2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)

Big Stream Processing Systems: An Experimental Evaluation

Elkhan Shahverdi, Ahmed Awad and Sherif Sakr
University of Taru, Estonia
{*elkhan.shahverdi, ahmed.awad, sherif.sakr*}*@ut.ee*

***Abstract*— As the world gets more instrumented and connected, we are witnessing a flood of digital data generated from various hardware (e.g., sensors) or software in the format of flowing streams of data. Real-time processing for such massive amounts of streaming data is a crucial requirement in several application domains including financial markets, surveillance systems, manufacturing, smart cities, and scalable monitoring infrastructure. In the last few years, several big stream processing engines have been introduced to tackle this challenge. In this article, we present an extensive experimental study of five popular systems in this domain, namely, *Apache Storm*, *Apache Flink*, *Apache Spark*, *Kafka Streams* and *Hazelcast Jet*. We report and analyze the performance characteristics of these systems. In addition, we report a set of insights and important lessons that we have learned from conducting our experiments.**

I. Introduction

Stream computing is an emerging paradigm necessitated by new data-generating scenarios, such as the ubiquity of mobile devices, location services, and sensor pervasiveness [12]. In general, stream processing systems support a large class of applications in which data are generated from multiple sources and are pushed asynchronously to servers which are responsible for processing them. With the increasing popularity of implementing Internet-Of-Things based applications on several application domains, implementing efficient and effective analytics on the generated massive amounts of data is a crucial requirement for realizing the *Smart X* phenomena (e.g., Smart Home, Smart Hospital, Smart City, ... , etc). In practice, it has been now well-acknowledged that the world's most valuable resource is no longer oil, but data.

For about a decade, Hadoop has been the defacto standard for big data processing platforms. However, both the research and industrial communities identified various limitations in Hadoop [16] and thus it has now been acknowledged that Hadoop can not be the *one-size-fits-all* solution for the various Big Data processing challenges. As a direct result of this, we have been witnessing the uptake of a new generation of general Big Data processing systems (e.g. `Spark` [17], `Flink` [2]) in addition to a collection of engines, which are dedicated to specific verticals [15].

In practice, in the domain of large scale stream processing, one of the main limitations of the Hadoop framework is that it requires the results of each single map or reduce task to be *materialized* into a local file before it can be processed by the following tasks [16]. This materialization step supports the implementation of a simple and elegant checkpointing/restarting fault tolerance mechanism. However, it dramatically hurts the performance of applications with real-time processing requirements. Thus, several systems have been introduced to tackle the challenge of big stream processing [15].

With the increasing usage and growing popularity of these systems, it has become necessary to assess the performance characteristics of these systems in order to understand their strengths and weaknesses. In practice, currently, when an end users/organizations need to implement a streaming-based solution for processing their data, they will face the challenge of having various systems which they need to choose from. Making this decision based on the large range of available options is not an easy task as the user needs to be guided with information about the features, functionalities and performance characteristics of these systems. In addition, the selection can vary from one end user to another based on the main requirements of the application scenario, business case, available hardware resource among other factors.

In this article, we present the detailed results of examining the performance characteristics of five popular systems in this domain, namely, *Apache Storm*, *Apache Flink*, *Apache Spark* (with its two modes, namely, micro-batch and structured streaming), *Kafka Streams* and *Hazelcast Jet*. For ensuring repeatability as one of the main targets of this work, we provide access to the source codes and results for the experiments of our study[1]. Our benchmark source code has been in a modularized way that automates the various stages of the benchmarking process and is easy to extend for including other systems.

The remainder of this paper is organized as follows. Section II provides an overview of Big Stream Processing systems with a main focus on the systems under test in this study. Section III describes the details of our experimental setup in terms of used benchmarks, hardware configurations, software configurations and data collections in addition to the detailed results of our experiments and lessons learned. We discuss the related work in Section IV before we conclude the paper in Section V.

II. Overview of Big Stream Processing Systems

In this section, we give an overview of the state-of-the-art Big Stream Processing systems with a main focus on the systems under test on this study.

[1] `https://github.com/DataSystemsGroupUT/Benchmarking-Big-Streams-Systems/`

2473-3490/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDEW.2019.00-35

A. Apache Storm

Apache Storm[2] has pioneered the wave of distributed and fault-tolerant stream processing systems. It has been created by Nathan Marz before it was acquired by Twitter and then released as an Apache open source project. The Storm system has been designed following the fundamental principles of Actor theory [4]. The basic primitives that Storm provides for performing stream transformations are *spouts* and *bolts*. A spout is a source of streams that produces data as tuples. A bolt consumes any number of input streams, carries out some processing, and possibly emits new streams. In particular, bolts represents the logical components of implementing the various stream processing operations (e.g., filtering, aggregation, joins). Complex stream transformations, such as the computation of a stream of trending topics from a stream of tweets, require multiple steps and thus multiple bolts. A *topology* is a graph of stream transformations where each node is a spout or bolt. Edges in the graph indicate which bolts are subscribing to which streams. Each node in a Storm topology executes in parallel.

