Comparative Evaluation of Spark and Flink Stream Processing

Contents

Abstract

Recent years have witnessed a big development improvements of Big Data frameworks such as 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 paper, we provide a performance comparison of streaming data processing in Spark and Flink, by measuring different performance metrics, namely latency and throughput. The experiment is done using multiple streaming workloads over real-world datasets.

1 Introduction

Big Data is now one of the most appealing topics in Information Technology, it has gained increasing attention of professionals in industry and academics, since 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 Spark and Flink that provide more and efficient data processing capabilities.

The Big Data is a method to process not just only just large datasets (volume), but also 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 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.

TODO: add paragraph about Yahoo benchmark and why do we still need evaluation 1- continuous dev of frameworks 2- explore more complex operation like groupby that shuffle the data.

The purpose of this paper 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 we try to cover the main programming API differences between the both frameworks. Informally, this work aims to help developer to determine which platform to choose in production.

The rest of this paper is organized as follows: Section 2 presents the main characteristics, APIs, and main data abstraction of Spark and Flink. Section 3 describes the experiment workloads and their implementation in each framework. Section 4 provides and compares the performance results of the different workloads. Finally, Section 5 gives the overall conclusion.

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

2 Technical Background

In this section, we present the architecture, key characteristics, programming APIs of Apache Spark, Apache Flink. The Apache Kafka platform is also discussed briefly.

2.1 Apache Spark

Apache Spark is an open source project that provide general framework for large-scale data processing [3]. It offers programming API 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 is the Resilient Distributed Datasets (RDDs), which is a in-memory collection of elements partitioned across cluster of computers that can be processed in parallel [4]. RDDs supports two categories of operations: transformations that drive new RDDs from the operated ones (e.g., map). And actions which return a value (e.g., count).

//TODO: DStreams, streaming model

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 [5]. It offers programming API in Java, Scala. Its stack includes the core DataStream and DataSet APIs with additional libraries such as Complex event processing for Flink (FlinkCEP), Machine Learning for Flink (FlinkML), and Flink Graph API (Gelly).

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

2.3 Apache Kafka

Apache Kafka is scalable, fault-tolerant distributed publish/subscribe streaming framework [6]. It manges the stream records in different categorizes (i.e., topics) that portioned and distributed over the servers in 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. It has been widely adopted, for example, Spark and Flink can receive data stream from Kafka.

3 Experiment Setup and Implementation

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

3.1 Data Stream Setup

Our streaming workloads read input data stream from Kafka. In order to simulate real-world uses cases 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.

In our experiment a data stream producer component reads the ADS-B datasets, then publish it to Kafka cluster to be consumed by the workloads in Spark and Flink. Figure 1 illustrates the data producer and the Kafka cluster setup. The data stream is portioned into four portions over two server, while the data producer publish the stream records randomly to Kafka partitions.

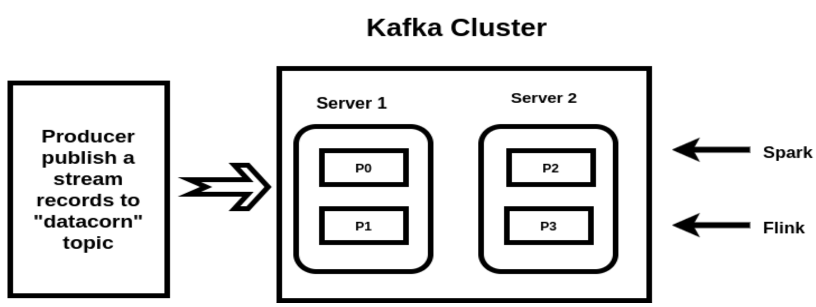

Figure 1: Data Stream Producer & Kafka Cluster Setup.

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. In our design choice, we try to cover some key points that related to the streaming processing tasks, and evaluate the corresponding solutions in Spark and Flink. The following are general key 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, and th 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 workloads.

