Hybrid Systems Benchmark

[BDAPRO.WS1617 - Technical University of Berlin]

Patrick Lehmann
TU-Berlin
Role: Spark
pat@my-lehmann.de

Andres Vivanco
TU-Berlin
Role: Flink
andresvivancov@gmail.com

Jeyhun Karimov
DFKI
Role: Mentor
Jeyhun.Karimov@dfki.de

ABSTRACT

The present project was done analyzing and comparing especially the latency and the throughput (events processed) in hybrid computation between two of the most popular Processing Streaming stacks on the market such as Apache Spark [3] and Apache Flink[4]. It is was done taking into account different parameters and considerations such as how is the behaviour when the workload is increased, how is the behaviour of the latency and elements processed in different windows size, and how is the performance when the initial historical data has different sizes in terms of elements or MB. All these use cases will allow us to have a better perspective of the performance for computing batch and data stream at same time in Apache Spark [3] and Apache Flink[4]. Finally both results will be analyzing altogether to observe important aspects or critical points for it.

General Terms

Hybrid Computation, latency, Windows Streaming, Streaming Processing, Apache Spark, Apache Flink, throughput, hybrid systems, Apache Kafka

Keywords

Flink, Spark, Kafka, benchmark, hybrid systems, hybrid computation

1. INTRODUCTION

Hybrid systems are special cases of streaming engines like Apache Flink[4], Apache Spark[3], and many more. In theses systems the streaming part is computed with another batch part. This is in most streaming systems not a built-in feature and may require some addition as in the case of Apache Flink. For other System like Spark that are build on microbatches it is in general easier to achieve a hybrid computation.
Therefore, it is from high interest to analyze those different approaches for hybrid systems and to find the advantages as well as the disadvantages for each system. This can be achieved by studying the latency and throughput of the two systems while changing different parameters.

2. BENCHMARKING DESIGN

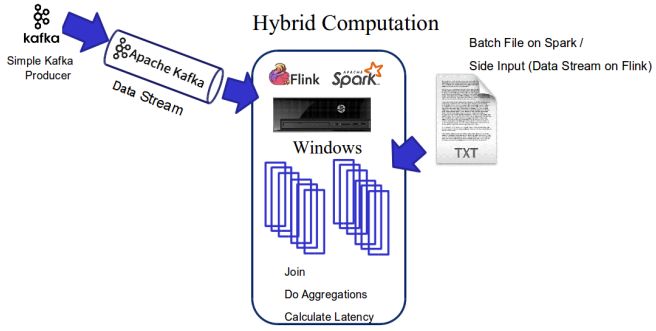

Figure 1: Benchmark design

The both system that are compared in this benchmark are Apache Flink[4] and Apache Spark[3] as seen in figure 1 which shows the general data flow for this benchmark. The use case for this benchmark is a simple join of a windowed stream with a historical data file. The historical data file contains several taxi ride events [1] in the USA with parameter like starting time, an id, and number of passengers. Each event is about 130 Bytes. The data stream is generated by the same file. Therefore, the result of the join will contain all elements that are joined. To create and receive the data stream Apache Kafka [2] is used.

To achieve best consistency the program skeleton for both systems is the following:

1. Read historical data
2. Set up Kafka connection
3. Receive taxi rides from Kafka
4. Assign timestamp while receiving
5. Create windows for data stream
6. Join windowed data stream with historical data

[1] http://dataartisans.github.io/flink-training/exercises/taxiData.html [6]
[2] https://kafka.apache.org/

7. Compute latency (current time − timestamp)

The experiments that were conducted analyzed the behaviour of both systems for the parameters the size of the historical file, the number of elements that are pushed to the systems (workload), and the window size.

2.1 Spark

To use Apache Spark as a hybrid system not much is needed to change as it is already built on microbatches. The data stream's abstraction in Spark is called a DStream [3]. Such a DStream is represented as a sequence of RDDs and these RDDs are basically microbatches.

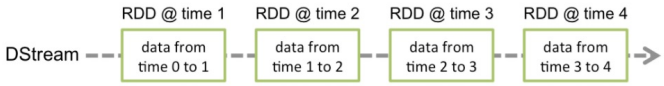

Figure 2: DStreams in Spark (Source: [3])

To make use of this microbatches for hybrid computation one has to use a transform function. Such a transform function is called on every RDD once and returns one for each RDD. Therefore, every RDD can be joined within the transform function with the historical data. As the joining is a keyed aggregation a transformToPair function was used for this specific use case.

```
1  //Creation of the Windows Stream
2  JavaPairDStream<String, String> joinedStream
3          = windowedStream.transformToPair(
4   new Function<JavaPairRDD<String, String>,
       JavaPairRDD<String, String>>(){
5  @Override
6    public JavaPairRDD<String, String> call
         (JavaPair<String, String> rdd) {
7  //Concatenation of historical data with DStream
8    JavaPairRDD<String, String> joined =
9                  rdd.join(cacheBatch)
10         .mapValues(x->x.1.concact
11                     ("," + String.valueOf(
12         System.currentTimeMillis()-
13                Long.valueOf(x.1.split(regex",")[9])));
14  return joined;
15  }
16 }
```

The historical data file also was cached to improve the performance as the system does not need to reread the large historical file.

2.2 Flink

2.2.1 State of the Art

When was done this project in February 2017, the latest stable release version of Apache Flink was v1.2.0 [4]. In this version was not possible to work or merge or join Data sets with Data Stream in a Windows Stream, these are different functionalities, and in the same way the Windows Stream approach just allow to join Data Streams with Data Streams but not with Data Sets. Hence the last released version of Apache Flink[4] did not support the approach of this project.

