Comparative Evaluation of Spark and Flink Stream Processing

Lab Report: Data Science and Big Data MA-INF 4306, Winter Term 2016/2017, University of Bonn

Ehab Qadah
Supervisor: PD Dr. Michael Mock.

March 5, 2017

Abstract

Recent years have witnessed a advances in the development of Big Data frameworks like Apache Spark and Apache Flink. For example, both frameworks support real-time streaming processing. While which platform is the best still an open question. In this report, we provide a performance comparison of streaming data processing in Apache Spark and Apache Flink, by measuring different performance metrics, namely latency and throughput. In addition, we cover some key aspects of streaming data applications and how they are handled in both frameworks. The experiment is done using multiple streaming workloads over real-world datasets.

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, as a result of the fact that the amount of data we generate is increasing exponentially [1] that generated from social media platforms, mobile phones, IoT and finical transactions, etc. Moreover, recent years showed a rapid development of Big Data frameworks such as Apache Spark and Apache Flink that provide more and efficient data processing capabilities.

The Big Data is a method to process not just only large datasets (volume), but also the data that arrives in high rate (velocity), and it comes in all kinds of 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 is a way to handle large, finite volumes of data (e.g., processing of historical transaction records), is more concerned with throughput. While the Stream processing is a method to manage fast and continuously incoming data (real-time), process it as it arrives (e.g., credit card fraud detection and network monitoring applications), where the low latency is required. The variety of data (structured and semi-structured) is included in the two processing models.

The are some performance evaluation of Apache Spark and Apache Flink and other framework such as Yahoo Streaming Benchmark [3]. We believe that we still need to carry our evaluation for two reasons: (i) both frameworks are continuously improving in 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, latency (i.e., performance) of stream processing in Apache Spark and Apache Flink. We use a datasets of airplanes trajectories provided in the context of Datacron project[1], in order to simulate real-world use cases. Using multiple streaming data workloads. Also we try to cover general aspects of streaming processing tasks (later details in Section 3) and how they can be solved in each platform. Informally, this work aims to help the developers to determine which platform to choose in production for different use cases.

The remainder of this report is organized as follows. In Section 2, we present the main characteristics, APIs, and main data abstraction of Apache Spark and Flink. Section 3 presents the general aspects of streaming processing, the design and implementation of experimental workloads. Section 4 provides and compares the performance results of the different workloads. Last, Section 5 gives the overall conclusion.

2 Technical Background

In this section, we present the architecture, key characteristics, 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 provide general framework for large-scale data processing [4]. It offers programming APIs in Java, Scala, Python and R. Its stack includes 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.

The main data abstraction in 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 [5]. RDDs supports two categories of operations: transformations that drive new RDDs from the operated ones (e.g., map), and actions which return a certain value (e.g., count). On the other hand, the Discretized Stream (DStream) is the data abstraction in Spark Streaming, which is series of RDDs that represent an input stream of data [6]. In other words, the Spark Streaming is batch-based processing model, which process the continuous stream of data by divided into micro-batches to be processed by the Spark engine. Figure 1 illustrates the general process flow of Spark Streaming.

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

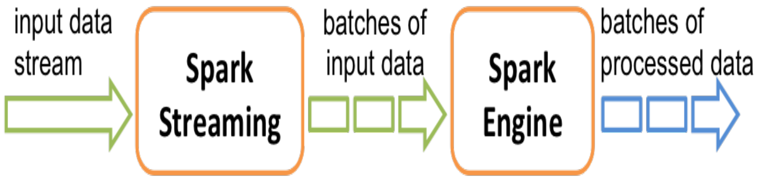

Figure 1: Process flow of Spark Streaming [6].

2.2 Apache Flink

Apache Flink is an open source project that provide large-scale, distributed stream processing platform, while the batch processing treated as special case of streaming applications (i.e., finite stream) [7]. 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 abstraction 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 case of DataStream.

The Flink's core is a distributed streaming dataflow engine, each Flink program is represented by a data flow graph (i.e., directed acyclic graph - DAG) that executed by the Flink's engine [8]. The data flow graphs are composed of stateful, parallel operations and intermediate data stream partitions. Figure 2 shows a data flow graph of Flink program.

