Evaluation and Comparison over Spark and Flink

Junyuan Chen
Electrical and Computer Engineering
University of Toronto
Toronto, Canada
junyuan.chen@mail.utoronto.ca

Dongfang Cui
Electrical and Computer Engineering
University of Toronto
Toronto, Canada
dongfang.cui@mail.utoronto.ca

Lan Luo
Electrical and Computer Engineering
University of Toronto
Toronto, Canada
lann.luo@mail.utoronto.ca

***Abstract*—This report is for the course project ECE1724 under the supervision of Professor Ashvin Goel, where we propose a project on the evaluation of Spark and Flink and the comparison between their performance results is covered as well.**

I. Introduction

A. The Project Goal

The goal of this project is to evaluate, compare, and discuss the performance of Apache Spark and Apache Flink from several aspects, including latency, throughput, and fault tolerance. We have designed and conducted evaluation experiments on streaming data-processing, fault tolerance capabilities(throughput and latency), and performance on simple data-processing tasks.

B. Problems and Potential Challenges

Living in this era of data, we need sophisticated frameworks and tools to process data more than ever. However, with many options available, it is hard to decide which is the correct choice to choose. Therefore, to fill up the absence of horizontal comparison, we decide to investigate the performance details of Spark and Flink frameworks.

Spark Streaming is based on micro-batch and Flink is native streaming. Our plan is to test their latency and throughput in a simulated real-life production environment.

They also have different fault-tolerance mechanisms, we want to test their performance under different types of failures. In addition, though a successful streaming framework needs to support various types of computations, in this project, we will only test a single kind of computational job using benchmark tools out of simplicity.

The first challenge for the evaluation project would be the deployment of the steaming frameworks and their dependencies on distributed environments. It includes storage systems like HDFS, resource management like Yarn, and data retrieving tools like Apache Kafka. Deploying such systems requires plenty of configuration work and machine resources.

Data generation is another critical for evaluations. It requires different types of datasets to simulate different streaming tasks, such as binary data for video streaming and JSON data for status monitoring. In addition, the data generator needs to produce a stable data stream at given throughput[1]. In the light of this, we leverage the Yahoo benchmark as our tool for data generation and computation distribution.

C. Expected Outcome and Result

Recently, Spark has involved structured processing to support stream processing but remained the primary feature of batch processing. However, in our project, we only focus on the comparison between Spark DStreaming and Flink.

We expected that Flink's latency would outperform Spark since Flink supports true native streaming while Spark still primarily supports micro-batching to deliver near real-time processing. Result in that Spark has a latency of sub-second level while Flink has latency at microseconds level[1]. Spark's minimum latency is limited by the framework itself and it requires a large scale of memory when the data size is at a very large level. So, for complex data-processing operations, Flink would be the better choice[2].

For fault tolerance, those systems used different techniques. Spark used to store write-ahead logs of all the data ahead of processing and with RDD [1], it allows the system to only recover at a fine-grained level by using the checkpoint and recompute mechanism. Meanwhile, Flink uses a consistent snapshot to achieve fault tolerance where it needs upstream replay and re-run from its checkpoint[3]. Result in that Spark might have a faster recovery process compare to Flink, but Flink has the feature of convenience of replaying and reprocessing streams after system changes

For low throughput, Spark is still a good choice since it theoretically has a faster recovery response with an acceptable latency and Spark provides a much more mature and adopted environment.

II. Background

A. Frameworks

Developed and extended from the MapReduce framework, Apache Spark[4] and Apache Flink[5] are two efficient open source data processing engines. Much research has been conducted over the topic of the comparison between the Apache Spark and Apache Flink on data processing. Performance evaluations of machine learning tasks on large-scale have been discussed as well[6].

These comparisons are currently focused on several perspectives, such as batch processing, stream processing, and structured stream processing. The existence of these different processing modes is due to the variety of data processing tasks. For example, batch processing is used when the data

are collected or stored in large files while stream processing is used when the data are continuous and need to be rapidly processed[7].

However, some research has shown that neither of these frameworks can outperform the other one in terms of all data types, sizes and task patterns[8]. For instance, in Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks, it directly compared the performance of Spark and Flink on a network containing up to 100 nodes and delivered a result that summarized the cases when each framework is superior[8].

For the streaming processing, in Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming, the research team chose to use Kafka and Redis to construct a full data pipeline to simulate the real-time production-level streaming scenarios, where the data pipeline is used as a streaming benchmark tool to compare the performance of Storm, Flink and Spark engines over the 99th percentile latency and throughput under various conditions[9].

Besides, there are several standalone researches which are focused on the distributed streaming computation benchmark framework. In Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks, the authors proposed a benchmark framework called Stream Bench, which is a message system serving as an agent on streaming data generation and consumption, where it covers up to 7 benchmark programs to tackle the daily typical streaming computation case[10].

Another benchmark of streaming computation engines was proposed and developed by Yahoo in 2015, which was intended to provide a real-world use case to compare the three most popular platforms(Storm, Flink and Spark) and has been released as an open source benchmark framework[11]. With this benchmark, they have found out that Spark 1.5.1 has both higher throughputs and higher latency than Flink 0.10.1[11].

In terms of the batch processing, in A comparison on scalability for batch big data processing on Apache Spark and Apache Flink, a comparative study over the scalability of Spark and Flink has been conducted, where the authors used the machine learning libraries, the FlinkML and the Spark MLlib, correspondingly for Flink and Spark to compare their performance for batch data processing[6]. The experimental results in the paper indicated that the Spark has better performance than Flink over batch processing[6].

