HAL
open science

Experimental Study on the Performance and Resource Utilization of Data Streaming Frameworks

Subarna Chatterjee, Christine Morin

To cite this version:

Subarna Chatterjee, Christine Morin. Experimental Study on the Performance and Resource Utilization of Data Streaming Frameworks. CCGrid 2018 - 18th IEEE/ACM Symposium on Cluster, Cloud and Grid Computing, May 2018, Washington, DC, United States. pp.143-152, 10.1109/CCGRID.2018.00029 . hal-01823697

HAL Id: hal-01823697

https://inria.hal.science/hal-01823697v1

Submitted on 26 Jun 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Experimental Study on the Performance and Resource Utilization of Data Streaming Frameworks

Subarna Chatterjee and Christine Morin
INRIA Centre Rennes - Bretagne Atlantique
Email: {subarna.chatterjee, christine.morin}@inria.fr

***Abstract*—With the advent of the Internet of Things (IoT), data stream processing have gained increased attention due to the ever-increasing need to process heterogeneous and voluminous data streams. This work addresses the problem of selecting a correct stream processing framework for a given application to be executed within a specific physical infrastructure. For this purpose, we focus on a thorough comparative analysis of three data stream processing platforms – Apache Flink, Apache Storm, and Twitter Heron (the enhanced version of Apache Storm), that are chosen based on their potential to process both streams and batches in real-time. The goal of the work is to enlighten the cloud-clients and the cloud-providers with the knowledge of the choice of the resource-efficient and requirement-adaptive streaming platform for a given application so that they can plan during allocation or assignment of Virtual Machines for application execution. For the comparative performance analysis of the chosen platforms, we have experimented using 8-node clusters on Grid5000 experimentation testbed and have selected a wide variety of applications ranging from a conventional benchmark to sensor-based IoT application and statistical batch processing application. In addition to the various performance metrics related to the elasticity and resource usage of the platforms, this work presents a comparative study of the "green-ness" of the streaming platforms by analyzing their power consumption – one of the first attempts of its kind. The obtained results are thoroughly analyzed to illustrate the functional behavior of these platforms under different computing scenarios.**

***Index terms*—** Stream processing, Apache Flink, Apache Spark, Twitter Heron, Internet of Things

I. Introduction

The inception of the Internet of Things (IoT) has witnessed massive data explosion due to the exponential growth of devices and the innumerable applications running within them [1], [2]. Therefore, data-intensive systems are gaining increasing importance today. Data stream processing comprises the heart of such systems which is why several stream processing frameworks co-exist today [3], [4]. Every stream processing framework is different in terms of the set of applications that it can serve, the unit (batch, or micro-batch, or streams) and nature (bounded or persistent) of dataset that it can handle, the type of processing it offers (at-least-once or at-most-once or exactly-once), the hardware requirements, and the scalability potential. Thus, it is extremely important to choose the correct streaming platform that would serve a specific application in a given available cluster. It is extremely difficult to choose "the" streaming framework that would suit the needs of an application and as well would have resource requirements within the limits that the underlying physical infrastructure can possibly offer. Despite the existence of several work that present a comparative study of data stream processing frameworks, such type of analysis regarding "how to efficiently choose a streaming platform for an application" is still to be discovered. Therefore, deriving such knowledge towards selection of application- and infrastructure-based streaming frameworks induces research interest and attention.

In this work, we focus on thoroughly comparing and analyzing the performance and resource utilization of three streaming platforms and eventually make an inference about the various applications that are suitable to be launched within each of these platforms and the various aspects of the physical infrastructure that each can support. Based on the potential to process both streams and batches in real-time and the popularity of usage, in this work, we have chosen Apache Flink [5], Apache Storm [6], and Twitter Heron [7] as the stream processing frameworks to be studied.

A. Motivation

As mentioned earlier, several prior works exist on comparative study and analysis of streaming platforms. However, there are some observed limitations of each of these works which are discussed as follows:

(i) *Recent release of Heron*: The open-source version of Twitter Heron was released in 2016. Therefore, the existing literature on comparative studies of streaming frameworks does not include any analysis of the performance of Heron. Heron was introduced as an enhanced version of Apache Storm and so it is extremely important to analyze and compare its performance with the other streaming frameworks.

(ii) *Analysis of energy-efficiency*: Existing works illustrate the comparison of the streaming frameworks in terms of the data throughput, the utilization of the communication network, the processing latency, and the fault-tolerance. However, a study and analysis on the energy consumption of the platforms is still to be found. The energy efficiency of a streaming platform is extremely important owing to the global responsibility towards the eco-friendly use of computers and thereby reducing the environmental impact of the IT operations.