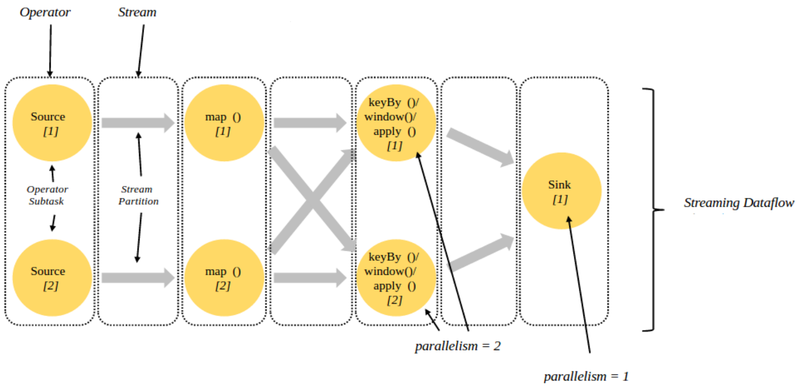

Figure 2: An example of data flow graph in Flink [7].

2.3 Apache Kafka

Apache Kafka is scalable, fault-tolerant distributed publish/subscribe streaming framework [9]. It manges the stream records in different categorizes (i.e., topics) that portioned and distributed over the servers of the Kafka cluster. It allows the data producers to publish stream of records to certain Kafka topic. While the consumer applications can subscribe to one or more topic to read the data stream. The consumers are grouped into groups based on the group name that 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 4 partitions of stream is distributed differentially between two groups of consumers based on the number of members. Apache Kafka has been widely adopted, for example, Spark and Flink can receive data stream from Kafka.

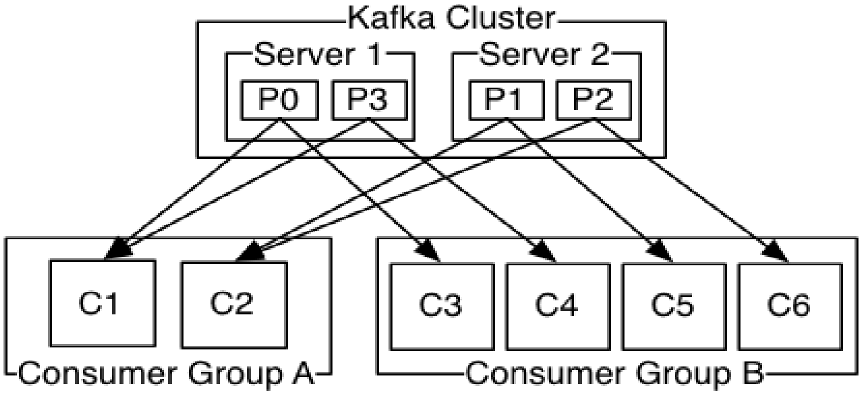

Figure 3: Distribution of stream partitions for consumer groups in Kafka [9].

3 Experiment Setup and Implementation

This section describes the data stream setup, the design of evaluation streaming workloads, and the implementation details in Spark and 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 present the aircraft's position over time, comprise 22 fields of data such as aircraft ID, date message generated, longitude, latitude, and altitude etc.

In our experiment a data stream producer component reads the ADS-B message datasets, then publish them to 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 portioned into four portions over two server, while the data producer publish the stream records randomly to Kafka partitions, and all ADS-B messages are published to same topic, which is `datacorn`. Moreover, the data producer attach the streaming time at the ADS-B message line before publishing it to the Kafka topic, to be used in the latency calculation.

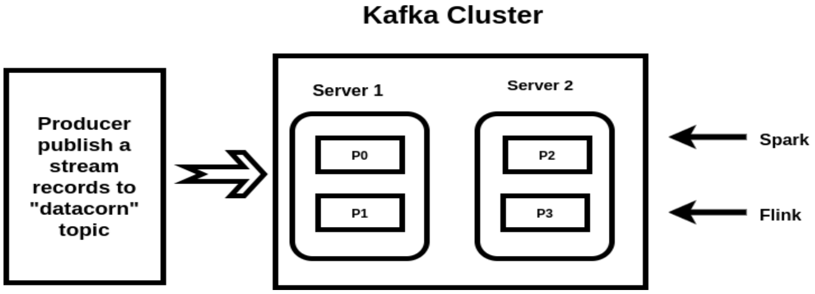

Figure 4: Data Stream Producer & Kafka Cluster Setup.

3.2 Evaluation Streaming Workloads

In our experimental evaluation, we developed two real-time stream processing workloads that read the ADS-B messages stream from Kafka in order to perform basic trajectories analysis methods. The workloads are designed in such way that cover general key aspects of real-time streaming 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's design:

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

We present the description of workloads and the relation with these aspects, also the implementation details of Spark and Flink solutions in Section 3.2.1 & 3.2.1. Afterward, In Section 4 we discuss the performance evaluation and analysis of these workloads in both frameworks.

