DDPS - Assignment #2

Pavlos Zakkas — Ivan Horokhovskyi

December 15, 2021

1 Introduction

In this assignment we were building a real-world data application. The idea was to build a distributed real-time streaming recommendation system. The application architecture is illustrated in Figure 1.

Firstly, the research question that we are interested in investigating is related to the number of Kafka and Spark nodes that should be used in the system in order to achieve a sustainable throughput. Secondly, fault tolerance of the system was inspected. Specifically, we are going to measure event and processing latency for different configurations of the system, and based on our findings we will conclude on how the relation between Kafka and Spark nodes affects the scalability of our system.

Moreover, an automatic deployment on DAS-5 cluster was implemented, but it can be easily changed for other node allocation system. Also, scripts to execute multiple experiments were made (see more in experiments section), while the code that was used to reproduce the experiments can be found on github repository.

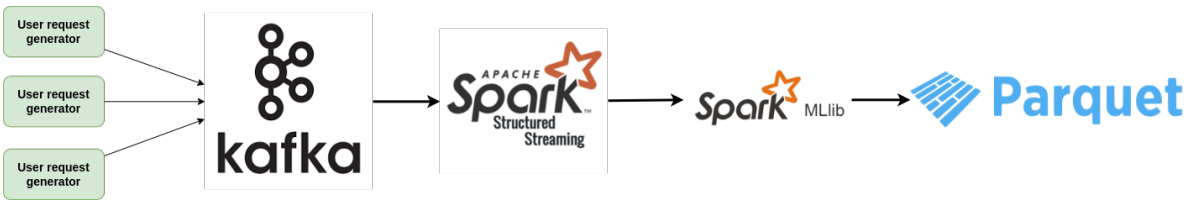

Figure 1: Dataflow schema

2 System design

The goal of the distributed recommendation system that we are going to implement is predicting the preference of a user towards a given movie. Specifically, a large load of requests will be simulated, and the system should be able to respond to this load without suffering from high latency, which can lead to a bad impact on users' experience. In the following sections we will present the metrics that we are aiming to measure, the input and output sources of our system, the design of generators, and the configuration of Kafka and Spark clusters.

2.1 Metrics

Usually in streaming systems 3 main metrics are measured namely event latency, processing latency and sustainable throughput.

Event latency - time interval between the time that the request has been generated and the time it was processed. Probably this metric is the main metric in production since it allows to measure latency from user perspective.

Processing latency - time interval between the time that the event was ingested to the system and the time it was processed.

Sustainable throughput - highest load of event traffic that a system can handle without a continuously increasing event-time latency.

2.2 Input and output sources

This system was meant to be production-ready and this is why we sometimes sacrificed speed in favor of reliability and usability. Unlike the article Benchmarking distributed stream data processing systems who claimed that using kafka and writing results to files is a bottleneck of the system, we realize that it is naive to use sockets (from fault tolerance and scalability perspective) and print results to standard output. Instead, in a real-world application results need to be stored properly, since the performed computations will be used to fulfil the business requirements of our application.

Since we do not have real workload of users and movies, we simulate the requests by creating JSON messages with the following fields: 'userId', 'movieId' and 'timestamp', which is the time that the JSON message was generated. This message flows through Kafka and reaches Spark, where the prediction of the user's preference for the given movie will be made.

Then, the results are written in apache parquet files for efficiency, while the structure of an output is shown in Table 1

Field	Data type	Description
userID	Integer	ID of a user from generator for whom we create a personal recommendation
movieID	Integer	ID of a movie from generator for which we create a personal recommendation
timestampGenerator	Timestamp	Time of creation of JSON message
timestampKafka	Timestamp	Time of receiving of a message by kafka
timestampSpark	Timestamp	Time of micro-batch query start in spark (used to measure processing time metrics)
prediction	Double	Estimated rating prediction of movieID for userID
EventLatency	Double	Event Latency
ProcessingLatency	Double	Processing Latency of Spark
ProcessedTimestamp	Timestamp	Time of micro-batch query end in spark (implemented as a Spark user defined function (UDF))

Table 1: Output parquet file schema

2.3 Data Generator

The data generator is used to simulate the user requests to the system. As we are interested in measuring performance metrics, a large load of messages will be produced, while the generation timestamp will be attached to each one ('timestampGenerator' field in Table 1). To achieve high load, we implemented the generator in such a way that a configurable number of instances can be deployed in order to produce multiple messages simultaneously. Specifically, 32 instances of generators were used, since they will be deployed on one node of DAS-5, in which 32 cores exist. Also, the messages were uniformly distributed over time and were sent via python-kafka library.