3.2.1 Statistics Computation per Trajectory

In first stream processing workload, we construct stream of trajectories by considering the aircraft's positions (i.e., ADS-B messages) that belong to the same aircraft as trajectory. Moreover, we continuously compute and aggregate statistics for each arriving position in a trajectory . As example of computed statistics indexes speed mean, mean of location coordinates, min and max altitude, etc. In this workload, we cover the parallel receiving of input data stream, state-full aggregation on input data stream, and preserving the right 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 Spark Streaming implementation of trajectory statistics computation read the Kafka stream using `KafkaUtils.createDirectStream` that creates `DStream` instance, in order to scale up and have parallel Kafka stream receivers, a multiple instances of `DStream` are created and combined together using `union` operation. The output stream of the `union` is 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 tuple with same aircraft ID are groped together to construct trajectories (tuples of Id and list of positions), by applying the `groupByKey` operation. Since streaming model of Spark is micro-batch based and stateless, `updateStateByKey` operation is must be used to manage the state between the batches. In context of this workload, the statistics computation is preformed within the custom `updateStateByKey` function, which uses the last position of trajectory from previous batch to aggregate and compute statistics values to each trajectory's position in the new arriving batch. The list of new positions must be sorted to preserve the correct order, since the tuples after `groupByKey` operation are shuffled across the cluster's nodes.

- **Implementation in Flink**

  The statistics computation of trajectories stream is solved in Flink by performing the following steps: a) read Kafka input stream b) parse the input stream records c) filter only the relevant messages d) combine the all messages related to the same trajectory e) compute the statistics for each trajectory.

The utilized Flink APIs to perform those steps are shown in Figure 3. First, create Kafka stream consumers that run in multiple parallel instances based on the configured `parallelism` factor. Second, stream elements (i.e., ADS-B message) are mapped to tuples of aircraft ID and position object through `map` transformation API. Third, the irrelevant messages are filtered by the `filter`. Fourth, tuples with same ID are combined together using the `keyBy` API function. Finally, the `reduce` API function is used to calculate the statistics and aggregate it.

Figure 2: Flink implementation of statistics computation workload.

//TODO: discuss differences between Flink and Spark.

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 trajectories stream. Given that the dataset of sectors (i.e., polygons) available as reference to check against it, also it used to assign the corresponding sector for certain aircraft's position. In this workload, providing the global model of data (sectors dataset) in streaming processing workload is tested.Moreover, preserving the stream records order and parallel streams consuming are covered.

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

- **Implementation in Spark**

  we implement this workload, based on the following operation stages: a) receive and build the trajectories stream b) manage the aggregate state and assign the sector for each trajectory. c) identify the change of sector per trajectory if exist

  figure 4 shows the Spark streaming APIs used to implement trajectory's sector change detection workload. First, the trajectories stream is built using `mapToPair`, `filter`, and `groupByKey` APIs as was discussed in statistics computation workload. Second, `updateStateByKey` API is used to assign corresponding sector for all new aircraft positions in micro batch and aggregate last position in previous batch. Finally, the detection of sector change is done by extra filtering transformation using `filter` API, by identifying the change of sectors in each trajectory. While the sector data set is provided using the `Broadcast` feature in Spark.

Figure 3: Spark streaming implementation of sector change detection workload.

- **Implementation in Flink**

The Flink implementation of sector change detection workload, uses the `FlinkKafkaConsumer09` that read the ADS-B messages stream from Kafka, which used to construct `DataStream` instance. The `DataStream` records are parsed to tuples of aircraft ID and aircraft's position, by a configured `map` function. The irrelevant tuples are filtered based on the message type using a `filter` operation. Afterward, the tuples with 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 new tuple, and 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 with different previous and current sector that represent change in sector case, they are filtered using defined `filter` transformation.

4 Performance Results

This section provides and compares the performance results of the different workloads.

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

[4] 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.

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

[6] Apache Kafka. Available: https://kafka.apache.org/ [Accessed February 2017].