(iii) *Choice of framework for different scenarios*: A streaming platform cannot be categorized as "good" or "bad" as every platform is differently designed to process specific stream types. Each platform is unique about the nature of data sets, the variable message processing semantics, and the differential hardware expectations. Therefore, it is extremely difficult,

yet crucial, to judicially select the streaming platform that not only suits the application requirements but also abides by the resource offerings of the physical infrastructure of the host organization (or an end-user). This might be of particular interest for cloud end-users while choosing Virtual Machines (VMs) in an infrastructure cloud for Infrastructure-as-a-Service (IaaS) or for cloud providers while allocating VMs to end-users in a Platform-as-a-Service (PaaS) cloud. An insight of the rationale behind the choice of a streaming platform is not yet discussed in prior existing works and is therefore one of the major focus of this work.

B. Contributions

The contributions of the work are multi-fold and are discussed as follows:
(i) *Heron is included*: The proposed work is one of the first attempts towards a Heron-inclusive comparative study and analysis of the popular streaming frameworks. In this work, we have selected Apache Flink, Apache Storm, and Twitter Heron as the frameworks for comparison. The rationale behind this choice is that, in addition to being open-source and the popularity of their usage, all these platforms have the ability to process true streams in real-time.
(ii) *Metrics studied*: This work compares Flink, Storm, and Heron with respect to a multitude of parameters. The work focuses to analyze the performance of the frameworks in terms of the volume and throughput of data streams that each framework can possibly handle. The impact of each framework on the operating system is analyzed by experimenting and studying the resource utilization of the platforms in terms of CPU utilization, memory consumption. The energy consumption of the platforms is also studied to understand the suitability of the platforms towards green computing. Last, but not the least, the fault-tolerance of the frameworks is also studied and analyzed.
(iii) *Applications tested*: In this work, we have selected a wide variety of real-life representative applications ranging from the conventional benchmark (word count application) to sensor-based IoT application (air quality monitoring application) to statistical batch processing (flight delay analysis application). Metric are thoroughly analyzed for each application and the variation of the metric is thoroughly studied and inferred.
(iv) *Insights on choice of framework*: One of the primary contributions of this work is to enlighten the readers with the insights and rules for selecting a particular framework for a particular streaming application to be run within a given infrastructure. Every metric for a particular application is analyzed from the perspective of resource utilization and quality of service thereby inferring the suitability of the platform under variable computing scenarios. These lessons can be very helpful in cloud computing scenarios while allocating or deploying VMs in PaaS or IaaS cloud, respectively.

II. Related Work

Many stream processing frameworks are discussed in the literature [8]–[17]. Nabi *et al.* [11] performed a direct comparison of IBM InfoSphere with Storm, whereas, Samosir *et al.* [12] focused on the comparison among Storm, Spark, Samza, and Hadoop ecosystems. Another work [13] on comparative experimentation between Flink and Spark proposed benchmarks for the platforms and validated the performance of the frameworks on the proposed benchmarks. But, since Spark is essentially a batch-processing platform, the effective comparative analysis among real-streaming platforms was still absent. Over time, both Flink and Storm gained popularity as real-streaming frameworks and consequently had a wide spectrum of usage within organizations and end-users [14], [15]. Based on the popularity of usage, in this work, we have chosen both Flink and Storm along with the latest platform Heron [7] for analysis.

Of late, some works specifically compared Flink and Storm. For example, Chintapalli *et al.* [16] developed a streaming benchmark for Storm, Flink and Spark and provided a performance comparison of the engines. However, the work mainly focused on the processing latency of the tuples but did not consider other parameters related to the performance. Lopez *et al.* [17] also analyzed the architecture of Storm, Flink and Spark. The authors performed a study on the throughput and fault-tolerance of the platforms, however, the work did not study the scalability of operators of each platforms and how that hugely affects the throughput of the incoming streams. The work also did not analyze the rseource consumption of the platform in terms of CPU or memory. In 2016, the open-source version of Twitter Heron was released as an enhanced version of Storm. Kulkarni *et al.* [7] proposed the new architecture of Heron and compared Storm and Heron in terms of throughput, end-to-end latency, and CPU utilization. However, analyses on the memory consumption, power consumption, and fault-tolerance were absent. Also, considering the recentness of Heron, it is imperative to study and compare it thoroughly with the existing, widely-used streaming processing frameworks. Therefore, in this work, we compare Flink, Storm and Heron for a wide range of parameters starting from the scalability of the operators, to tuple statistics, and throughput, to resource utilization, energy-efficiency, and fault-tolerance.