2.4 Kafka cluster

Apache Kafka is a broker-message system, in which producers send messages to specific topics and then Kafka delivers these messages to the consumers that are subscribed to such topics. Kafka can be deployed as a distributed system of multiple nodes in order to balance the load of messages. In order to investigate how the number of nodes affects the performance of the system we configured our

experiments in such a way that the number of nodes in the cluster is configurable. Thus, experiments containing from 1 up to 3 nodes will be executed. Finally, the timestamp of the moment that the message was received by Kafka was recorded ('timestampKafka' field in Table 1). The time difference between message is generated and gets to Kafka is relatively small, as values around 10^{-5} seconds were recorded.

Also, Kafka topics can be partitioned, meaning that a topic can be spread over a number of buckets located on different Kafka brokers. This is a configuration of high importance since it affects the scalability of our system. As a result, a new topic was created for each experiment and the partitions that were used per topic were equal to the number of Kafka nodes, in order to take advantage of all the nodes of the cluster.

2.5 Spark cluster

Apache Spark is a unified analytics engine for large-scale data processing. To support our recommendation system we are going to load a pre-trained model to Spark runtime and use it to give potential review of a movie for a pair of user and movie ids. Since the main scope of our experiments is measuring systems scalability metrics we have not tuned the model, as model accuracy is not one of our goals. Concerning model implementation, we used the ALS model implemented in Spark MLlib and the Movies Kaggle Dataset.

Besides prediction, which is the main and more time consuming processing step of Spark's flow, we also calculated 'ProcessedTimestamp' (see in Table 1) and event and processing latency. To measure 'ProcessedTimestamp, we used a Spark user defined function (UDF) to find the timestamp during the last moment of processing of a message. This is supposed to be the time that the processing of a user request ends. Regarding event time latency, we can subtract the 'ProcessedTimestamp' from the time that the request was generated ('timestampGenerator'). Finally, for processing time latency, we use the difference between the 'ProcessedTimestamp' and the 'timestampSpark', which is the moment when the query was ingested from Kafka system to Spark.

The results of the above measurements are stored to parquet files in disk, in order to further investigate them after the end of the experiment and perform our analysis related to the scalability of our system.

2.6 Software & Hardware

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.

The Software that was used is summarized below:

1. Python 3.9.5
   (a) kafka-python 2.0.2
   (b) numpy 1.21.2
   (c) dataclasses 0.8
2. Apache Spark 3.2
3. Apache Kafka 2.13
4. Apache Zookeeper 3.6.3
5. Scala version 2.12.15
6. OpenJDK 64-Bit Server VM, Java 1.8.0_161

3 Experiments

3.1 Setup

The purpose of our experimentation is to investigate the affect of different combinations of Kafka and Spark nodes in the performance of the system, and specifically in the event and processing latency.

After experimenting manually with several configurations, we identified specific loads of messages that can be handled by our system. Then, we configured our automated experiments in such a way that the data generator produces the same load of messages in order to compare the performance of different combinations of Kafka and Spark nodes. All seven combinations of Kafka and Spark nodes from 1 to 3 were tested ((1,1), (1,2), (2,1), (2,2), (2,3), (3,2), (3,3)). Also, after experimentation, we identified that the maximum sustainable throughput for our weakest system (1-Kafka, 1-Spark nodes) was just above 10 million messages in 10 minutes. As a result, 5 and 10 millions messages were generated in 10 minutes for all configurations of the system, while the 3-Kafka, 3-Spark nodes configuration was tested for 20 millions messages too. The different combinations that were examined did not exceed the 4 nodes per cluster, since limitations were presented during node allocation due to cluster load by other users.

