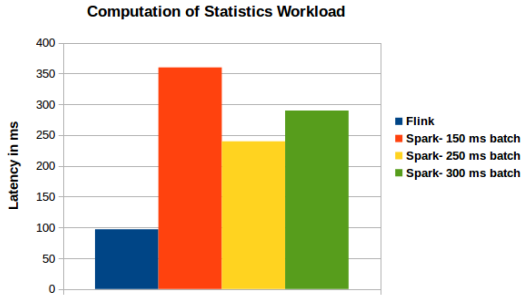
This section presents the results of performance evaluation in Spark (2.0.2) and Flink (1.1.3) for the streaming workloads that were described in Section 3.2. Specifically, we provide the measurements of latency and throughput of the benchmarking workloads execution in Spark Streaming and Flink. We carried the experiment by setting up the Kafka stream as described in Section 3.1 on a local machine[3] and execute the different workloads on the same machine (i.e., local mode) to obtain the performance results.

[3] The experiments have been carried on single machine with Ubuntu 16.04 LTS os, Intel Core i5-6200U CPU @ 2.30GHz 4 processor, and 16 GB of memory.

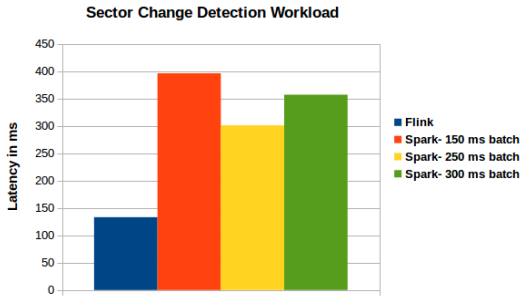
We consider the time delay between the streaming time of a record in Kafka and the time when the record is processed by a workload as the platform's processing latency (measured in milliseconds). And the number of processed stream's records (aircraft positions) per second within each streaming workload is the throughput measure. Since the Spark Streaming is micro-batch based, we examine the effect of different batch intervals of the Spark Streaming on the performance measurements.

4.1 Latency Results

Figure 5 shows the results of the latency measurements of the streaming workloads in Spark and Flink. The presented values present the average value of multiple trails and measurements. Measurements for different batch durations are given in Spark Streaming to find the optimal batch duration, since the large batch times give higher latency rates, also we noticed that reducing the batch time to smaller values increases the latency because of the overhead of managing more micro batches by Spark, this behavior also was observed in [5]. Results show that Flink outperforms Spark Streaming in term of processing latency in the two evaluation workloads. The two framework give higher latency in the sectors change detection workload than statistics computation workload due to the sector assignment for each position by traversing the 20,000 sectors until a match is found.



(a) Latency of Statistics Computation



(b) Latency of Sector Change Detection

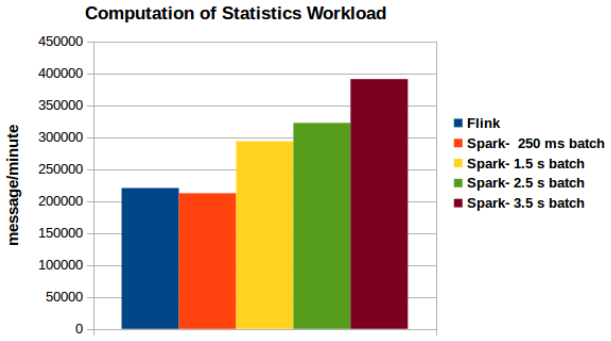
Figure 5: Latency Measurements

4.2 Throughput Results

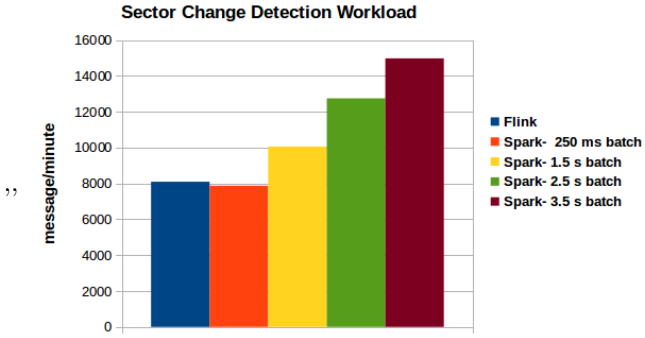
The measurements of throughput for the two streaming workloads are given in Figure 6. The presented values reflect the average of throughput (message/minute) in Spark streaming and Flink for multiple executions. Also for Spark streaming we tested different batch times including the throughput of the best latency batch time. Results show that Spark Streaming has higher throughput than Flink; increasing the batch duration of Spark Streaming gives better throughput. Moreover, the throughput rates of Spark Streaming and Flink at the low latency are nearly similar (rates of Flink little higher). The two framework have a lower throughput in the second workload than the first one due to the sector assignment for each position.

5 Conclusion

In this report, we present an experimental evaluation of the stream processing performance and capabilities of Apache Spark and Apache Flink. We carried our experiment over aircraft positions datasets provided by the Datacorn project, we developed two streaming workloads to cover general aspects of real-time stream



(a) Statistics Computation



,,

(b) Sector Change Detection

Figure 6: Throughput Measurements

processing tasks. Additionally, we measured the latency and throughput of those workloads solution of Spark and Flink. Results show that Flink outperforms Spark Streaming in term of processing latency. In contrast, Spark Streaming provides better throughput rates than Flink by increasing the batch duration, while Flink gives a similar throughput to Spark Streaming with small batch sizes. Our results in terms of latency and throughput are consistent with the Yahoo Streaming Benchmark results.

To sum up, Flink's processing model is well-suited to the stream processing tasks that require low latency, also there is no need to manage the Stream's state and the batch interval comparing to Spark streaming. Moreover, Flink operates over the individual record of the data stream, while Spark Streaming processes the input data stream as micro-batches. Thus we can state that Flink is a true stream processing framework. A side remark, we have noticed that finding online resources related to Spark is easier than to Flink.

In future work, we plan to carry out our experiments over cluster of com-

puters not just only on single computer. In addition, we will cover more aspects of real-time data stream processing and develop more evaluation workloads.

References

[1] Gantz, John, and David Reinsel. "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east." IDC iView: IDC Analyze the future 2007.2012 (2012): 1-16.

[2] Laney, D 2001 3D Data Management: Controlling Data Volume, Velocity, and Variety. META Group.

[3] Apache Spark. Available: https://spark.apache.org/ [Accessed February 2017].

[4] Apache Flink. Available: https://flink.apache.org/ [Accessed February 2017].

[5] Chintapalli, Sanket, et al. "Benchmarking streaming computation engines: Storm, Flink and Spark streaming." Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International. IEEE, 2016.

[6] Apache Storm. Available: http://storm.apache.org/ [Accessed March 2017].

[7] Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.

[8] Spark Streaming. Available: http://spark.apache.org/docs/latest/streaming-programming-guide.html [Accessed March 2017].

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

[10] Apache Kafka. Available: https://kafka.apache.org/intro.html [Accessed February 2017].