Additionally, we also aim to provide insights on the usability of the frameworks for specific application requirements and examine their suitability across different physical computing infrastructure – which is one of the first attempts of its kind towards this direction. This will precisely enlighten IaaS cloud-end users to wisely choose the correct streaming platform in order to run a particular application within a given set of VMs or assist the cloud-providers to rationally allocate VMs equipped with a particular stream processing framework to PaaS cloud-users for running a specific streaming application.

III. Analyzed Platforms

In this Section, we present a brief background and architecture of the chosen platforms under comparison.

A. Apache Flink

Apache Flink is a very popular stream processing framework that comprises of a data source, a set of transformations,

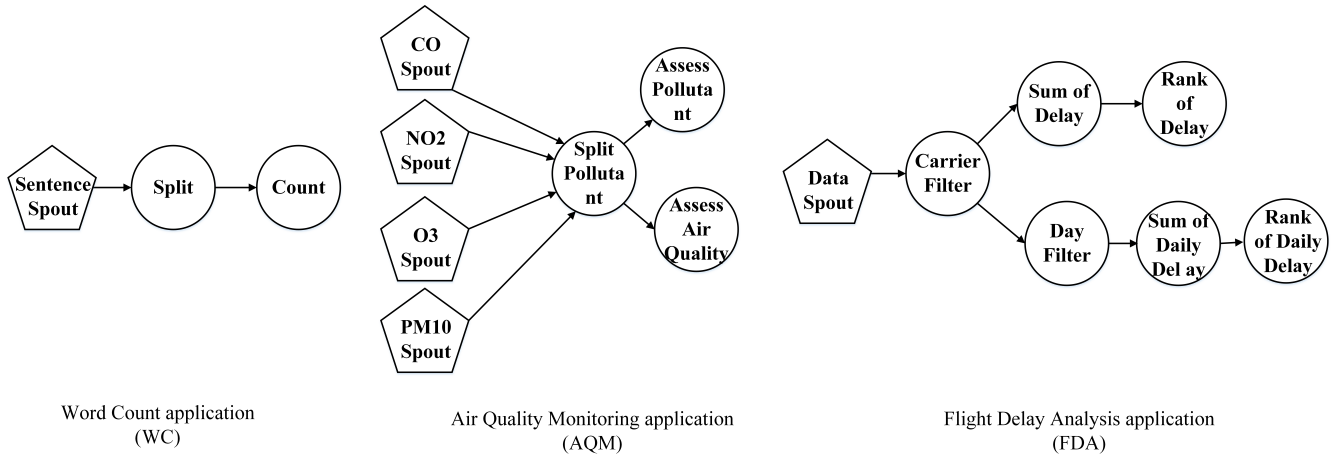


Figure 1: Topology/DAG representation of the applications used

and a data sink [5]. The master-slave architecture of Flink comprises of a Job Manager (JM) that controls the job, schedules tasks, and coordinates with the slave nodes which are also called Task Managers (TMs). A TM comprises of TM slots each of which can run a pipeline of tasks. A TM can possess as many TM slots as the number of CPU cores of the machine. Thus, the parallelism of a Flink cluster can maximally be equal to the total number of TM slots of all the TMs.

In this work, we have set up a Flink 1.2.1 cluster and every physical node was configured to use 24 TM slots and the default parallelism was set to its maximum. The number of network buffers of each TM is set to the product of the square of the number of CPU cores and four times the number of TMs* and the heap memory of a TM is set to 10 GB. The Flink data sources are configured as Kafka 0.10.2 consumers and the Kafka producers are designed to generate data streams from synthetic sources or files mounted in Hadoop Distributed File System (HDFS), depending on the application.

B. Apache Storm

A Storm topology is a DAG in which the vertices comprise of spouts and bolts. Spouts are data sources that generate data streams and bolts are the intermediate and terminal vertices that perform specific computation as per the topology [6]. The vertices of a topology are essentially run as tasks with threads called the executors which are, in turn, scheduled within multiple JVM processes called workers. In a clustered deployment of Storm, the master node called nimbus, performs jobs related to scheduling, monitoring, and management. The slave nodes are called supervisors and they execute tasks.

