DDPS - Assignment #1

Pavlos Zakkas — Ivan Horokhovskyi

October 29, 2021

1 Introduction

For the assignment of **Reproducibility Study**, experiments from the "Benchmarking Distributed Stream Data Processing Systems" article were reproduced.

The authors of the article tried to address the lack of detailed performance evaluation on streaming data processing systems. As a result, they experimented with three widely used open-source systems, namely Apache Spark, Flink and Storm. Specifically, they measured performance metrics on windowed operations, including processing and event time latency, scalability and throughput.

Our motivation behind picking this article was based mostly on the following reasons:

1. Compared to the other articles, it is the most recent (published in 2018), and this could give us the chance to get familiar with quite recent research experiments
2. Performance evaluation is an interesting domain, and there seem to be several challenges to make valuable research
3. Three well spread tools are used (Apache Storm, Apache Spark, and Apache Flink)
4. Lots of extensions can be made including Apache Kafka, BI tools, and more

Regarding our experiments, we mainly used Apache Spark to reproduce them. It is also important to mention that we also got familiar with Apache Flink, but the results of it are not included into the report due to specific issues that will be mentioned below.

Finally, the code that was used to reproduce the experiments can be found on github repository.

2 Experiments

2.1 Authors' experiments

In this section, it is important to identify the experiments that were performed by the authors and also the main design decisions that were made.

At first, they mention that on-the-fly generation of data is preferable for performance evaluation, compared to message brokers such as Apache Kafka, as the the data exchange between the message broker and the streaming system may easily become the bottleneck. As a result, they used in-memory data generators with configurable generation rate in order to produce data faster than the ingestion rate of the systems under test (SUT). However it is strange to have such benchmarking since in the vast majority of the cases mentioned streaming systems are used with message brokers, this is why it is not 100% clear what is the practical use of the experiment.

Moreover, it is constantly mentioned that queues were used between the generator and the SUT and that the target system was completely isolated in order not to affect its performance. Nevertheless, there is no clear description of what kind of queues were used, in order to produce high load without though creating a bottleneck between the data generation and the target system.

Specifically, for Apache Spark the supported built-in streaming sources are either basic streams including files and sockets or advanced streams such as Apache Kafka or Kinesis. Based on the facts that firstly we should limit the bottleneck between data generation and target system, and that

Software	Version
Spark	2.12.15
Scala	3.2.0
Python	3.9.5

Table 1: Software used

secondly there is no need for fault tolerance as we are only interested in performance evaluation, we decided to use streaming through sockets. Last but not least, adding queue inside the data generator doesn't seem to have any benefit as it could only delay the delivery of the messages.

In addition, there was made a variety of experiments performed by the authors. Windowed aggregation and join queries of gem packs purchases and ads streams were used, while the performance metrics that were evaluated are based on processing and event time latency, sustainable throughput of the systems and scalability of each system while increasing the number of nodes. Specifically, below you may find the list of experiments that were used by the authors in order to measure the aforementioned metrics:

1. Windowed aggregations with 8 seconds window length, 4 seconds slide and 4 seconds batch size
2. Windowed joins in which length and slide is not mentioned. Probably the same values as aggregation were used.
3. Queries with large windows of 60 seconds length and slide
4. Data skew in which events of a single key were streamed
5. Fluctuating workloads for windowed aggregations and joins
6. Comparison between event-time and processing-time latency
7. Observing backpressure and short-term spikes
8. Measuring throughput over time

For the execution of the experiments, a cluster of 20 nodes with 2.40GHz Intel(R) Xeon(R) CPU E5620, 16 cores and 16GB RAM was used. A dedicated master with either 2, 4, or 8 worker or driver nodes were deployed for comparison, while the system clocks were synchronized via a local NTP server. 16 instances of data generators were used in order to produce 100M events with constant speed while the messages were generated with normal distribution on key field.

Finally, concerning the presented results, it is important to mention that the number of iterations made for each experiment is not mentioned. Also, plots and tables regarding latency do not clearly mention what kind of latency they measure. From the values presented though, we assume that it is related to processing time latency. Tables of latency statistics contain minimum, average, maximum and 90, 95 and 99 quantiles statistics, which seem to be appropriate kind of measurements in order to give an overview of the performance. Nevertheless, the plots of latency over time do not seem to be of the same quality, as the measurements recorded do not refer to min, max or quantile values.