Storm has a Master/Slave architecture where the master node is called *Nimbus* and worker nodes are called *Supervisors*. The master node is responsible for assigning the tasks to the different worker nodes in addition to monitoring the execution of the whole job. All the messaging communication between the Nimbus and the Supervisors are managed through an Apache ZooKeeper[3] cluster. Storm has influenced the design of several other following systems such as Apache Heron[4] and Alibaba JStorm[5]

B. Apache Flink

Apache Flink[6] is one of the most popular distributed stream processing engines [2]. The architecture of Flink follows the common master/slave architecture with three main components:

- *Job Manager*: It represents the coordinator node (master node) of the distributed execution is that manages the data flow over the task managers on the slaves nodes.
- *Task Manager*: It is responsible for executing the operators that receive and produce streams, delivering their status to Job Manager, and exchanging the data streams between the operators (task managers)
- *Client*: It transforms the program code to a data flow graph which is submitted to the Job Manager for execution.

The system can connect to and process data streams from various data sources (e.g., Kafka, Flume, ZeroMQ) where data streams can be transformed using a set of data stream-based APIs[7]. Flink supports APIs for different programming languages including Java, Scala and Python. Flink can run as a completely independent framework but can run on top of HDFS and YARN as well. Flink can also directly allocate hardware resources from infrastructure-as-as-service clouds, such as Amazon EC2.

C. Spark Streaming

The Spark project has been introduced as a general-purpose Big Data processing engine that can be used for many types of data processing scenarios [17]. The fundamental programming abstraction of Spark is called Resilient Distributed Datasets (RDD) which represents a logical collection of data partitioned across machines that are created by referencing datasets in external storage systems, or by applying various and rich coarse-grained transformations (e.g., Map, Filter, Reduce, Join) on existing RDDs. The Spark system provided an API extension that adds support for continuous stream processing. Initially, Spark streaming relied on the *micro-batch* processing mechanism which collects all data that arrive within a certain period of time and runs a regular batch program on the collected data. In particular, Spark provided the *DStream* API[8] as a Spark streaming abstraction over RDDs where RDDs in a DStream contain data of a given batch interval. During the execution of the batch task, the dataset for the next mini-batch is gathered. Therefore, it can be considered as a batch processing mechanism with controlled window time for stream processing. Recently, from version 2.2.0, Spark provided the new structured streaming feature[9] that supports real-time processing instead of micro-batches [1]. In particular, Spark Structured Steaming manages streaming data as of a relational table where data is continuously appended to this table. Thus, for the end user, the programming interfaces of Structured Steaming is similar to the ones of batch processing while Spark hides and transparently manages all the details of continuous processing of streaming data.

D. Kafka Streams

Apache Kafka[10] has been introduced by Linkedin, in 2011, as a fault-tolerant distributed messaging system [8]. In particular, it provides low-latency, high-throughput publish and subscribe pipelines. Kafka is mostly used for applications that transform or react to the data streams with real-time requirements. It provides reliable data pipelines for data transfers among applications. In 2016, Kafka Streams[11] has been introduced as a new component of Kafka which is designed to support building applications that transforms Kafka input stream of topics to Kafka output stream of topics in a distributed and fault-tolerant way. It supports windowing, join and aggregation operations on event-time processing. It is designed as a library and thus it does not have any external dependency and does not require a dedicated cluster environment. In practice, streaming applications use the Kafka

[2] http://storm.apache.org/
[3] https://zookeeper.apache.org/
[4] https://apache.github.io/incubator-heron/
[5] http://jstorm.io/
[6] https://flink.apache.org/
[7] https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/datastream_api.html
[8] https://spark.apache.org/docs/2.2.1/api/java/org/apache/spark/streaming/dstream/DStream.html
[9] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html
[10] https://kafka.apache.org/
[11] https://kafka.apache.org/documentation/streams/

TABLE I
SUMMARY OF THE MAIN FEATURES OF THE SYSTEMS UNDER TEST IN OUR STUDY

	Storm	Flink	Spark DStream	Spark Structured Streaming	Kafka Streams	Hazelcast Jet
Year	2011	2015	2013	2016	2016	2018
Initial Creator	BackType	TU-Berlin	UC Berkely	Databricks	Confluent	Hazelcast
Processing Model	Real-time	Real-time	Micro-Batch	Real-time	Real-time	Real-time
Programming Model	DAG	Dataflow	Monad	DAG	DAG	DAG
Resource Management	Standalone, YARN, Mesos	Standalone, YARN, Mesos, Docker	Standalone, YARN, Mesos	Standalone, YARN, Mesos	Standalone, YARN, Aurora, Mesos	Standalone
Language Support	Java, Thrift	Java, Scala, Python, SQL	Java, Scala, Python	Java, Scala, SQL, R	Java, Python	Java

Streams library with its embedded a high-level DSL or its API depending on the application's needs.

E. Hazelcast Jet

`Hazelcast Jet`[12] has been introduced as a new stream processing engine which is built on top of the Hazelcast IMDG (In-Memory Data Grid)[13]. It has been designed as a lightweight library that can be embedded in any application to manage a data processing microservice. The library provides APIs that supports several operations including filter, group and map. To model the application, Jet uses Directed Acyclic Graphs (DAGs) where nodes represent computation steps. These computation steps can be executed in parallel by more than one instances of the streaming processor. Vertices are connected with each other via edges that represent the flow of the data and describe 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. Thus, there are always concurrent queues running amongst processor instances and they are completely wait-free. The main focus of Hazelcast Jet is to achieve high performance. Thus, it relies on the use of cooperative multi-threading, and thus, instead of the operating system, the Jet engine is the one which decides how many tasks or threads to run depending on available cores during the runtime. Regarding connectors, for now, Hazelcast Jet only supports Hazelcast IMDG, where HDFS and Kafka libraries are being actively developed.