There are two important facts that can help to solve the approach needy for working with Hybrid computation on Apache Flink[4]. The first one is Flink Improvement Proposals (FLIP) [5] that is a collection of documented new big proposal that will be analyzed, discussed, and implemented on Apache Flink. One of these enhancements is the FLIP âĂŞ 17[5] that describes the functionality of Side Inputs for Data Stream API. With these new libraries is possible to achieve the goal of this project hence it allows hybrid computations on Apache Flink [4]

Another important fact is the Scalable Online machine learning for predictive analytics and real time interactive visualization named Proteus Engine[2], that is a new framework specialized on online machine learning based mailny on Apache Flink founded with contribution of the European Union for solving machine leargning necessities with big data. Proteus engine[2] combines batch and streaming processing and do it in a scalable and real time mode. Proteus [2] also integrates the API features described on the FLIP -17[5].

Considering these two facts was possible to implement the hybrid computation benchmarking based mainly in Java programming languages on Apache Flink[4].

2.2.2 SideInput on Apache Flink

Based on the State of the Art for achieving our project, we needed to use the Side Input API realised on the Proteus Engine Repository[2] that contain this functionaliy. There are three options inside of Side Input API functionality which are:

1. newForwardedSideInput
   (DataStream<TYPE> sideInput)
2. newKeyedSideInputet
   (DataStream<TYPE> sideInput,int field)
3. newBroadcastedSideInput
   (DataStream<TYPE> sideInput)

For our case of use, we will use the functionality *newBroadcastedSideInput (DataStream< TYPE> sideInput)* , due that it is the most closet functionality comparing with Apache Spark[3].

2.2.3 Code Source for Hybrid Computation

There are three important parts in the code of our project. Firstly we need to read the Historical Data (Batch File), it can be possible just with a single Read Text File function.

```
1
2    //args [1] is the path where is located the
         historic data file"
3       String datah = args[1];
4    //Read from Historical Data
```

[3]http://spark.apache.org/docs/latest/streaming-programming-guide.html

```
    DataStream<String> historicalData =
        env.readTextFile(datah);
  //Mapping of the taxi ride ID
    DataStream<Long> historicalID =
        historicalData.map(new
        Aggregations.MapHistoricalDatatoID3());
  //Creation of the SideInput as broadcast mode
    final SideInput<Long> sideInputHistoricalData =
        env.newBroadcastedSideInput(historicalID);
```

Secondly we will read the events from Kafka, these data streams will be accessed using an Apache KakKfa[7] consumer 08 as kafka client in Apache Flink[4].

```

// configure the Kafka consumer
  Properties kafkaProps = new Properties();
  kafkaProps.setProperty("zookeeper.connect",
      LOCAL_ZOOKEEPER_HOST);
  kafkaProps.setProperty("bootstrap.servers",
      LOCAL_KAFKA_BROKER);
  kafkaProps.setProperty("group.id", RIDE_SPEED_GROUP);
// always read the Kafka topic from the latest event
  kafkaProps.setProperty("auto.offset.reset",
      "latest");
//reading latest event
  kafkaProps.setProperty("enable.auto.commit", "true");
// create a Kafka consumer
    FlinkKafkaConsumer08<String> consumer = new
        FlinkKafkaConsumer08<>(
    "spark-winagg5",
    new FlinkSimpleStringTsSchema(), kafkaProps);
// create a TaxiRide data stream
  DataStream<String> rides = env.addSource(consumer);
```

Thirdly, we will create Windows on Apache Flink[4], and inside of the Windows we will comparing with the events of the Data Historic (Side Input)

```
//Processing the Data Stream
 DataStream<Tuple15<Integer,Boolean,String,String,Float
 ,Float,Float,Float,  Short,Long,Integer,Long, Long,
     Long, Long>>
//Maping the taxi rides events
  taxiside = rides.map(new
      Aggregations.MapPassengerR())
 //Assigning timestamp
     .assignTimestampsAndWatermarks(new
         AscendingTimestampExtractor
     <Tuple15<Integer,Boolean,String,String,Float,
     Float,Float,Float,Short,Long,Integer,Long,
     Long, Long, Long>>() {
@Override
     public long
         extractAscendingTimestamp(Tuple15<Integer,Boolean,
     String,String,Float,Float,Float,Float,Short,Long,
     Integer,Long,Long,Long, Long> element) {
         return element.f9;
            }
         })
//grouping all values by rideId on position 0
 .keyBy(0)
// generationg tumbling windows
 .timeWindow(Time.milliseconds(Long.valueOf(args[0])))
//sums the 1s and the passengers for the whole window
 .reduce(new Aggregations.SumAllValuesR())
 .map(new RichMapFunction<Tuple15<Integer,Boolean
  ,String,String,   Float,Float,Float,Float,Short,
  Long,Integer,Long, Long, Long, Long>,
   Tuple15<Integer,Boolean,String,String,Float,Float,
   Float,Float,Short,Long,Integer,Long,
   Long, Long, Long>>()
   {
@Override
  public Tuple15<Integer,Boolean,String,String,
  Float,Float,Float, Float,Short,Long,
  Integer,Long, Long, Long, Long>
  map(Tuple15<Integer,Boolean,String,String,
  Float,Float,Float,Float,Short,Long,
  Integer,Long, Long, Long, Long> t)
  throws Exception {
//Generating the ArrayList with the side input data
  ArrayList<Long> side = (ArrayList<Long>)
    getRuntimeContext()
    .getSideInput(sideInputHistoricalData);
           Long millis = System.currentTimeMillis();
           Long duration = millis - t.f12;
           Long seconds = duration/1000;
           Long idrides = t.f0.longValue();
//Iterative of the side input array list with the
    datastream
 for (Long s: side) {
   if (s == idrides) {
     t = new Tuple15<Integer,Boolean,String,String,
     Float,Float,Float,Float,Short,Long,
     Integer,Long, Long, Long, Long>
     (t.f0,t.f1,t.f2,t.f3,t.f4,t.f5,
     t.f6,t.f7,t.f8,t.f9,t.f10,
     t.f11,
         t.f12,(System.currentTimeMillis()-t.f12)/1000,
   (System.currentTimeMillis()-t.f12));
       }
     }
 return t;
  }
})
//Calling to Side Input
    .withSideInput(sideInputHistoricalData);
```