Each of the aforementioned experiments was executed three times and the results were aggregated in windows of 10 seconds. For each window, the average event and processing latency of the batch was measured along with the corresponding variance which is presented as an error bar on each point of the plot. It is important to note also that the variance between different iterations of the same experiment was not significant.

3.2 Event and Processing latency

The two most important metrics of our usecase are event and processing latency, since the former is the time that the user should wait for the system's response and the latter measures the core processing time needed by Spark to make the prediction.

In all cases, the system experienced some warming up period at the beginning, in which higher latency was observed till the stabilization of the system. This trend is presented in figure 2, where we can also observe how different configurations of the system reacted during that start of the experiment. As shown, systems with only 1 Kafka node needed more time to stabilize, and even when they stabilized, the recorded latency was larger than the systems with 2 or 3 Kafka nodes. On the other hand, increasing Spark nodes lead to slightly worse event and processing latency, which can be attributed to the fact that the overhead to setup the Spark cluster for the given task increased with no significant benefit due to computational complexity is not high.

The results during the main part of the experiment, when the systems were already stabilized, are shown in figure 3 and 4 for event and processing latency respectively. In these plots, the variance of each recorded value over the 10-second window aggregation is also presented. On the left graphs we can see the performance of the system for the load of 5 million messages, while on the right the same system was tested for 10 million messages. As we can see, in all system configurations both processing and event latency for 10 million messages are slightly larger than for 5 million messages, which is expected due to higher load. The highest difference was approximately 15% and was recorded for the 1-Kafka, 1-Spark nodes system. For other systems, the difference was only 1-5%, when the load was doubled, which leads to the conclusion that systems with more nodes can handle both amounts of messages sufficiently. This observation can be further supported while examining the performance of the 3-Kafka, 3-Spark cluster for 5, 10 and 20 million messages in figure 5.

The most interesting findings though can be observed if we examine the scalability of our system by comparing in one plot the systems with different combinations of Kafka and Spark nodes. For that purpose, the figure 6 was used, in which we present the average processing and event latency during the main period of the experiment. In favor of greater clarity, we decided to remove variance of data