Furthermore, in An analysis of technological frameworks for data streams, the research group conducted analysis by making performance and latency comparisons between Spark streaming, Spark structured Streaming and Flink, where important factors such as processing guarantees, fault tolerance, and platform maturity have been considered[12].

B. Stream Processing

Unlike the batch processing, stream processing requires the real-time processing of generated or produced data flow. Nowadays a lot of data is produced in the form of streaming, such as the IoT devices sensor data, social media user activity data, or online trading market data. This data is treated as events. There are many crucial use cases of stream processing in our daily life. For instance, the retail industry needs purchases to be processed in real-time, the healthcare industry needs patient monitoring status data to be handled in real-time, and the financial industry needs transactions to be validated in real-time.

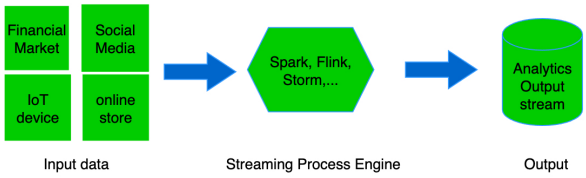

Fig. 1. How Streaming Processing WorkFlow

Fig.1 briefly shows the streaming processing's workflow. Typically, streaming processing requires capturing data in real-time for a variety of sources, processing in real-time, and eventually routing to different places for different purposes such as data analytic tasks.

C. Benchmarks

In the performance evaluation, benchmarking is the foundation of quantitative analysis for a computation system. In the following part we introduce two existing data streaming processing networks for Apache Spark and Apache Flink.

1) Yahoo! Streaming Benchmark: The Yahoo! Streaming Benchmark is a tool for testing mainstream streaming processing platforms, which is designed as an advertisement streaming data processing application. Back to early years in the last decade, different technology and frameworks on processing streaming data have been emerging. In order to build a real-world streaming benchmark, the engineering team at Yahoo! decided to develop the Yahoo! streaming benchmark, which is a streaming computation engine to compare the frameworks in the streaming processing landscape. In the initial evaluation the Yahoo! streaming benchmark was limited to three platforms, Apache Store, Apache Spark, and Apache Flink. Later on, other systems such as Spark Structured Streaming have been added into the scope of the benchmark.

The overall design idea of the Yahoo! Streaming Benchmark is to simulate a production-level environment. Thus the benchmark builds a complete pipeline as a simple advertisement application. The benchmark generated streaming data, leveraged Kafka as the event distributor, and used Zookeeper for its configuration and synchronization service. The workload is distributed onto different worker node servers and a Redis server is used to handle the real time queries.Kafka is widely used to process streaming data and it was chosen to be the input broker of this benchmark. Kafka can take input

from distributed data resources. Apache Zookeeper is used to coordinate and synchronize the distributed cluster. Redis is used as the distributed key-value database.

2) Intel HiBench: HiBench is another benchmark tool of large scale data processing which includes the streaming benchmark of Spark and Flink. Developed by Intel, the HiBench contains various workloads for Hadoop, Spark and Flink, such as wordCount, PageRank or even the K Means clustering Machine learning tasks. It contains several streaming workloads for Spark Streaming, Flink, and Storm as well.

III. Approach and Methodology

This section presents our approaches and methodologies used in the project. This project is aimed to compare the performances of streaming frameworks in real-world applications. The most common use case of streaming systems is data analysis applications. Given an example of online advertisement, the client servers emit real-time advertisement click events to the streaming system, then streaming jobs will parse and analyze the data from the events and send results to external storage or databases. The main design idea is to simulate this process in the project.

A. The Evaluation Framework

To simulate this process in the experiment, we set up a system with a data generator to produce events, a message broker to send events to the streaming framework, the streaming frameworks to process events, and a database to save process results. In Fig. 2, we present the overall framework.

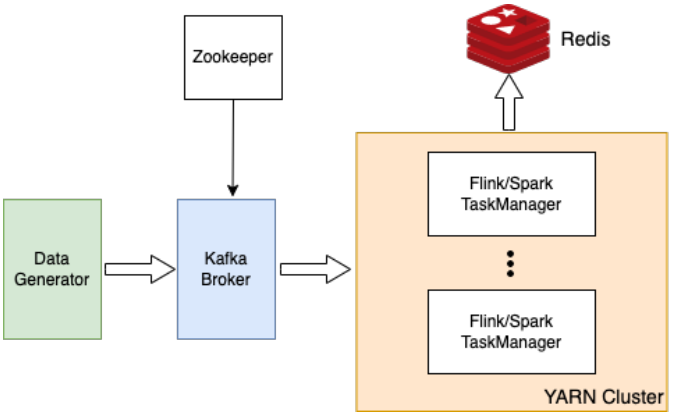

Fig. 2. Overall framework

Data generation is a vital part of our experiments. A stable and controllable upstream is essential for the accuracy of the evaluation results. Also, we hope the dataset could represent the real-world streaming data. The data generator is designed as follows: a program will generate random values using real-world data patterns. The generator has two control parameters, the generated events per second and total generating time. This approach has several benefits. The first is that the in-memory generation is easy to control generating speed and maintaining stable throughputs without being affected by file systems. Secondly, it maintains the patterns of real-world datasets.