In our work, we have setup a Storm 0.10.2 cluster in which a single node serves as the nimbus with services of zookeeper 3.4.11 running below it. The other nodes are the Storm supervisors that execute worker processes. We have spawned multiple executor threads (30, 27, and 42 per worker process for WC, AQM, and FDA, respectively) within worker processes and have set only one task per thread to reduce multi-level scheduling of tasks, threads, and processes.

*https://flink.apache.org/faq.html

C. Twitter Heron

A Heron topology is similar to a Storm topology as it also has spouts and bolts. However, the components inside nodes are different from Storm [7]. In a clustered deployment of Heron, the master node, called the Topology Master, runs a container which contains processes that manage the entire topology and allow consistent data processing within slave nodes. The slave nodes of the cluster run several containers comprising of processes called Heron Instances (HIs) which are essentially the spouts and bolts of the topology. Heron is dependent on a scheduler to schedule these containers and a distributed file system where the topology package is to be uploaded for use by the cluster.

In our work, we have setup a Heron 0.14.5 cluster with a single Topology Master. The framework is configured with an Aurora scheduler 0.12.0 executing on top of Mesos with zookeeper services running below it. We configured HDFS 2.7.4 over the scheduler and setup Heron at the topmost layer. Every Heron container is configured to use 26 GB of RAM, 4 GB of disk, and 6 CPU cores.

IV. Data Sets and Applications Used

In this section, we present the different data sets and applications that we have used for the purpose of experimentation.

A. Word Count Application (WC)

This is the conventional benchmark of streaming platforms for counting the frequency of words from a stream of sentences. Here, we have used a synthetic data set of infinite number of sentences with an arrival rate of 1000 words per second per source thread. The topology or the Directed Acyclic Graph (DAG) representation of WC comprises of three nodes, as shown in Figure 1, where the spout releases sentences followed by the next node that splits them into words and finally counting the frequency of the words.

Table I: Setup of experimentation

Parameters	**Values**		
	Flink	Storm	Heron
Image used	Debian Jessie-x64-base-2017030114	Debian Jessie-x64-base-2017030114	Debian Jessie-x64-base-2017030114
Resources per node	2 CPUs Intel Xeon E5-2630, 6 cores/CPU, 32GB RAM, 557GB HDD, 10Gbps ethernet	2 CPUs Intel Xeon E5-2630, 6 cores/CPU, 32GB RAM, 557GB HDD, 10Gbps ethernet	2 CPUs Intel Xeon E5-2630, 6 cores/CPU, 32GB RAM, 557GB HDD, 10Gbps ethernet
Cluster size	8	8	8
Cluster name	taurus	taurus	taurus
Version	1.2.1	0.10.2	0.14.5
Topology uptime	3 hours	3 hours	3 hours
Operator parallelism	Maximally allowed	Maximally allowed	Maximally allowed
No. of workers /containers	–	12	6

B. Air Quality Monitoring Application (AQM)

The AQM application is for monitoring air quality based on selectively chosen criteria air pollutants suggested by the air quality standards and report of the United States Environmental Protection Agency (EPA) [18]. The criteria air pollutants are selected as carbon monoxide (CO), nitrogen dioxide (NO2), ozone (O3), and particulate matter (PM10). The acceptable threshold of these pollutants are set as per the meteorological measurements obtained from EPA. The DAG representation of the application is indicated in Figure 1. As the sensor data streams are generated, the individual pollutants are separated from the heterogeneous streams and are assessed with the overall air quality.

The application receives sensor based data of CO, NO2, O3, and PM10 and determines the level of hazard for both the individual parameters and the overall air quality. The sensor data is obtained from The Urban Observatory, a real deployment by the Newcastle University for obtaining long-term sensor feeds[†]. We have collected data for a span of 1 hour with a log rate at one per millisecond from environmental sensors measuring CO, NO2, O3, and PM10. The data is replicated to keep up the topology for 3 hours.

C. Flight Delay Analysis Application (FDA)

The FDA application deals with the raw data on airline on-time statistics and delay causes released by the U.S. Department of Transportation's (DOT) Bureau of Transportation Statistics (BTS). The data set comprises of data from 1987 to 2008[‡]. However, in this work, we have used the data set of only year 2008 comprising of information for more than 7 million flights throughout the year.

The FDA application ranks all the 1492 flight carriers in terms of the yearly and daily delays incurred due to seven different causes - arrival, departure, carrier delay, weather causes, National Aviation System (NAS) delay, security reasons, and late aircraft delay. The topology of the application is depicted in Figure 1. As the flight data streams are generated, the streams are filtered based on the different carriers and subsequently dealt with in two different pipelines. In the first, the carriers are ranked as per the different delays that have occurred and in the second pipeline, the carriers are ranked as per the different causes of daily delays.

