UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

**Elkhan Shahverdi**

# Comparative Evaluation for the Performance of Big Stream Processing Systems

**Master's Thesis (30 ECTS)**

Supervisor: Sherif Sakr

Tartu 2018

## Comparative Evaluation for the Performance of Big Stream Processing Systems

**Abstract:**

Nowadays data is growing with tremendous acceleration, and this growing data must be processed properly if we want to have control over it. It pushes us to think about data stream processing. Most of the time, a data-intensive fraud detecting, trading, manufacturing, military and intelligence systems require processing data immediately (real-time). These kinds of systems need considerably ssophisticated pattern matching and correlations. However, other uses of stream processing have also emerged over time. In this thesis, we will benchmark to compare and contrast Apache Flink, Apache Storm, Heron, Kafka an Apache Spark stream processing engines. In these applications and domains, there is a crucial requirement to collect, process, and analyze significant streams of data to extract valuable information. This thesis aims to conduct an empirical evaluation and benchmarking of the state-of-the-art of big stream processing systems.

**Keywords:**

Stream Processing, Batch Processing, Benchmark, Apache Flink, Apache Spark, Apache Storm, Apache Heron, Apache Kafka, Kafka Stream

**CERCS:**

P170 Computer Science, Numerical Analysis, Systems, Control

## Big Stream'i Töötlemissüsteemide Toimivuse Võrdlev Hindamine

**Luhikokkuvõte**:

Andmete hulk kasvab tänapäeval meeletu kiirusega ning seda andmete hulka tuleb korrektselt töödelda, et saavutada kontroll andmete üle. Antud olukord sunnib meid mõtlema andmevoo töötlemise peale. Enamasti nõuavad andmemahuline pettuse tuvastus-, kaubandus-, tootmis-, sõjanduse ja luure süsteemid pidevat andmete analüüsi (reaalajas). Sellist tüüpi süsteemid nõuavad kõrgetasemel ist mustrite sobitamist ja korrelatsioone. Aja jooksul on ilmnenud erinevaid andmevoo töötlemise võimalusi. Antud teesis tehakse jõudlustest Apache Flink, Apache Storm, Heron, Kafka ja Apache Spark andmevoo töötlemismootoritega ning tulemusi võrreldakse ja vastandatakse omavahel. Nendes rakendustes ja domeenides on väga oluline nõue koguda, menetleda ning analüüsida olulisi andmevooge, et eraldada sealt väärtusliku informatsiooni. Antud teesi eesmärk on läbi viia empiiriline hindamine ning võrdlemine kõrgtasemel andmevoo töötlemissüsteemide vahel.

**Võtmesõnad**:

Andmevoo töötlemine, Partii töölemine , Jõudlustest, Apache Flink, Apache Spark, Apache Storm, Apache Heron, Apache Kafka, Kafka Stream

**CERCS:**

P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

## Table of Contents

# 1 Introduction

New technologies are changing the world faster than one can imagine. These changes can be found in every field, and each of them brings tons of new data to us and most of the time the information we retrieve is not self-explanatory. At this point, the importance of Big Data and its process in a productive way emerge. It is not a coincidence that data is defined as 'the petrol of the economy' nowadays. Thus, big companies and institutions are very interested in investigating on Big Data, and they all realize that whoever takes the lead in this sector, will have a significant role in the market control.

Big Data challenges include capturing data, data storage, data analysis, search, sharing, transfer, visualization, querying, updating, information privacy and data source. To ease the management of Big Data challenges are mostly grouped in two main aspects by engineers: Data Storage and Data Process. In this master thesis, we will mainly concentrate on Data Process. A proper Data Process management can optimize the amount of the data to be stored as well, by reducing repeated or unnecessary information. However, today's good algorithms or methods for Data Process can easily be stale tomorrow.

Before [1] getting started with Big Data Process, it is essential to have a look at the so-called '3V of Big Data': Volume, Velocity, Variety. Volume problem can be easily understood by the fact that 9/10 part of the data currently existing on the Internet, was created in the last two and half years [2, 3]. Velocity is mostly about the amount of data going through the Internet in a single time unit. Because of the inevitable increment of internet users, we know that this number increases radically every day. Regarding Variety, we must consider that there are billions of video, audio and text files generated by different devices, and even each kind may contain variations in the file formats.

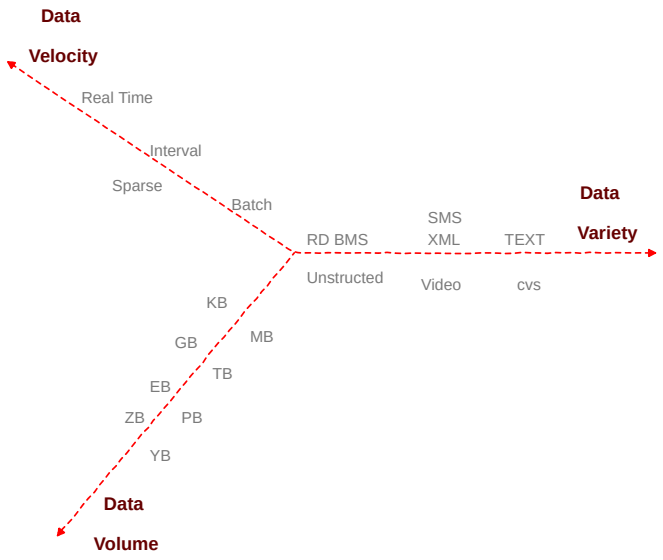



Figure 1. Big Data 3Vs

With this research document, we will try to provide some information about the Data Process on different frameworks. In the following sections of this document, you will find information about the purpose of the benchmark, how it is exactly done, previous works in this specific topic, as well as some useful results regarding choosing the adequate framework for users' needs.

## 1.1 Scope

As we already mentioned, Data Process contains various steps. In this master thesis, we will mostly focus on Batch and Stream Processes and behaviors of different frameworks regarding them. To obtain more meaningful results, we chose the most well-known frameworks such as Flink, Spark, Storm, Heron, and Kafka. The streaming benchmark simulated an advertisement analytics pipeline, and the job of the benchmark was to read advertising log events and process them in the shortest potential time. The same amount of data- from 10K to 100K advertisement log events per second have been utilized for all frameworks, and results were evaluated concerning latency, throughput, and resource consumption. All the implementation process was done on CPU optimized Ubuntu[1] running servers of DigitalOcean[2].

We ran a specific algorithm to implement streaming machine learning algorithms such as regression, classification, and clustering on Big Stream processing environments. The data that was used for this research was similar to real-world advertisement log events, and the amount of the data was close to our servers' capacity. Although we acknowledge that another implementation of this benchmark with a more extensive dataset could provide more trustful results, we are confident that all the tests we have accomplished gave us a bright idea about the performance of the mentioned frameworks.

## 1.2 Motivation

In the Big Data world, what is important is not the amount of the data are worked with, but how that data is handled. Processing Data means manipulating all of it in a way to produce useful information depending on own purposes. Combining big data with high-powered analyses, business-related tasks such as customer-based content management, risk management, fraud and real-time failure detection, etc. can efficiently be accomplished. Thus, the most challenging and critical part is not just about if you can extract the information that your analytics will use, also if you do it efficiently. The term 'Derived Data' is what you will have at the end of the day [4].

There are many derived data processing types. Batch Processing and Stream Processing are the most valuable types, and our benchmark is focused on them. Below in Figure 2, you can find brief definitions of the two processes that we are going to talk about:

[1] https://www.ubuntu.com/
[2] https://www.digitalocean.com/

**Batch Processing**: Batch processing loads a significant amount of data, runs specific jobs and algorithms on it, generates output data. The most relevant performance measures are the throughput and latency. Batch Process is useful for long-term strategies, and the process itself requires a considerable amount of time comparing with stream processing. Since this process is based on one (or several but not too many) huge file/record, the need for Map Reducer tools is expected.

**Stream Processing**: Stream processing is an online data [4] processing which runs jobs on flooding data. The stream processors consume the input and produce an output. Furthermore, the analyses are done for each event or small event groups in real time. Due to the I/O bottlenecks of Distributed File Systems, Map Reduce is not preferred in this process. Moreover, since records per unit are not big at all, you will most likely want to keep them in memory. Running micro/macro batch processes during stream processing is also probable.

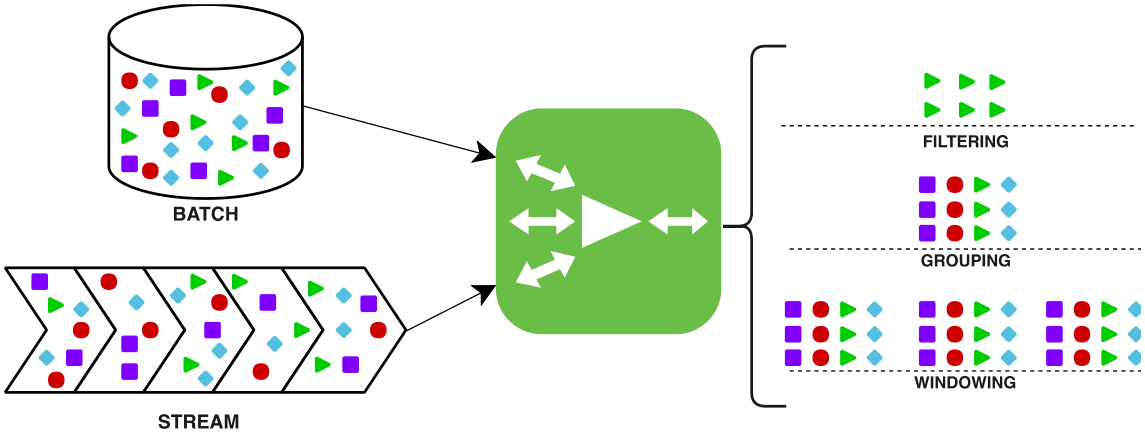



Figure 2. Stream and Batch Processing

Existing engines behave differently in each step of the data processing because of the way they are designed. To choose the most convenient framework; the user should define the problem clearly and consider the engines' performance in that specific case. With this paper, we aim to show the performance evaluation of previously mentioned engines based on three main characteristics of Big Data: Volume, Velocity, and Variety. We hope that after reading our research paper, readers will get some ideas about Big Data Streaming process, the main features of each engine that have been used, and finally their more and less powerful sides.

## 1.3 Research Problem

The main purpose of this master thesis is to find the proper engines for users' needs. To achieve it, we had to have an idea about behaves of different frameworks, how they are affected by their configuration, which of '3V' aspects they have a better performance, etc. To put all these notions together and also have a structured research path, we came up with the following questions which we believe that describes the principal goal of this thesis:

1) Which Big Data Benchmarking standard is more suitable for Big Data Stream Processing?
2) What is the most effective configuration of Apache Flink, Spark, Storm, and Kafka?
3) What is the volume, velocity and variety capacity on fixed clustered implementation of these stream processor?
4) Which stream processors are more efficient for clustered deployment?

We hope readers will find answers to these questions after studying this paper. However, in future works part, you can see that we are not going to stop searching and we will always try to reach the most satisfying answers for Big Data Process users.

## 1.4 Structure

This document contains six chapters. First of all, we begin with giving some general formation about the scope and the purpose of this master thesis, and the methods we have followed to achieve the final results. In the following section, we provide the necessary background knowledge every reader should have before reading this document. Most of them are about the frameworks and the environment that was used during the research. Even though we believe that we shared enough resources for the users to understand the concept of each framework, it is not reasonable to expect that the readers will become an expert on these technologies.

Later on, in the third section, a reader can find previous works related to our research area. Although the benchmarks are not the same, just close to each other, those papers and research works were very enlightening for us, and we think they can be very useful for the reader as well. Contribution section is where we talk about our work and benchmark, its environment, implementation, configuration and management of each framework for our requirements, and other small improvements we have accomplished. Since the frameworks are kind of similar at some point and share identical terminology, to avoid confusions, we separated that section into subsections. In Section 5.2, we evaluate the results obtained from the benchmarking and efficiently share them with the reader. We discuss the results and interpret the possible reasons for the expected and unexpected figures at the end of the research.

Finally, we conclude the achieved results, gains and the aim of this master thesis, and add some words about future works can be done in the related area.

# 2 Background

Before getting started with research details, we would like to let readers fulfill their knowledge about the technology we have used. In this section, first we will provide some middle-level terminologies that have been mentioned in this paper, and then we will talk about the technologies which were mostly used to achieve our goals.

## 2.1 Streaming

Nowadays, everything we see on the internet or in the technology world is data, and that data is being transmitted from one point to another one without cease. Transactions we do through our bank's mobile app, 'likes' and 'comments' of our social media accounts, messages and millions of other things are examples of the data being generated as a stream. There are two types of streaming; Bounded and Unbounded. An unbounded stream is a stream, which never ends (or we do not know when it will), and event data is being processed continuously, which means future events are not important for the events that are being handled now. In this case, all events are similar, and the only way to differentiate them is their creation or received time. In contrast, start and end times are defined in bounded streams. Data is handled in batches, and thus, this kind of streams are also called batch processing. Below. in Figure 3, you can find an illustration of two mentioned types of streams:

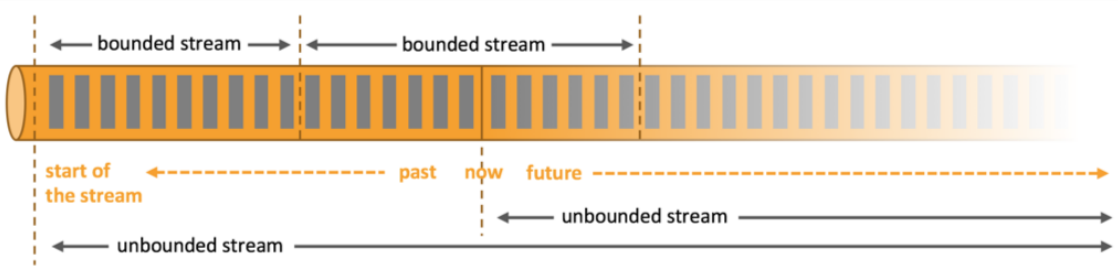



Figure 3. Streaming Types [5]

### 2.1.1 Use case

Users can easily develop and run diverse types of applications with stream processors. We can group those applications into three main categories.

First of them would be event-driven applications which are considered as a derivative of traditional transactional applications. The main difference between these two is that event-drive applications have separated compute and storage layers, and they also have forever running event listeners. Instead of connecting to a remote Database, this kind of applications use their local data which causes a better performance regarding throughput and latency. Social media websites, business process monitoring systems can be good examples of event-driven applications. The bottleneck of this kind of applications is how time and state are handled by the stream processors. Streaming engines provide event-

time, customizable window logic, as well as save points. Savepoints are externally stored checkpoints that are used to stop, resume, and update the application.

The second type of applications is mostly used is Data-Analytics Applications. These applications' job is to extract or produce useful information from the stream's raw data. Most of the time results are emitted as reports or written to a storage system. Streaming analytical applications have more advantages over batch analytical applications. Customer interaction, Internet/Web Search applications are some examples of Data Analytics Applications.

Data Pipeline Applications is the last type we are going to talk about the use of Streaming. In this case, applications are responsible for transforming, enrich and re-locate it from one storage to another one. They must be able to read continuously read the date from a source and move it to the destination storage with the possible latency. Data Pipeline Applications provides larger use cases and more useful data than other applications. The most of stream engines have a SQL Interface for this kind of applications which also supports user-defined functions. Most of the e-commerce applications are Data Pipeline Applications.

## 2.2 Apache Flink

Apache Flink is one of the most popular distributed processing engines which can be run in all common cluster environments. What makes Flink so famous is that it is computation speed is very close to memory-speed [5]. It is an open-source software which the license is held by 'The Apache Software Foundation[3].' Flink is a powerful alternate of MapReduce and is very well paired with HDFS[4]. Applications which are meant to use Flink Streaming can be programmed with Java[5], Scala[6], Python[7] via using DataStream or DataSet APIs.

### 2.2.1 Architecture

Flink has a famous master and slaves' structure where the master is the center of the Flink's component stack. Master is the owner of JobManager which comes up when Flink file system is started. JobManager is the coordinator, and it controls the data flow which is used by one or more TaskManagers on slaves.

Apache Flink clustered [5], distributed and fault-tolerate infrastructure implemented minimum three different type of processes.

*The Client:* The Client transforms program code to a data flow graph and submits it to the Job Manager

[3] https://www.apache.org/
[4] https://hadoop.apache.org/
[5] https://java.com/
[6] https://www.scala-lang.org/
[7] https://www.python.org/

*Job Manager:* Job Manager is the coordinator of the distributed execution.

*Task Manager*: Task Manager executes operators that produce streams, deliver their status to Job Manager, and exchange the data streams between operators.