2.2 Main experiments & design

First of all hardware and software comparison was done. Authors are using Spark 2.0.1 which used to be outdated by the time of article publication since Spark 2.3 was already released. Used software can be observed in Table 1. Libraries versions will be provided in the requirements files of the project. For each experiment we did at least 5 repetitions and averaged the results because there were no significant fluctuations.

The query on which experiments were based is window aggregation, the key for group by is numeric values distributed normally with std equal to 1 (author don't mention the std they used). Regarding experiments it was decided to reproduce following experiments:

2.2.1 Windowed Aggregations

Idea - Basic window aggregation experiment by which it is easy to compare baseline with authors results.

This experiment was repeated varying the nodes quantity and batch size parameter. Ten iterations were made for each configuration in different time slots, and the results seemed to be significantly similar, so we concentrated on the average values for latency, as adding quantiles would make the plots more noisy without any benefit.

2.2.2 Event-time vs processing-time latency.

Idea - to compare the differences within two main streaming processing metrics.
Authors conduct experiments with aggregation query (8s,4s) on a 2-node cluster.

2.2.3 Queries with large window

Idea - since window and slide size have a huge impact on the framework performance especially in Spark. Due to the framework policy the windowed operation results are cached to be usable in later stages of a job. This process aggressively consumes memory and if it is not enough of memory - the disk starts being used which affects performance negatively.

Authors results does not match with our prior knowledge. Moreover, it is quite spread use-case in real life tasks.

This experiment was done in 2 different ways, with streaming batch size 1 and 4 seconds. It was assumed with higher batch sizes Spark can handle higher workloads and big windows sacrificing low latencies. The authors of the article used 4 seconds, so it was decided to additionally make experiments with 1 second batch size to check the mentioned above theory.

2.2.4 Data skew

Idea - in this experiment all data had same key, so framework will needs manage partitions efficiently. Spark, forcing all partitions to send their reduced value to a specific computing slot can easily cause a network to become a bottleneck when partition size is big. Therefore, Spark adopts tree reduce and tree aggregate communication pattern to minimize the communication and data shuffling.

2.3 Additional experimentation

Additionally to the experiments that were made by the authors, it was tried to experiment with the following alternative implementations.

2.3.1 Streaming from multiple sockets

As shown later in the Results section, the recorded event-time and processing-time latencies were smaller than what the authors recorded. The main reason for this is the fact that we could not achieve the same data load of events. Even with 32 processes of data generator and a huge number of messages being produced, the generation rate could not exceed the value of 150.000 messages per second. That made us conclude that the usage of a single socket is the bottleneck for this kind of experiments.

To overcome this issue, we tried to use a larger number of sockets and implement a union of data streams in Spark in order to combine data from all those sockets simultaneously. The execution of this experiment did not work as expected, because Spark could not apply effectively a union of streams when using sockets. As a result, even though the implementation was made, we could not provide any experiment results.

2.3.2 DataFrames API

In the article authors used RDD API, nowadays Spark benefits from using structured DataFrame and Dataset APIs. Catalyst is an optimizer which leverages advanced programming language features (e.g. Scala's pattern matching and quasi quotes) in a novel way to build an extensible query optimization.

Unfortunately DataFrames do not support socket streaming and this is why we tried to implement structured streaming using Datasets. Nevertheless, results were surprising, as Datasets performed in approximately 10 times worse than regular RDD DStreams. It was assumed that such behavior was connected to non optimal Spark socket usage since it is used only as test input data source which might not been intended to use with high loads.

2.3.3 Flink

In addition to Spark, some experiments were made using Apache Flink. Even though the APIs of both frameworks are pretty similar, under the hood they perform pretty different, as Spark uses micro-batches even in streaming jobs, while Flink is able process rows after rows of data in real time which reduces the latency in comparison to Spark. Another important difference between frameworks is Data Flow. Flink is able to provide intermediate results via broadcasting variables.

Due to the lack of time only Spark experiments were left in the final version, even though configuration, aggregation script and some parameter tuning was done.