In this project, we use Apache Kafka as a message broker. Unlike the message queues that only deliver messages once, the message brokers persistent data and makes messages available for reprocessing[14]. There are some debates about message brokers. They argue that message brokers like Kafka will slow down the streaming systems due to the repartitioning between message brokers and streaming frameworks[15]. In our case, unlike previous work[15], this project plan to test the fault-tolerance of streaming frameworks, which makes a message broker supporting replay function essential in the experiments. Besides, the repartitioning problems can be addressed by careful configuration.

We use Redis to store the final result. Redis is an open-source, in-memory data store which is often used as a database, cache, and message broker[21]. Redis is famous for its speed and is able to handle more than 400,000 requests per second on a single machine[21], which is sufficient for the experiment's requirements.

We choose the AWS platform to construct the cluster as it provides a wide selection of computation resources and rich deployment tools. The Amazon EMR is a big data tool that provides an easy setup for distributed computing environments. We use EMR to construct a cluster that uses the Hadoop framework as the basic stack, including HDFS as storage, YARN as resource management, and Zookeeper for coordination. On top of it, We deployed Apache Spark[4] and Apache Flink[5], and Apache Kafka.

B. The Streaming Task

Rather than testing a single operation like grep or word count, We choose the same streaming task from the Yahoo benchmark, which includes a series of operations and is closer to real-world applications. Those operations include[15]

1) Read an event from Kafka.
2) Deserialize the JSON string.
3) Filter out irrelevant events
4) Take a projection of the relevant fields
5) Join each event by ad_id with its associated campaign_id.
6) Take a windowed count of events per campaign and store each window in Redis along with a timestamp of the time the window was last updated in Redis.

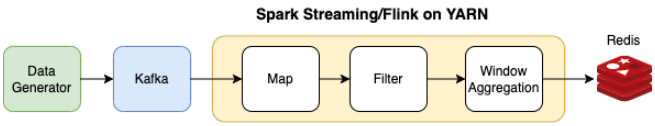

Fig. 3. Stream Task

C. The metrics

The metrics focus on the performance of streaming systems. There will be two metrics used in the evaluation: throughput and latency. Throughput describes how much data the streaming frameworks have processed in the given time. We use

events per second to evaluate the throughputs. And latency is defined by the time from the events generated to the events processed and sunk in Redis.

IV. Status

After we conducted detailed research on the pros and cons of both frameworks, we decided to set up both frameworks in the same environment and strictly control all the other variables for precise evaluation. AWS EMR service is used as the platform of our cluster deployment.

The deployment of both frameworks: Spark and Flink are deployed along with Hadoop on the AWS EC2 instances via AWS EMR service. Both frameworks have the same environment and computation power since we deployed both of them on AWS EMR with the same release version.

The Yahoo streaming benchmark is used to establish the entire experiment environment and testing pipeline.

Different experiment results are obtained with different cluster parameter settings. Some of these results fit our original expectation while others are suggesting different conclusions. We have covered the details in the following Experiment Section.

V. Experiment

A. Evaluation

To evaluate both frameworks via different aspects to find which framework is a better and certain situation and since Spark has evolved to support stream processing, we can evaluate them and compare them on the performance of stream processing, we plan to set different throughputs levels to both frameworks.

We make sure the same data source and a similar data pipeline as input are used. For large throughput, we set both frameworks continuously processing the data and conduct a chart to visualize the latency and compare the performance between large throughputs. We will record and compare the CPU/memory utilization to get rid of the effect of other variables such as data pipeline and purely look at the insight to support our observation.

For small throughput, we process the data and conduct a graph to compare the execution time for both frameworks, we will also record and compare the CPU/memory utilization to support our observation. For fault tolerance, we will simulate different failures, record and compare how both frameworks deal with failures, for example, how they perform on recovery, any data loss, and how long does those frameworks take for them to recover. [10]

B. Environment Set Up

The following sections describe our environment. Note that the word 'container' refers to the yarn container, which is an isolation process space. It is used for our parallel distributed computation. We take the 1 job manager by default and do not count it into the total number of containers in this report. For example, 8 containers means using 8 task managers plus 1 job manager.

1) Cluster Topology: We start a cluster with 3 EC2 instances, 1 master instance, and 2 worker instances. We can easily resize the worker instance number through the EMR service. The later experiments show that 3 instances are enough for our purpose. The Hadoop framework along with YARN are installed in all 3 instances. Spark and Flink are configured to use YARN as the resource provider. Data generators, Kafka, and Redis are deployed on the same machine, that is the master instance. The instance types are listed in Table I. The IO-optimized class of EC2 instances was selected for the master node with enough CPU cores and network bandwidth to support the high disk I/O and network needs for Kafka

Node	number	type	core	memory	network	storage
master	1	i3.2xlarge	8	61G	10G	SSD
worker	2	c5.xlarge	4	8G	10G	EBS

TABLE I
EC2 Instance Hardware

2) Versions Compatibility and Configuration: This project has used various software with dependencies on each other, which makes it difficult to handle version compatibility. We have encountered several incompatible failures during the environment setup. Table II lists the compatible versions we used for this project.

Name	version
Hadoop	3.2.1
Flink	1.11.0
Spark	3.0.0
Kafka	2.4.1
Redis	6.0.5
Scala	2.12.10

TABLE II
EC2 Instance Hardware

There are numerous configuration parameters for all these engines. Some tips for better performance are the Kafka partition numbers should be a multiple of the streaming framework parallelism, or it may have a data imbalance problem. Also, the parallelism of the streaming framework in single instances shouldn't be more than the number of machine cores.