F. Other Systems

Table I summarizes the main features of the systems under test in our study. In addition to these systems, there are some other systems that have been introduced in the domain of big stream processing. For example, *Apache S4*[14] (Simple Scalable Streaming System) is one of the early systems that has been introduced as a distributed, scalable and partially fault-tolerant system which provides the programmers with the ability to develop stream processing applications [13]. The design of S4 is inspired by both the MapReduce framework [5] and the IBM System S [6]. One of the main limitation of the runtime of Apache S4 is the lack of reliable data delivery which can be a problem for several types of applications. In S4, applications are written in terms of small execution units which are designed to be reusable, generic and configurable so that it can be utilized by various applications. The project was retired in 2014.

Apache Heron[15] has been introduced by Twitter in 2015 as a successor and re-implementation of the `Storm` project with better performance, lower resource consumption, better scalability and improvement of different architectural components including the job scheduler [9]. In particular, Heron is designed to run on top of `YARN`, `Mesos`, and `ECS (Amazon EC2 Docker Container Service)`. This is a main difference with Storm as Storm relies on the master node, *Nimbus*, as an integral component that is responsible for scheduling management. Heron's APIs has been designed to be fully compatible with Storm APIs to ease the application migration process from Storm to Heron.

Apache Samza[16] is another distributed stream processing platform that was originally developed by LinkedIn and then donated to the Apache Software Foundation. Samza uses Apache YARN for distributed resource allocation and scheduling. It also uses Apache Kafka as its distributed message broker. Samza provides an API for creating and running stream tasks on a cluster managed by YARN. The system is optimized for handling large messages and provides file system persistence for messages.

Infosphere Streams[17] is a component of the IBM Big Data analytics platform that allows user-developed applications to quickly ingest, analyze and correlate information as it arrives from thousands of data stream sources. The system is designed to handle up to millions of events or messages per second. It provides a programming model and IDE for defining data-sources, and software analytic modules called operators that are fused into processing execution units.

JStorm[18] is another distributed and fault-tolerant realtime stream computation system that has been presented by Alibaba. JStorm represents a complete rewrite of the original `Storm` engine in Java with the promise of better performance.

AthenaX[19] has been released by `Uber Technologies` as a streaming analytics platform that supports building streaming

[12] `https://hazelcast.com/products/jet/`
[13] `http://docs.hazelcast.org/docs/latest-dev/manual/html-single/index.html`
[14] `http://incubator.apache.org/projects/s4.html`
[15] `https://apache.github.io/incubator-heron/`
[16] `http://samza.apache.org/`
[17] `http://www-03.ibm.com/software/products/en/ibm-streams`
[18] `http://jstorm.io/`
[19] `https://athenax.readthedocs.io/`

TABLE II

SYSTEMS THAT HAVE BEEN USED IN OUR EXPERIMENTS IN COMPARISON WITH THE YAHOO! STREAMING BENCHMARK

System	Yahoo! Benchmark	Our Benchmark
Storm	0.9.7	1.2.1
Flink	1.1.3	1.5.0
Spark	1.6.2	2.3.0
Kafka Stream	Not Included	1.1.0
Hazelcast Jet	Not Included	0.6
Redis	3.0.5	4.0.8
Kafka Broker	0.8.2.1	0.11.0.2

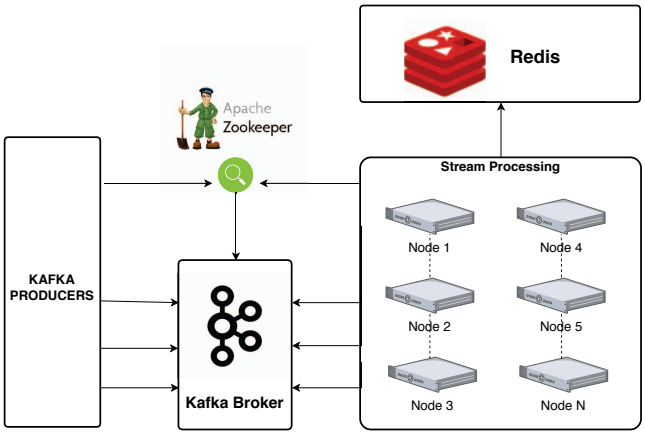


Fig. 1. Benchmark Architecture

application using Structured Query Language (SQL). `Apache Apex`[20] has been introduced as a Hadoop YARN native platform for stream processing. It provides a library of operators that help the end-user to build their streaming applications. It also provides many connectors for messaging systems, databases and file systems.

III. EXPERIMENTAL EVALUATION

A. Experimental Setup

Our experimental evaluation is based on the `Yahoo! Streaming benchmarking` [3] that simulated the application of processing advertisement clickstreams. In particular, we used the same application scenario and workloads. However, we are benchmarking new versions of the Apache Storm, Apache Flink and Apache Spark systems. In addition, we are benchmarking two additional systems that have not been included in the previous study, namely, Kafka Streams and Hazelcast Jet. Table II illustrates the comparison between the systems under test of the Yahoo! Streaming benchmarking and our benchmarking study. Furthermore, while the original benchmarking has mainly considered the throughput and latency metrics, in our benchmark, we additionally considered the resource consumption aspect.