Flink is also a layered system, and its architecture contains various components which are built on top of each other. For example, the runtime layer is responsible for receiving JobGraphs which is a generic parallel data flow with arbitrary tasks that consume and produce data streams [5]. JobGraphs are generated in the API layer and are executed according to available deployment options. The following figure which can be found in Flink's official documentation [5] understandably illustrates the components:

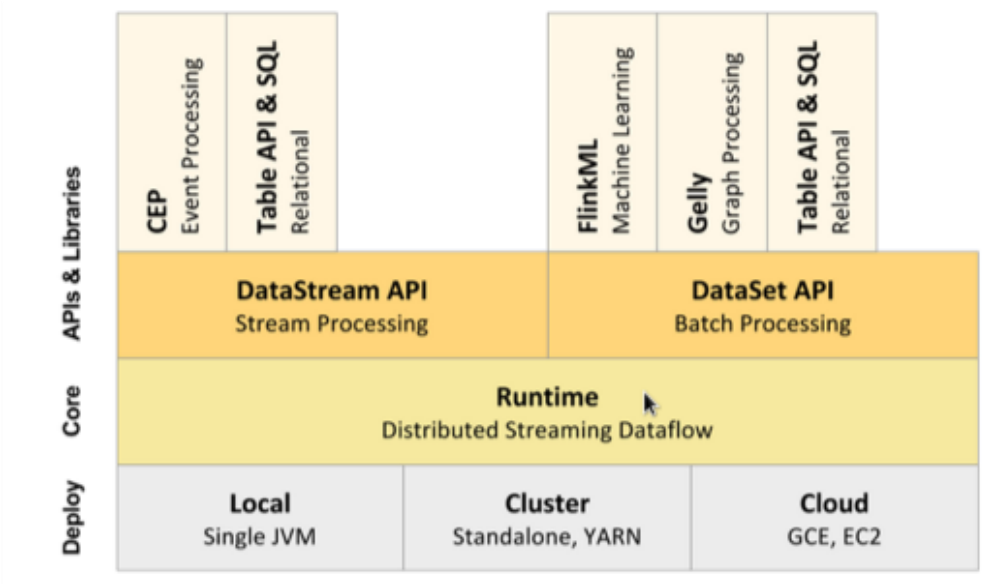



Figure 4. Flink Architecture [5]

### 2.2.2 Flink APIs

Most of the developer-friendly features of Flink come with its APIs. As a layer, APIs are on top of the Core tier [5].

**DataSet API**: Processes text or CSV files that have been generated on different sources, as well as the data that retrieved from a local collection, and lets the user do several operations on it, such as mapping, filtering, joining and grouping. Mostly used for distributed tasks and runs the batch process in streaming runtime.

**DataStream API**: To achieve real-time batch data processing, does the filtering, updates, defining windows, joins, etc. Can receive any kind of data from message queues, sockets, file systems.

**Table API**: The main concept of Table is to let the user write simple SQL queries in high layers of Flink instead of complex SQL Queries to process the data. Tables can be created with DataStream and DataSet APIs by using Table Environments. Registered tables can be retrieved by simple SQL queries.

**Gelly**: Is a motor which lets users create, transform and process graphs. Graphs are represented by DataSets. Those DataSets are made of vertices and edges. The API itself contains several functions as well as graph algorithms, and also supports iterative processing.

**FlinkML**: Machine Learning library for Flink users. It currently contains Supervised Learning algorithms such as SVM, Multiple Linear Regression, Optimization Framework, as well as k-Nearest Neighbours join from Unsupervised Learning.

**FlinkCEP**: Complex Event Processing library on top of Flink. It lets users catch event patterns within unbounded streams, as well as filtering and combining them.

## 2.3 Apache Storm

Storm [6, 7]is a distributed, fault-tolerance computing system supporting streaming data processing. By replaying data that wasn't successfully processed previously, Storm provides guaranteed data processing. The main difference of this engine is that it runs own 'topologies' instead of traditional MapReduce tasks. We talk about topologies in the next section. The Storm is scalable, compatible with many programming languages, no data loss, and noticeable fast for processing large data sets. Twitter, Yahoo, Spotify, Yelp are some of Apache Storm's famous users.

### 2.3.1 Architecture

When talking about Storm's architecture [6], first of all, we should mention that although it is very similar to Hadoop's, they are not the same. Instead of jobs of Hadoop, there are topologies in Storm which run forever (until killed by a user) to process messages continuously, unlikely the Hadoop worker tasks.

Storm [6, 8] also has master-worker node structure, where the master is called Nimbus and workers are called Supervisors. The master node is responsible for assigning tasks to different machines, codes amongst clusters, as well as monitoring failures. To monitor the message processing tasks, Nimbus uses Apache ZooKeeper services. Workers of Storm clusters are used to run daemons called Supervisors. These nodes listen to Nimbus's messaged to assign a job to own machine or to stop them if necessary. All the messaging between Nimbus and Supervisors are handled through Apache ZooKeeper cluster.

Another important point of Storm architecture is about its topology. A topology is what has to be created for real-time computation. The processing logic of topologies are called 'Bolt's which receive data from 'Spouts' that are entry points of the topologies:

Spouts are responsible for reading data as tuples, from different storages such as databases, distributed file systems, messaging frameworks and emit it to Bolts for actual runs. Depending on the ability to replay the data, Spouts are classified into two groups: Reliable and Unreliable. In the first case, when there is a failure in the process of the data, tuples are recovered from the source and processed again.

All the real job including filtering, aggregation and joins, inside topologies are done on Bolts. Generally, in cases when a topology has to do complex work, it is divided into multiple Bolts which communicates amongst each other as shown in Figure 5.

In Storm, topologies are always submitted to clusters and run inside them. For running topologies, there are mainly three types of entities:

*Worker Process*: Belongs to a specific topology and runs executors inside its topology. In most cases, one topology contains more than one worker process.

*Executor*: Is a thread that has been generated by s Worker Process. Executor processes run tasks for Bolts and Spouts.

*Task*: Is the entity which processes data and is created by executors. Thus, in Storm applications, a number of tasks are always equal (by default one task per executor) or greater than the number of executors.

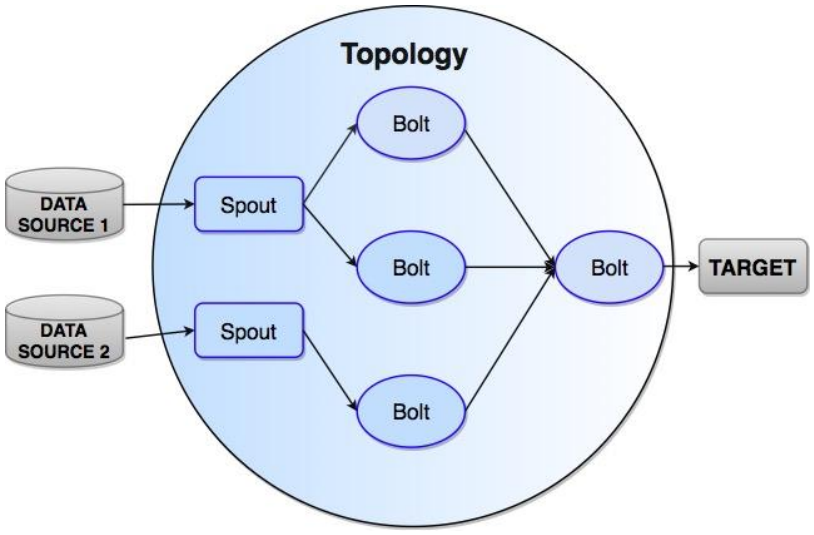



Figure 5. Storm Architecture [6]

## 2.4 Apache Spark

Spark is one of the biggest alternatives to Hadoop, and there are quite big communities which prefers Spark over Hadoop. Spark is open-source, and it is developed in Scala. When talking about this framework, the first thing to mention is that coding with Spark is a way easier than comparing other frameworks we have talked about. Besides development, the runtime of Spark is quite fast as well. On its website [9], developers claim that sometimes it is 100 times faster than Hadoop's MapReduce regarding memory processes. This lets us say that users can use Hadoop (HDFS) as storage of old data but processing them via Spark will be easier and faster. Overall, we can say that Spark's simple programming model captures batch, streaming and interactive workloads.

### 2.4.1 Programming Model

Another advantage of Spark is that you can use it with traditional programming languages such as Scala, Java, and Python. If you have already existing project, transforming it into a Spark [10, 9] runnable doesn't require too much work-around either. You only have to decide which parts must be parallelized and apply the logic only to this part. Spark also has some libraries which are frequently used by developers. The key point of Spark programming is Resilient Distributed Dataset (RDD). RDDs are fault tolerant and partitioned across a cluster, and that is why they can be handled in parallel. While programming, users can create RDDs using some operations which are called transformations, and later they will contain a collection of objects. Below you can find a piece of code to estimate Pi value by 'throwing darts' method implemented in Python using Spark:

In this example, RDD will contain a number range and then will filter them by checking if they are inside the circle or no. Here, as we mentioned before, no calculation will be done until the 'count' action is called. Once it is called, RDD will be created, filtered, and 'count' action will be performed.

RDDs are lazily evaluated, and it lets Spark to find an efficient plan for computations. Since results of RDD operations are RDDs too, these transformations are not computed immediately. Instead, when an action is being performed, Spark checks all the transformations introduced and creates an optimized execution plan which sometimes builds up better modularity than the programmer thought of. The execution is performed only once for the whole graph of transformations. It is worthy to emphasize that RDDs shares the data amongst computation nodes and they are only called when there is an action taking place. However, as programmer's wish, RDDs can be persisted in the memory for rapid use (if data is too big for the memory, Spark will locate it on the disk as well).

```
def inside(p):
    x, y = random.random(), random.random()
    return x*x + y*y < 1

count = sc.parallelize(xrange(0, NUM_SAMPLES)) \
            .filter(inside).count()
print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)
```

Figure 6. Spark Programming Model [9]

Besides sharing and parallelizing data options, another powerful point of RDDs is automatically recovering from failures. Instead of the traditional way of fault tolerance where computing systems had replication or checkpoints, Spark provides a different approach- Lineage [10, 9]. This process is done by saving a track of transformation graphs and rerunning these operations on base data. This strategy is more efficient regarding running time and storage, in data-intensive workloads. The reason is very clear, writing data into RAM is significantly faster than the writing it over the network, and the recovery process is done in parallel on different nodes.

### 2.4.2 Spark Streaming

We know that modern distributed stream processors take three sequential steps for execution. First receiving the data, then process it, and finally emit the output. In the case of Spark Streaming, it is a bit different. Instead of retrieving the data one by one, Spark splits it into batches (RDDs). In other words, receivers accept the data in parallel and locate it in the nodes. Then Spark assigned tasks dynamically to these nodes depending on the required data of each task. It allows applications to perform better load-balancing and fast fault-recovery.

In practice, regarding throughput, Apache Spark has noticeable higher performance compared with other frameworks. Talking about the latency, Spark's speed is at a few hundreds of milliseconds which is quite low and does not make Spark less-used as a batch processor over end-to-end processors.

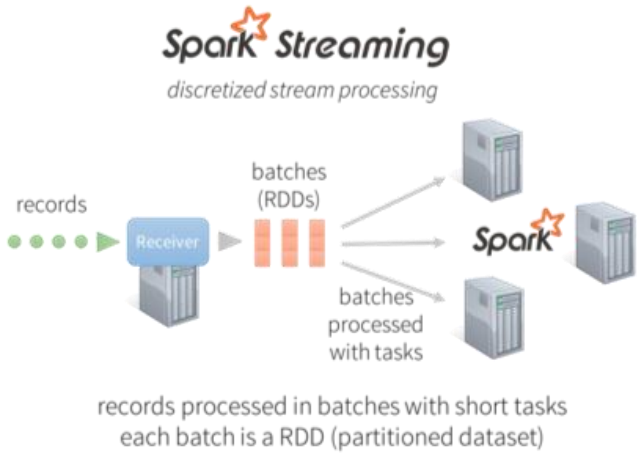



Figure 7. Spark Streaming [25]

**DStream API** is a Spark abstraction over RDDs. When we check beyond of the Dstream, we will definitely see DStream is the sequence of some amount of RDDs. As you see it from Figure 7, RDDs in a DStream contains data a given batch interval. An operation which is applying to DStream same time applying to sub RDDs in DStream. This applied RDD transformation handling by Spark Engine. That is why we can tell that DStream is Spark abstraction over RDDs.

### 2.4.3 Spark Structured Streaming

Structured Streaming [9] is the new high-level API in Spark Engine which started implement from version 2.2.0. It is using Spark SQL for processing data. It creates an opportunity for process data with a basic streaming function like a filter, group, aggregate, event-time windows and stream to batch join. Internally Structured Streaming using micro-batches for processing data like Spark Streaming. However, from version 2.3.0 Structured streaming support Continuous Processing with low latency. It means a new version of Spark is using real-time processing like a Flink, Storm, etc. You could process your data in 1ms end-to-end latency with structured streaming. Spark Structured Steaming behave stream like a

table which data appended to this table continuously. Figure 8. That is why the programming interfaces of Structured Steaming look like batch processing. You can create your streaming calculation as a batch processing, but Spark run it on continuous streaming. Spark Structured streaming provide DataFrame and Datasets API for accessing and calculation batch bounded data and streaming unbounded data.

## 2.5 Apache Heron

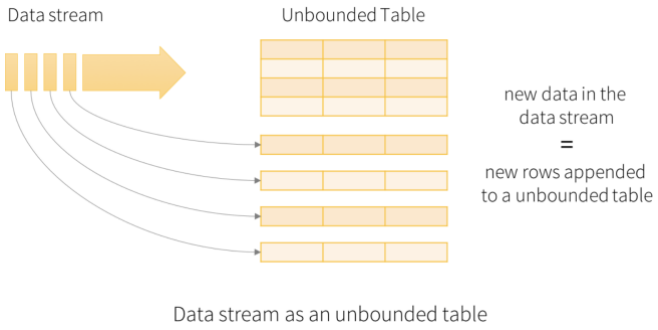



Figure 8. Structured Streaming [25]

Heron [11] is a fault-tolerance streaming engine released by Twitter[8] in 2016 and has been used actively by the providers for over three years. Different parts of Storm are written in different programming languages such as Java, Scala, Python, and C++. It is a successor to Storm; thus, users can use any API of it on Heron as well. The main difference between Heron and Storm is, on Heron, there are more functionalities that are not implemented on Storm, such as job schedulers. Same as many other frameworks, Heron was also thought to be easy-to-develop, handle a big amount of data, increase developer's productivity, and have more efficient performance.

### 2.5.1 Architecture

Developers [11] can compose DAGs of real-time query execution logic which are called topologies. Later these topologies are submitted to the job scheduling system to be executed. Same as the Storm, Heron clusters also contain Spouts and Bolts, where Spouts are connected to the data source and responsible for injecting it into the topology, and Bolts are where the data is processed. For example, in the case of Twitter's word counter in tweets, there is one Spout which receives the tweets from tweet sources. After that received tweet is transferred to the first bolt, where it is split into words. Moreover, finally, counter Bolts counts the words and returns the final number. We can say that all topology logic is very similar to the Storm's topology.

It is also possible that some Bolts inside a topology will receive more data than it can handle. For example, in our tweet counter case, the Spout can accept more data than parser Bolt can process since the latter does more computation than the Spout. For this

[8] https://about.twitter.com/en_us/company.html

kind of cases, it is possible to define parallelism capacity for each node in Heron topologies. These numbers are associated with nodes, and they specify the number of instances required the CPU in parallel. Moreover, at that point, another issue is about transferring data amongst instances. Let's say Spout knows that the data should be transferred to the next Bolt, but if it does not send the data to the proper instance, it would be chaos. Heron solves this problem with the strategy called Grouping. There are several types of grouping which are listed below:

*All*: Data is transferred from an instance to all instances of the downstream bolt.

*Shuffle*: In this case, all instances of any node, can send data to any of instances of the next node.

*Direct*: In this case, the sender decides itself to which instance the data should be transferred.

*Fields*: Decisions are based on some field values. Predefined values are hashed, and when the data is ready, specific field value 's hash is computed and sent to the downstream node's proper instance based on that hash value.

*Global*: Each instance is assigned to one another instance, and it only sends the data to it.

Heron also takes advantages of ZooKeeper State Manager for the coordination of the tasks on clusters. We talk about it in the following paragraph of this section.

Although beside being reliable Heron also has proved that it reduces the hardware resources significantly and processing latency, increase throughput, the main known disadvantage of this engine is that it is dependent on Mesos[9]. If a user does not already have a Meson infrastructure installed, it is not easy to handle this requirement. That is why it is recommended that, if you have Storm system already in use, you can easily stick to it, unless you have a huge demand as Twitter does, of stream processing.