C. Latency Experiment

A particular streaming framework of the benchmark would return a raw latency and an event count file for different throughput. The event count file is the count of events for different time windows. The raw latency file records the time from the last event of a time window emitted to Kafka to the moment the event is written to Redis in microseconds.

The default way to calculate the latency of the framework on processing is to use the raw latency file's value for one event, then minus the duration for each window. In our case, the latency is (raw latency – 10000) since the duration for each time window is 10s which is 10000ms. This default calculation gets rid of the time for the framework to read data from Kafka. However, this is not the actual time cost by the streaming framework to process data. This default calculation does not

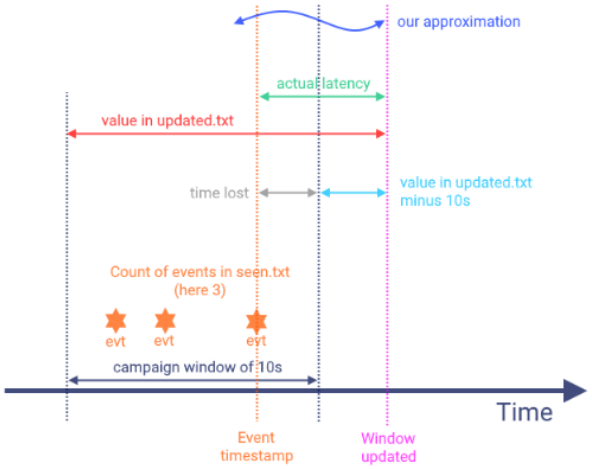

Fig. 4. Latency Calculation Explanation [19]

consider the loss of time between the last event and the end of the window[19].

Thus, we further introduce measurement to get the approximate latency for a framework to process data. We assume that events are read evenly distributed in a given time window. We then use time windows time divided by event count to get the duration of one event and then add the compensation term to the default method. The formula looks like this:

latency = (raw latency - 10000) + (10000 / event count)

The calculation still has some minor deviations because the events are not read completely evenly distributed and the interval for checking updates is one second. But it is good enough to get the approximate latency since this term is applied to all the frameworks we test. We give a view of what the latency looks like with respect to the structure of the benchmark in figure 4.

To get the latency we run the test environment for 5 minutes. First, we utilized the resources of 2 containers to process the task.

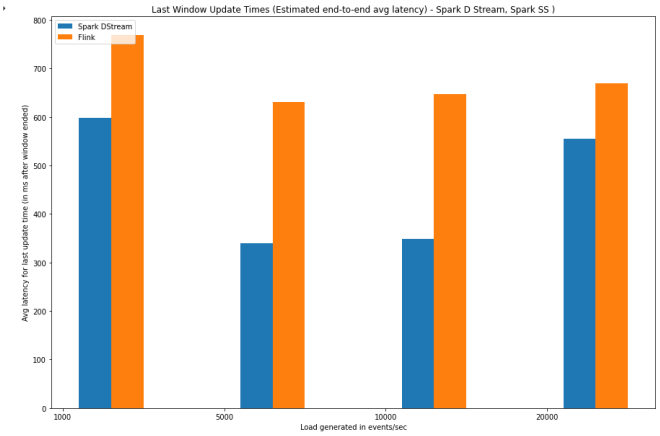

Fig. 5. Actual latency vs throughput with 7 containers

Figure 5 compares the average latency between Flink and Spark DStream at a different throughput level which is 1000 events/sec, 5000 events/sec, 10000 events/sec, and 20000 events/sec. We conducted a table for a clear view of the data.

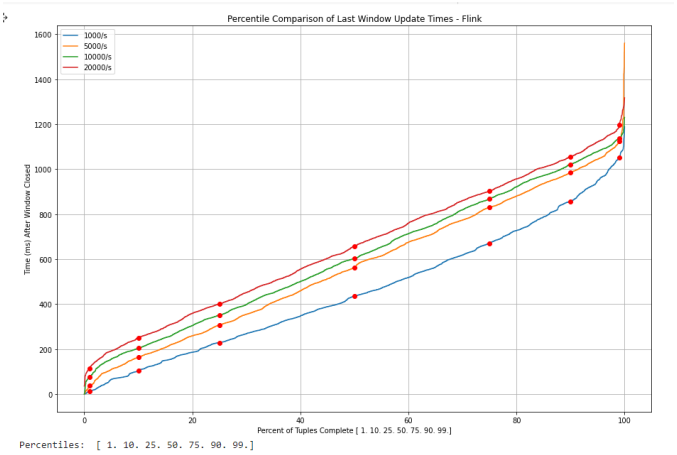

Fig. 6. Percentile of time after window close 7 containers Flink

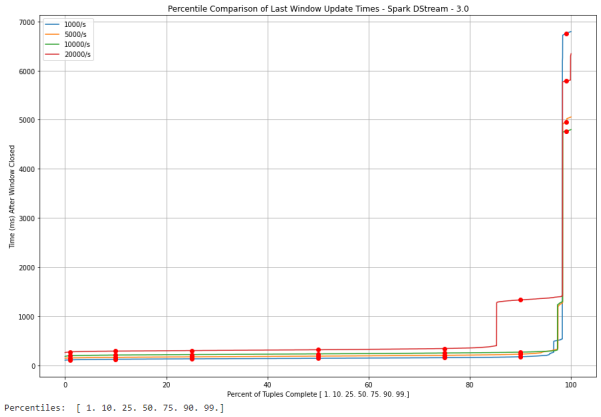

Fig. 7. Percentile of time after window close 7 containers Spark DStream

Figure 6 further proves latency at the 50th percentile (median) for different throughputs is ranged from 410 ms to 620ms for Flink. On the graph, we could see the pattern from 1st percentile to 99th percentile is increasing in a linear manner with a small slope. This gives us the incline that our result is reasonable since Flink's pure native streaming engine gives a relatively smooth line as it processes the data.