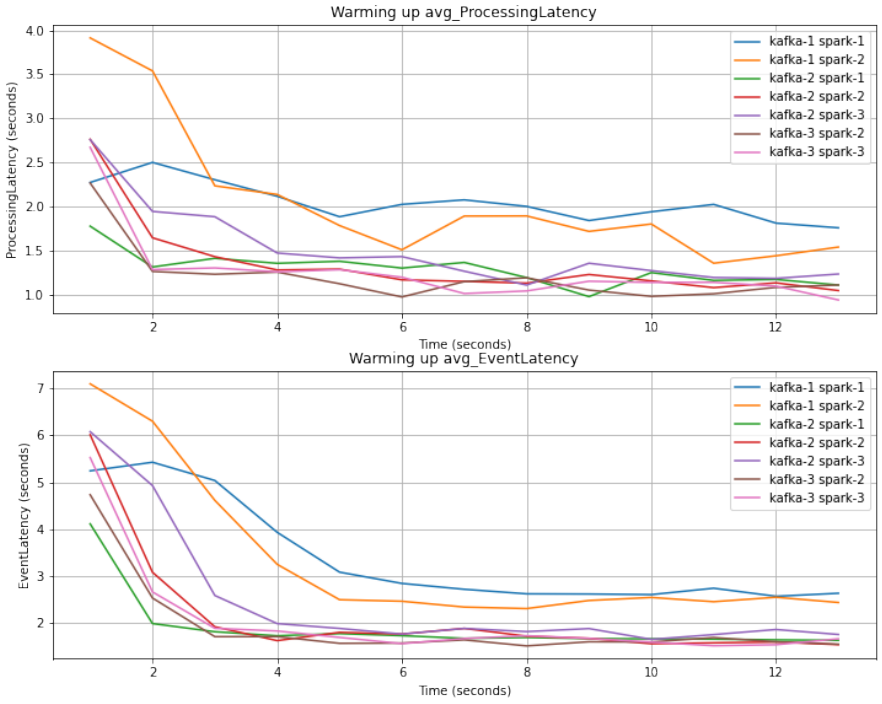

Figure 2: Event and Processing Latency during warm-up period

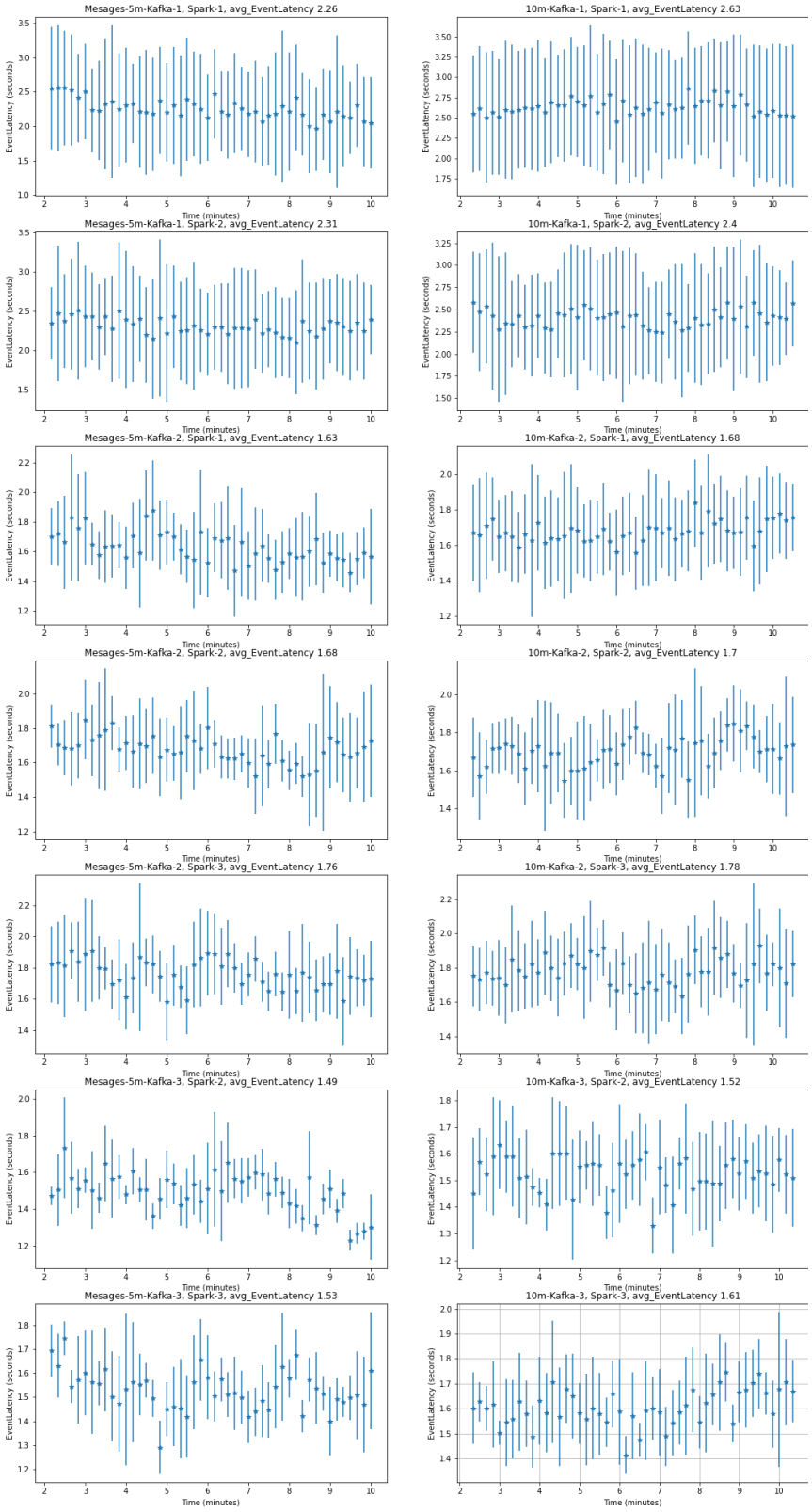

Figure 3: Event Latency per different load of messages

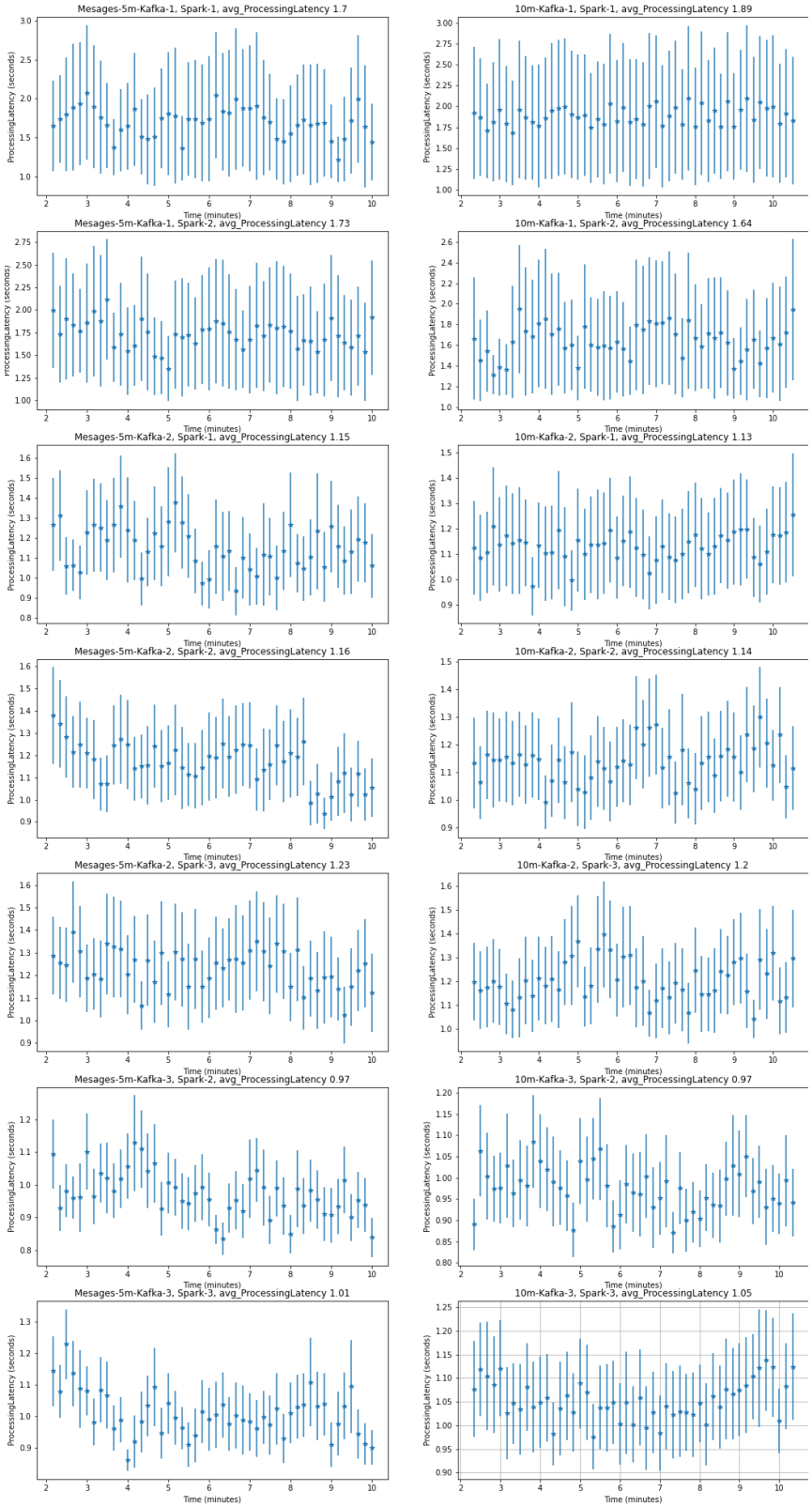

Figure 4: Processing Latency per different load of messages

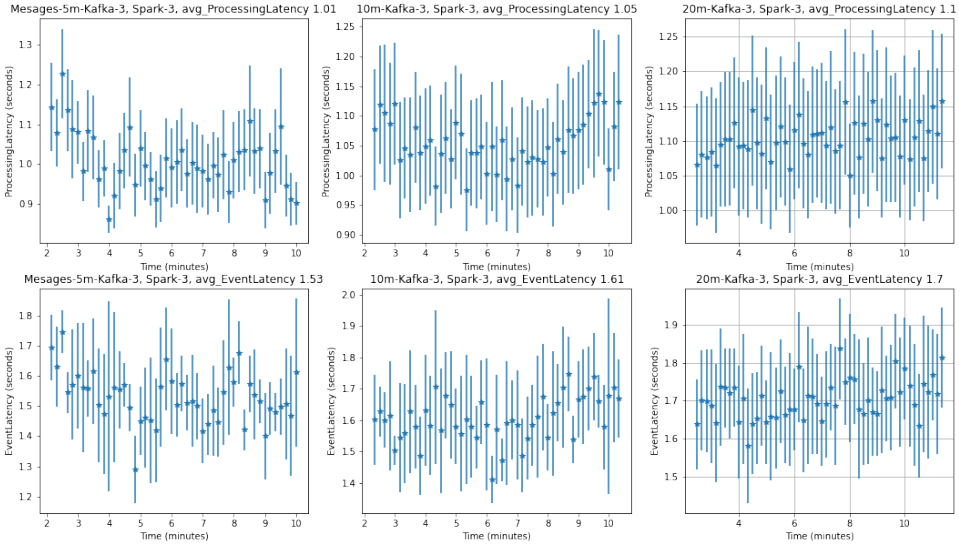

Figure 5: Event and Processing Latency for 3-Kafka, 3-Spark system

points from these plots.

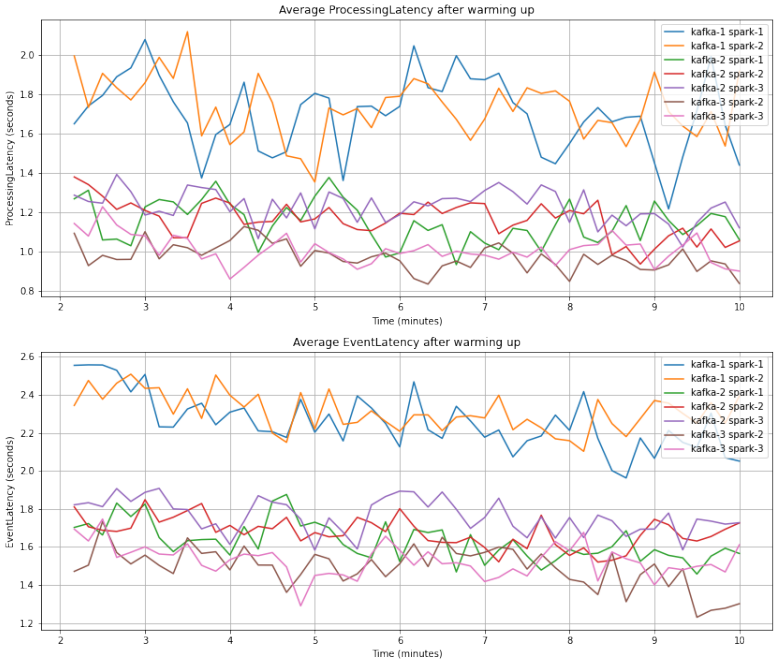

Figure 6: Processing and Event Latency for different system configurations

As observed, increasing Kafka nodes leads to a significant decrease in both processing and event latency. For instance, as depicted in figure 3, increasing Kafka nodes from 1 to 2 or from 2 to 3 leads to almost 30% and 12% decrease in average latency respectively. On the other hand, increasing Spark nodes does not benefit the system performance significantly. It is important to note that in our case, combinations of 2-Spark, 1-Kafka or 3-Spark,2-Kafka nodes perform slightly worse than combinations of 1-Kafka, 1-Spark and 2-Kafka, 2-Spark respectively. This means that while increasing Spark nodes, the communication overhead added to Spark cluster is bigger than the additional computing power that is provided. Of course, this observation is related to the type of task that Spark is requested to execute. In our case, predicting the user's review on a given movie does not seem to be a very hard task for Spark, and as a consequence increasing number of nodes does not benefit the performance. In other applications though, where the processing task is more time consuming or complex, using more Spark nodes could result in decreasing the processing latency even more.