Our experiments are executed on a cluster of Digital Ocean's CPU optimized droplets[21]. Table III describes the configuration of our experimental cluster. All the droplets

[20]`https://apex.apache.org/`

[21]`https://www.digitalocean.com/products/droplets/`

TABLE III

CLUSTER CONFIGURATION

Node Group	Count	Characteristics	Role
Load	10	2 vCPU 4 GB Memory	Generating streaming data
Stream	10	16 vCPU 32 GB Memory	Stream Processors
Message Broker	10	16 vCPU 32 GB Memory	Hosting Kafka
Coordination	10	4 vCPU 8 GB Memory	Hosting of ZooKeeper and Management Services
Data Storage	10	4 vCPU 8 GB Memory	Hosting Redis

of our cluster were running `64-bit Ubuntu 16.04.4` as their underlying operating system. Figure 1 illustrates the architecture of our benchmark where events that are generated on loader droplets are initially sent to Kafka's message broker. Once messages are in the queue, consumer nodes, hosting the stream processors, start to read the sequence of events in parallel. Finally, the results are stored on a `Redis` Database[22]. In this architecture, the coordination and service management are maintained by the Apache ZooKeeper nodes. The github repository of our benchmark provides all the scripts which are designed to support fast and easily configurable extensible benchmark execution.

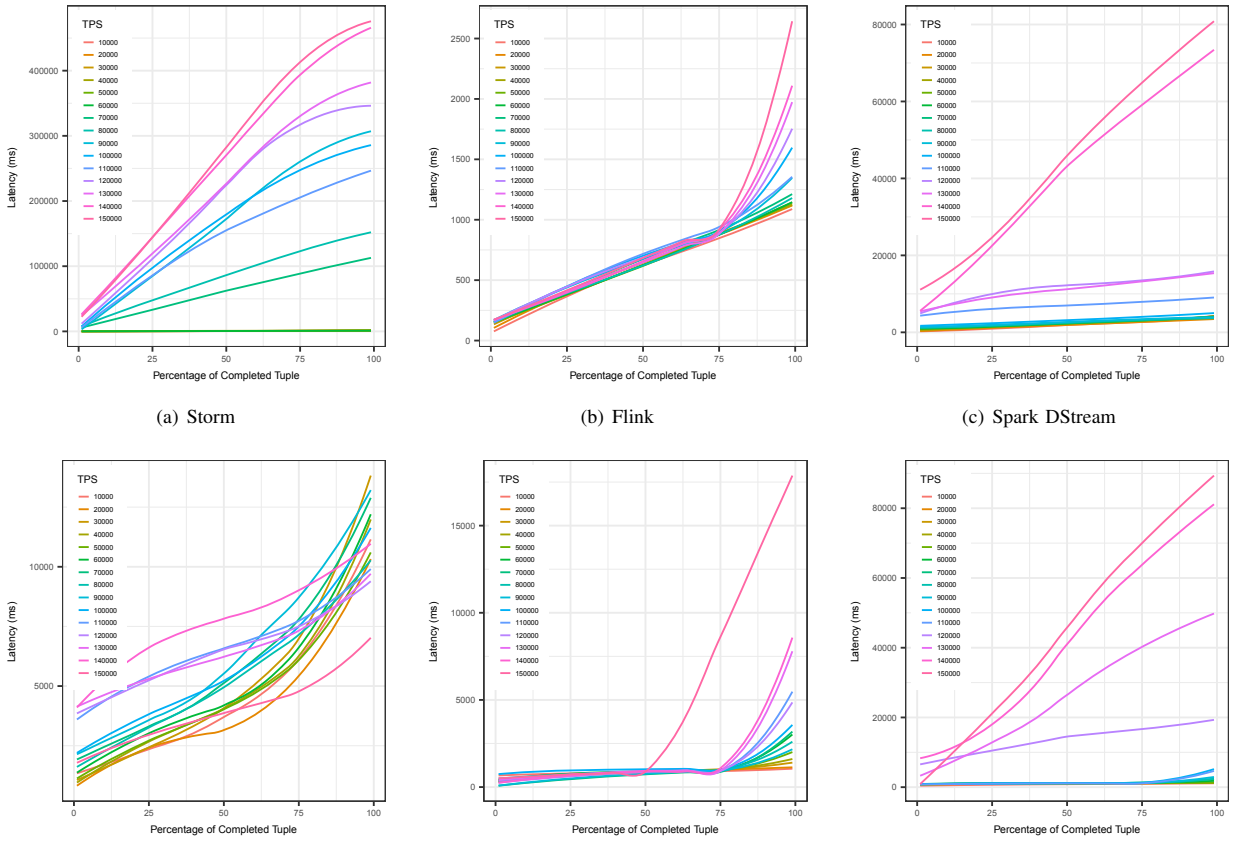
B. Experimental Results

Using the setup given in the previous section, we run the experiments for 600 seconds for data generation. Meanwhile, data are consumed by the different systems. We generated data simulating 100 campaigns. The pipeline was partitioning the data by campaign and performing aggregation over a 10 seconds window. After data generation, we wait another 60 seconds after data generation completion to allow all environments to process queued events in Kafka. After that, we shutdown the jobs and collect performance metrics as will be shown below. The experiment has been repeated with different data generation rates from $10K$ to $150K$ transaction per second (TPS) with a step of $10K$ TPS.