[†]http://uoweb1.ncl.ac.uk/explore/

[‡]http://stat-computing.org/dataexpo/2009/the-data.html

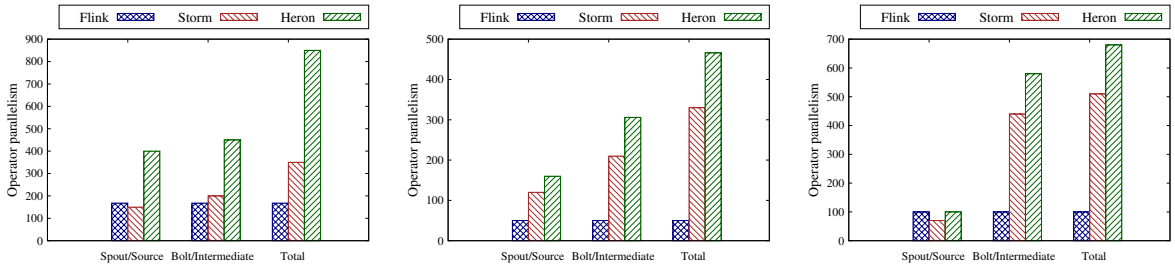
V. Performance Evaluation

In this section, we describe the details of the experimentation and analyze the results obtained. We have performed the experiments in an 8-node cluster on Grid5000 [19] and have configured every topology to stay alive for 3 hours. The details of the resources used are highlighted in Table I. The details of the methodology used for the collection of the metrics is described in the subsequent subsection.

A. Methodology

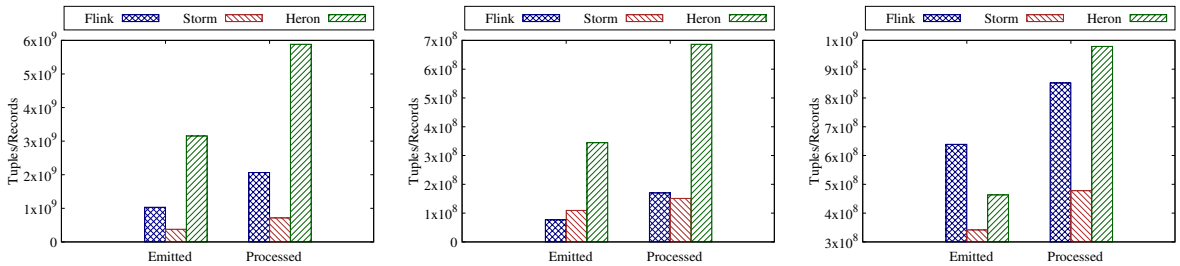
The definitions of operator parallelism, emitted tuples/records, and processed tuples/records are included in Definitions 1, 2, and 3, respectively. The metrics of operator parallelism and the tuple statistics are collected from the REpresentational State Transfer (REST) API of the respective frameworks as per the definitions through a json file which is subsequently parsed and plotted. The operator parallelism for spout/source is plotted as the summation of the parallelism of all the spouts/sources in the topology and the same is applied for bolt/intermediate operators. The tuple/record throughput is computed as the rate of processed tuples/records every minute and is also obtained from the REST API of the frameworks.

The measurements for the power consumption of the frameworks are provided by the Kwapi tool [20] of Grid5000 testbed. The power values obtained over time are converted