Overall, the system that handled the loads of 5 and 10 millions messages with the least event and processing latency over time was the 3-Kafka, 2-Spark configuration, which outperformed even the 3-Kafka, 3-Spark configuration. Based on the aforementioned findings though, this result is expected, since using one more Spark node will not boost the system.

3.3 Fault tolerance

In this set of experiments we were testing system durability since both Kafka and Spark claim to have one. In this experimental setup 3 Spark and 3 Kafka nodes were used along with 20 million messages per 10 minutes load. After system starts, passes through warming up period and processes a bit more messages, we connected to one or more nodes and killed either Spark or Kafka process. Results of the experiments can be observed on figures 7 and 8 for event and processing latency respectively.

Firstly, we have tried to kill 1 of the Spark nodes. Since Spark had large margin of safety (due to not very computationally expensive tasks), the influence of this was really minor and we were not able to distinguish the node killing time from the plots after the experiment. This is why we made another experiment, in which we killed 2 out of 3 Spark nodes (we killed the first 2 nodes since during measurements we noticed that Spark loads its nodes sequentially). In this experiment processing time increased by 12%, but there was no peak noticed, so we can say that Spark did a great job stabilizing.

Secondly, we were measuring how Kafka can deal with node loss. Due to the specifications of the environment we worked with, we were not able to have more nodes and moreover usually network was the limiting factor in our experiments. We put partitions equal to the number of nodes in order to squeeze out the system as much as possible. This is why when we killed one of the Kafka nodes, Kafka cluster paused its work and was just idling till the moment the node recovered. We restarted Kafka process on the node within one minute. Overall, on the figures a massive peak can be observed, since average event and processing latency increased in 28.5 and 16.5 times respectively. In this case, the system needed almost 5 minutes to reach the latency of the system before the termination of the process.

3.4 Future work

- For the future work, experiments with more nodes can be done to see more combinations and give optimal configuration for larger cluster sizes.
- Investigate how streaming window watermarking influences the performance and do experiments with late coming events.
- Try more intensive computations, so Spark will have more load and adding more Spark nodes would be reasonable.
- Try experiments on cluster with higher network bandwidth to reduce network bottleneck and load cluster on full.