Besides, Figure 7 gives latency at the 50th percentile (median) for different throughputs ranging from 300ms to 450ms for Spark DStream. It also shows us that Spark's micro-batch streaming is using a stepwise increase. It depends on the situation if one batch is sufficient enough to process all the events [20].

Table 3 shows the detailed statistical number in the comparison. The result shows that Spark DStream has a slight advantage over Flink in this situation. But as we mentioned above, due to the error we might have in the calculation. And in fact, the result shows latency at high throughputs is close.

Throughput(events/s)/Latency(ms)	Flink	Spark DStream
1000	780	600
5000	610	320
10000	640	340
20000	650	720

TABLE III
THROUGHPUT/LATENCY OF FLINK AND SPARK DSTREAM 7 CONTAINERS

We consider that Spark DStream might have a slight advantage over Flink at low throughput.

For the purpose of validation, we conducted another experiment where we reduce the utilized resources to 2 contains but keep all the others set up as the same. In the meantime, we divide the interval of throughput in more detail to validate our hypothesis.

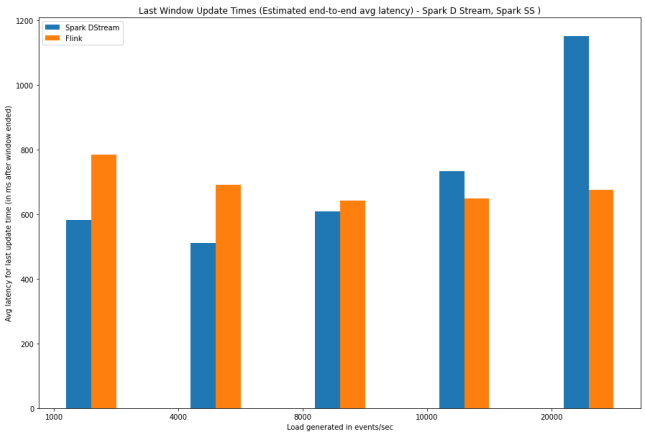

Fig. 8. Actual latency vs throughput 2 containers

Figure 8 shows the latency for Spark DStream at high throughput could not follow up, but still remains its advantage at low throughput.

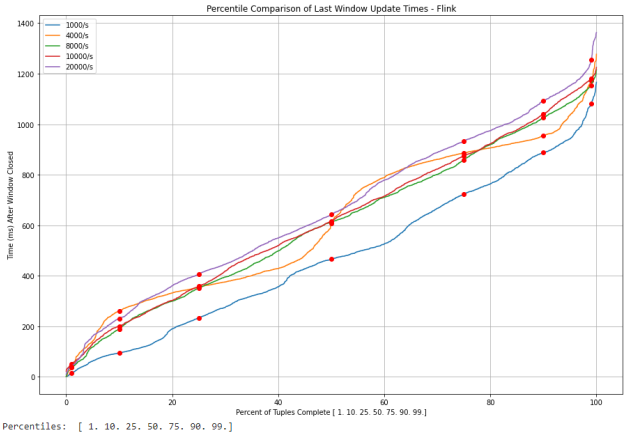

Fig. 9. Percentile of time after window close 2 containers Flink

Figure 9 and Figure 10 validate the nature of Spark and Flink streaming engines.

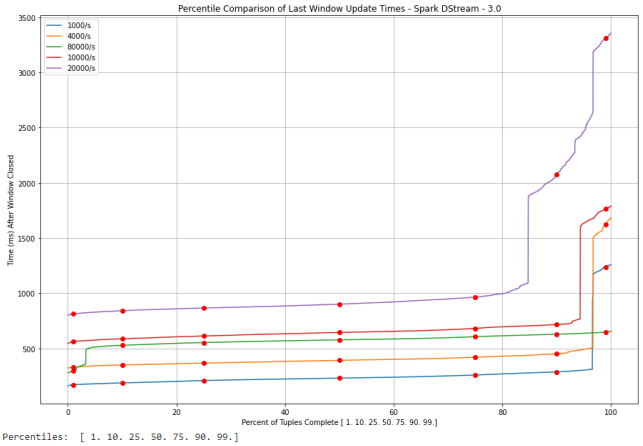

Fig. 10. Percentile of time after window close 2 containers Spark DStream

Throughput(events/s)/Latency(ms)	Flink	Spark DStream
1000	780	590
4000	700	570
8000	630	620
10000	640	720
20000	650	1180

TABLE IV
THROUGHPUT/LATENCY OF FLINK AND SPARK DSTREAM 2 CONTAINERS

Through Table 4, we found an irregular result when the throughput is 20000 events/sec. The same situation happens in Figure 6 where we found that the line is significantly higher compared to other results, especially after the 75th percentile. This gives us the doubt that Spark DStream reached its limit to effectively process the data compared to Flink. Arguments have been made that Flink might outperform Spark at high throughput. This hypothesis is made based on the nature of the streaming engine of these two different frameworks, which micro-batch has the disadvantage of processing large throughput where one batch is not enough to process all the data thus causing extra latency. A new experiment is conducted where we further control the resource distributed to each framework by reducing it to only one container. Details of this experiment will be discussed in section Throughput.