Finally we will print and export the results into a csv file and executing the Flink job

```

//Printing the results
  taxiside.print();
//Exporting the results into a text file
  taxiside.writeAsText("/share/hadoop/kafka/kafkaAV/out
  /windowflink/latency/outputfile.txt",
  FileSystem.WriteMode.OVERWRITE).setParallelism(1);
// execute the transformation pipeline
  env.execute("Flink Hybrid Benchmarking");
```

3. RESULTS

In the following section the results of the experiments for both systems will be shown. First the behaviour regarding the different parameters of each systems is analyzed separately. Afterwards a comparison of Apache Spark vs Apache Flink is shown.

3.1 Spark

3.1.1 Historical data size

The first experiment that was conducted has the goal to analyze the behaviour of Spark regarding the size of the historic data (batch file). In figure 3 the latency is shown. For a smaller historic data the system was performing decent with a latency around 1000 ms and slightly increasing until a file size of about 25 MB (200000 tuples). At this point the latency is increasing drastically and from a practical perspective not usable. This high latency results from a delay in the beginning, where the computation takes too long, that the system cannot recover fast enough or at all.
A look at the number of actual processed elements (throughput) for the same experiments in figure 4 shows that Spark can process all incoming elements until that critical point of about 25 MB. Afterwards the system cuts down the processed elements and has still that large latency. Therefore, hybrid systems in Spark work decent until a critical point. From that critical point the system is not practicable usable. Although the system could be tuned with for example more node or cores, it is to expect that at some point the system‘s performance crashes. Basically just the graph would be shifted.

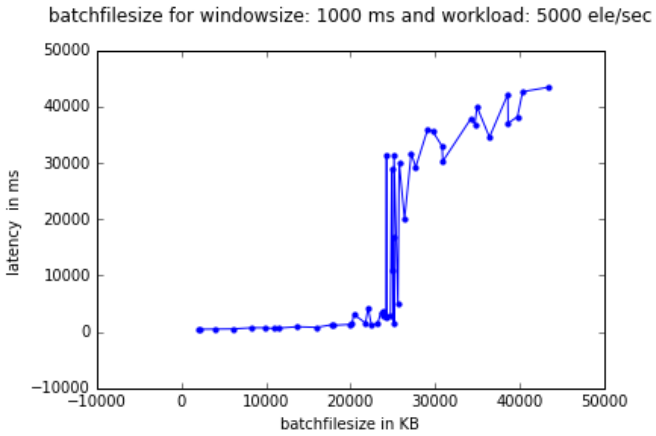

Figure 3: Latency for increasing historical file size

3.1.2 Behaviour one experiment

To analyze the behaviour of the backpressuring mechanism of Spark the latency and throughput for one experiment was conducted. The backpressuring is a built-in mechanism in streaming systems that lowers the processing rate of it to avoid a large latency. This mechanism should in an optimal system always find very fast the optimal processing rate for a specific computation. As seen in the previous graphs that mechanism is not working optimal as it has a very large latency.
In figure 5 one can see that in the beginning of the experiment the number of processed elements in each window if

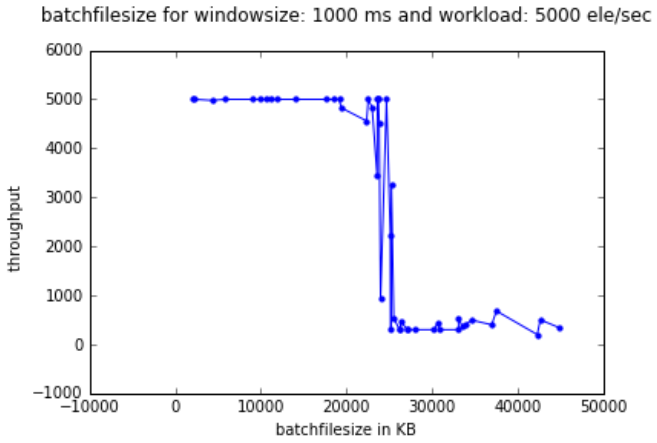

Figure 4: Throughput for increasing historical file size

very high regarding the workload. Basically it tries to process all elements. As this results in a delay of the computation the system needs to reduce the processed elements to recover of the delay. After about 7 minutes the system find a stable situation in which the systemÂt's throughput fluctuate around 5000 elements.
The figure 6 shows the latency over the time for the same experiment. To see here is that until the stable point, as in the previous graph, the latency is very high and inconsistent. After that the latencies are in a more usable range. Therefore, to measure the real latency for an experiment it might be better to measure it from the point where the system gets stable. The problem with this method is that it is not always clear at which point the system is stable and if it gets in a stable situation overall. Also at conducting the experiment it would not be clear how long it takes to find the stable situation as with bigger complexity it takes longer to find it and therefore the duration is not clear. This makes the measurement of the latencies not very exact.
In the following experiments as in the previous one, all elements were included in the computation of the latencies to achieve better consistent results, although they might have larger latencies.