2.4 Implementation and Deployment

In order to execute the aforementioned experiments, a data generator, a Spark implementation and scripts to deploy those services on DAS-5 were developed.

2.4.1 Data generator

The data generator was developed in Python. The main responsibilities of this service is to establish the socket connection with the system under test, and then stream the data with the event-time timestamp attached. In order to achieve constant streaming, as authors mentioned, the number of messages, and the available time needed to send those messages are passed as parameters to the generator. From those values we can easily calculate the generation rate as well. Moreover, the host and the port of the socket are also passed as parameters, whereas in order to achieve higher load to the socket, we parallelized the execution of the generator by using a configurable parameter related to the number of processes that will run simultaneously. Finally, the main process is configurable to wait for a given amount of time before exiting in order to ensure that the experiment has finished.

2.4.2 Apache Spark

Apache Spark system was developed using Scala (not to deal with Python translation overheads) in order to aggregate the streamed purchases based on specific window. The main implementations that were made include reading from one socket by using RDDs and either a (8s, 4s) window or a large window (60s, 60s). The batch size could also be easily changed while configuring Spark's streaming context. Also, additional experimentation was made by implementing union of multiple sockets or usage of structured streaming data with Spark datasets.

In order to compute event-time latency, the event-time attached by the generator to the most recent message of a window was subtracted by the timestamp acquired just after the processing of the window. Those metrics are saved as text files by using Spark's related operation as a last step after window processing.

To gather more metrics, the 'onStageCompleted' method of 'SparkListener' was overridden and 'taskMetrics' provided by Spark were used to measure the records read, the size of the results and especially the difference between the completion time and the submission time of each window, which is equal to the processing-time latency. Except for the stages related to window processing of input messages, there were also stages with a very small execution time that were related to saving data as text files. Those stages were ignored while analyzing the results, as they are not relevant to our

windowed operations performance evaluation. The logs from the above metrics are structured as a csv format and are redirected to an output file during execution.

The above implementations are packaged in a JAR file, and are submitted as spark tasks by configuring the host and the port of the socket along with the target directory to store the generated files.

2.4.3 Deployment

Regarding deployment, each component (data generator, spark) contains a deployment directory in which python scripts were developed to initialize the generator and the spark cluster respectively. Specific arguments can be passed as parameters when calling those scripts in order to define the desired configuration of the system. Moreover, variables with regards to the environment of the systems are stored to related files, while for Spark cluster, we maintain two kinds of environment variables, including local and DAS-5 configuration.

For a complete run of an experiment, python scripts in a top-level directory named 'deployment' are used, which coordinate the whole execution. Specifically, the main script that runs the given experiment, follows the below process:

- reserves the required nodes on DAS-5
- deploys data generator on the given node and port, which will produce the parameterized number of messages during the given available time
- initializes the spark cluster on the given set of nodes, by configuring 'spark-env' and 'workers' spark config files, based on the reserved nodes and the environment variables of DAS-5.
- submits a spark task based on the experiment name that is also passed as a parameter to the script.
- cancels the reservation of the nodes after the parameterized waiting time of the experiment has finished.

3 Results

3.1 Scalability measurements

Regarding aggregation of purchases stream with window of 8 seconds length and 4 seconds slide, the same experiment was executed for Spark cluster of 1, 2, 4, and 8 nodes.

Processing time latency of those experiments can be seen in Figure 1, while the comparison between different number of nodes is clearly visible in Figure 2. As expected, while increasing the number of nodes the processing time latency decreases. This is more visible when comparing 1-node cluster to 2-nodes cluster where the latency difference is almost double. While comparing 2, 4 and 8 nodes clusters though, the difference is smaller due to the throughput bottleneck but we might say that it is still visible.

As a result, the trend seems to be similar to what the authors recorded. Moreover, as we can see, there are some spikes especially for small amount of nodes, which is also observed in authors' plots, and is probably related to the time needed by the system to process a whole window when it is gathered. Also, the largest spikes were present in the first 2% of the recorded values. Those values were removed from the plot though, as we consider that they were related to the warmup of the target system, which was also mentioned in the article, as the authors used even 25% of the data as warmup.