Latency and throughput. Fig. 2 shows the latency of the different systems with respect to the percentage of processed events (tuples) for the different data generation rates. At a 99% percentile, `Storm` provides the worst latency of about 400 seconds at data generation rate from $140K$ TPS (Fig. 2(a)). Latency is under 10 seconds until data rate of $60K$ TPS which is quite low compared with the other systems. `Spark DStream` comes next with latency above 70 seconds for data generation rate from $140K$ TPS, as shown in Fig. 2(c). With data rates below $110K$ TPS, the 99% latency drops to around 6 seconds. `HazelCast Jet` has a latency above 80 seconds for data rates from $140K$ TPS for the 99% percentile. For data rates below $110K$ TPS, the 99% latency is below 5 seconds (Fig. 2(f)).

`Flink` provides an outstanding latency below 3 seconds for 99% of processed tuples at the highest data generation rate. Until $100K$ TPS data rate, latency increases linearly. The next best latency is given by `Kafka Streams` with latency around 7 seconds for 99% completion percentile at data rate up

[22]`https://redis.io/`



(a) Storm (b) Flink (c) Spark DStream

(d) Spark Structured Streaming (e) Kafka Streams (f) Hazelcast Jet

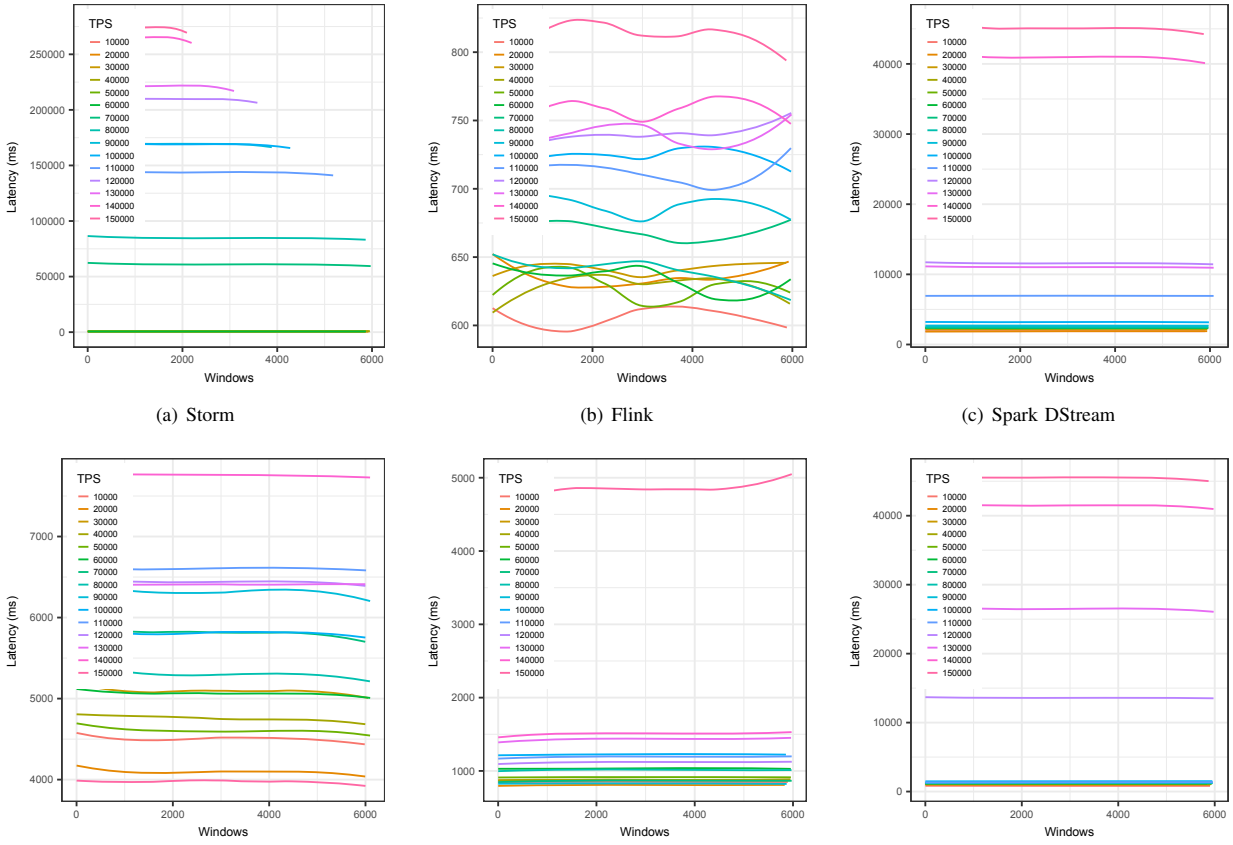
Fig. 2. Latency versus percentage of completed tuples

to $140K$ TPS. Only for $150K$ TPS, the latency jumps to about 18 seconds. `Spark Structured Streaming` is between the high-latency group of HazelCast Jet, Spark DStreams, and Storm and the low-latency group containing Flink and Kafka Streams. `Spark Structured Streaming` latency ranges between 7 and 14 seconds over the range of data generation rates. Notably, for emission rates $120K$ and above, Spark Structured Streaming latency is better than middle-range rates from $60K$ to $100K$.