Other research works may suggest Flink's performance is better than Spark in terms of latency. The reason for this could be that other research work uses simple work such as word-Count while Yahoo Streaming Benchmark uses complex tasks. Another reason could be the usage of window aggregation, which may reduce the difference gap of latency performance of Spark and Flink.

D. Throughput Experiment

As for the performance under different throughput levels, it can be observed that Spark and Flink have demonstrated different properties. Experiments over throughput have been conducted with different experimental parameters and we have gathered different results. Overall, Flink's performance is more stable than Spark under different throughput. It can be

observed from Fig.5 and Fig.8 that the Spark's performance fluctuates heavily along with the changing throughput level while that of Flink remains relatively consistent. Moreover, as shown in the Fig.8, it can be noticed that when throughput is beyond 20000 events/sec, the latency of Spark Dstream soars. We suspect that the Spark DStream has a bottleneck in throughput in our testing environment. Therefore it demonstrates such behavior.

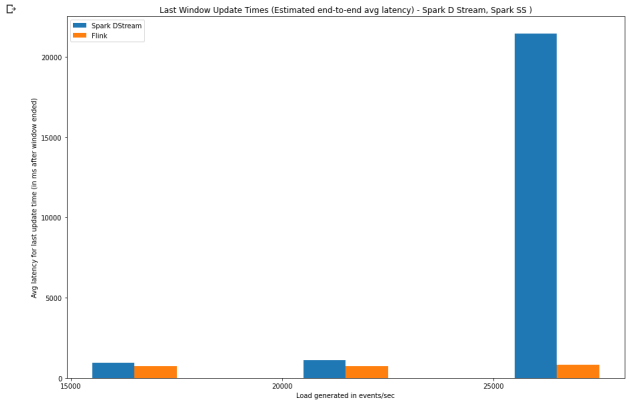

Fig. 11. Actual latency vs throughput 1 container

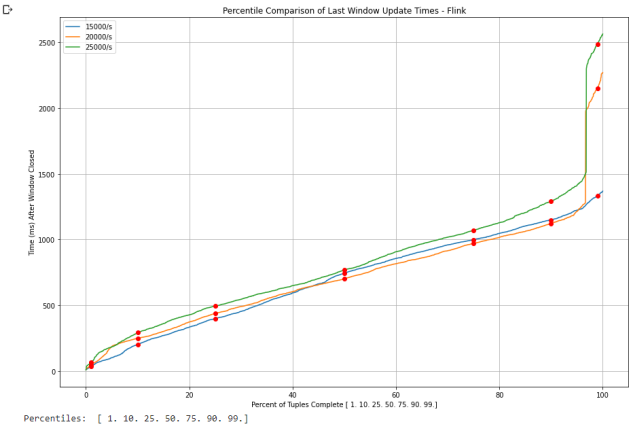

Fig. 12. Percentile of time after window close 1 container Flink

To validate this idea, another set of experiments was made with only one container with higher testing throughput level. As shown in Fig.11, after a throughput level of 25000 events/sec, the latency of Spark DStream soared dramatically. From this result, it can be certain that there is a bottleneck for Spark for high throughput. This conclusion can also be validated from Fig.12 and Fig.13. In Fig.12 Flink's performance(Latency time after window closed) has similar behavior with different throughput. However, on the contrary, in Fig.13, Spark DStream's performance under throughput of 25000 events/s (the green line) has abnormal behavior.

Additionally, it has also been tried to set throughput to 30000 events/sec for both settings. This time, the result shows that the latency at 30000 events/sec is extremely high for

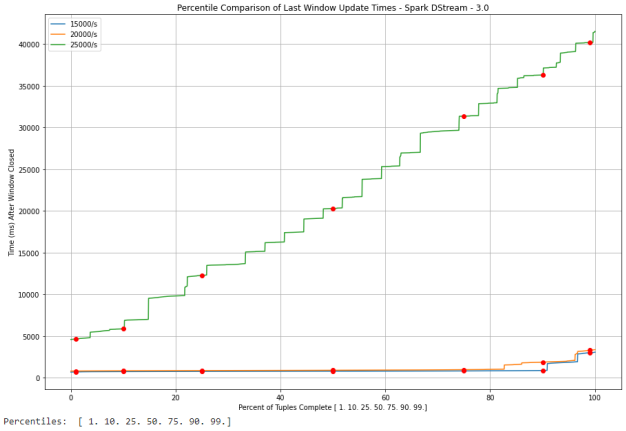

Fig. 13. Percentile of time after window close 1 container Spark DStream

both Flink and Spark. The number is around 10 times higher compared to latency at 20000 events/sec. Though further experiments have been done where we upgrade the worker , we still found that the result remains the same. Therefore, there exists such a probability that the bottleneck is caused by Kafka. Through the debug process, we found that there are backlogs in Kafka. After a series of testing, we confirmed that the limit of Kafka's throughput on a single machine is around 27000 – 30000 events/sec. Beyond this point, the bottleneck caused by Kafka will lead both frameworks to latency spikes.