On the other hand, there is a significant difference in the scale of recorded values, as our observed latencies were much smaller than the latencies presented in the article. The main reason that explains this observation, is that using a single socket to generate data, combined with differences that might occur in the network, resulted in achieving lower load to the spark cluster. That was also the main reason that we tried to use multiple sockets as described in section 2.3.1. Finally, due to lower load, it was not possible to measure the maximum sustainable throughput of the system.

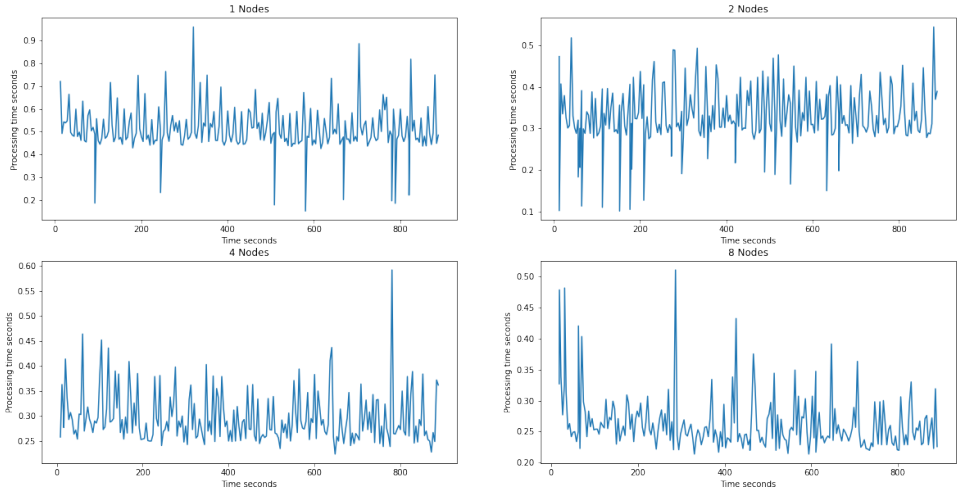

Figure 1: Processing time latency measurements for different nodes quantity

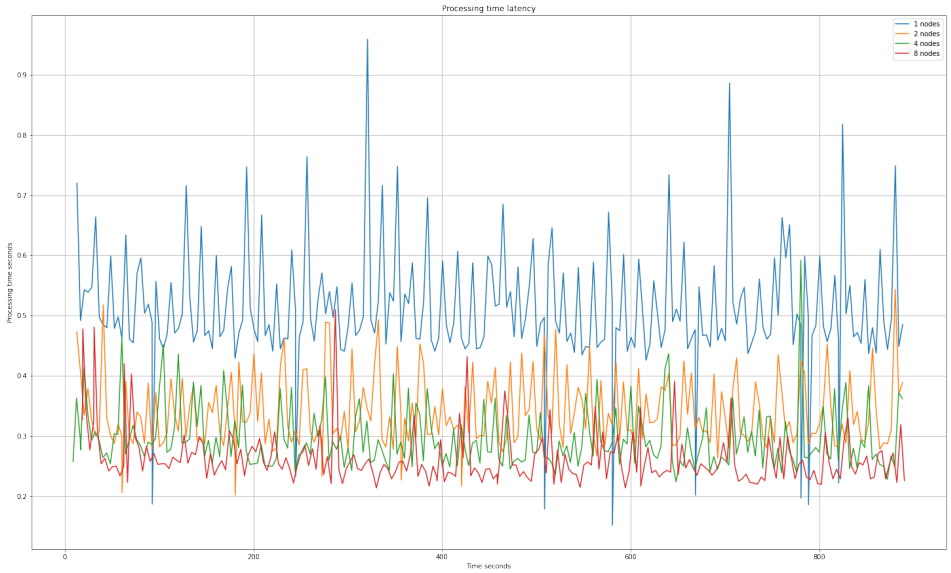

Figure 2: Processing latency / nodes quantity

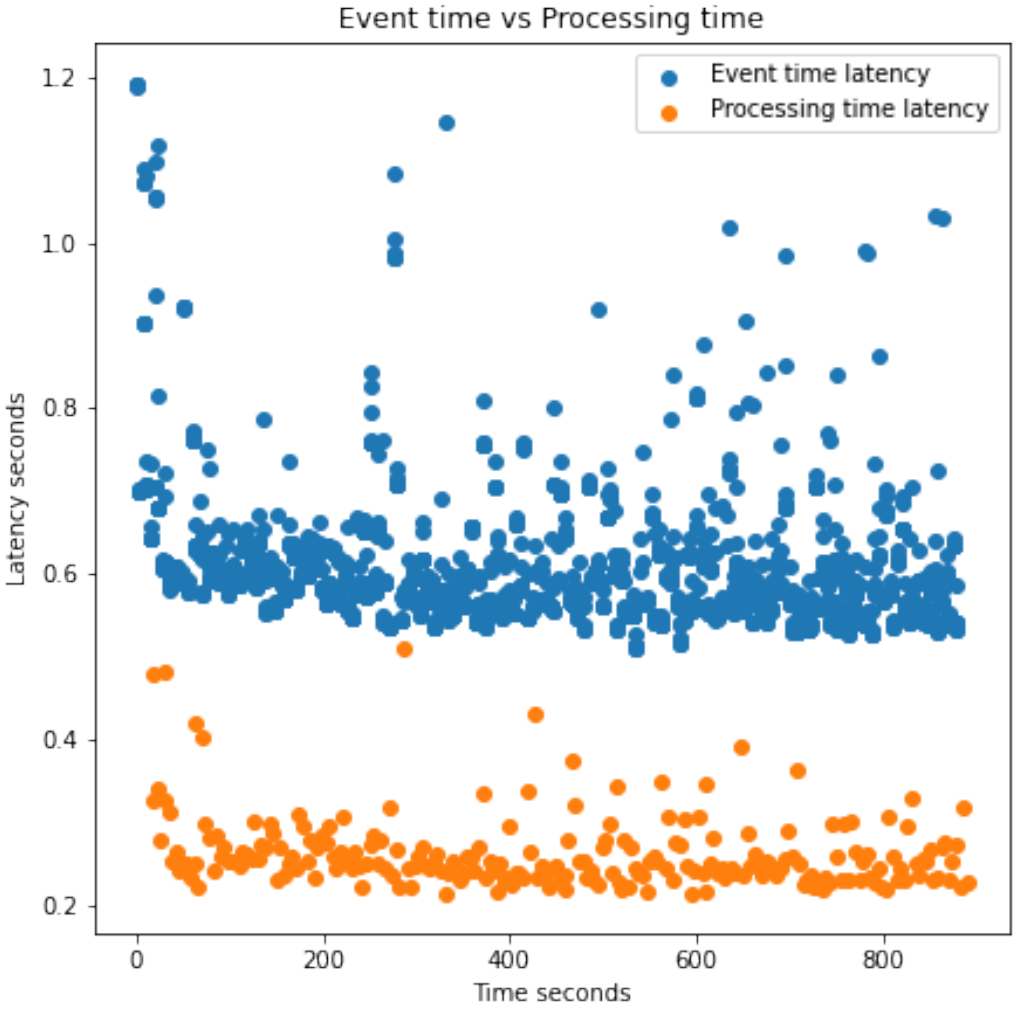

Figure 3: Event vs Processing time

3.2 Event vs Processing time latency

Authors of the article proposed to count time latency as difference between maximum start time and the end of aggregation. Even though it may seem counter intuitive from the first glance, there is definitely a point in calculating maximum because before getting the last event (with maximum time), the aggregation can not be calculated and it is unfair to call data collection as latency.

The results of this experiment can be observed on Figure 3. Event time latency includes processing time and is probably more important metrics in production since it defines the time in which the user interacts with a system and should be minimized. The authors of the article had different numbers from which were received in our experiments due to the throughput differences. In general the trend is the same, as it can be observed that input tuples spend most of the time in driver queues waiting to be processed.