## 2.6 Apache Kafka

Kafka [12] is also a distributed system used by many companies in production that handle petabytes of data every day because it is fault-tolerant and entirely scalable. It has been created and open-sourced by Linkedin in 2011 as a messaging queue. Since that time, Kafka evolved significantly, and nowadays it provides low-latency, high-throughput publish and subscribe pipelines. Kafka is mostly used for the applications that transform or react to the data streams, or applications that need real-time, reliable data pipelines to transfer the data between other applications.

Before diving in, it is essential to mention a few things from Kafka's architecture. First of all, same as some other engines, it is run on clusters on multiple servers, and these clusters contain records which are grouped into categories called topics. Inside topics, each record

[9] http://mesos.apache.org/

has three fields such as key, value, and timestamp. Topics usually have more than one subscriber meaning that the data written to it will be read by multiple consumers.

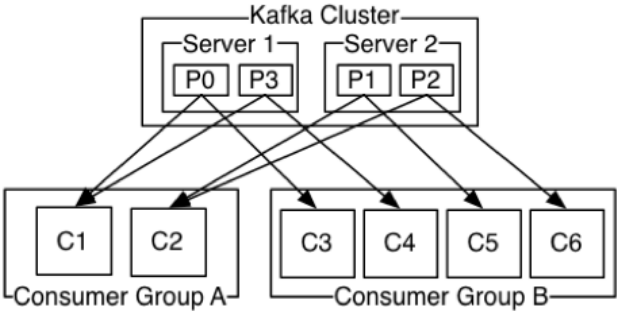



Figure 9. Kafka cluster [12]

Apache Kafka also has several APIs:

**Producer API**: Applications send (publish) data to topics in the Kafka cluster using this API.

**Consumer API**: By using this API, applications can subscribe to several topics and process the records that are produced to them.

**Streams API**: Applications can have a stream processor using this API. It receives streams of input topics and emits output topics.

**Connector API**: Allows applications to connect the data source to Kafka topics. The connection is bidirectional.

### 2.6.1 Producers and Consumers

The data on Kafka [12] topics are written and read by the producers and the consumers accordingly. Besides publishing the data on the topics depending on their choice, producers are also responsible for choosing the correct partition inside the topic. Consumers' structure is a bit more complicated than that. They can be assorted into different Consumer Groups and data from one producer is received by only one member of the group. Consumer instances do not have to be on the same server. In the case that there is only one consumer group for the whole system, then records will be handled by a different instance every time, and in this way, it will be an effective load-balancer. However, if each instance has a diverse consumer group, then each record will be transferred from all producers to all instances.

Just writing, reading, and storing data is not makes Kafka one of the most used streaming engines, but how it handles it in real-time. In Kafka world, anything from reading input and emitting an output topic is the job of the streaming processor. The basic Kafka streaming processor applications can easily be implemented only by using Producer and Consumer APIs of it. For more complicated cases, fully integrated Streams API can be used, in order to join the streams or compute aggregation of them.

Kafka is a combination of a distributed system like HDFS which allows batch processing with static files, and traditional enterprise messaging system that processes future messages you are subscribed to. Past and future data are both handled in the same way in Kafka. This combination brings low-latency and reliability which the stream processors will transform the data as it arrives.

Pros of the Apache Kafka is the:

- It is the fast, scalable partitioned, replicated messaging system which can be scale-out easily.
- It is offering high throughput and low latency for producer and consumer.
- It is supporting multi-producer and fault tolerance for consumers.
- It is store messages on disk that is why memory consumption is so less and can be useful for batch processing.

### 2.6.2 Kafka Streams

Kafka Streams [12] is one of the most powerful components of Kafka, and it is used for building applications which transforms Kafka input topics to Kafka output topics in a distributed and fault-tolerant way. There are some key characteristics that make Kafka an adequate option for stream processing applications.

First of them is its performance and power. It is highly scalable, fault-tolerant, and it supports windowing, joins and aggregation operations on event-time processing. These being said, it is necessary to mention that Kafka Streams is not a framework, but a library and that is why it does not have any external dependency and doesn’t require dedicated clustering or such a thing. For this reason, it is considered as the best alternative of Apache Storm. Although Kafka Streams is a new library comparing it with Apache Kafka itself, it has no integration problem with Kafka, nor with existing applications, and deployments can be managed without applying an artificial rule. Moreover, Kafka Streams provides low processing latency, and it never creates micro batches while processing a stream.

Furthermore, Kafka Streams offers good usability for developers. It is possible to use the library with a high-level DSL, as well as with a low-level API depending on the programmer's needs. In the first case, users can use basic operations provided by the library such as map, filter, join, etc., where they can have maximum control and more flexibility in the second case. Even beginners can easily write a basic application and run it on a single machine without installing or understanding distributed stream processing clustering.

In summary, we can say that Kafka Streams is a lightweight, real-time, scalable library that simplifies working with stream processing applications. It can easily be embedded or integrated into any application, which is more difficult with framework-based stream processing tools.

## 2.7 Hazelcast Jet

Jet is Hazelcast's [13] first and a very successful open-sourced third generation big data processing engine. It is built on top of another open-source Hazelcast product IMDG [14]. Because it is just a lightweight library, Jet can be embedded in any application to manage a data processing microservice. The library provides APIs containing several Transforms which cover some useful data operations such as filter, group, map, etc.

As some other stream processing engines, in its core Jet also uses Directed Acyclic Graphs. Nodes or vertices as it is called in the Jet system, represent computation steps. These computations can be done in parallel by more than one instances of the streaming processor. Then, vertices are connected with each other via edges. Edges represent the flow of the data, how it is routed from the source vertex to the downstream node. They are implemented in a way to buffer the data produced by an upstream node and then let the downstream vertex to pull it. It means there are always concurrent queues running amongst processor instances and they are completely wait-free.

Hazelcast Jet's first goal is to achieve high performance, and this is managed by the use of cooperative multithreading. The main idea behind it is that, instead of the Operating System, Jet engine is the one who decides how many tasks or threads to run depending on available cores during runtime. Basic processing units are called tasklets, and before they are run, their data is always available in the queue.

Currently Hazelcast Jet is available for Stream and Batch processing applications. For upcoming releases, it is expected [14] that Hazelcast will provide more features for Stream processing. Regarding connectors, for now, it only supports Hazelcast IMDG, where HDFS and Kafka libraries are being actively developed.

Table 1. Comparison of Stream Processing Systems

| Stream Processing Systems | Flink | Storm | Spark Stream | Spark Structured Stream | Kafka Stream | Heron | Hazelcast Jet |
|---|---|---|---|---|---|---|---|
| **Year** | 2015 | 2011 | 2013 | 2016 | 2016 | 2015 | 2017 |
| **Creator** | DFG | BackType | AMPLab, UC Berkeley | | Confluent | Twitter | Hazelcast |
| **Processing Model** | real-time | Real-time | Micro-batches | Real-time, micro batches | Real-time | Real-time | Real-time |
| **Programming Model** | Dataflow | DAG | Monad | DAG | DAG | DAG | DAG |
| **Stream Partitioning** | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| **Distributed Cluster** | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| **Resource Management** | Standalone, Docker, Mesos, YARN | Standalone, YARN, Mesos | Standalone, YARN, Mesos | Standalone, YARN, Mesos | Standalone, | Standalone, Aurora, Mesos, YARN | Standalone |
| **Coordination** | Built-In, Zookeeper | Zookeeper | Built-In, ZooKeeper | Built-In, ZooKeeper | Zookeeper | Local File System, Zookeeper | Built-In, Zookeeper |
| **Programming Language** | Java, Scala, Python, SQL | Java, over Thrift | Scala, Java, Python | Scala, Java, Python, R | Java, Scala | Java, Python | Java |
| **Implementation Language** | Java, Scala | Java, Clojure | Scala, Java | Scala, Java | Java, Scala | Java | Java |
| **Fault Tolerance** | Yes | Yes | Yes | Yes | Yes | Yes | Yes |

## 2.8 Apache Zookeeper

Although Apache ZooKeeper [8] was developed at Yahoo as a sub-project of Hadoop for streaming the processes of big data on clusters, nowadays it is one of the leading Apache software by own. ZooKeeper can be defined as a centralized coordination service which allows development of distributed systems. This software can be used to maintain different parts of a distributed system such as configuration and location information, synchronization, hierarchical naming, etc. ZooKeeper is compatible with Java and C by using its native interfaces, as well as there is a variety of client bindings with Python, Ruby[10] and Go[11].

ZooKeeper's namespace is very similar to standard file storage systems. Names are sequences of path values which are separated from each other by a slash ('/'). Node names are unique in this system. However, in ZooKeeper each node can contain some data associated with itself, which in this case the whole structure can be thought as a file system where directories can act as files as well. Nodes of ZooKeeper are called znodes. Data read and write operations are allowed and done automatically on znodes. These operation permissions are controlled by Access Control Lists (ACL) that is stored on the znode. Besides that, znodes also have watches which are triggered to inform a client about the changes on the znodes.

Since one of the main ideas of ZooKeeper [8] is providing an easy-to-implement interface for developers, there are only a few operations that are available:

- *Create*: creates a node in the tree
- *Delete*: deletes a node from the tree
- *Exists*: checks if a node already exists in the tree
- *Get Data*: reads the node's data
- *Set Data*: writes data to a node
- *Get children*: returns list of child nodes of a node
- *Sync*: waits for data to be reproduced

ZooKeeper is replicated, and the database that contains the entire tree data is in-memory as well. Changes and writes are saved into a disk for recoverability before they appear in the in-memory Database. Moreover, all the requests from clients are forwards to the single server which is called the leader. Rest of the servers are the followers, and they are responsible for delivering messages from the leader and agree upon message delivery. Replacement of leaders in case of failure and synchronization of followers are handled by the messaging layer.

Finally, we can say that ZooKeeper is being successfully used by many big companies, and it is known as reliable, simple, ordered and fast engine.

[10] https://www.ruby-lang.org/en/
[11] https://golang.org/

## 2.9 Redis

Redis [15] is mostly known as a Database or a Message Broker by developers, which is actually an open-source, in-memory data storage system. Many modern programming languages support Redis bindings such as Java, Python, C, Ruby, Scala, etc. Redis [15] working with data structures as lists, hashes, sets, sorted sets with range queries, hyperlogs, bitmaps, etc. Operations that are supported by Redis- intersection, union, etc., are available depending on the data type it will take place on. Redis is implemented in C, and it is available on Linux available servers, where there are also some possible ways to run it on Windows.

Redis is also most-known NoSQL Database amongst developers. Data is stored in the key-value structure on Redis, where keys are unique, and no value can be accessed without specifying its key. Regarding replication, it is possible to create a master and have several slaves on Redis system, where a slave can be master of another slave. Redis commands are considered simple, and this software is used by Microsoft in Azure[12], and it is available in the Amazon Web Services[13] portfolio. In this research, we took advantage of Redis in the enrichment process of our data, for data lookup and as well as saved the final results.

[12] https://azure.microsoft.com/
[13] https://aws.amazon.com/

# 3 Related Work

When organizations would like to implement a Big Data processing solution they have too many technology options to use which all have common functionalities. Before choosing technologies, they need to know the feature, performance, risks, and functionalities of them. These factors depend on the business case which one they would like to implement on these technologies. Choosing the right solution is the most important thing. At these moments a standardized benchmark can help them out to evaluate these technologies then can build the right solution with the adequate one. The standardized benchmarks help us to understand the performance of a particular software stack on specific hardware configurations. Academia and Big Data industry are developing new benchmark in the particular technology. Many standard performance organizations like a TPC, SPEC, SPC and specific companies IBM, Yahoo, Google, Twitter, Facebook follow the same approach for developing benchmarks. Their benchmark strategy is targeted to acceptance of their benchmark across many software and hardware vendors.

Before checking details of the benchmark, we need to understand what the primary requirement of stream processing engines is. There are so many different benchmarks, and all of them evaluate stream processing engines from various aspects. According to Stonebraker [16], there are eight different characteristics and requirements of stream processing engines:

**Keep the Data Moving**

That is the primary essential requirement of stream processing. How efficient stream engines keep data moving? How much latency they proceed? How often costly storage operation they are processing?

**Query using SQL on Streams (StreamSQL)**

The streaming engine must provide a query mechanism to retrieve data from the streaming pipeline. Most of the streaming processing engines developed in low-level programming languages. While using low-level programming languages for querying data, it makes the system more complicated and greater the high value on development and maintenance.

SQL is the most common language that for traditional DBMS. It would be better to run a query on the streaming pipeline which looks like SQL, with some kind of an API.

**Handle Stream Imperfections (Delayed, Missing and Out-of-Order Data)**

The traditional storage engines run queries on last isolated snapshot of the data, but in streaming engines, it does not work in this way. While querying data, it is possible, but in the stream, processing queries are running on flooding data, and it is possible that queries can affect the entire system.

In a stream processing system, letting a program wait infinitely is never a good idea. For this reason, every calculation timeout should be allowed, so the application may continue to be partial. Any stream processing system must have such time intervals for all blocking operation.

Usually, a time window closes the window when a message received with a greater timestamp of window close time. Dealing with out-of-order mechanism must act that the data with greater timestamp may not a reason for closing windows.

**Generate Predictable Outcomes**

Streaming processing systems must produce predictable outcomes. This requirement is essential for fault-tolerance and recovery. Streaming processing systems need avoiding reprocessing of the data.

**Integrate Stored and Streaming Data**

Stream processing engines must not be valid to process data only on a streaming pipeline. It can enrich data with historical data as well. That is why accessing data from integrated storage is also a primary requirement of streaming processing engines.

**Guarantee Data Safety and Availability**

High Availability is a significant critical requirement for stream processing systems. The system has to work in a replicated way in order to avoid unpredictable hardware errors.

**Partition and Scale Applications Automatically**

Streaming processing system must be compatible to deploy multi-processor and multiple machines service environments. Streaming must handle load balancing amongst servers. Partition of the streaming pipeline should never generate a high latency.

**Process and Respond Instantaneously**

What this last requirement says is that the stream processing system must have a well-optimized mechanism with minimal execution time to provide a real-time response for applications with large volumes.

**BigBench** [17, 18, 19] is the Benchmarking standard which one produced by Transaction Processing Performance Council (TPC). The main difference between BigBench 1.0 and BigBench 2.0 is the coverage. BigBench 1.0 is the only Big Data analytic benchmark standard, but BigBench 2.0 is covering the all big data pipeline like stream processing, key-value processing, graph processing, ETL and Big Data analytics. Here we will concentrate on BigBench 2.0. The BigBench 2.0 is benchmarking Big Data system and observes the system's volume, velocity and variety characteristics. It includes a data generator for structured, semi-structured, and unstructured data. The BigBench 2.0 data volumes can dynamically vary based on a scale factor. The simulated workload of BigBench 2.0 has covered 30 queries to scale the Big Data analytics from the different aspect. The BigBench consists of four steps:

- System setup
- Data generation
- Data load
- Execute application workload

TPC committee still is working towards standardizing it as the most common TPC Big Data benchmark

**BigFrame** [20] is a benchmark generator offering a benchmarking solution for Big Data analytics. It consists of structured data adapted from the TPC-DS benchmark (retail business model) and unstructured data. The benchmark divided into two different sections:

- Offline
- Real-Time

With offline analytic section BigFrame benchmarks historical data and continuous query. Historical workflow is processed at a scheduled time.

Real-time workflow is processing in real-time. It allows near real-time decision making based on instant sales. BigFrame is more suitable for benchmarking Lambda Architecture. It scales Batch and Streams processing at the same time.

TPC have so many different benchmarks, but there are two benchmarks which impress us more than others. That is the StreamBench and Yahoo's streaming benchmark which is benchmarking engines with real-world applications. In Table 2. comparison of those benchmarks attached. We will go to the deep in these benchmarks. Because in the future we will going to perform more alike benchmark.