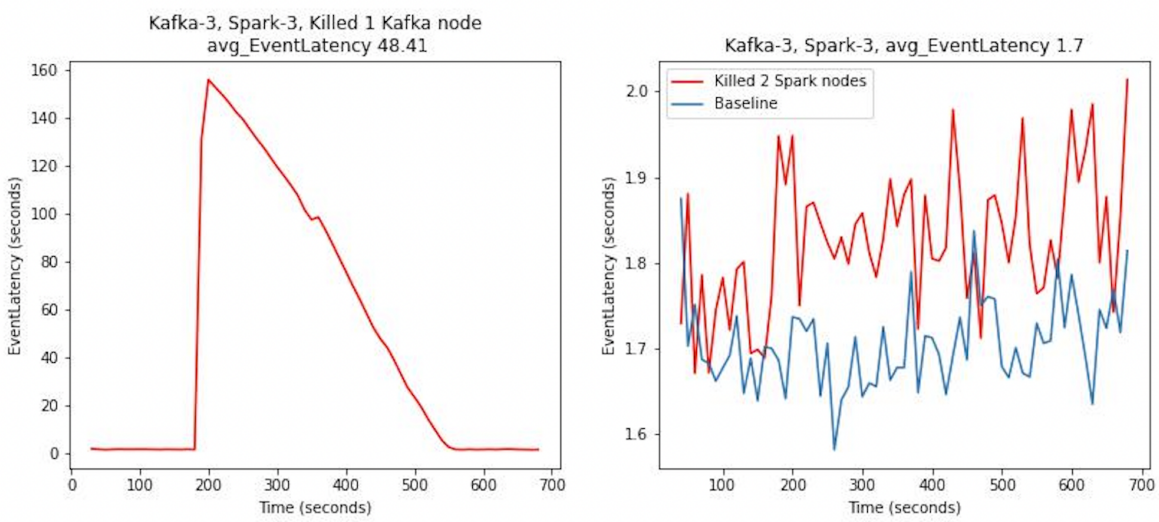

Figure 7: Event latency for fault tolerance experiment

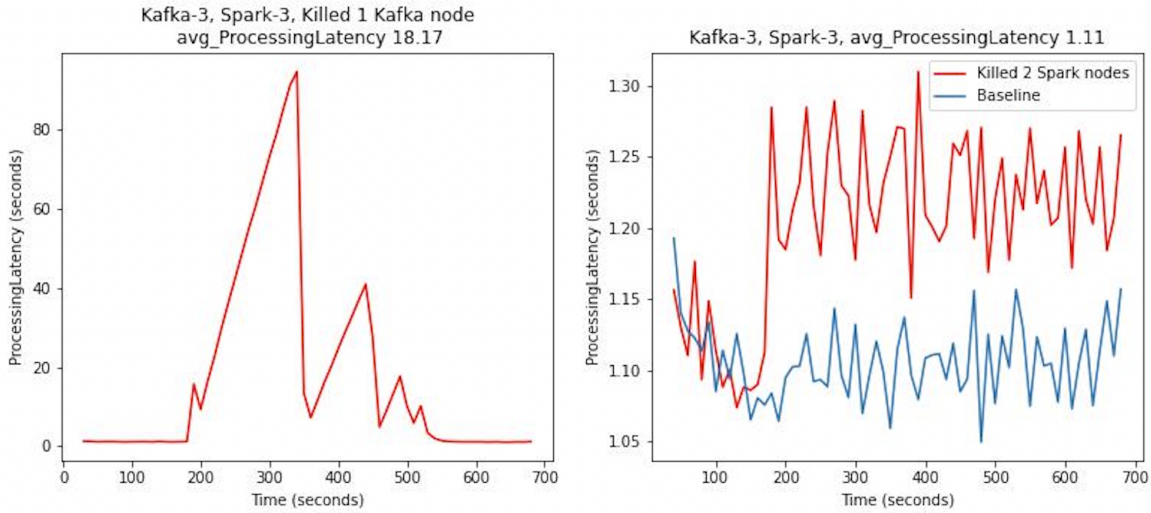

Figure 8: Processing latency for fault tolerance experiment

4 Conclusion

Overall after the experiments we realize that cluster configuration and calibration is a complicated process which requires testing to achieve best performance. Moreover, network influence can be crucial for high load systems. In addition, it is a bit counterintuitive but sometimes increasing number of nodes can slow down the computation and decrease key metrics. In our case, this could be observed when we compare clusters, where the number of Spark nodes was bigger than the number Kafka nodes. It was assumed that such behaviour can be connected with some overheads which distributed systems features which are necessary for scalability. In our case, increasing Kafka nodes lead to a significant improvement of the system's performance and specifically the optimal configuration was achieved with 3 Kafka and 2 Spark nodes. Regarding fault tolerance, Spark node termination experiment illustrates that modern distributed processing systems are durable to node failures and can handle them automatically without of major performance drop.

Section	Main contributor
Data generator	Pavlos
Deployment scripts	Pavlos
Spark	Ivan
Execution	Both
Plotting	Ivan
Analyzing	Both
Report	Both

Table 2: Work division

5 Appendix

For this assignment, we both worked on all of the sections of development and experimenting either by reviewing or implementing the needed parts.

In the Table 2 you may find a table which mentions who was the main contributor to each part of our assignment. The work was done and reviewed together and the division is kind of nominal.

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://kafka.apache.org/
4. https://zookeeper.apache.org/