3.1.3 Workload

The behaviour of the latency for an increasing workload (number of produced elements) is similar to the one of the historic data file. Figure 7 also shows that Spark can perform decent for low workload. Like in the other experiment a critical point exists from that the systems‘s performance is drastically decreasing.

3.1.4 Window size

This experiment for an increasing window size is shown in figure 8. One can see in that figure that an increase of the window size also results in an increase of the latency. This effect is not as drastically as for the other parameters, but still has a noticeable effect.

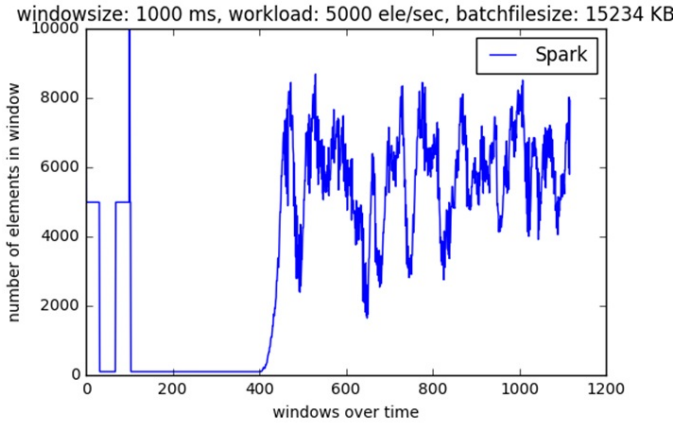

Figure 5: Throughput for one experiment

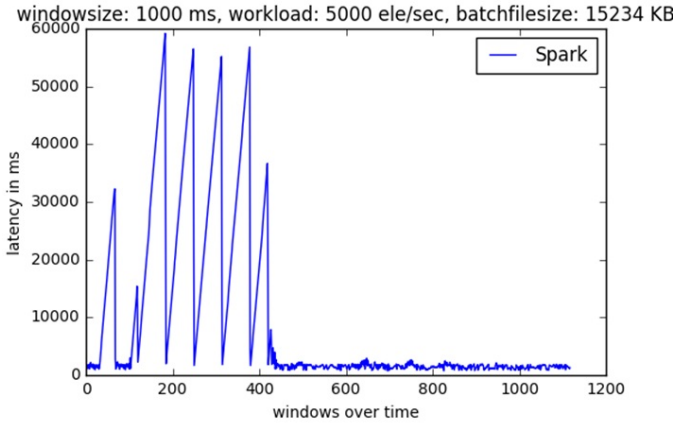

Figure 6: Latency for one experiment

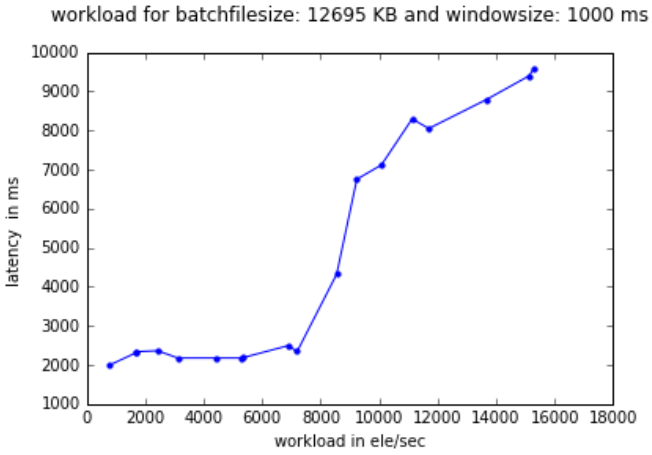

Figure 7: Latency for increasing workload

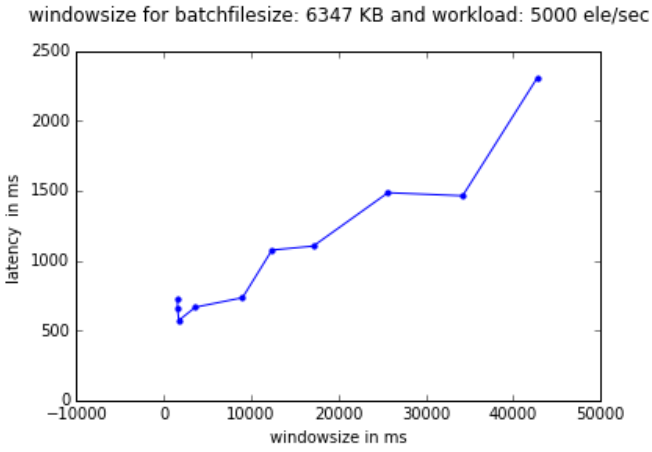

Figure 8: Latency for increasing window size

3.1.5 Increasing workload at runtime

Another way to measure the performance is to start with a very low produced workload that is slowly increased through the experiment. Therefore, one can see what is the maximal workload for which the system performs well. To achieve that two experiments were conducted with different size of the historical data and the maximal produced workload. The workload started from 100 elements per second to a maximum of 10000 and 40000. Each experiment was three hours long and the workload was increased after every three minutes.