3.2.1 Statistics Computation per Trajectory

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 example of computed statistics quantities speed mean, mean of location coordinates, min and max altitude, etc. In this workload, we cover the parallel receiving of input data stream, statefull aggregation over the input 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` that represents the stream of position messages, in order to scale up and have parallel Kafka stream receivers, a multiple instances of `DStream` must be created and combined together using the `union` operation. The output stream of the `union` is further processed using `mapToPair`, to map each position message to tuple of aircraft ID and position message. Then, the irrelevant tuples are filtered based on the position message type using a `filter` transformation. Afterward, the tuples that share the same aircraft ID are groped together to construct the 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, `updateStateByKey` operation must be used to maintain the state of the trajectories stream. In the context of this workload, the state is the aggregated statistics of trajectory, so the statistics computation is preformed within the custom `updateStateByKey` function that continuously applied for every batch. The state update function uses the last position of trajectory from previous batch to aggregate and compute statistics quantities to each trajectory's position in the new data batch, and the new positions list is kept as the new state for the trajectory. 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 represent the Kafka stream of aircrafts positions, it implicitly runs multiple parallel Kafka stream consumers, which makes the Kafka partitions balanced between the instances. Second, the stream 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 resulted tuples after the filtration transformation are grouped based on the tuple's ID using keyBy operator that constructs the KeyedStream of trajectories. Finally, a reduce transformation on the KeyedStream is used to calculate the statistics for each new coming trajectory's position, by utilizing the aggregated statistics of previous position, while old position is discarded and the new position with attached computed statistics is used to aggregate the statistics of a certain trajectory.

In summary, Flink handles the parallel consumers of Kafka stream implicitly, while a multiple DStream must be created in Spark Streaming and union them to have parallel Kafka receivers. The Flink's streaming processing is stateful that means there is no need to manage the stream state as the case in Spark streaming. A sort action is required in Spark Streaming to preserve the correct order of the position messages inside the state update function.

3.2.2 Air Sector Change Detection

The goal of sector change detection workload is to detect the enter or leaving of aircraft from one air sector to other, by processing the real-time stream of aircraft positions. Given that the dataset of sectors (i.e., polygons) is available as reference to assign the corresponding sector for certain aircraft's position. In this workload, we test the aspect of providing the global model of data (sectors dataset) in streaming processing workload. Moreover, the preserving of the stream's records order and consuming of parallel Kafka streams points are also 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 trajectories stream and assigning the sectors for each position of a trajectory, then a change between the assigned sectors of two consecutive aircraft's positions is marked as an transition of sector for certain trajectory. First, a multiple instances of DStream are created using KafkaUtils.createDirectStream to read in parallel from Kafka, after all input data streams are combined using union function. Second, the records (i.e., ADS-B messages) of the resulted union stream are parsed and mapped to tuples of (aircraft ID, position message) using defined mapToPair transformation. Third, the unrelated tuples are filtered based on the message type through a filter operation. Fourth, the trajectories stream 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 assign the corresponding sector for each new position, the old positions of a certain trajectory from previous batch are discarded, but only the last position of old batch is kept to be used alongside with the new positions. In addition, the new positions are sorted 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 efficient matter. Finally, the detection of sector change is done by an extra filtering transformation using a `filter` operator, by checking 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 used to construct `DataStream` instance. The `DataStream`'s records are mapped to tuples of (aircraft ID and aircraft's position), by a configured `map` transformation. The irrelevant tuples are filtered based on the message type using a `filter` operation. Afterward, the tuples that shares a common IDs are grouped together by the `keyBy` operator. Since the Flink streaming processes the stream records as it come, a defined `reduce` operator assign corresponding sector of position on new tuple, while the old tuple with same ID is used to retrieve the previous sector to attach it to the new tuple, while the old tuple is discarded. Finally, the tuples (i.e., tuple of (aircraft ID, position message)) with different previous and current sector are filtered using defined `filter` transformation.

To sum up, the same differences between Flink and Spark Streaming regarding the state management and parallel consumers of Kafka stream as was discussed in Section 3.2.1. Another key difference between them, is how to provide the global data model (e.g., sectors) in a streaming workload. Spark Streaming offers the `Broadcast` feature to handle this issue in efficient way, while the global data model is manually provided to the stream operations in Flink.

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 were described in Section 3.2. Specifically, we provide the measurements of latency and throughput of the benchmarking workloads execution in Spark and Flink. We carried the experiment by setup the Kafka stream as described in Section 3.1 on local machine[2], and run the different workloads on the same machine (i.e., local mode) to obtain the performance results.

[2] 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 record in Kafka and the finish processing time in a workload as the platform's processing latency (measured in milliseconds). On the other hand, the number of processed stream's records (aircrafts position messages) 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 Measurements

4.2 Throughput Measurements

5 Conclusion

This section gives the overall conclusion.

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

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

[5] 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.

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

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

[8] 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).

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