Fig. 3 provides another view on latency with respect to the number of windows and data generation rates. We have 6000 windows as we run the applications for 600 seconds and group the data by 10 seconds tumbling windows partitioned by 100 campaigns. Overall, the charts goes in harmony with the charts in Fig. 2. Almost all systems have constant latency for a given data generation rate. For Storm (Fig. 3(a)), we can notice that from data rates of $70K$ and above, Storm fails to complete processing all the data after 660 seconds of application start. For a rate of $100K$ and above, Strom is left with more than 33% of windows, i.e. data, unprocessed with latency above 160 seconds. Flink shows a fluctuating latency as windows evolve (Fig. 3(b)). However, this is a very refined view as the latency scale is on a step of 50 milliseconds whereas other systems' scales are at least one order of magnitude higher.

Fig. 4 shows a comparative latency of the benchmarked systems except Storm. We excluded Storm because of the very high latency which would affect the scale. The comparative latency are provided for the 90% in Fig 4(a) and 99% in Fig. 4(b) respectively. For the 90% latency, we can notice an almost constant latency for Flink across the different data generation rates. Closely is Spark structured streaming which does not change much across the different rates. Kafka streams latency increases dramatically starting from $120K$ TPS whereas latency of Spark DStream and Jet go higher at earlier rates of $100K$ TPS. At 99% latency, cf. Fig. 4(b), we can notice a slight change in the latency of Flink and Kafka streams.

Resource Consumption Percentage Figures. 5 and 6 show the resource consumption, for the six systems when running the benchmark with a data generation rate of $60K$ and $150K$ TPS respectively, along with Kafka queues serving them with data, cf. Table III. Figures 5(a) and 6(a) report CPU consumption of the stream processing systems whereas for the two rates. Figures 5(b) and 6(b) report Kafka servers CPU consumption. Memory consumption of streaming servers is shown in figures 5(c) and 6(c), while memory consumption of Kafka servers is given in Figures 5(d) and 6(d). Storm has the highest initial CPU consumption, 8% of the available 160



(a) Storm (b) Flink (c) Spark DStream

(d) Spark Structured Streaming (e) Kafka Streams (f) Hazelcast Jet

Fig. 3. Loess Regression of Latencies

cores. After the first 200 seconds, it stabilizes around 3%. Almost all other systems' usage drops towards 0% after 600 seconds as they are almost processed their load. Storm is an exception at data rate $150K$ TPS as by the time of experiment termination, it still has unprocessed load. Looking at memory consumption, we can notice that all systems are below 15% that is $48GB$ out of the available $320GB$. Flink is an exception which goes up to 60% after the first 400 seconds. We have collected resource consumption for all data generation rates. They are not reported here for space limitation. Interested readers can find complete charts in the Github repository we pointed to earlier[23].

C. Lessons Learned

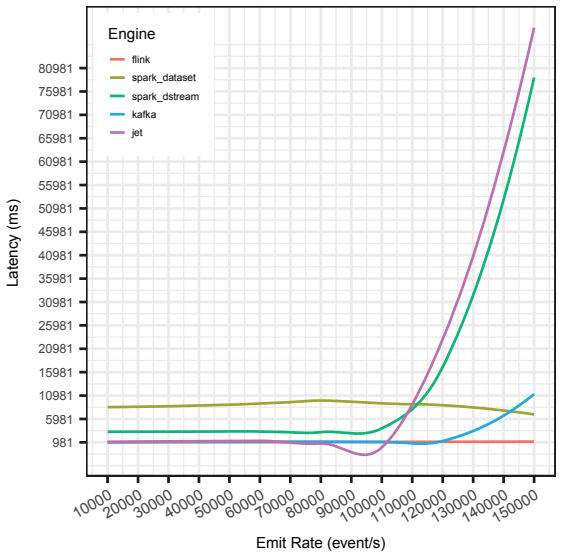
In this section, we report some of the lessons we learned during our benchmarking study. Some of the reported lessons are about the systems we included in our study while some other lessons are about the systems that we could not include in our study for various reasons.

`JStorm` is one system that we tried to include in our study. However, we faced several challenges starting from obsolete and some times missing connector libraries, mainly Kafka connector was outdated. Also, we had to invest a lot of time in building the system from its source code and tracing script and configuration files from the source code level. This is due to the poor documentation in English. Most the documentation is given in Chinese. Thus, we could not include JStorm in our study. `Heron` was also on our plan. However, it is a very complex system to setup. One needs to play with so many configuration files and several conflicting dependencies in addition to the need to lookup several log files to debug. Moreover, there was no sufficient troubleshooting documentation. We did not include `Infosphere Streams` in our study as we have been mainly focusing on the open source systems.