The figure 9 shows an experiment with a lower maximum workload and a lower historical data file. The red line in this figure represents the number of elements that were produced for at that specific time, while the green one shows how many of these elements were actually processed by Spark. That both lines are nearly almost the same means that the system can process all elements through the whole experiment. This results also in a very low latency (blue line). One also can see that after each increase of the workload the processed elements fluctuate as the system's backpressuring tries to find the optimal number of elements. For a higher workload the fluctuation is generally bigger and it takes longer to find the best number, which is in this experiment the maximum number of produced elements.

In figure 10 an experiment with higher throughput and higher maximal workload was conducted. Similar to the previous experiment the green and the red graph were in the beginning the same. But at about a workload of 20000 elements per second the system did not significantly increase the number of processed elements and they stayed stable about that number. This is done by the backpressuring mechanism to achieve best performance. This results into a low latency throughout the experiment. With an workload of 20000 elements per second the system now performs better than is was when the workload would have started with that amount.

Hence, with the method of an increasing workload at runtime, one can achieve a better performance. This happens because the backpressuring of Spark needs a long time to find the optimal number of processed element. If there are just small increases in the workload the system can better react to them and find faster the optimal number.

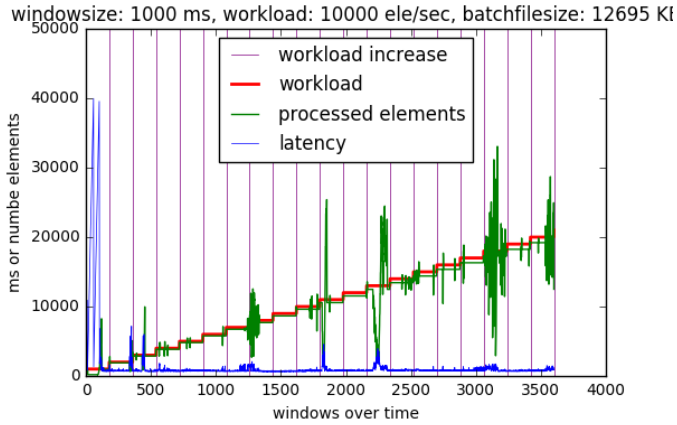

Figure 9: Increasing workload at runtime with low input

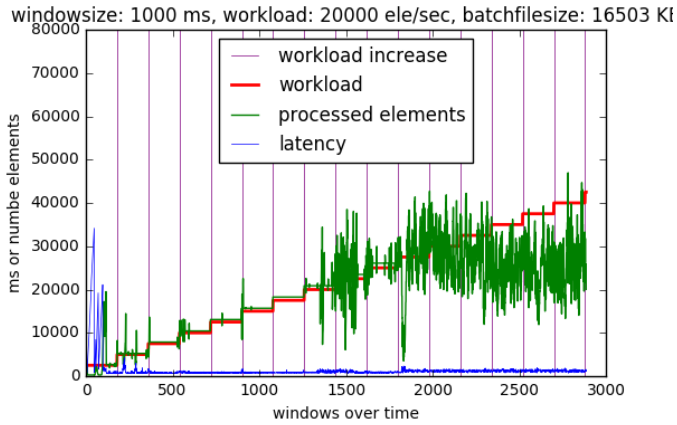

Figure 10: Increasing workload at runtime with high input

3.1.6 Conclusion - Spark

In the previous experiments it was shown that Spark performs for hybrid systems decent regarding the latency and throughput until a critical point. After that critical point the system‘s performance is very bad and not usable. This occurs for all parameters, although for the window size has the smalles effect on the performance. Furthermore it was shown that because of its bad backpressure mechanism it takes a long time to find the optimal number of processed elements. For conducting experiments it might be better to do long run experiments and just use the data after finding a stable situation. This might take very long for complex problems and it is not clear after which time such a stable situation occurs.
With the experiment of the increasing workload, results could be achieved that were much better than the ones with a directly high workload. This is explained by the fact that with lower increases of the workload the system‘s backpressuring can more easily react to it and find much faster the optimal number of elements. Therefore, the system does not lead into a critical point as the system is not increasing the number of processed elements while maintaining a low latency.

3.2 Flink

3.2.1 Historical data size

In our first experiment we will measure the behaviour of the latency considering the size of the historical data (Side Input) as a variable, it will be changing through time in the generation of windows. In the Figure 11 we can observe that in whole cases that we did the experiment, considering a sparse range from data historical size from 15KB(100 events) to 1277000 KB (1.2GB)(1000000 events) the latency remains relatively constant from a range 75 milliseconds to 110 milliseconds We can conclude that the processing latency of Apache Flink[4] is extraordinary low, because the latency stays bellow 120 milliseconds. Unfortunately, based on the same experiments and if we aim to the number of actual processed elements (throughput) on the Figure 12 we can appreciate that Apache Flink[4] can deal and process whole incoming elements until the critical of about 25MB (200 000 events). Thereafter stay to reject several incoming events periodically and for example at the point of 25MB processed 5000 from 5000 events/sec incoming events but when is a historical data of 63MB it processed 1885 from 5000 events /sec, and in our high data historical size with 1.2GB, it processed just 956 from 5000 events/sec.