Table 2. Comparison of Existing Benchmark

| Benchmark | Real World Applications | Micro Benchmarks | Criteria | Engines |
|---|---|---|---|---|
| StreamBench (2016) | - | 3 | Throughput, Latency | Storm, Spark, Flink |
| Yahoo Streaming Benchmark | 1 | - | Throughput, Latency | Storm, Spark, Flink, Apex |

## 3.1 StreamBench

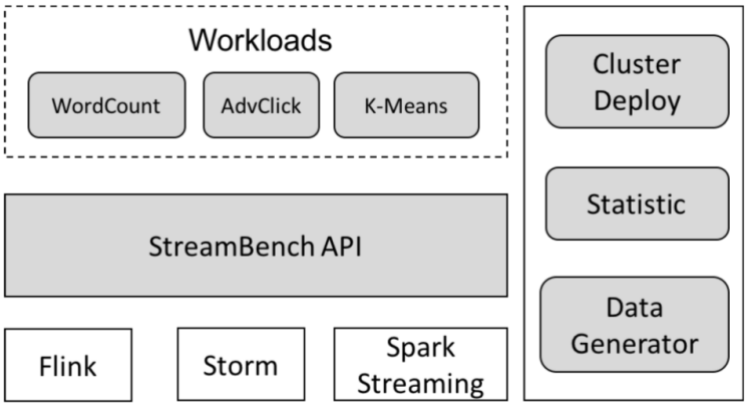



Figure 10. StreamBench Architecture [20]

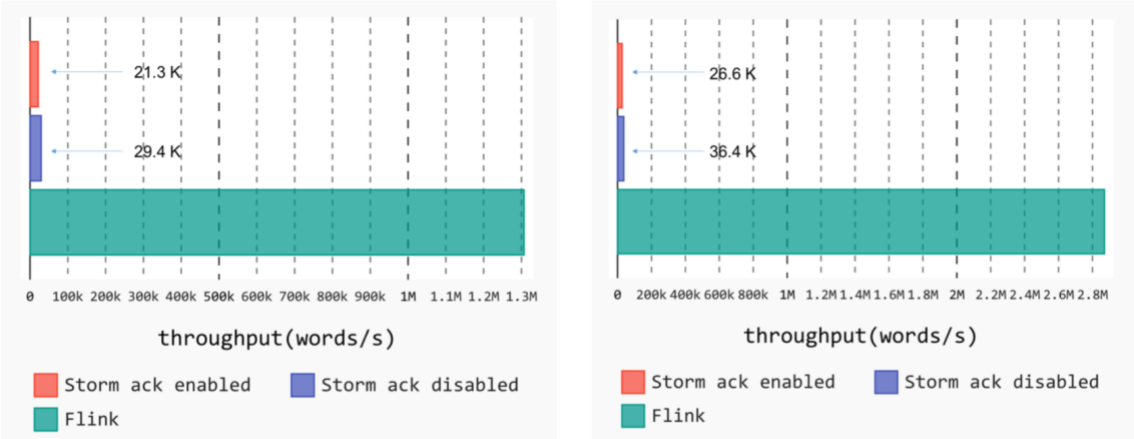
Yangjun Wang [21] simulated Advertisement Click, Word Count and K-Means algorithm on Spark, Storm, Flink engines. The architecture of StreamBench is illustrated in Figure 10. StreamBench Architecture [20]. Wang created core java application by using related engines API.

The Benchmark contains three micro-benchmarks. In AdsClick benchmark Wang simulated view and click of advertisement events. Wang's implementation analysis relation between view and click events. In both Wang's declared id field for the advertisement event. Both streams have joined by using advertisement id. The advertisement appears in both stream in the close time frame he marked that advertisement as a valid click. Thus, he counted valid clicks for billing customer of those advertisements.

Wang's [21] another micro-benchmark implementation is called WordCount. In that implementation, generated data is aggregated in a specific time window. While computing the curve, the computation node that counts the word with the highest frequency may be the bottleneck. Inspired by MapReduce Defragmenter, he designed another WordCount version of the streaming processor along with the window operator. Windows are usually event groups at a specific period. During the reduction phase of Windowed WordCount, the first words are grouped and re-clustered. At a given time, local pre-collection results are stored in calculation nodes. When windows closed, the word counts are keyed reduced to calculate the final results. For last micro-benchmarks, her run the K-means clustering algorithm for points.

Wang's implemented all this infrastructure on virtual servers which run with Ubuntu 14.04 LTS. His implementation contains eight slave nodes and one master nodes for running stream processors. As a messaging queue, he installed five Kafka brokers. During his benchmark, he uses Spark-1.5.1, Storm-0.10.0 and Flink-0.10.1 specific version of stream processing engines.

With Wang's WordCount example it is clear to compare the throughput and latency of Flink And Storm. For a precise result, Wang has run 2 type of WordCount benchmarks; one is Offline and another one Online. Wang designed Offline WordCount benchmark for testing both system's throughput. Before running stream processors, he had generated and stored some amount of data in Kafka and then started the processors to process data. In this way, he compares the throughput of Flink and Storm. He runs the same WordCount benchmark

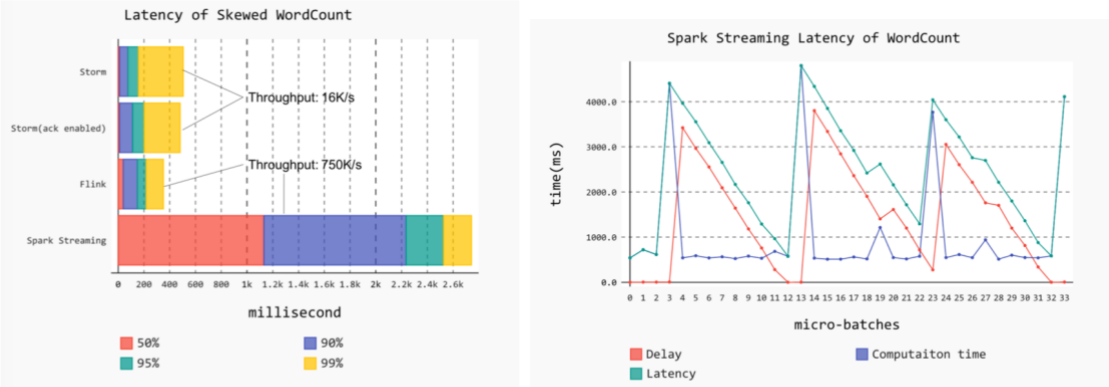



(a) Skewed WordCount (b) Uniform WordCount

Figure 11. Throughput of Offline WordCount [20]

with ack and without ack on Storm. Storm performs more efficient performance while disabled ack. It is clear from the Figure 11. Flink throughput was ten times higher than Storm throughput in this benchmark. Wang did not involve Spark to this benchmark because due to spark's micro batch structure there are a lot of benchmarks scenarios with the same structure.

During Online WordCount benchmark Stream processors have been started before generator's start and Kafka was cleaned up. With this benchmark latency of Storm and Flink have been traced. While running Spark benchmark default configuration for the




(a) Records latency (b) Micro-batches latency

Figure 12. Latency of Online WordCount [20]

micro-batch interval (1 seconds) and checkpoint intervals (10 seconds) didn't change. The

result of the Online benchmark is shown in Figure 12. Because of checkpointing latency result of spark is bumpy. Every 10 seconds spark tried to write information to the storage to achieve fault-tolerance. These operations in the Spark engine consume significant resources. The throughput of data which latency was in Figure 12b is the 1.4M word/s. Figure 12a shows us median latency for Storm (ack enabled) was ten milliseconds, meantime the same metrics for Flink was 39 milliseconds. The 95th percentile latency was 201 milliseconds meantime for spark same metric result was 217.

In AdsClick benchmark click events occur within 20 seconds after the corresponding view event. Kafka [12] doesn't keep the order of messages in the partitioned topic that is why 5 seconds window time set. It means when a click event happens it can join a view event in the future. Below in Table 3, you can find Advertisement click performance table. From Wang's AdsClick benchmark results show us, Flink is working with higher throughput and lower latency. 90% percentile Flink was 637ms meantime Storm was 2116ms.

Table 3. AdsClick Performance

| | Maximum Throughput | Latency | | |
|---|---|---|---|---|
| | | Throughout | Median | 90% |
| Storm (ack disabled) | 8.4K/s | 4.2K/s | 14ms | 2116ms |
| Flink | 63K/s | 33K/s | 230ms | 637ms |
| Spark Streaming (20s/5s) | < 2K/s | Ø | Ø | Ø |
| Spark Streaming (60s/30s) | 20K/s | 20K/s | ~20s | ~24s |

## 3.2 Yahoo Stream Benchmark

At Yahoo, [7, 22] they had implemented Apache Storm before developing this benchmark. Their benchmark scenario and infrastructure are the same for Storm, Spark, Flink, and since they tested them with a real-world application, they got realistic results.

Yahoo [22] has implemented benchmarking with Apache Kafka[14] and Redis[15]. They have simulated advertisement analytics pipeline, where there was IDs for campaigns and advertisements in pipeline data. Then benchmark consumed data from Kafka servers in JSON format and merged it to Redis in-memory storage. During these processes, system aggregated and stored relevant events in Redis in-memory as well. Subprocesses that has been completed during the benchmark scenario are shown below:

- Consume events from the Kafka topic
- Deserialize the JSON data.

[14] https://kafka.apache.org/
[15] https://redis.io/

- Filter out irrelevant events
- Take a projection of the relevant fields
- Merge each event with the covalent row in Redis in-memory store
- Take the window count of the events group by the campaign and store it in Redis in-memory store by their campaign id and timestamp.

Yahoo [22] benchmarked multi-node infrastructure. Each node processors were Intel E5530, 2.4GHz, 16 cores (8 physical) and 24 GB memory, and in total, 30 nodes have been used. They [7] distributed 30 nodes amongst Spark, Storm, and Flink, and have configured 5 Kafka nodes with 5 data partitions, 1 Redis, and 3 ZooKeeper[16]. In that infrastructure, they run 100 campaigns, with ten ads per campaigns. Kafka producers were able to produce 17,000 events per second. In the beginning, they have cleaned all Kafka topics and loaded initial data to Redis. Later on, they have started stream engines and producers, and after half an hour, producers were stopped. When all Kafka topics were consumed and all

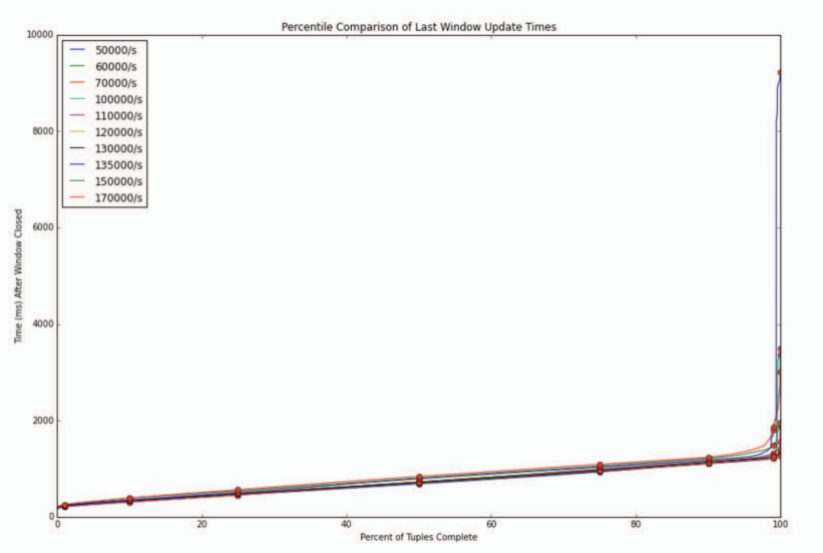



Figure 13. Flink Performance [7]

stream pipeline processes, the system was shut down.

Apache Flink [22] benchmark was developed in Java using DataStream API. In Flink benchmark Kafka event consume rate changed between 50000 events/sec and 170000 events/sec. For each emits rate, events processing latency of Flink is visualized in Figure 13. We can understand that until around 99% event processing latency increases linearly.

[16] https://zookeeper.apache.org/

Apache Spark [22] benchmark was developed in Scala[17]. Micro-batch structure of Apache Spark is making it different from Flink. While Flink benchmark was updating the Redis database in every second, in Spark it was benchmarked with 3 and 10 seconds frames. Kafka event consume rate was 100000 events/sec. They have got two different performance results for 3 and 10 seconds Micro-batch processing. As Figure 15. Shows us, 10 seconds version 90% events have been processed in the first batch. However, they got better results by reducing the batch size, and they have divided it into 3, four sub-batches. Full results are illustrated in Figure 15.

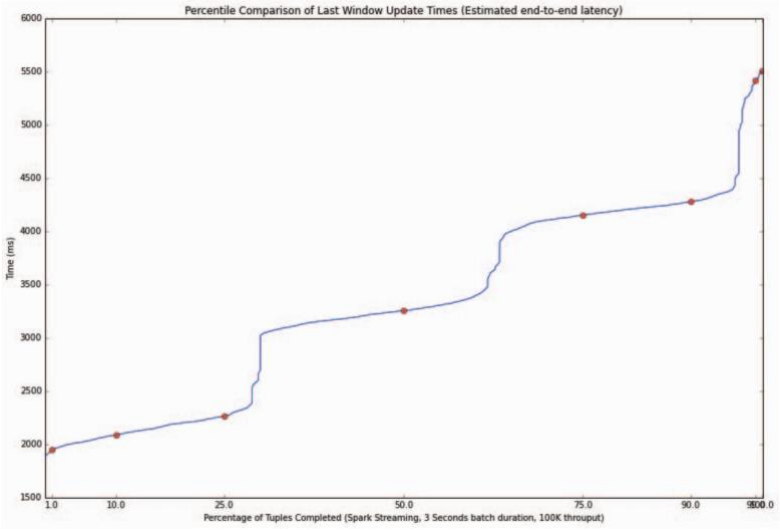



Figure 14. 3 Second batch duration Spark Streaming Performance [7]

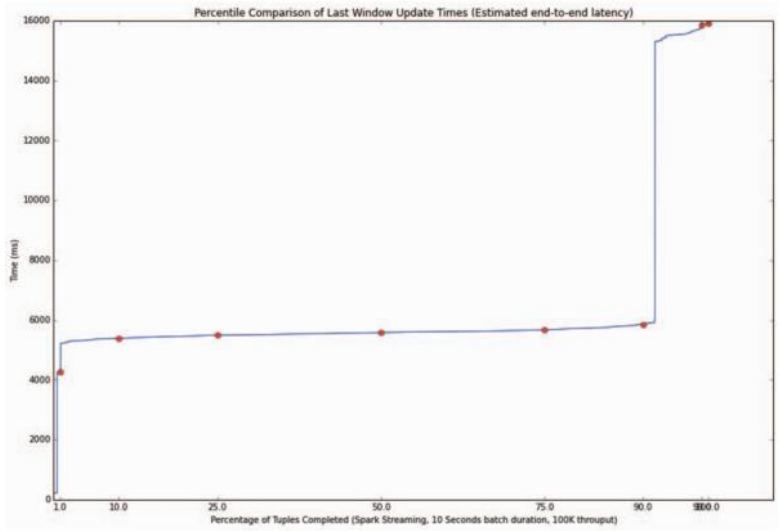



Figure 15. 10 Second batch duration Spark Streaming Performance [7]

[17] https://www.scala-lang.org/

# 4 Contribution

First of all, we would like to share how we prepared our environment for achieving the benchmark of this master thesis. In this section, we will provide some information about the installation process of required software, their configuration, benchmark implementations and how we optimized these steps. Repository of source code has been attached in Appendix III.

As we have previously mentioned, this research benchmark got its main concept from Yahoo!'s benchmarking work from 2015 [7]. However, we didn't want to make a new replicate of the same benchmark, as well as extend it by using some other technologies. Some of the engines, frameworks, and libraries that we worked with are the same as the mentioned benchmark's, where some of them are more modern tools which even didn't exist 2-3 years and are included into our tests. Moreover, for the engines that were also tested by Yahoo!, we used their newer versions. We acknowledge that three years is a long time regarding today's technologies development. There were noticeable improvements and changes in the more recent releases of those engines which we could not skip.

Table 4. Comparison Engines with Yahoo's Benchmark

| **Tool Name** | **Yahoo! Version** | **Our Version** | **Important Changes** |
|---|---|---|---|
| Flink | 1.1.3 | 1.5.0 | Dynamic Scaling / Key Groups; Kafka Producer Flushes on Checkpoint; Table API and Streaming SQL Enhancements; Async I/O, etc. |
| Spark | 1.6.2 | 2.3.0 | API Stability; Unifying DataFrame and Dataset; New user-defined Functions; Scalable Partition Handling; Continuous Processing, Structured Streaming and etc. |
| Storm | 0.9.7 | 1.2.1 | Simple KafkaSpout Configuration; Support for bolt+spout memory configuration; Miscellaneous bugs fixes and improvements. |
| Redis | 3.0.5 | 4.0.8 | A new replication engine; Native data types RDB format changes; Many other bug fixes and performance improvements. |
| Kafka Broker | 0.8.2.1 | 0.11.0.2 | Support for Kafka Stream; Several bug fixes and performance enhancements. |
| Kafka Stream | not tested | 1.1.0 | ~ |

## 4.1 Environment

After deciding which engines, we would use, we continued with looking for servers that suit best for our needs. We ended up with using Digital Ocean's CPU optimized droplets

which perform much better results for CPU intensive projects- such as in our case, than regular droplets. All the droplets had 64-bit Ubuntu 16.04.4 as an Operating System running on them. One of the advantages of using these droplets was that configuration of the servers was easy-to-change. For example, during the time that benchmarks were running, we kept the droplets' RAM and CPU values at the highest. However, when hard processes finished, and we did not need any high performance from the droplets, we decreased them. On Digital Ocean's CPU optimized droplets, these operations can be done through URL's, unless you want to change the disk size- which we did not have to regulate anyway. The script doing this task is provided within the benchmark repository[18] as well.