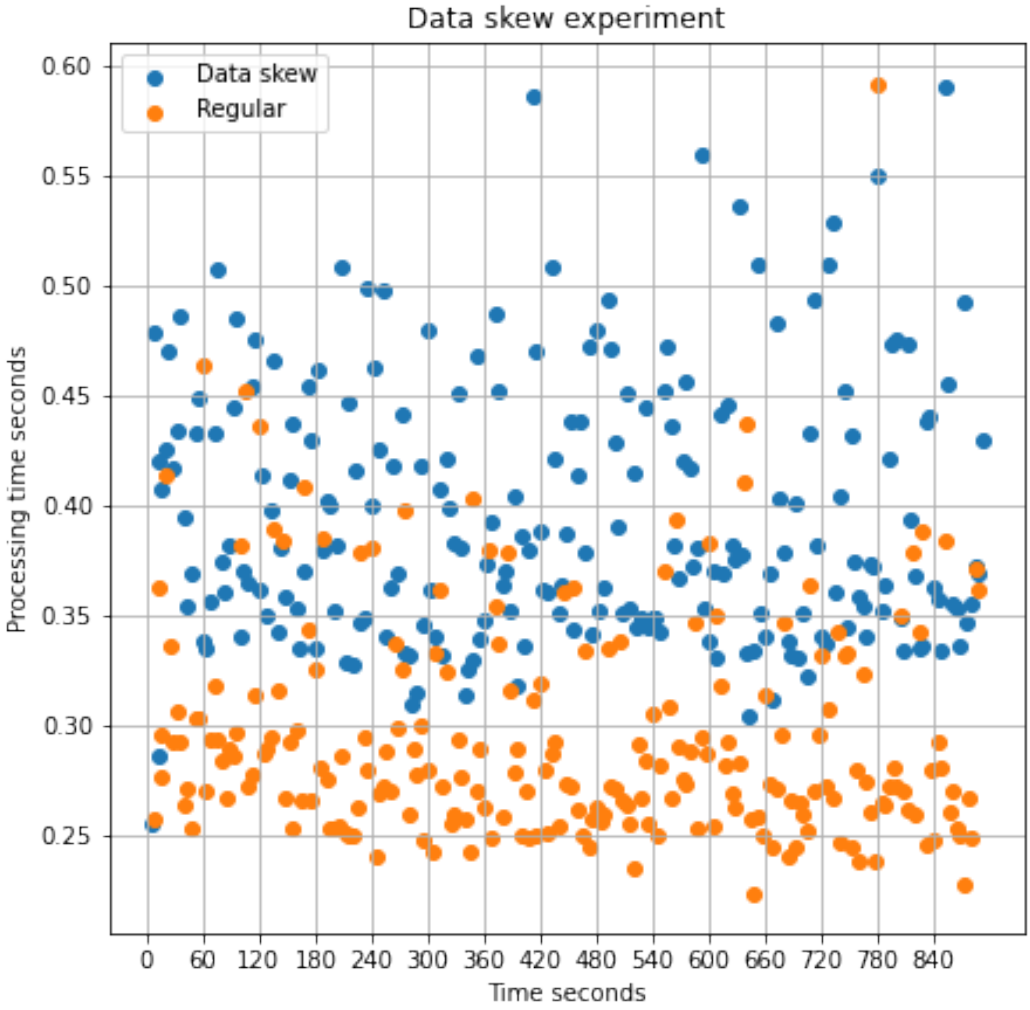

Figure 4: Data skew

3.3 Data skew

After configuring the data generator to produce data for a single key, we executed the windowed aggregation query in a 4-node Spark cluster as the authors. As shown in figure 4, the processing time latency of events with a single key, was slightly higher compared to the the events with normally distributed key, but the difference was not very significant. The authors of the article did not include any plot related to this experiment, but they mentioned that Spark can handle skewed data efficiently, and our experimentation results seem to support this point as well.

3.4 Batch size

Experimentation results from streaming data with batch sizes of 1 and 4 seconds can be shown in Figure 5. As we can see, applying batch size of 1 second, increased the performance of the system in terms of processing latency, which is expected based on the fact that Spark can read data faster and as a result process them more efficiently.