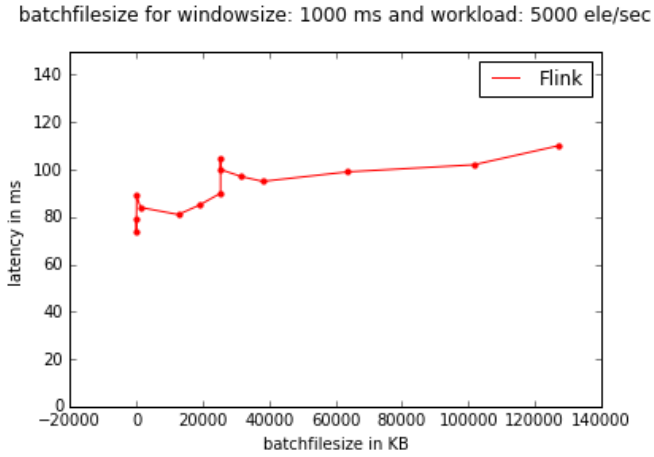

Figure 11: Latency for increasing historical file size

Hence, hybrid computation in Apache Flink[4] works very well with very-low latency for all cases, but regarding on the throughput (elements processed) it went down widely with the increasing of the historical data size.

3.2.2 Behaviour one experiment

In this section we will analyze the behaviour of one experiment taking into account the next constants: windows size : 1000 milliseconds, workload: 5000 elements / seconds, and a batchfile size (Input Side) of 25800 KB. In the Figure 14 we are going to analyse the latency in each windows generated or grouped by the keyed rideId. Also we will analyze with this same experiments the number of elements that were processed considering rideId as a key for each window in the range of 1000 milliseconds showed on the Figure 13

Analyzing the results in the Figure 14 we can observe that the latency was relatively stable in a range of 0 to 300 mil-

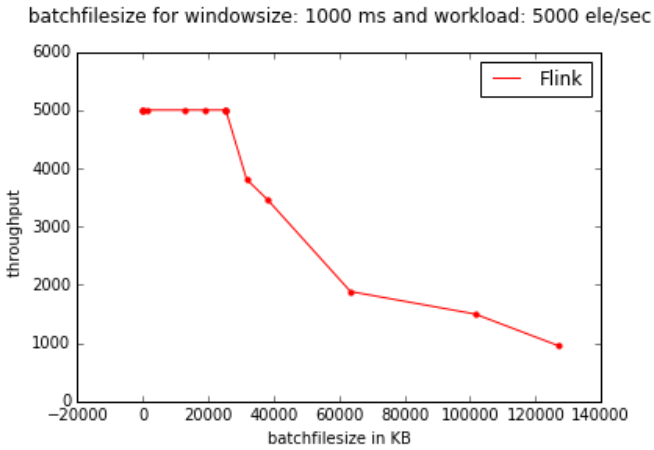

Figure 12: Throughput for increasing historical file size

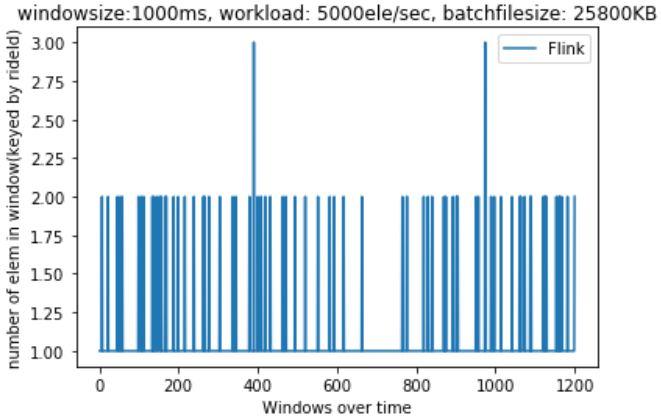

Figure 13: Throughput for one experiment

liseconds, with a average latency around 100 milliseconds, that is an extraordinary low latency of Apache Flink[4].

Regarding on the number of elements processed in this same experiment, showed in the Figure 13 we can observer that due that we created the windows keyed by rideId, we have a huge sparse data with many different rideId, and usually in one second, considering 5000 events sent by kafka, there are one or two times the events with the same rideId. Its important to clarify that in this experiments all events were processed in its totality successfully.

3.2.3 Workload

These experiments were did with the next constants: windows size : 1000 milliseconds,and a batchfile size (Input Side) of 12695 KB. We were increasing frequently the workload from 100 elements / seconds until 30 000 elements / seconds. The Figure 16 showed the behaviour of the latency (number of produced elements by second), it is similar to the previous experiment of changing the historical data on the Figure 11, hence the latency remains steady in a range of 80 milliseconds to 130 milliseconds, with an average of 110

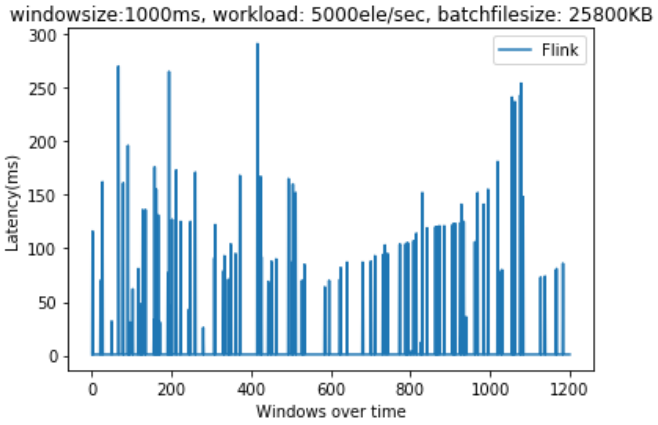

Figure 14: Latency for one experiment

milliseconds in total of latency for this experiment, which in terms of very low latency that an streaming processing framework needs to have, it is outstanding.