E. fault tolerance experiment

We also test the fault tolerance mechanism of Spark Streaming and Flink. As our streaming frameworks are deployed on YARN, we should first introduce the YARN fault tolerance mechanism. There are several types of failure and YARN has different strategies for those failures. In brief, if ApplicationManager fails, the ResourceManager will restart it[24]. And if the containers fail, ResourceManager will report the failure to ApplicationMaster and it's up to the framework for the recovery. Therefore, to test the fault tolerance of the streaming framework, we decide to kill the container while the task is running.

Spark Streaming and Flink are both running with 1 container. Fig. 14 is the Flink throughput per second after we kill the container(TaskManager). As the figure shows, Flink takes about 65 seconds to recover from the failure. Fig.15 is the latency of Spark Streaming during the failure. Spark takes 21 seconds to recover from failure, which is only one-third of Flink's recovery time. We further analyzed the latency of Spark Streaming, which is constructed in two parts: the processing time and scheduling delay. Fig.16 shows the two kinds of latency separately. We can see that the processing time only has one spike in the moment failure happened, and then reduced to normal quickly. The total latency is mainly due to the scheduling delay. Our hypothesis is that Spark Streaming is based on micro-batch and the batch model enables Spark Streaming to recompute data quickly. So once the scheduling service recovered, the total latency fell back to normal. While

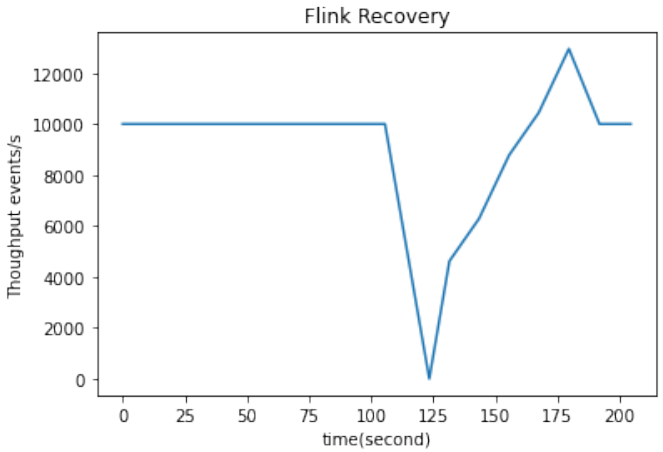

Fig. 14. Flink throughput recovery after failure

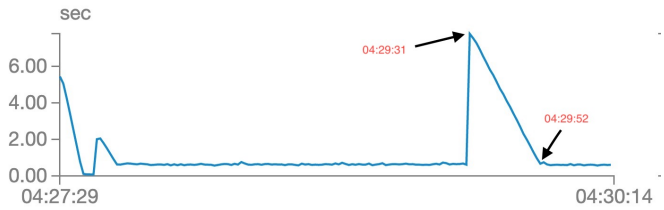

Fig. 15. Spark Streaming latency recovery after failure

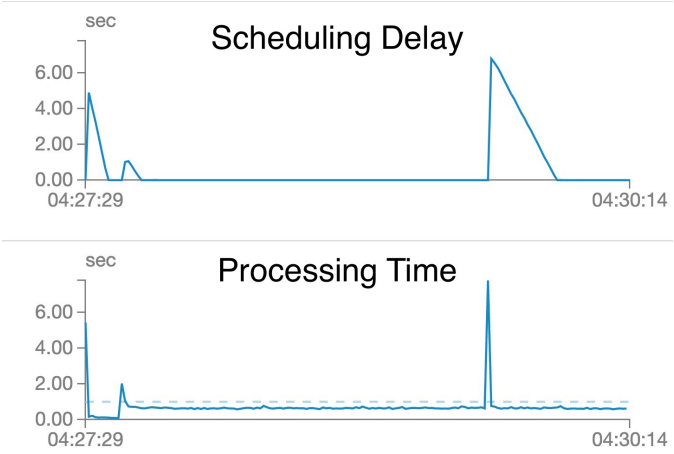

Fig. 16. Spark Streaming latency detail

Flink is based on the continuous model. As the Fig.14 shows that the throughput recovered gradually and took more time to return to normal status.

VI. Conclusion

In summary, to investigate the performance difference between Spark and Flink, we leveraged the Yahoo Streaming Benchmark to conduct experiments over the important metrics over these two frameworks. As discussed in the previous section, several conclusions have been drawn from the experiment's results on the latency,throughput, and fault tolerance. For latency, when throughput is low, Spark has slightly lower latency than Flink. When it comes to higher throughput, however, Spark has higher latency than Flink. For the throughput, we have found that overall Flink's performance is more stable than Spark and Spark DStream has bottleneck when throughput is beyond 20000/s. For fault tolerance, due to different processing and fault tolerance mechanisms, Spark has better fault tolerance ability. In our test setup, Spark is around 3 times faster to recover from failure than Flink.

In conclusion, the fault-tolerance results of our experiments match our expected outcome, but the results of latency and throughput suggest a different conclusion than our expected outcome.

VII. Future Work

Due to the limitation of Kafka, we could only conduct an accurate result below 30000 events/sec. In the future, we plan to deploy multiple Kafka nodes in a distributed manner instead of a single node of Kafka to overcome the limitation. We could then calculate the accurate throughput of each framework in a real world situation.