Afterward, choosing an adequate number of droplets and their distribution was the next step. In order to get more trustful results and to generate more real-world events, we decided to use several droplets as data loaders. Besides that, we also had to reserve some droplets for message brokers. You can find the full list of the node groups, their purpose of use, and characteristics at the highest performance mode in Table 5.

Table 5. DigitalOcena Droplets

| **Node Group** | **Count** | **Characteristics** | **Purpose** |
|---|---|---|---|
| Load | 10 | 2 vCPU,<br>4 GB Memory | Generating real-world ads events |
| Stream | 10 | 16 vCPU,<br>32 GB Memory | Stream processors |
| Message Broker | 5 | 16 vCPU,<br>32 GB Memory | Host of Kafka |
| Zookeepers | 3 | 4 vCPU,<br>8 GB Memory | Host of ZooKeeper & Manager Server |
| Redis | 1 | 4 vCPU,<br>8 GB Memory | Host of Redis Database |

In the next sub-section, we are providing more information about the installation process of necessary software, their configuration, and how we optimized these steps in order to have a half-automated structure to run benchmarks faster.

## 4.2 Benchmark Architecture

While checking the benchmark-repository users will find some bash scripts which are very useful for them to achieve fast, easy-configurable benchmarks. The very first of them is 'initialSetup.sh' which sets up the necessary dependencies as it is understood from its name. After dependencies, when users have to install the tools they want to do tests with, another script called 'stream-bench.sh' can be used. It will download compressed files of the introduced engines, decompress and install them without a user interrupt.

[18] https://github.com/elkhan-shahverdi/streaming-benchmarks

In our case, we wanted to build an architecture where events that are generated on loader droplets would be sent to Kafka's message broker in the first hand. Once messages are in the queue, consumer nodes- which are stream processors, in this case, starts to read them in parallel. In the end, the results are saved on Redis Database. As we mentioned previously,

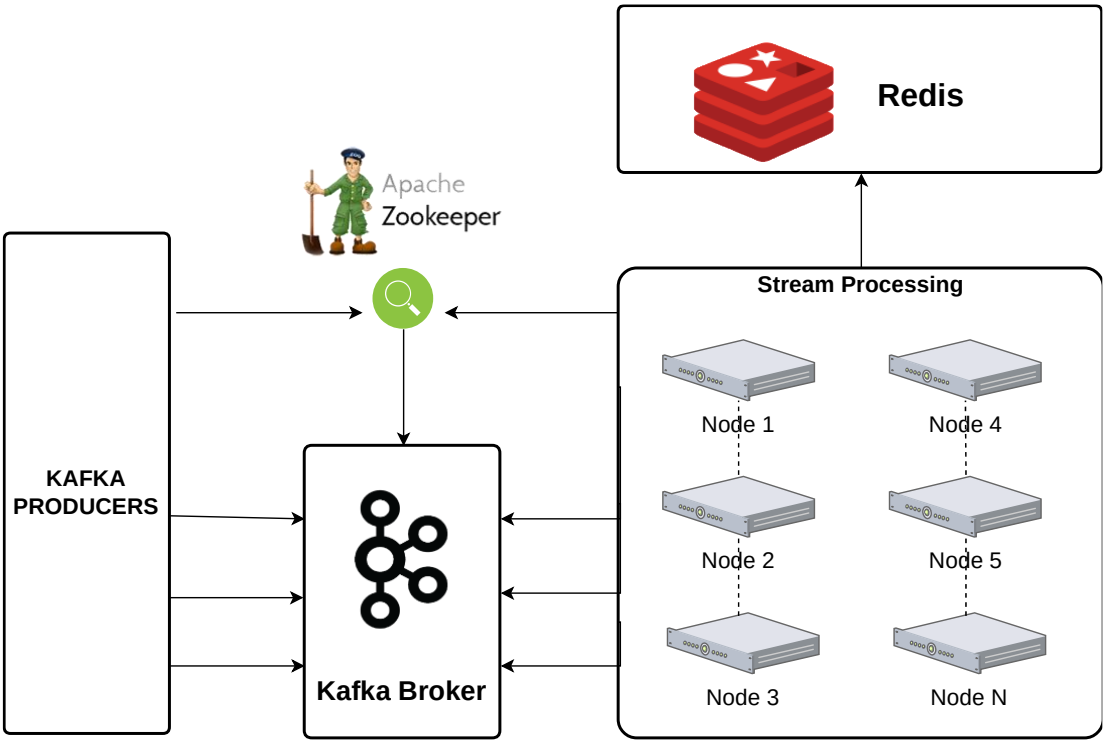



Figure 16. Benchmark Architecture

service management was done by Apache ZooKeeper. In the following Figure 16, the benchmark architecture is illustrated to make it more understandable for readers.

At this point, we had all the necessary programs installed on different droplets, and it was time to configure them for our needs and run. Since in the beginning, we were not sure how many different combinations we would try of configurations, we decided to ease this process by writing more scripts.

## 4.3 Environment Setup

To begin with, we created two scripts where one of them would set the initial configurations of each engine, framework or library, and the second one could run the benchmark with different arguments. In the remote repository, 'setup.sh' can be found under the root directory.

Setup script starts with bash commands of Apache Flink and sets some required variables of it. Later, we define ten nodes to be used by Flink where nine of them would be slaves, and one would behave as the master. Later, we continue with Apache Spark and apply the same amount of master and slaves. Other execution flags of Spark engine such as executor core, executor memory, etc. are also defined and set in this section. Moving on, we added

commands for Storm and Kafka setup. It is important to notice that inside these two sections we configure ZooKeeper and its connection with the stream processors as well.

The script continues with ‘start’ and ‘stop’ operations of the engines and their dependencies. Without going deep to these commands, we would like to emphasize that content of the setup script is quite easy to understand and update if needed. If in the future any researcher wants to develop this benchmark with some more engines or just with newer releases of the current frameworks or libraries, it can be achieved just by modifying existing variables or adding a few more similar lines in case of new tools. Until now we have talked about automatization of the installation process of our engines and getting them ready to run. In the next section, we share our work regarding running another script, which allowed us to execute different tools with, several arguments and an easy configuration.

## 4.4 Benchmark Execution

Knowing that we have all the pieces of the puzzle in the place, we moved on by finding a way to start running the benchmark step. We realized that we might need to run benchmarks more than once and it could require some small changes on every run. Thus, we decided to write a script where we could have all the commands together, with parametric functions that we can change some runtime arguments, etc. The script can be found under the root directory of the benchmark repository as well (‘remote.sh’).

First of all, we started by defining the most frequently used variables such as emit rate parameters, sleep time intervals, SSH credentials, etc. We thought that having these parameters at the beginning of the file could help us with changing them easily when needed. Then, we listed our engines and prepared their possible commands that would be used during benchmarking. Although in most of the time it was engines’ ‘start’ and ‘stop’ commands, in some special cases they required more specific commands as well, such as creating a temporary directory and removing it when the program stops, etc.

Once we had all the necessary commands, we started to group them in logically related functions. For example, we added a function to start all the tools to run sequentially for a pre-defined time, but in an infinite loop. It would require stopping command afterward when satisfactory results are obtained. There are also more functions for running droplets, shutting them down, starting and stopping each engine, monitoring, collecting results, etc. Before listing the full command list of the script, we would like to emphasize that, it is a script containing more than six hundred lines of bash commands, and it was written with the only purpose; benchmark of stream engines must be as easy as configuring them on existing bash files, running, monitoring and retrieving the results by less than 10 commands in total. Now let’s have a look at the full list of arguments that can be called with ‘remote.sh’ script.

As it is provided in Table 6, to run a benchmark all is missing is a program that implements a good algorithm for tests. We believe that we made it easy for anyone who wants to run

a benchmark on stream processing area. Next, we are sharing our basic program that was used for this research.

Table 6. Command descriptions

| **Command** | **Description** |
|---|---|
| <tool_name> | Starts a benchmark for the introduced stream engine including its monitoring. Possible values are *flink*, *spark*, *storm*, *kafka*, *heron*, *all*. |
| start <tool_name> | Starts only the tool, engine, library, or program itself with introduced name. Possible values are all tool names*, redis, zoo, prepare, load.* |
| stop <tool_name> | Starts only the tool, engine, library, or program itself with introduced name. Possible values are all of the ones for start and *stopAll.* |
| result | Collects results from Streaming, Kafka and Redis servers respectively. |
| load | Starts data loading. |
| push | Pushes git changes to the remote servers and runs all of them. |
| report | Collects the results and draws meaningful charts on PDFs. |
| build | In case of changes in the code, this command can be run to re-build maven projects in the remote servers. |
| clean | Clean the last obtained results. |
| test | Runs basic tests for the benchmark. |
| resize up | Increases capacity of available Remote Droplets. |
| resize down | Decreases capacity of available Remote Droplets. |
| power off | Turns off the remote droplets. |
| power on | Turns on the remote droplets. |
| reboot | Reboots remote droplets. |
| shutdown | Runs a command to shut down all the Servers from Digital Ocean. |

## 4.5 Implementations

Let's have a deeper look on Benchmark implementations. For running Benchmark, first of all, we need to start the Redis, and after starting it, we have to run Clojure script with -n parameter for setting up Redis for the new real-time simulation. This step must run on only one node, only once before starting multiple data loaders (-r) on multiple nodes. Once we have these steps done, we can start ZooKeeper and then Kafka. If Kafka and Zookeeper are running, then it is time for creating advertisement events topics on Kafka. Later on, benchmark scenarios have been submitted to the stream processing systems. When all these steps are completed, we will be ready for the start of the streaming engines and the data generators. What that Clojure script is doing that it will populate Redis with the generated campaign which in the future will be used by data loaders while creating advertisement events. In other words, the data loader will load data to Kafka by checking

convenient campaign from Redis. After starting all the environments and data loaders, submitted scenarios on Stream Processing engines will start to process data which had been loading to advertisement topic on Kafka Servers. Next question can be like that how we are processing data how we are calculating Throughput, Latency, and Resource Consumption over this scenario.

Yahoo simulated processing of advertisement clickstream. During reading the explanation, the structure can be followed up over Figure 17.

The First processing unit of our clickstream scenario is consuming advertisement event clickstream topic from Kafka. After consuming row data from Kafka, we are parsing it to Java POJO, json or tuple.

The Second processing unit of our clickstream scenario is filtering data by *event_type*. While data generator generates an event, it is assigning event type randomly among this list (view, click, purchase). In our scenario, we are filtering, and processing only views events.

The Third processing unit of our clickstream scenario is the projection of the event's useful attributes. Because there is additional information like user id, page id, ad type, IP address and event type which will not need any more in our scenario. In this step, we will eliminate such kind of useless attributes.

In the Forth processing unit, we enrich our events with its campaign id. After starting Redis, we have run the setup script. It populated that campaign id which can be attached to our events by ad id in our fourth step. In this step, we are creating a connection with Redis to lookup the campaign id of our advertisement. If there is not appropriate campaign with these ads, then we are going to eliminate this advertisements event.

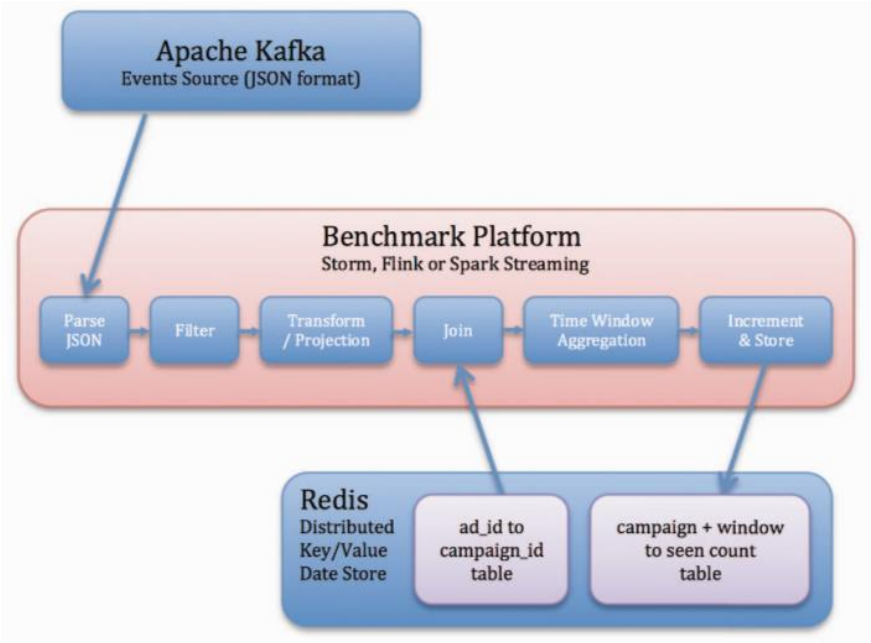



Figure 17. Benchmark Design [7]

The fifth processing unit of our benchmark scenario is the aggregations. In our benchmarks, there is a time-divisor property in our configuration file, which is defining the size of the time window for aggregation. The processing unit is aggregating events by campaign id and time window.

The sixth processing units of our benchmark scenario are the storing result. We are storing all our aggregation results in Redis. After finishing Benchmark before stopping Redis, we are loading this data into the file system.

Yahoo's [7] aggregation and storing technics doing its jobs in two ways. The first one of them is the *CampaignProcessorCommon* class which has been created under benchmarks-common projects. It is the common helper class which is handling fifth and 6th step by itself. Talking about how it works, we can say that it contains a *campaign_windows LinkedHashMap* and a *need_flush* Set local fields. After the 4th steps when we transmit data to inside these classes, it will aggregate events and store result in *campaign_windows* map and meantime local flusher thread will control the closed window for storing the result of that windows in Redis.

The second way of aggregating data is using the native aggregate, *groupBy*, and *reduceByKey* methods which Stream Engines provide with their APIs.

For the benchmarks of Flink, Kafka, Storm, and Heron, we have used the first implementation. However, Spark DStream and Structured Streaming benchmarks used Native Stream processing APIs. Let's go to deep with our Benchmark implementations with the mentioned Engines. Before we forked the benchmark, Yahoo had implemented four SPS; Flink, Storm, Spark DStream, and Apex[19]. You could check the version of engines from the Table 4.

Before implementing new engine benchmarks, we were going to upgrade engines which have been added by Yahoo. Upgrades of Kafka version has affected all benchmarks, and we were gone to update all related dependency of existed projects. We exclude the Apex from our benchmark. After running the Storm benchmark due to the low performance, we decided not to compare it with our Engines benchmark.

Flink benchmark has been developed by Yahoo which we included it in our benchmarks as well. When running Flink benchmark in our Environment, we set the parallelism to 144 (hosts * core). Because of the Kafka version upgrade, we changed Data source from *FlinkKafkaConsumer082* to *FlinkKafkaConsumer011*. In Flink Benchmark data has been transmitted in Tuples.

Before choosing new Stream Processing engines, we had to decide amongst Kafka, Spark Structured Streaming, Heron, and Hazecast Jet. Due to Hazelcast Jet's recently developed implementation and some missing features, we ended up with skipping it after some basic tests.

[19] https://apex.apache.org/

Besides the fact that Kafka has been involved as a message broker, Kafka Stream was used as a Stream processing engine as well. Thanks to its APIs, implementation did not require too much work-around. Kafka Stream API provides us *StreamsBuilder* class to create Stream processing bolts. While processing our advertisements stream, we have used *mapValues* to parse our JSON strings to POJO object. We took advantage of Kafka's filter method and *mapValues* for projections for events' properties. *RedisJoinBolt* class has been created and extended Transformer abstract class. This class helped us to enrich the ads events by campaign id retrieved from Redis. Kafka Stream gives us the opportunity to create our custom Processors with *AbstractProcessor* abstract class. We have used this class to construct our common aggregate class and invoke it for each incoming event.

After updating Spark to the latest version, we noticed the new Structured Streaming API, and we decided to include Spark's new concept in our benchmarks. We have implemented it under the *spark-cp-benchmark* module.