Applying 4 seconds as a batch size, which is equal to the sliding window in our experiment, may

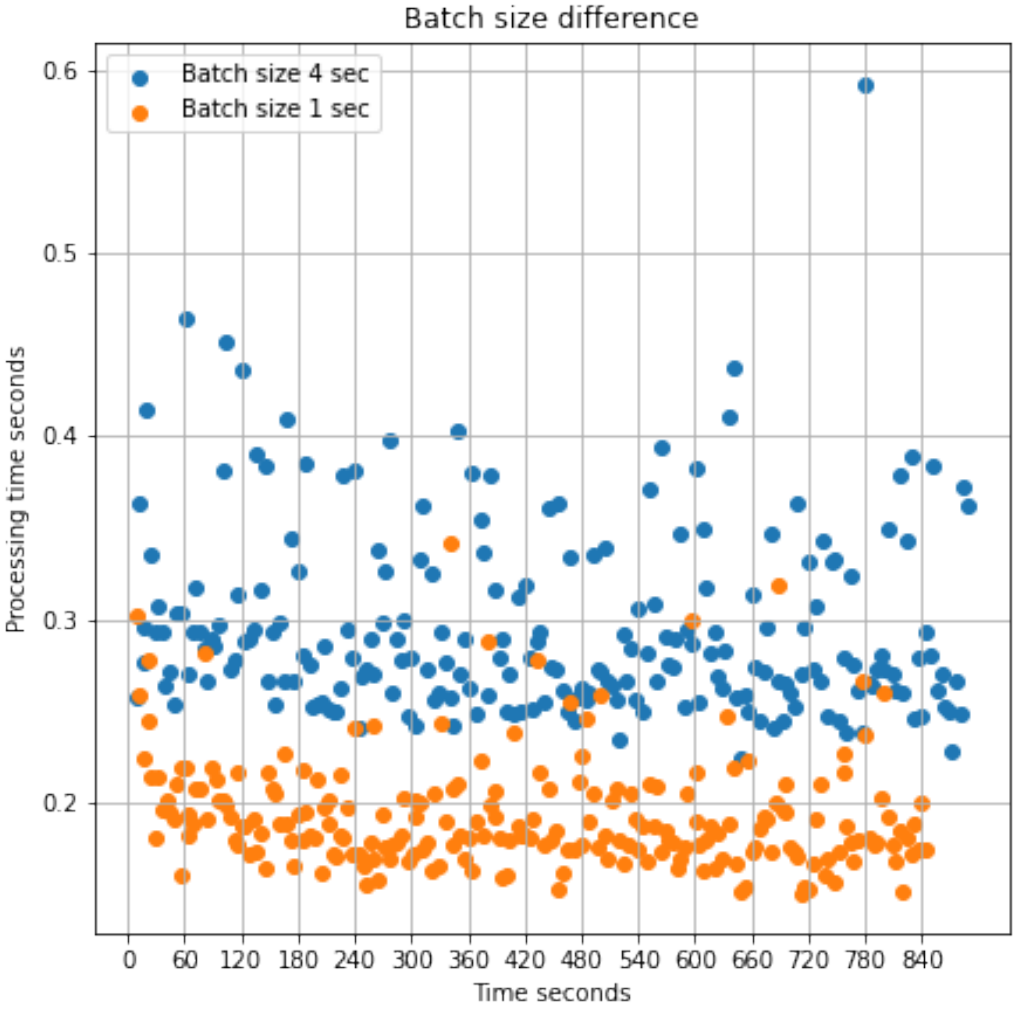

Figure 5: Processing time latency for different batch sizes

add a delay to the processing of each window, which was probably not considered thoroughly by the authors as they mainly applied 4-seconds batches in their experiments. Of course, performing the same experiment with higher load could give further insight into this statement.

4 Conclusion

In this assignment, we ~~struggled~~ worked with distributed streaming systems, and we have investigated Apache Spark RDD and Structured streaming APIs. In addition, we have made a couple of experiments with Apache Flink framework. The key metrics that were measured were event time latency, processing time latency and throughput. Finally, we investigated influence of window size, batch size, data skew and scalability.

Overall it is hard to reproduce results due to the software and hardware differences. Also, sometimes authors neglect some details which can dramatically influence on reproducibility. In our case, this detail was the system input source and networking configuration between the generator and the system. Moreover, various teams may have different opinions about some details which can influence realization

of experiment and change the results.

References

1. Karimov, Jeyhun, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. "Benchmarking distributed stream data processing systems." In 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp. 1507-1518. IEEE, 2018.
2. https://spark.apache.org/docs/latest/streaming-programming-guide.html
3. https://flink.apache.org/