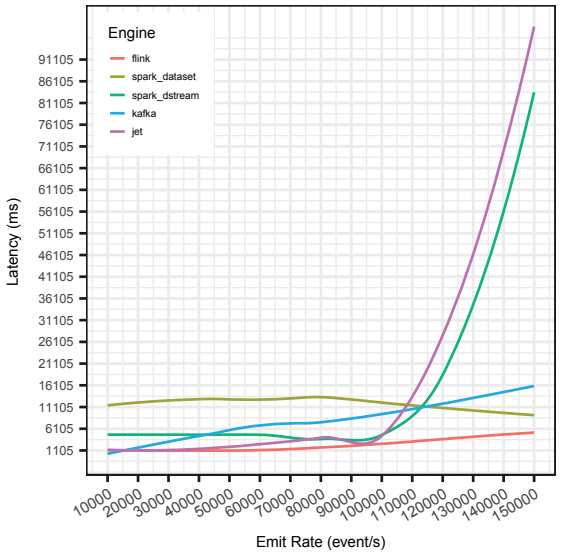
`Hazelcast` Jet is the youngest member in big stream processing systems family. It has low-latency as long as the data rates is less than $110K$ TPS. For higher rates, latency grows exponentially. The deployment and configuration of Jet is as easy as other mature systems such as Flink, Kafka streams and Spark. However, Jet runs on its own resource manager and scheduler. This may make it harder to integrate within big data processing ecosystem.

From our experience, Kafka streams was the simplest system to deploy and troubleshoot. It has decent and sufficient

[23] `https://github.com/DataSystemsGroupUT/Benchmarking-Big-Streams-Systems/blob/master/result/`



(a) 90% Percentile Latency

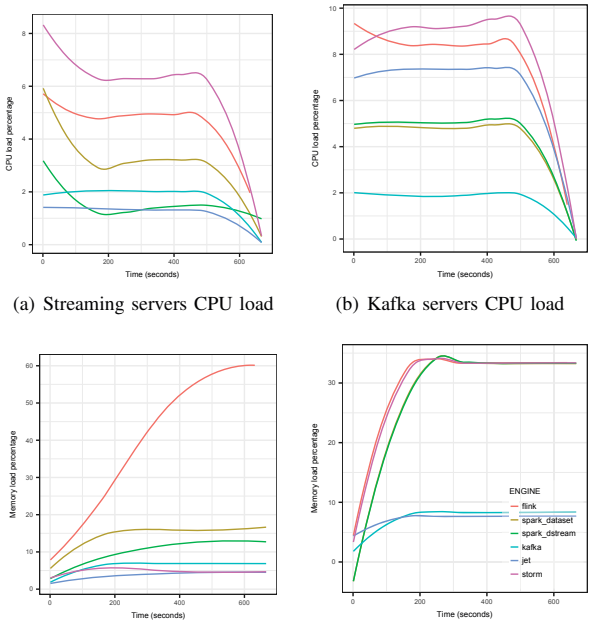


(b) 99% Percentile Latency

Fig. 4. Comparative Latency

documentation as well as decent performance in runtime.

Which system to use is a hard question with no single answer. However, if data generation late is low, e.g. up to $60K$ TPS, and you have a simple stateless pipeline, e.g. alarm or fraud detection, Storm is still a system to consider especially if resource consumption is a concern as well as stability and community support. If throughput is more important than latency and you need to uniformly use libraries like machine learning on both stream and batch data, Spark DStream represents a good choice. When low latency is the primary decision driver, Flink can effectively achieve this purpose. If the low latency goal can be sacrificed a bit for a better resource consumption, Kafka streams and Hazelcast Jet represent good options to choose from. Finally, if usability is the main concern, Spark Structured Streaming can effectively



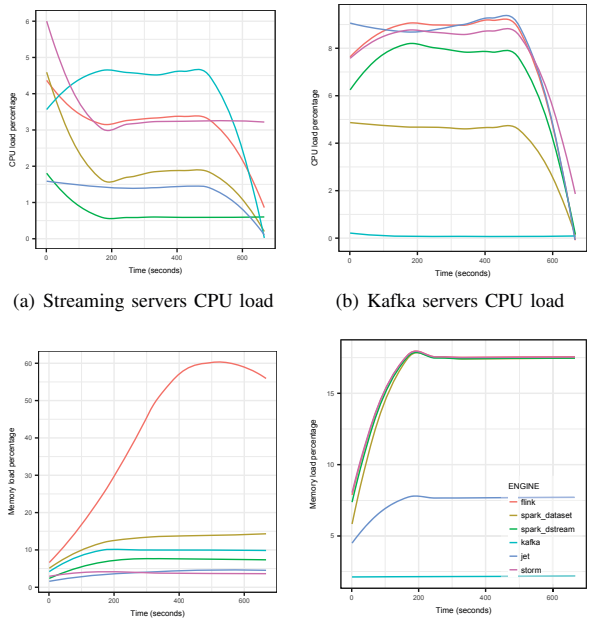
(a) Streaming servers CPU load

(b) Kafka servers CPU load

(c) Streaming servers memory consumption

(d) Kafka servers memory consumption

Fig. 5. Resource Consumption at $60K$ TPS



(a) Streaming servers CPU load

(b) Kafka servers CPU load

(c) Streaming servers memory consumption

(d) Kafka servers memory consumption

Fig. 6. Resource Consumption at $150K$ TPS

fulfill this requirement as it provides SQL-like interface to build processing pipelines.