One of the biggest advantages of Structured streaming is, processing units as known as bolts, can be defined by using SQL. For example, we have used "CAST (value as string)" query for parsing data into String when it is received from Kafka. Filtering methods of Bolts were similar as SQL syntax too. In this implementation, we, have aggregate data stream by Spark Structured APIs. For group operations, we called *groupByKey* by *campaign_id* and *window_time* parameters, *count* method for counting advertisements of appropriate windows, and finally *as("count")* method to attach count of advertisement to the streaming again. By the end of this process, we save the results to Redis Database with "update" output mode of the writeStream.

# 5 Experiment

After creating our Droplets in DigitalOcean, we were going to test our benchmark scenario with several different configurations. One by one we installed and ran all clustered engines. While running benchmark we did not use any resource management platform such as Yarn, Mesos, etc. All engines have been installed in standalone mode. Let's talk about the details of configurations of our implementations. You can check our configuration script (setup.sh) from our repository.

## 5.1 Experimental Design

In our infrastructure, we have installed 3 Zookeeper and 5 Kafka servers. We used the same Zookeeper instances as service managers for the Storm and Heron Engines. For Zookeeper servers, we had a heap size of 7GB heaps in maximum.

Furthermore, we have used 5 Kafka Servers with 32G maximum heap size. To achieve nearly the best performance of Kafka, we have changed the default properties of our Kafka engines. We have tested large topic partitions in our benchmark, and that is why our network thread size changed to 20 to receive and send more network requests concurrently. Kafka's disk I/O thread count has been increased up to 8. Nevertheless, [12] flushing data to the disk is more expensive to process. Kafka servers has enough memory to keep and process messages in Memory, and because of that, we set a huge interval for flushing data to the disk space. During benchmark, the flush interval of a number of messages was 10 million, where of the time was 100 seconds. The same configuration has been used for Kafka Streams as well except the Kafka Engine version which was 1.1.0. This Kafka version could not be used for engines as message broker because of missing Kafka-client support for consumers.

Flink was implemented in the master-slave model, where there were a single master and nine slave nodes. Flink heap size of task managers has been set to 30GB and job managers heap size has been set to 15GB. The number of task slots for slaves was set to 16 since all of our stream servers have 16 vCPUs. Before installation, we have set public SSH keys among all our benchmark servers with our installation scripts. That is why after listing all our slave nodes to slaves and master node to masters file our installations for our Flink stream processing environment have been finished.

Later, Spark was implemented with the same model- a single master and nine slave nodes. As we mentioned before, we did not run Spark Streaming with the default configuration. We have increased heap size and executor count and made it more compatible with our benchmarks and environment. For our Spark engines, a number of cores for the executors have been defined as 16 and memory per executor was of 30GB. On every slave, the total number of cores to be used has been set to 16, and total memory to be used by executors on workers was 30GB. The micro batch size- a vital parameter of Structured Streaming and DStream with Spark, was set to 3 seconds same as in Yahoo’s benchmark. We have run our

benchmark with such properties on Spark Streaming Engines, and we believe that it was the adequate configuration for our case.

Finally, Storm engine has been implemented and configured in standalone mode. As the previous streaming engines, we also thought about optimizing configuration for this tool. We tried to adopt Storm to our benchmark and servers. A single node allocated as a nimbus for clustered Storm and 3 ZooKeeper servers were used as service managers. For childopts of supervisors and workers 16GB memory has been allocated. However, 24GB memory has been allocated for nimbus's childopts. We have tested our Strom benchmarks with different combinations of worker count, Acker counts, and topic partitions. Comparing all cases, we obtained the highest throughput and lowest latency with 36 workers, 9 ackers, and 100 Kafka Topic partitions. This was the only case that we have set Kafka topic partitions to 100. However, considering all the engines, we did not get a satisfactory result for comparing it with Kafka, Flink, and Spark. Therefore, in the next chapter, we will compare throughput, latency and resource consumption of Flink, Kafka and Spark Engines in our benchmarks and talk about their performances regarding mentioned characteristics.

## 5.2 Stream Experimental Result

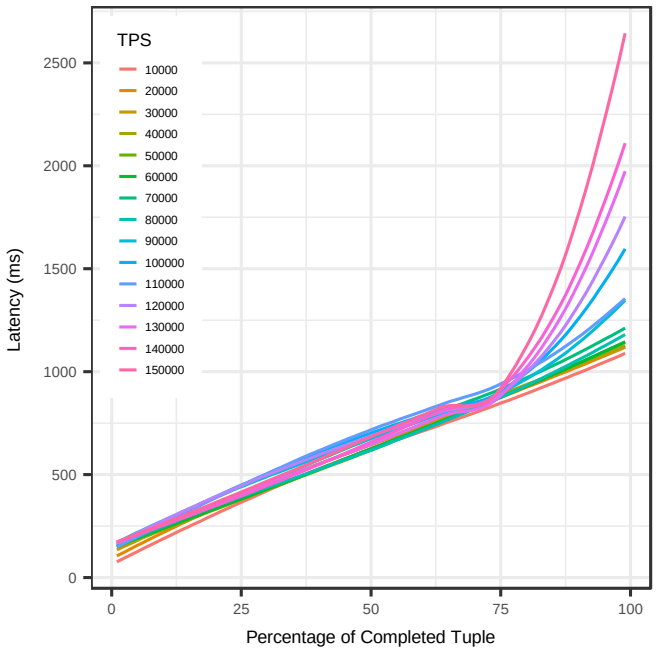
You can find all the metrics we obtained from our benchmark. The duration between start and stop time of data loaders has been set to 600 seconds. However, after stopping data loaders, we have waited 60 seconds all environment up and running for processing queued event in the brokers. Data loaders emit rate has been varied between 10K TPS and 150K TPS. Emit rate has been increased by the step of 10K transactions per seconds. All the chart and graphs provided in the next sections were illustrated based on the experiments.

### 5.2.1 Latency and Throughput

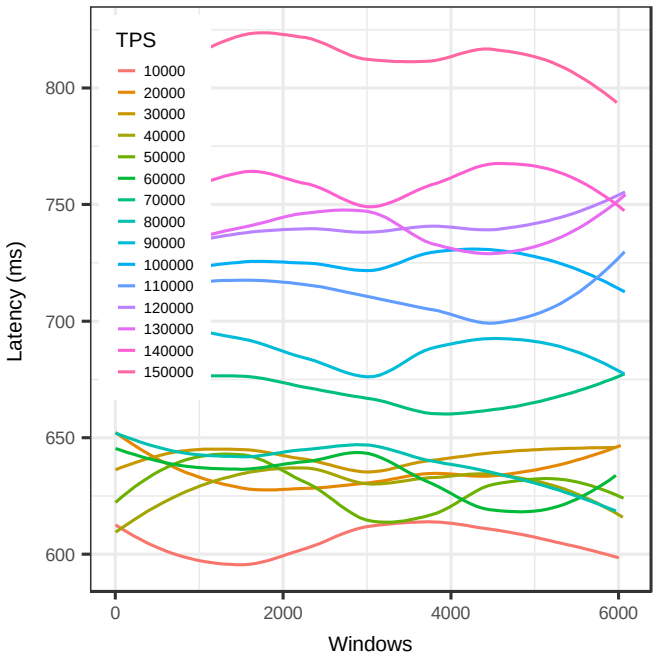
In this section, we will evaluate the result of Benchmarks and will try to find bottleneck of each stream processing engines.

**Flink Benchmarks Results**

TPS range for Flink benchmark was between 10K and 150K. In Figure 18, information about Flink percentile of latency and windows latency can be found as a chart. From the figure, we can conclude that below 75% percentile all benchmarks behaved similarly, regarding latency was independent on emit rate. Above 75% percentile, the latency of each emits rate between 100K - 150K, varied between 1 second and 2.5 seconds. However, latency until 100K emit rate, increased linearly. In Figure 18b, we can see latencies of all windows. Because of three facts, our windows count was approximately 6000. These reasons were; a) we had defined window size as 10 seconds, b) the benchmark duration was 600 seconds c) Total amount of campaigns was 100. We observe that Flink has the bottleneck when TPS is 150K. Because when we check the chart, we will see that 150K TPS has been increased latency more than others. In the percentile graph since after 140K emits rate 90% percentile of latency was more than two seconds.




(a) Percentile of latency

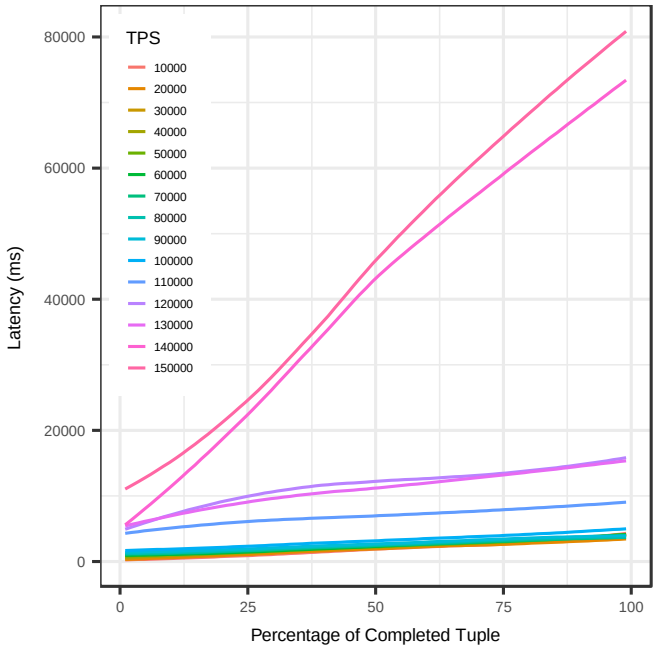



(b) Loess regression of latencies

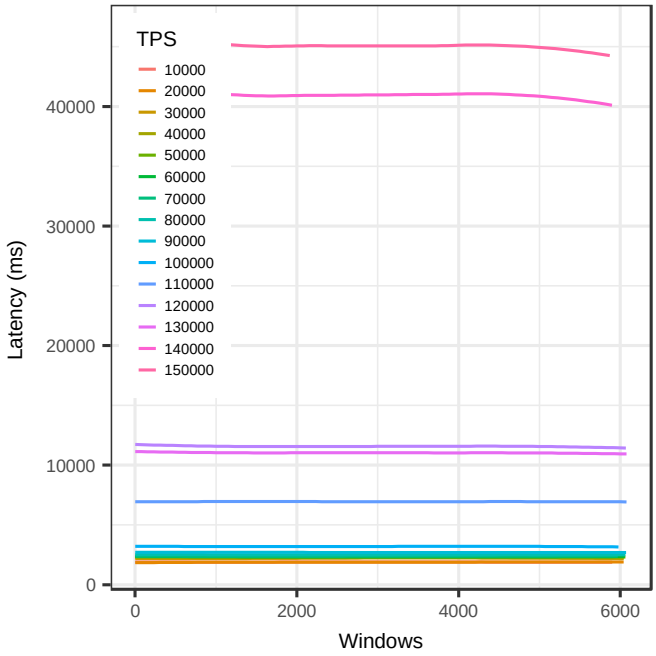
Figure 18. Flink Benchmark latency reports

**Spark DStream Benchmarks Results**

Data loader emit rate varied between 10K TPS and 150K TPS for Spark DStream Benchmark. Three seconds micro-batch size has been defined for Spark DStream Streaming. From Figure 19, we can check the percentile and regression latency of spark streaming. From our benchmark, we can say that Spark DStream bottleneck was around 130K emit rate. After 130K emit rate percentile latency radically increased. When we check, we observe that after 130K emit rate, the latency of windows jumped around from 10 seconds to 40




(a) Percentile of Latency




(b) Loess regression of latencies

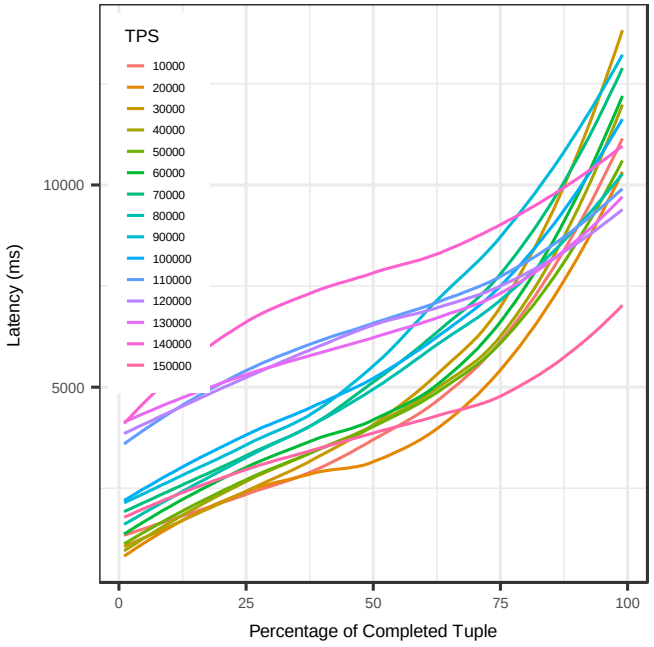
Figure 19. Spark DStream Benchmark latency report

seconds. If we excluded the 140K and 150K emit rate, (Appendix II, Figure 25) latency of emitting rate between 110K and 130K would be higher than the others. Thus, we can say that Spark DStream had an excellent performance below 110K emit rate.

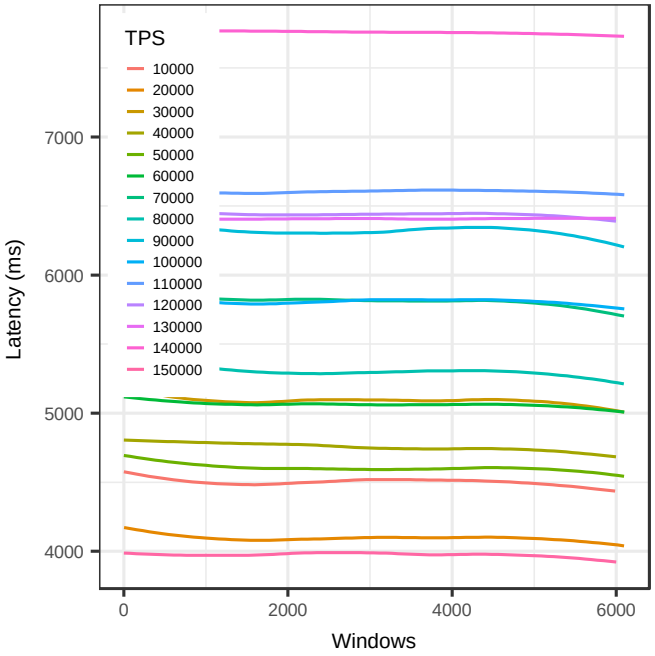
**Spark Structured Streaming Benchmarks Results**

Data load emit rate varied between 10K TPS and 150K TPS for Spark Structured Streaming benchmark and 3 seconds micro-batch size has been defined for the benchmark. In Figure 20, we can see the percentile latency and latency of windows of Spark structured

streaming. Spark Structured streaming throughput is higher than Spark DStream. While emit rate is above 140K, the latency of DStream is greater than Structured Streaming. However, having the latency more than 7 seconds is not good while micro-batch size has been chosen as 3 seconds. On the graph, we see that the latency is always above 7 seconds where the TPS is 150K.