However, we can appreciate that the elements processed (throughput) in the Figure 15 start to decreased significantly when Apache Flink[4] commences to receive and process more than 5000 events per second, until the point that for example when it receive a workload of 25000 elements per seconds it just can process 8494 events from 25 events received from Kafka. In other words the performance decreasing gradually with more that 5000 events per seconds as workload.

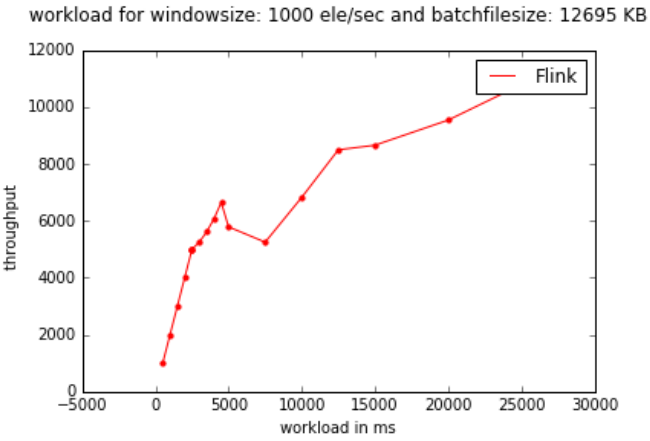

Figure 15: Troughput for increasing workload

3.2.4 Window size

These experiments were approached considering the Windows size as a variable, we can observe on the Figure 17 that we have an special case when the windows size is 1000 milliseconds, always in several experiments on the cluster or in stand alone mode we obtained a latency around 100 milliseconds, after that with rising of the windows size from 2000 milliseconds to 35 milliseconds or more, we had an stable latency in the range of 600 milliseconds to 750 mil-

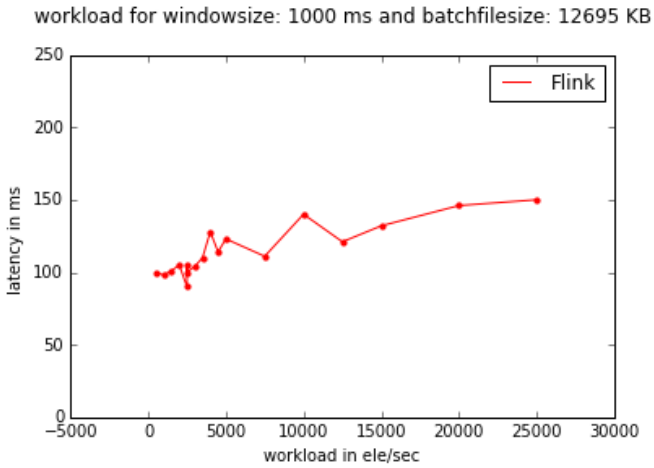

Figure 16: Latency for increasing workload

liseconds, that comparing with another streaming processing frameworks, Apache Flink[4] has a remarkable and important very law latency, it could be due the nature of streaming processing.

Figure 17: Latency for increasing workload

3.2.5 Cluster Size

These experiments were approached considering the cluster size or number of threads as a variable, we can observe on the Figure 18 that we have an special case when the cluster size is in stand alone, and when we increasing the number of threads until 250 the latency remains stable in a range behind of 140 milliseconds for whole cases. I think that it happened due that that the historical data was 25800KB, and easily could be managed or processed just in the memory of one cluster.

3.2.6 Conclusion - Flink

For working in these experiments inside of the scope of hybrid computation, we needed to work with the Side Input for data streams API, which is a new functionality available on

Figure 18: Latency for increasing cluster size

Proteus engine and in Flink improvements proposals (Flip 17), and not part of an official released version of Apache Flink that at this time was the version 1.1.4.

The specific function used in Side Input was the newBroadcastedSideInput, that broadcasted the historical data to whole nodes and threads of the cluster , and after that we could concatenate or compare it with each events received from the kafka stream producer.

According to the experiments computed Apache Flink demonstrated in whole cases switching several parameters such as (workload, side windows, historical data size) provide a very-low latency.Nevertheless when Apache Flink receive more than 200000 events per seconds, it start to not process gradually several events. In the same way there is a critical point for not processing whole eventes when the historical data file is major than 200 000 events.

We can conclude that Apache Flink had an outstanding performance, and considering that in real work not many companies or customers neeeds to process more than 200 000 elements per second.

3.3 Apache Spark vs Apache Flink

The next graphs describe the behaviour of the streaming frameworks Apache Spark vs Apache Flink considering variations in the workload, windows size and historical data size. As seen in the previous results Flink outperforms Spark regarding the latency as it handles much better and faster the backpressuring. Therefore, the throughput was used to analyze both systems.

3.3.1 Benchmarking of Historical Data

The figure 19 describes the behaviour of the throughput changing the historical data size between Apache Spark and Apache Flink.