IV. Related Work

Some work have attempted to tackle the challenge of benchmarking Big streaming systems. For example, Lu et

al. [11] have introduced the `StreamBench` benchmark[24] for assessing and comparing the performance characteristics of Big stream processing framework. They introduced seven benchmark programs (e.g., Grep, Wordcount, Distinct count, Projection) that attempt to address typical stream computing scenarios and core operations. Based on the defined benchmark programs, four workload suites have been specified for measuring various aspects of the systems under test, namely, performance workload suite, multi-recipient performance suite, fault tolerance suite and durability suite. For the performance workload suite, two main metrics have been considered: throughput and latency. The authors have used their benchmark to compare the performance of early versions of two systems: Apache Storm and Apache Spark Streaming. Qian et al. [14] have presented a study that used StreamBench to compare the performance of three systems: Apache Spark, Apache Storm and Apache Samza.

The `Yahoo! Streaming Benchmark`[25] has been designed to measure the performance of Big streaming systems using a full data pipeline that closely mimics the real-world production scenarios [3]. In particular, the benchmark design simulates an advertisement analytics pipeline and uses `Kafka` and `Redis` for the purpose of data fetching and storage in building the data processing pipeline. Chintapalli et al. [3] have used their proposed benchmark for measuring the performance of three systems: Apache Storm, Apache Flink and Spark Streaming. Lopez et al. [10] presented another study for comparing the performance of Apache Storm, Apache Flink and Spark Streaming. In particular, they conducted two experiments by simulating threats detection on network traffic to evaluate the throughput efficiency and the resilience to node failures. Recently, Karimov et al. [7] presented a study for comparing the performance of three systems: Apache Storm, Apache Spark and Apache Flink. The study focused on measuring the throughput and latency of windowed operations.

In general, there is no *one-size-fits-all* benchmarking study that can cover all systems, application scenarios and metrics. Each study considers some representative systems and focuses on some aspects to cover. In addition, each study represents a snapshot for the-state-of-the-art. However, in practice, there are continuous updates with new versions of the systems with better performance and more supported features that need to be assessed and reflected in the literature of the benchmarking studies. Pervious studies have mainly focused on benchmarking three main systems (Storm, Flink, Spark). Thus, to the best of our knowledge, our study is the first study that benchmarked this group of Big streaming systems (Apache Storm, Apache Flink, Apache Spark, Apache Heron, Kafka Streams and Hazelcast Jet) where the performance of three of these systems (Apache Heron, Kafka Streams and Hazelcast Jet) have been benchmarked for the first time. We have automated our benchmarking process to ease the process of adding other systems and workloads to our benchmark in the future.

[24] `https://github.com/wangyangjun/StreamBench`
[25] `https://github.com/yahoo/streaming-benchmarks`

V. Conclusion

We are witnessing a continuous increase in the usage of Big Data processing systems, in general, and specifically that of Big streaming systems. Big streaming systems have attracted a lot of interest from the academia and the industry for several purposes and application domains. This interest is expected to continue growing significantly with the rising momentum of IoT-based applications. These applications are expected to generate massive amounts of streaming data with crucial needs of real-time processing. To this end, we conducted an extensive experimental study for the performance characteristics of six popular systems in this domain. The results of our experiments have shown some interesting characteristics about the performance of the evaluated systems. In addition, our analysis for the detailed results has provided a set of useful insights. As a future work, we are planning to extend our benchmark to include more systems. In addition, we are planning to consider more detailed aspects of streaming systems such as load balancing, efficiency of handling our-of-order events and efficiency of performing stream join operations.

Acknowledgment

This work is funded by the European Regional Development Funds via the Mobilitas Plus programme (grant MOBTT75)

References

[1] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *SIGMOD*, 2018.
[2] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.
[3] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *IPDPS Workshops*, 2016.
[4] W. D. Clinger. Foundations of Actor Semantics. Technical report, Cambridge, MA, USA, 1981.
[5] J. Dean and S. Ghemawa. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.
[6] B. Gedik, H. Andrade, K. Wu, P. S. Yu, and M. Doo. SPADE: the system s declarative stream processing engine. In *SIGMOD*, 2008.
[7] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking distributed stream processing engines. *arXiv preprint arXiv:1802.08496*, 2018.
[8] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, 2011.
[9] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *SIGMOD*, 2015.
[10] M. A. Lopez, A. G. P. Lobato, and O. C. M. Duarte. A performance comparison of open-source stream processing platforms. In *(GLOBECOM)*, pages 1–6. IEEE, 2016.
[11] R. Lu, G. Wu, B. Xie, and J. Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *(UCC)*, 2014.
[12] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.
[13] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *ICDMW*, 2010.
[14] S. Qian, G. Wu, J. Huang, and T. Das. Benchmarking modern distributed streaming platforms. In *Industrial Technology (ICIT), 2016 IEEE International Conference on*, pages 592–598. IEEE, 2016.
[15] S. Sakr. *Big data 2.0 processing systems: a survey*. Springer, 2016.
[16] S. Sakr, A. Liu, and A. G. Fayoumi. The family of mapreduce and large-scale data processing systems. *ACM Computing Surveys*, 46(1):11, 2013.
[17] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.