(a) Percentile of Latency

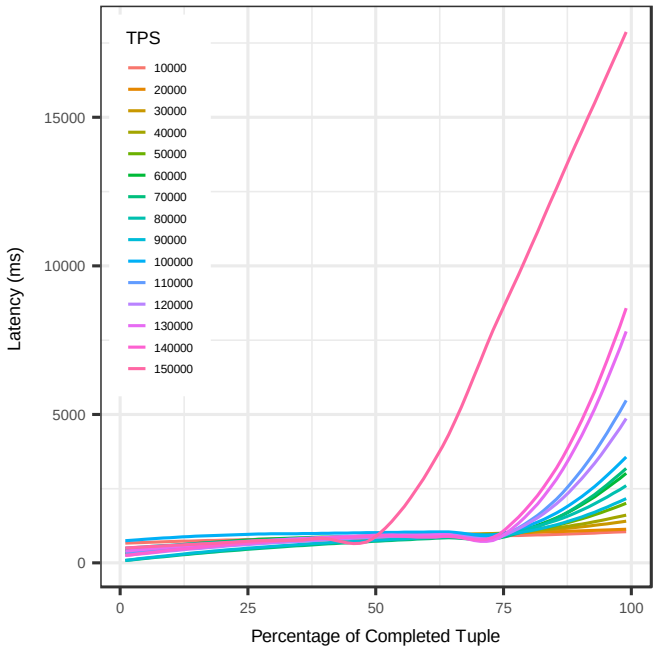



(b) Loess regression of latencies

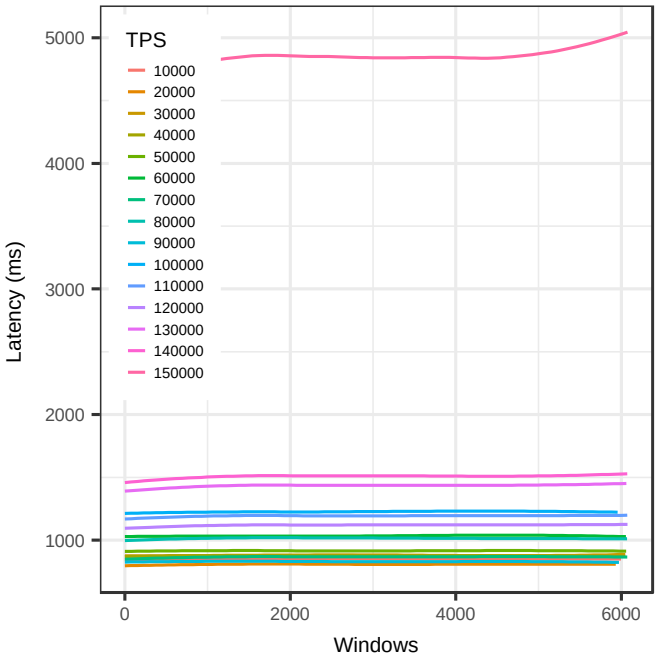
Figure 20. Spark Structured Streaming Benchmark latency report

**Kafka Stream Benchmarks Results**

Same as for the previous cases, data load emit rate was defined to change between 10K TPS and 150K TPS for Kafka Streaming benchmark as well. In Figure 21, percentile and window latencies values are illustrated. Before the evaluation and comparison of Kafka with other engines, there is a benefit to emphasize that we have created five partitions topic. Because only ten servers have been attached to the Kafka benchmark in total as



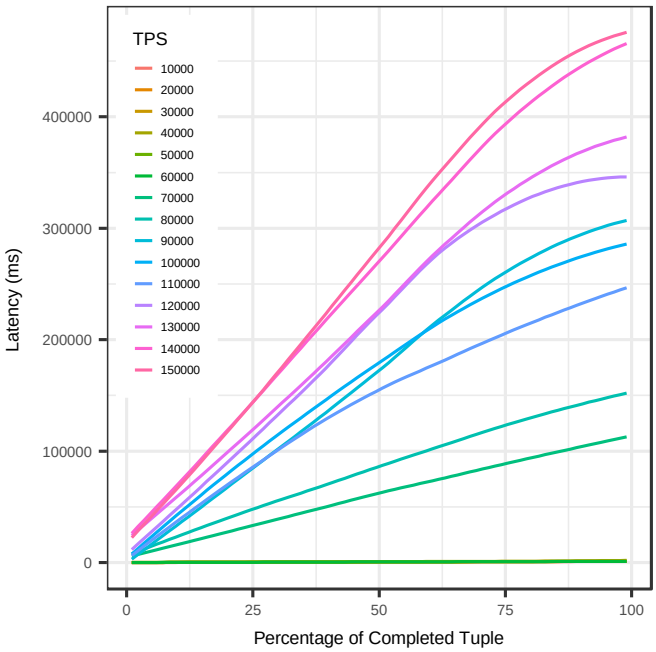

(a) Percentile of latency



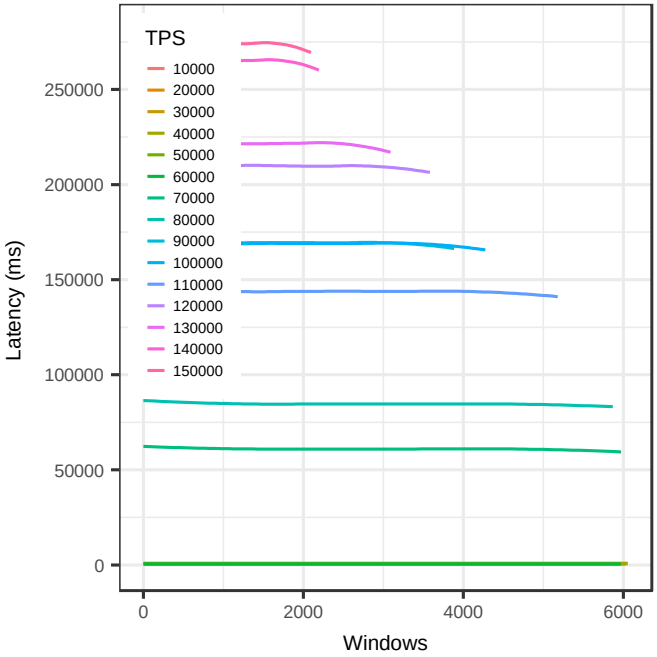

(b) Loess regression of latencies

Figure 21. Kafka Stream Benchmark latency report

stream and broker servers, where there were 15 of them in other benchmarks. From Figure 21, we can easily observe that 150K emit rate is the bottleneck of Kafka. Same as the Flink, below 75% percentile latency increases linearly, where the benchmark is excluded by emit rate 150K. With our implementation, Kafka's highest throughput with low latency was 140K TPS, which means any value above that is a bottleneck. When Data loader emitting rate is 150K after 50% percentile latency radically increases.




(a) Strom percentile of latency




(b) Loess regression of latencies

Figure 22. Storm Benchmark latency report

**Storm Benchmarks Results**

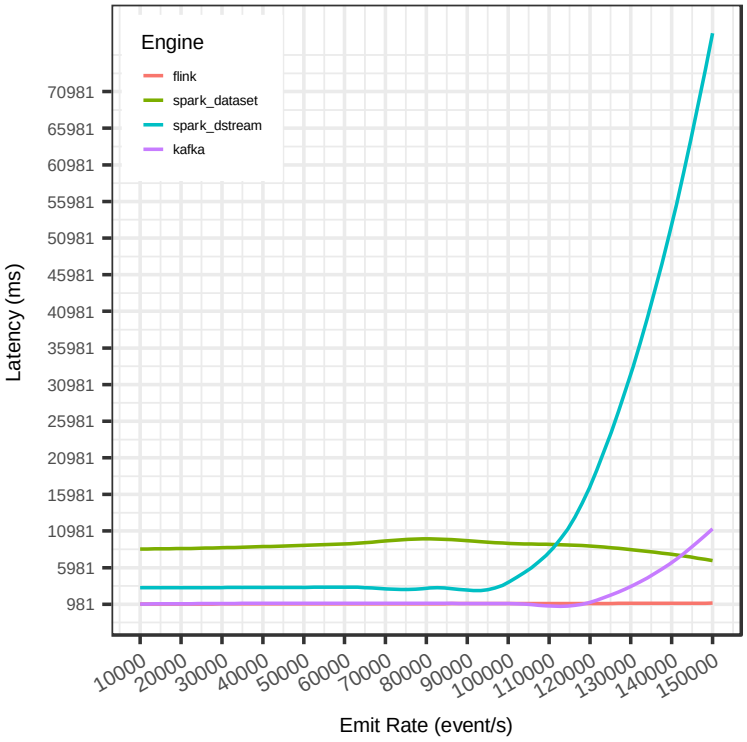
From the Figure 22, we can see the low performance of Storm. Although, Storm's performance below 60K TPS is comparable with Flink and (Appendix II, Figure 26), Storm was the engine which we caught the earliest bottleneck. If we take a look Figure 22b, we can see after 60K TPS latency increased over the 60 seconds and then Storm engine left unprocessed event behind it. When TPS was maximum Storm engine left unprocessed more than half of events.

### 5.2.2 Comparative Latency

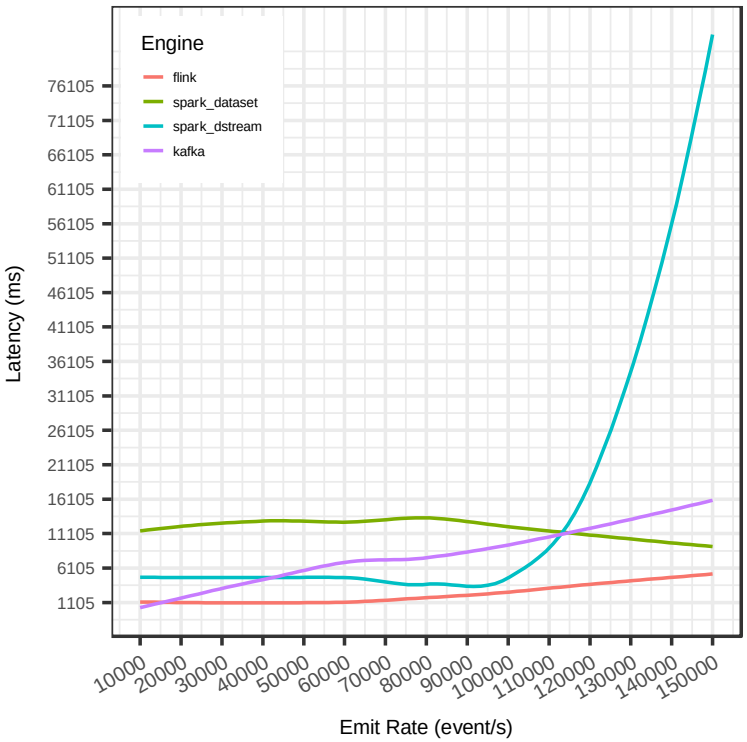
In this section, we will compare performances of our streaming engines regarding latency on several emit rates. In Figure 23, you can see how the relationship between the latency of each stream depending on throughput for 90 and 99 percentiles. Although the general view is similar, for both cases, we would like to emphasize some points regarding the charts. Starting with 90%, we observe that Spark DStream has a drastically changed after 100K emit rate where it keeps having increasing latency until 150K. Although Spark Dataset has more constant latency in most of the time, we see that DStream has at least two times better performance than it for the emit rates below 100K. We also observe a reduction in Spark's Dataset with a larger amount of data which is even better than lower emit rates. Even though we realize that before interpreting it, it is important to test it with larger data emit rates, we think that Spark Dataset has more scalable performance independent of emitting rate. Another attention-grabbing point in this chart is about Flink's almost-linear performance. It has the best performance regarding latency compared with any other tool, for any emit rate. We will talk about reasons underlying it at the end of this section after having a look at resource consumption results. Kafka is the only engine which competes with Flink for emitting rates below 120K. We see hyperbolic increment after that breakpoint.

While reviewing results from 99% percentile graph, we obtain the same ideas from the previous comparison. Spark DStream starts with higher latency and continues very similarly and reaches to its point of failure at the same emit rate again. Spark Dataset follows the same shape as well, where we see that it reaches to its lowest at the ends of the chart. It lets us say that Spark Dataset should be tested at much larger emit rates to achieve better ideas about its performance. The most noticeable point between two charts of 90 and 99 % percentile is about Kafka's latency. Although it was quite stable in the previous case, we see that in the second case it starts to increase close-to-linearly since the beginnings of the graph. Regarding this, we can say that 90% of the cases latency was low and didn't vary a lot, but when it varied, the gap was too big that it affected the average latency of 99% percentile. Finally, we see that Flink has performed better than any other engine again, even though it has a small linear increase after 70K.

Overall, we can inference that for the emit rates below 100K, Kafka and Flink has a good, and Spark DStream not bad performance, where Spark Dataset is approximately two times




(a) 90% percentile latency




(b) 99% percentile latency

Figure 23. Percentile latency report

slower than them. For higher emit rates, Kafka and Spark DStream are not performing trustful at all, where Flink has a linearly increasing and Spark Dataset has a linearly decreasing latency performance.

### 5.2.3 Resource Consumption

In this section, we want to share a comparison of the performance of the tools regarding resource- CPU and Memory usage, consumption. Figure 24, can give the reader a general idea about the resource usage of each engine. However, we want to talk about some interesting points in these charts. Please bear in mind that we reported these statistics for each server group separately. In the left side of the Figure, results of 10 Streaming servers are illustrated, where in the right side, they belong to the 5 Kafka servers that were used as message brokers.

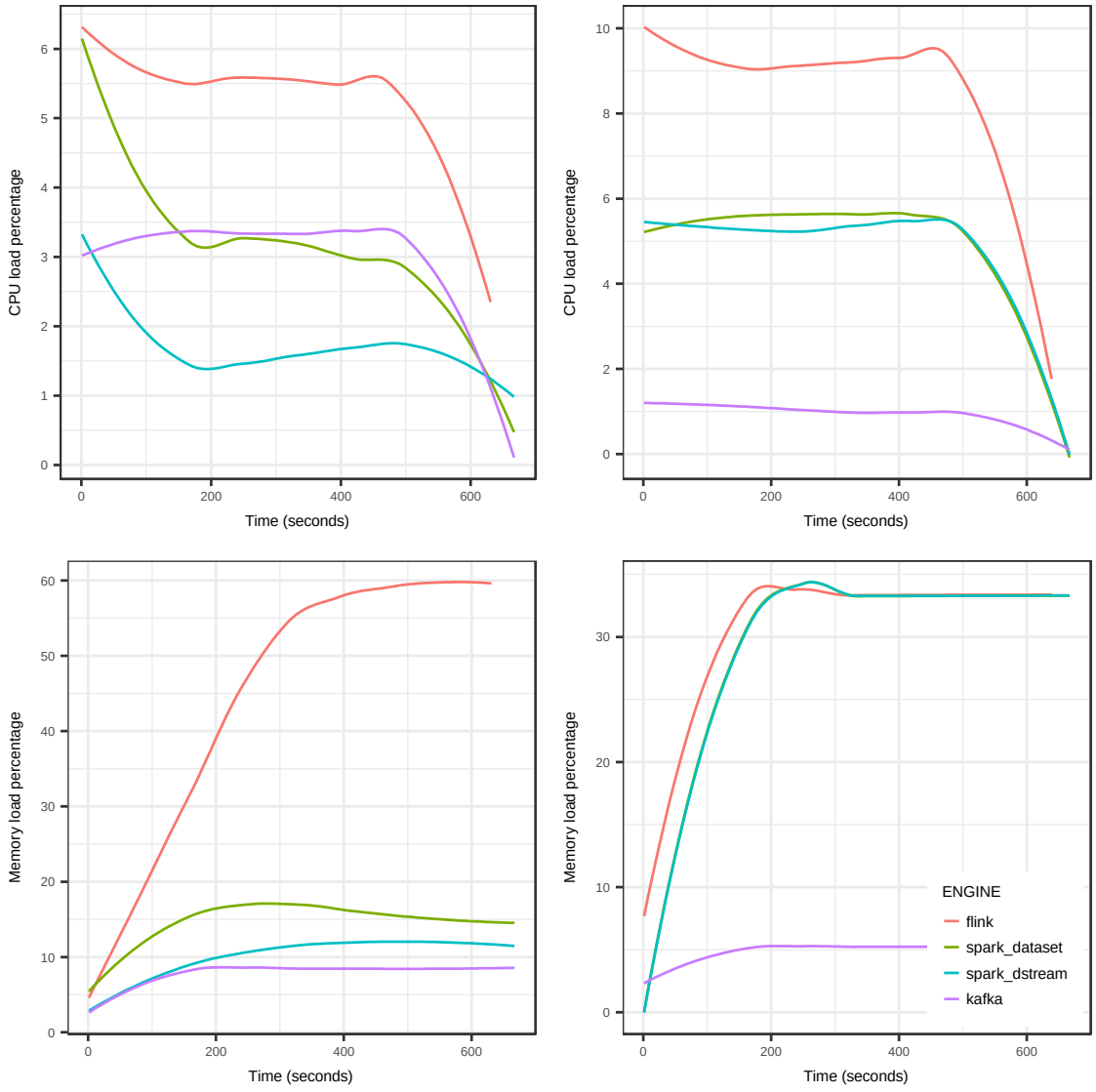
All of the Stream and Kafka servers have 16 cores CPU and 32 GB RAM. In Figure 24, average CPU and Memory usage of these servers are illustrated. Memory consumption usage percentage is evaluated out of 320 GB for stream servers and 160 GB for Kafka servers, where CPU usage percentage is based on total amount of 160 cores for stream servers and 80 cores for Kafka servers. To explain it more clearly with an example, we can think of the case where the chart indicates 6% of CPU usage and 10% Memory consumption. We can conclude from it that, at that particular time, the benchmark was consuming ten cores and 32GB of RAM respectively.