Based on the previous results we can conclude that in several cases Apache Flink had a outstanding and stable very low latency, by the literature[1] the nature of the Flink processing engine has a better performance and offer a lower

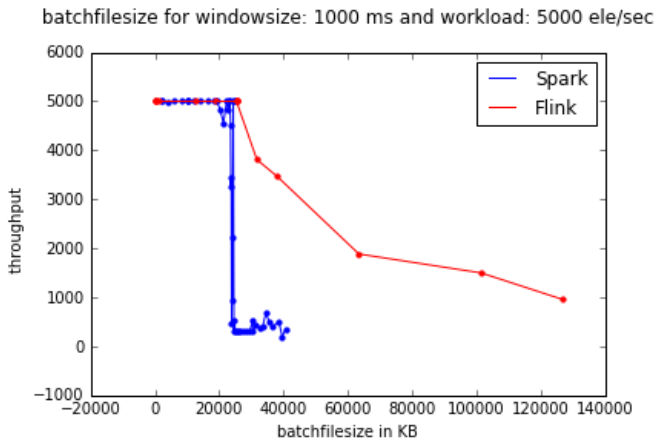

Figure 19: Benchmarking of Historical Data size vs throughput

latency for dealing with data stream rather than the Apache Spark that aims to work with DStreams or micro-batching approach. Considering the throughput in both cases Apache Flink and Apache Spark have in the beginning (low historical data) a good performance for processing workload of 5000 elements per seconds, but when the data size is enhanced Apache Spark reduces drastically the number of processed elements. This critical point was described in the previous chapter about the results of Spark. Meanwhile Apache Flink decrease slowly the acceptation of process more incoming events major of 5000. This shows first that Flink has a better performance regarding a larger complexity of the problem and second that it handles better the backpressuring as it, opposite to Spark, manages to slightly reduce the number of processed elements.

3.3.2 *Benchmarking of Window size*

The figure 20 describes the behaviour of the throughput changing the windows size between Apache Spark and Apache Flink.

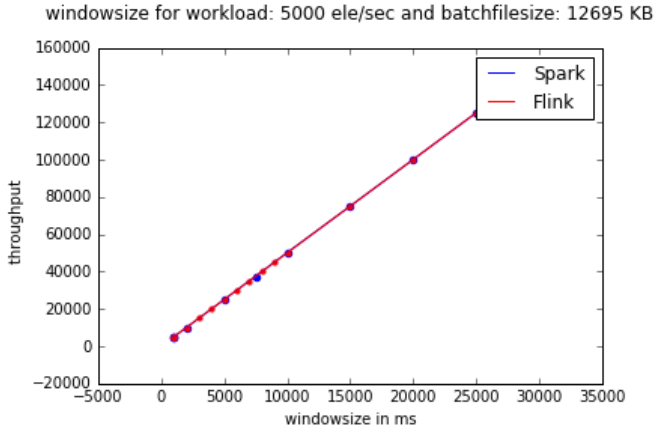

Figure 20: Benchmarking of Window size vs throughput

The throughput for both systems was at any time at the maximum of incoming elements and therefore for both systems the same. This can be explained by the small complexity of the experiment. As the latency of Spark is linearly increasing while for Flink it stays rather stable, means that Flink also performs better in this case.

3.3.3 *Benchmarking of Workload*

The figure 21 describes the behaviour of the throughput changing the workload (speed of events produced) between Apache Spark and Apache Flink.

Figure 21: Benchmarking of Window size vs throughput

This figure shows that both systems can keep up in the beginning with the increasing workload. As it gets larger Spark can even process slightly more elements between a workload of 5000 to 10000 elements per second. Afterwards Spark reached again its critical point and can not keep up at all. Flink on the other hand can slightly increase its processed elements, although it can not process all incoming elements.

4. CONCLUSION

The integration of data of batch and streaming computation is pivotal to important business processes. It is the main reason that Apache Spark and Apache Flink (in progress) are offering this capability in its streaming frameworks. It was hown that the usage of hybrid computations for systems, like Spark, that are based on microbatches is easier to achieve. This project aimed to benchmark the latency and elements processed (throughput) as measurements for comparing the performance of these two popular streaming platforms.

Considering the several experiments it was shown that Flink has a much better latency compared to Spark. This latency was, regarding the different parameters, more or less constant or just increasing very slowly. This resulted from the better usage of the backpressure mechanism of Flink as it can reduce more efficiently the number of processed elements for an increasing complexity of the experiments. Spark's performance was characterized by a critical point from which the system could not keep up with the computation, which resulted in an extremely high latency and low

throughput. Before that critical point Spark has a similar performance regarding the thoughput as Flink, but also a higher latency.

5. REFERENCES

[1] D. O. J. Ellingwood. Hadoop, storm, samza, spark, and flink: Big data frameworks compared. https://www.digitalocean.com/community/tutorials/hadoop-storm-samza-spark-and-flink-big-data-frameworks-compared, 2017. [Online; accessed 1- March-2017].

[2] P. Engine. Proteus engine. https://github.com/proteus-h2020/proteus-engine, 2017. [Online; accessed 1- March-2017].

[3] A. S. Foundation. Apache spark. http://spark.apache.org/. [Online; accessed 1- March -2017].

[4] A. S. Foundation. Apache flink. https://flink.apache.org/, 2017. [Online; accessed 1- March-2017].

[5] A. S. Foundation. Apache flink - flink improvement proposals (flip). https://cwiki.apache.org/confluence/display/FLINK/Flink+Improvement+Proposals, 2017. [Online; accessed 1- March-2017].

[6] A. S. Foundation. Apache flink - taxi ride training exercise. http://dataartisans.github.io/flink-training/exercises/taxiData.html, 2017. [Online; accessed 1- March-2017].

[7] A. S. Foundation. Apache kafka. https://kafka.apache.org/, 2017. [Online; accessed 1- March-2017].