Spark introduced a new streaming engine called Spark Structured Streaming to replace its old DStream engine, As the newer competitor of Flink, Spark SS also supports native streaming.[22] In the future, we could add Spark SS to our evaluation list where we could compare it with its older version DStream to see what is the advantage of SS over DStream. Also, we could compare it with Flink to suggest framework choices in different situations.

In the future, we plan to test the loss of messages due to failure by manually creating a failure and testing the number of messages lost for both frameworks. This is included in the section on fault tolerance to further conduct an evaluation of the capability of both frameworks to handle failure

In the future, we plan to add storm to the evaluation list. Compared to Flink and Spark, the system only provides at-least-once semantics [23]. We could conduct experiments to compare the design trade-off of at-least-once semantic and exactly-once semantic.

References

[1] Macrometa. Sept 10 2021. Apache Spark Vs Flink Retrieved from https://www.macrometa.com/event-stream-processing/spark-vs-flinkSection-4

[2] Zaharia, Matei, et al. "Discretized streams: Fault-tolerant streaming computation at scale." Proceedings of the twenty-fourth ACM symposium on operating systems principles. 2013.

[3] Carbone, Paris Katsifodimos, Asterios Kth, † Sweden, Sics Ewen, Stephan Markl, Volker Haridi, Seif Tzoumas, Kostas. (2015). Apache Flink™: Stream and Batch Processing in a Single Engine. IEEE Data Engineering Bulletin. 38.

[4] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretizedstreams: an efficient and fault-tolerant model for stream processing on large clusters," in Presented as part of the, 2012.

[5] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, andK. Tzoumas, "Apache flink: Stream and batch processing in a singleengine," IEEE Data Engineering Bulletin, 2015.

[6] García-Gil, D., Ramírez-Gallego, S., García, S. et al. A comparison on scalability for batch big data processing on Apache Spark and Apache Flink. Big Data Anal 2, 1 (2017). https://doi.org/10.1186/s41044-016-0020-2

[7] M. Saadoon, S.H. Ab, H. Hamid, H.H.M. Sofian, Z.H. Altarturi, N. Nasuha Azizul Fault tolerance in big data storage and processing systems: A review on challenges and solutions Ain Shams Eng J (2021), 10.1016/j.asej.2021.06.024

[8] O. Marcu, A. Costan, G. Antoniu and M. S. Pérez-Hernández, "Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks," 2016 IEEE International Conference on Cluster Computing (CLUSTER), 2016, pp. 433-442, doi: 10.1109/CLUSTER.2016.22.

[9] S. Chintapalli et al., "Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming," 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016, pp. 1789-1792, doi: 10.1109/IPDPSW.2016.138.

[10] R. Lu, G. Wu, B. Xie and J. Hu, "Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks," 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, 2014, pp. 69-78, doi: 10.1109/UCC.2014.15.

[11] "Benchmarking Streaming Computation Engines at... —Yahoo Engineering," [accessed 6-January-2016]. [Online]. Available:http://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

[12] Puentes, F., Pérez-Godoy, M.D., González, P. et al. An analysis of technological frameworks for data streams. Prog Artif Intell 9, 239–261 (2020). https://doi.org/10.1007/s13748-020-00210-6

[13] J. Kreps, N. Narkhede, J. Rao et al., "Kafka: A distributed messaging system for log processing," in Proceedings of the NetDB, 2011, pp. 1–7.

[14] R. Ranjan, "Streaming big data processing in datacenter clouds," IEEE Cloud Computing, vol. 1, no. 1, pp. 78–83, 2014.

[15] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen and V. Markl, "Benchmarking Distributed Stream Data Processing Systems," 2018 IEEE 34th International Conference on Data Engineering (ICDE), 2018, pp. 1507-1518, doi: 10.1109/ICDE.2018.00169.

[16] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretizedstreams: an efficient and fault-tolerant model for stream processing on large clusters," in Presented as part of the, 2012.

[17] M. Armbrust, T. Das, Joseph Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18). Association for Computing Machinery, New York, NY, USA, 601–613. DOI:https://doi.org/10.1145/3183713.3190664

[18] S.Perera, A. Perera, K. Hakimzadeh, "Reproducible Experiments for Comparing Apache Flink and Apache Spark on Public Clouds", 2016

[19] Julian, Matschinske, Benchmarking Slang, Apache Storm and Apache Spark, Mar, 4, 2019, [Online], Available: https://bitspark.de/blog/benchmarking-slang

[20] S. Chintapalli et al., "Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming," 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016, pp. 1789-1792, doi: 10.1109/IPDPSW.2016.138.

[21] "Redis Project," [Online]. Available: https://redis.io/

[22] "Structured Streaming Programming Guide" [Online]. Available: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

[23] Toshniwal, Anki, Taneja, Siddarth,Shukla, Amit, Ramasamy, Karthik, Patel, Jignesh, Kulkarni, Sanjeev, Jackson, Jason, Gade, Krishna , Fu, Maosong , Donham, Jake , Bhagat, Nikunj , Mittal, Sailesh , Ryaboy, Dmitriy. (2014). Storm@twitter. 10.1145/2588555.2595641.

[24] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: yet another resource negotiator. In Proceedings of the 4th annual Symposium on Cloud Computing (SOCC '13). Association for Computing Machinery, New York, NY, USA, Article 5, 1–16. DOI:https://doi.org/10.1145/2523616.2523633