First of all, we would like to start with CPU consumption of two different server groups by engines with emit rate at 90K. Talking about stream servers which we have installed streaming engines, we see that the highest average CPU use belongs to Flink. Although its CPU usage decreases considerably after 400 seconds, we see that at the beginning of the process it was two times higher than Spark tools and three times higher than Kafka Streaming. Another interesting point in the chart is about Spark Dataset's performance. We can see that it starts with almost same as Flink- above 6 % and keeps decreasing constantly and reaches to its low by the end of the process. Kafka Streaming has an average and constant CPU consumption- approximately 3.5 %, since the beginning and slows down after 500 seconds. The best performance regarding CPU load of Stream Servers has been achieved by Spark DStream. We see that it starts at 3% and constantly decreases down to 1.5 % during the first 200 seconds. Although it keeps increasing after this time, it never goes above 2% which is at least two times better performance than the other engines.

From the right-upper chart, we can see that the CPU Load of Kafka message broker servers were a bit different than the Stream Servers. However, Apache Flink still has the highest CPU consumption compared with the other engines. We observe that it starts at 10 % and is always above 9% during the first 500 seconds of the benchmark. From the chart, we can see that Kafka Streaming has the lowest consumption of CPU resources. At this point, we want to point out the fact that, while running the benchmark for Kafka Streaming, although we reserved, 15 Servers in total, in most of the time not more than 10 of them were in use. Use of servers was managed by Kafka itself, and we did not add or remove more servers to the cluster manually. Since some droplets were not actively part of the benchmark, their average CPU consumption is lower with a big gap than the other engines. We can also

observe that Spark engines had very similar results in this case. They both start above 5% and then slow down after 500 seconds when the process is about to finish.

The second term of resource consumption is about memory load of both- Stream and Kafka, servers during benchmarks. While checking the first chart, we see Flink's remarkable low performance even from the first glance. It is observed that Flink had more than 50 % of the memory load after 300 seconds, which is six times of Kafka Stream's memory load. Kafka Streams always performed below 10% of Memory load on Stream Servers, and it is also about the fact that some servers were not used all the time actively by Kafka. Spark Dataset performed the average Memory use which was around 17% in its peak. However, we can say that it loaded 15% of the Memory constantly during the last 500 seconds. The most scalable performance here is achieved by Spark DStream which increases the memory use with a very small percentage. Although it reached 8% of memory use in the 100th seconds, we see that it never goes above 15% during the benchmark.




(a) Stream Servers (b) Kafka Servers

Figure 24. Resource Consumption for 90k TPS

Because Flink [5] uses on-heap memory as a memory segment and keeps the data processing on binary representation and off-heap, as well as reduces Garbage Collector's job to the minimum, the memory is loaded more than average during Flink benchmark comparing with the rest of the engines. On the other hand, this memory management helps Flink gain high throughput and low latency. Flink loads Garbage Collector of JVM in minimum levels and provides better performance in this way. However, Spark and Kafka applications rely on JVM GC for memory management. However, as Spark and Kafka [12, 9] applications, JVM's garbage collector push the boundary of performance and creates low memory consumption for both systems. Because of the micro-batching process with 3 seconds interval, Spark has a higher memory consumption compared with Kafka.

The graph that Kafka Servers' memory use is illustrated shows that engines loaded memory in different rates and in a different way. Kafka Streams is the only one which still has a very scalable performance where it does not load the memory more than 5% at any moment. Flink's performance is the same as of the Stream Servers with the only difference that the highest and average use is around 37%. The unexpected behavior for us was about Spark tools performance. Dataset and DStream both reach the 35% of the memory use in less than 200 seconds and after that keep constant use around 35%, which is the same as Flink's.

# 6 Conclusions

Finally, we will conclude the experiments and results we obtained regarding the thesis of this master research. First of all, we would like to emphasize that the results that are shared represent only some part of the benchmarks we ran. We have run more configuration versions for each engine, where 15 different emit rates were tested for each of them as well. In the contribution section, we only included the most meaningful results regarding the main concept. Besides the performance regarding latency, throughput and resource consumption, we would like to share that Heron and Hazelcast tools had the most complex setup configuration and maintenance feature to manage.

Regarding the performance of the tools we have tested, we can say that each of them had pros and cons depending on the environment and other factors. We think that as real-time stream processors Flink and Kafka are most noticeable ones. We have already seen how low latency Flink provides for high emit rates comparing with the rest of the engines. However, we have also seen that it is because it exploits resources such as CPU and Memory use. Although Kafka did not load the servers as much Flink did, its latency was not far from the latter one. At this point, we can say that, if a user needs real-time processing, with low latency, Kafka is a very optimal choice. Moreover, for the cases where real-time is the most important factor-such of network monitoring systems, fraud detections, etc., and there are no resource limitations, Flink must be preferred over the rest.

Spark tools had lower performance for real-time processing. Although DStream used fewer resource consumptions and performed well at the beginning of the process, it exhibited its bottleneck at emitting rate 90K which is low for real-time applications. Thus, we evaluate DStream with ‘failed’ in our benchmark. Talking about Structured Streaming of Spark, we should consider that we had micro batching for this engine. Batch size was 3 seconds, and thus, although 6 seconds of latency is above average compared with other engines, considering its throughput, we think this new tool performed well. It can be chosen for continuous processing because of its sustainability as well.

Overall, it can be said that for real-time processing, depending on system requirements and hardware characteristics, Flink or Kafka can be used for large emit rates. For continuous processing, Spark’s Structured Streaming can be adequate because of its high throughput rate.

# 7 Future Work

Although we believe that we have tested the most well-known stream processing tools, there are several engine or frameworks that deserves to be involved future replications of this master research. One of them is Heron which we have started and worked within the local, but we do not have remote results included to this master thesis. From our observations based on the local run, we truly believe that Heron can compete with today's leader engines, and this master research can be extended by its tests. Another interesting framework to be added to this benchmark could be Hazelcast Jet which we talked about in this paper previously. Besides these two, new technologies can be followed, and a researcher can include more tools since the infrastructure is very convenient for it. The very first example of them can be Apache Samza because it has good bonds with Apache Kafka. Moreover, benchmarks with new versions of engines that have been used during this master thesis would be another interesting approach. For example, newer versions of Spark starting from v2.3.0, Structured Streaming supports continuous processing which is a must have for the future replications of this benchmark.

As a part of future work, benchmarks of Flink and Kafka can be refactored by using native methods such as *grupBy*, *reduceGroup,* and *groupByKey* and be included to the benchmark.

One of the most powerful sides of our research was that we compared the tools regarding latency, throughput and resource consumption, which are three main concepts in stream processing world. However, monitoring network usages of different engines, comparing their behavior amongst nodes and clusters can be a useful way to extend this research with.

Another approach could be about the program that was used for the tests. As we have already mentioned our implementation did not have all the operations that are used very frequently in the big data world. One of them is 'join' operation, which can easily be added to this benchmark by dividing 'click' and 'view' events to be logged on two different stream pipelines. A 'join' operation can later be used to gather all information based on their 'advertisement_id.'

Finally, there are some requirements which we mentioned in the 'Related Work' section of this paper but didn't refer to our master thesis. For example, scalability and rebalancing tests could also be very interesting for big data processors. It is possible that in the future improvements we increase the total amount of nodes in the system and more metrics to evaluate these two features of the tools we have used.

# 8 References


[1] H. Rui, K. J. Lizy ja Z. Jianfeng, „Benchmarking Big Data Systems: A Review,“ *IEEE Transactions on Services Computing*, kd. 11, nr 3, pp. 580-597, 2018.

[2] J. Waite, „10 Key Marketing Trends for 2017,“ IBM.

[3] L. Doug, „3D Data Management: Controlling Data Volume, Velocity, and Variety,“ META Group Inc, Stamford, 2001.

[4] M. Kleppmann, „Designing Data-Intensive Applications,“ %1 *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*, Sebastopol, O'Reilly Media Inc, 2017, pp. 383-239.

[5] "Apache Flink Project," [Online]. Available: https://flink.apache.org/.

[6] "Apache Storm Project," [Online]. Available: http://storm.apache.org/.

[7] C. Sanket, D. Derek, E. Bobby, F. Reza, G. Thomas, H. Mark, L. Zhuo, N. Kyle, P. Kishorkumar, J. P. Boyang ja P. Paul, „Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming.,“ *IEEE*, 2016.

[8] "Apache Zookeeper Project," [Online]. Available: https://zookeeper.apache.org/.

[9] "Apache Spark Project," [Online]. Available: https://spark.apache.org/.

[10] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing.," *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation table of contents*, pp. 2-2, 25 04 2012.

[11] "Apache Heron Project," [Online]. Available: https://apache.github.io/incubator-heron/.

[12] "Apache Kafka Project," [Online]. Available: https://kafka.apache.org/.

[13] Hazelcast, "Hazelcast Jet," [Online]. Available: https://jet.hazelcast.org/.

[14] G. Can, "Introducing Hazelcast Jet - DZone Java," 11 February 2017. [Online]. Available: https://dzone.com/articles/introducing-hazelcast-jet.

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

[16] S. Michael, Ç. Uğur ja Z. Stan, „The 8 Requirements of Real-Time Stream Processing,“ *SIGMOD Rec.*, kd. 34, nr 4, pp. 42-47, 2005.

[17] R. Tilmann, F. Michael, D. Manuel, H.-A. Jacobsen and B. Gowda, "The Vision of BigBench 2.0," *Proceedings of the Fourth Workshop on Data Analytics in the Cloud*, p. 4, 31 05 2015.

[18] C. Paul, G. Bhaskar, L. Seetha, N. Chinmayi, N. Patrick, P. John and P. Meikel, "From BigBench to TPCx-BB: Standardization of a Big Data Benchmark," in *Performance Evaluation and Benchmarking. Traditional - Big Data - Internet of Things*, vol. 10080, New Delhi, Springer, Cham, 2017.

[19] "Transaction Processing Performance Council," [Online]. Available: http://www.tpc.org/.

[20] T. Ivanov, "Big Data Benchmark Compendium," in *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things*, Kohala Coast, HI: Springer International Publishing, 2016, pp. 137-146.

[21] Y. Wang, „Stream Processing Systems Benchmark: StreamBench,“ Aalto University, Espoo, 2016.

[22] Yahoo, "Yahoo Streaming Benchmark," Yahoo, [Online]. Available: https://github.com/yahoo/streaming-benchmarks.

[23] C. Boden, A. Spina, T. Rabl and V. Markl, "Benchmarking Data Flow Systems for Scalable Machine Learning," *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond,* pp. 1-3, 2017.

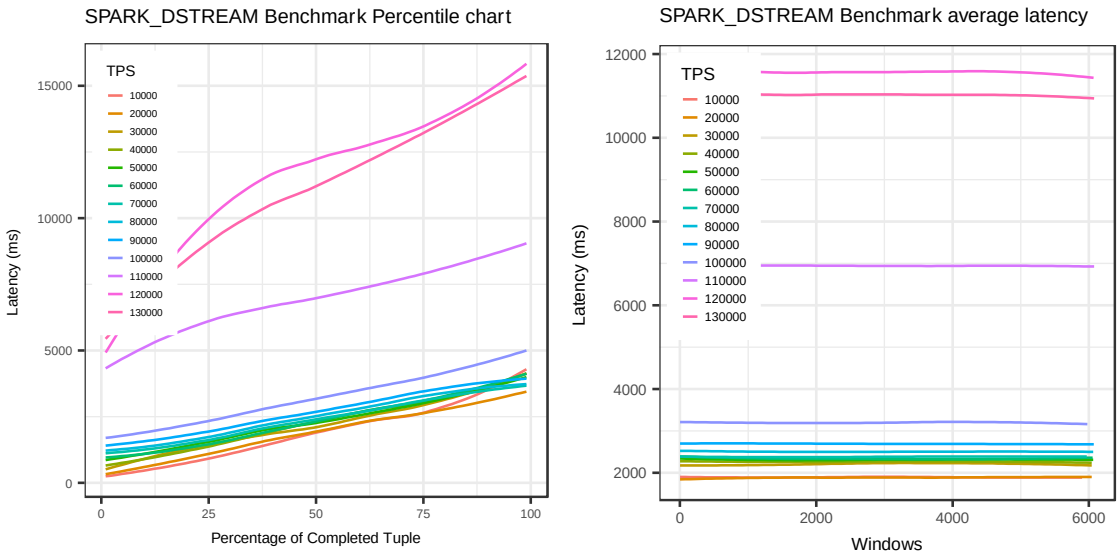
[24] D. Tathagata, Z. Matei and W. Patrick, "Diving into Apache Spark Streaming's Execution Model," 30 07 2015. [Online]. Available: https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html.

# Appendix

## I. Abbreviation

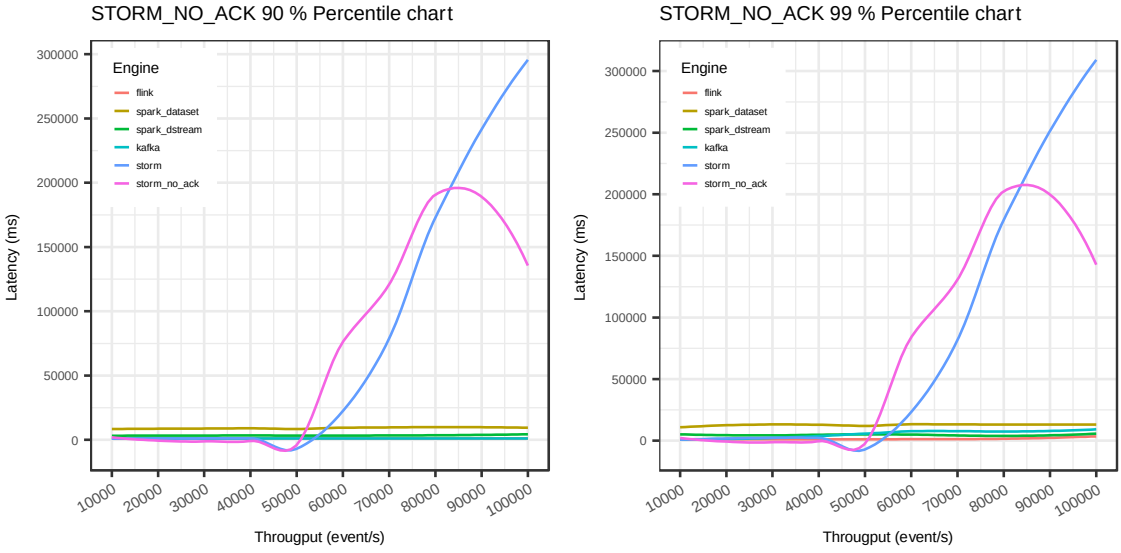
| | |
|---|---|
| **SSH** | Secure Shell |
| **SPS** | Stream Processing Systems |
| **SPE** | Stream Processing Engines |
| **TPC** | Transaction Processing Performance Council |
| **TPS** | Transaction Per Seconds |
| **POJO** | Plain Old Java Object |
| **CPU** | Central Processing Unit |
| **RAM** | Random-access Memory |
| **JSON** | JavaScript Object Notation |
| **ML** | Machine Learning |
| **CEP** | Complex Event Processing |
| **HDFS** | Hadoop Distributed File System |
| **JVM** | Java Virtual Machine |
| **API** | Application Programming Interfaces |
| **RDD** | Resilient Distributed Dataset |
| **DAG** | Directed acyclic graph |
| **SPEC** | Standard Performance Evaluation Corporation |
| **SPC** | Storage Performance Council |
| **DBMS** | Database Management System |
| **ETL** | Extract Transform and Load |
| **SQL** | Structured Query Language |

## II. Benchmark Result Charts




(a) 90% percentile latency (b) 99% Latency of Windows

Figure 25. Latency report of Spark DStream since 130K




(a) 90% percentile latency (b) 99% Latency of Windows

Figure 26. Latency report of All Engines (Storm Included)

## III. Source Code

The link to the GitHub Repository of the project is shown below:

https://github.com/elkhan-shahverdi/streaming-benchmarks

- **spark-benchmark:** Contains Spark Dataset benchmark codebase.
- **spark-cp-benchmark:** Contains Spark Structured Streaming benchmark codebase.
- **kafka-benchmark:** Kafka Stream benchmark codebase.
- **heron-benchmark:** Heron benchmark codebase.
- **storm-benchmark:** Storm benchmark codebase.
- **hazelcast-benchmark:** Hazelcast Jet benchmark code base
- **streaming-benchmark-common:** Common libraries codebase
- **conf:** Contains local and remote benchmark configurations.
- **data:** Contains data generator Clojure scripts.
- **reporting:** Contains reporting R scripts**.**
- **result:** Contains result of benchmarks and generated reports.

## IV. License

**Non-exclusive licence to reproduce thesis and make thesis public**

I, **Elkhan Shahverdi**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until the expiry of the term of validity of the copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until the expiry of the term of validity of the copyright,

of my thesis

**Comparative Evaluation for the Performance of Big Stream Processing Systems**,

supervised by Sherif Sakr,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **09.08.2018**