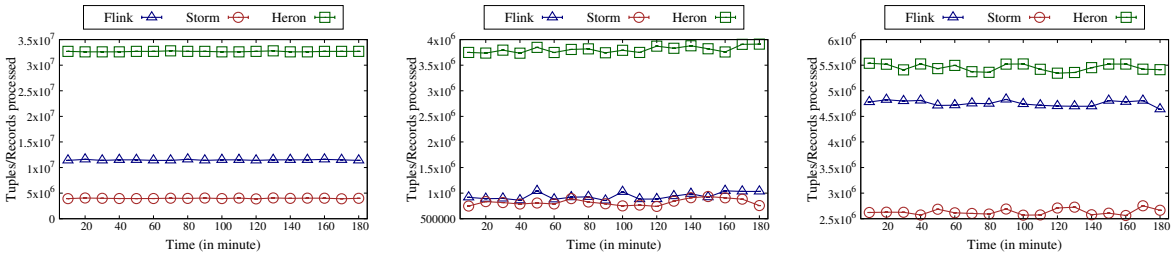
(a) WC (b) AQM (c) FDA

Figure 2: Comparative analysis of parallelism of operators



(a) WC (b) AQM (c) FDA

Figure 3: Comparative analysis of tuple statistics



(a) WC (b) AQM (c) FDA

Figure 4: Comparative analysis of tuple throughput

(by following formulative multiplication) to energy values and plotted. The energy consumption of the cluster are computed by summing up the respective consumption of every node of the cluster. The energy consumption of the slave is plotted as the average over all the slaves. The measurements for the CPU utilization and the memory consumption are provided by the Ganglia tool [21] of Grid5000 testbed[§]. The fault-tolerance of the framework is expressed in terms of the recovery time of the topology when one or more of the slaves are killed. In the experimentation, we killed $1-3$ nodes and noted the recovery time of the topologies derived from the logical plan of the topology provided by the REST API.

Having described the experimental setup and the methodology, we now study the metrics individually and analyze their variation across the three different applications.

[§]For analyses of the metrics for cluster, master, and slave perspectives, the same methodology as that of energy consumption is used.

B. Stream Processing

In this subsection, we focus on three different metrics of the stream processing frameworks – operator parallelism, tuple statistics, and tuple throughput.

Definition 1. Operator parallelism is defined as the number of source/spout threads and the intermediate/bolt threads that can be scaled up within a given experimental setup and can be efficiently supported and maintained by the frameworks without incurring overheads due to thread/task scheduling, migration, garbage collection, and excessive memory consumption[¶].

The comparative study of operator parallelism supported by the chosen frameworks is depicted in Figure 2. In every subfigure, we highlight the parallelism achieved by the spouts,

[¶]For each framework, every application topology is separately tuned by taking into account the incurred overhead thereby allowing the parallelism of operators to be at its maximum.

the bolts, and in total. In this regard, we observe that, while both Storm and Heron have variable parallelism for spouts and bolts and the total reflects the summation of the parallelism of spouts and bolts, Flink has non-variant parallelism throughout. The primary reason is that, Flink supports as many TM slots as the number of CPU cores. Hence, for a master-slave configuration of a 8-node cluster for WC application, there is a total of 168 slots exhausting the 24 CPU cores of each of the 7 slave nodes. Each slot contains a thread that can schedule multiple tasks within it. Thus, for the entire runtime of an application comprising of a series of parallel and sequential Flink operations, there is always a maximum of 168 threads with several tasks corresponding to different operations scheduled. For both of the AQM and FDA application, Flink managed to support less than its capacity of threads, as shown in Figures 2(b) and 2(c). Unlike WC which is essentially a pipelining application allowing the flow of streams without significant processing, both AQM and FDA have significant data processing and analytics. Pulling up all the 168 threads create increased lag in processing the records thereby leading to backpressure. The difference in the parallelism of Flink operators in AQM and FDA is due to the variation of the arrival rate of the streams. AQM being a sensor-based application, the arrival rate of the sensor streams is non-uniform over time, and hence the parallelism is set keeping in mind the mean arrival rate of the sensor streams (1 record per sensor per 3 ms) which is low due to the periodic sleep cycle of sensors. On the contrary, being an application for statistical analysis, FDA receives large batches of flight data after a window time of 1 ms (100000 records per source per 10 ms).

Unlike Flink, there is no hardware-controlled upper-bound of the number of operator instances for both Storm and Heron and each framework can schedule as many as the topology can handle without backpressure. For both Storm and Heron, the number of bolts is always tuned equal to at least the number of spouts for efficient processing. Since WC has less data analytics, it is possible to afford a high spout parallelism owning to the subsequent flow of streams across the bolts with very less processing. However, increasing the number of spouts to more than 200 was observed to slow down Storm due to backpressure rising from a large number of tuples that could not be handled. Further, the bolts also cannot be hugely increased because the operators in Storm are essentially not threads but tasks that needed to be scheduled within threads within Storm worker processes. Hence, the multi-level scheduling controls the parallelism of operators as well in a framework. On the other hand, in Heron, operator parallelism is the number of HIs supported. As mentioned before, HIs being separate Java processes that work for an application, the overhead due to multi-level scheduling is absent. Thus, Heron supports larger parallelism of operators, which is however reduced in Figures 2(b) and (c) with the increase in the complexity of application level data processing in AQM and FDA. Both Storm and Heron exhibits larger parallelism of operators in FDA because of the same reason as that of Flink.

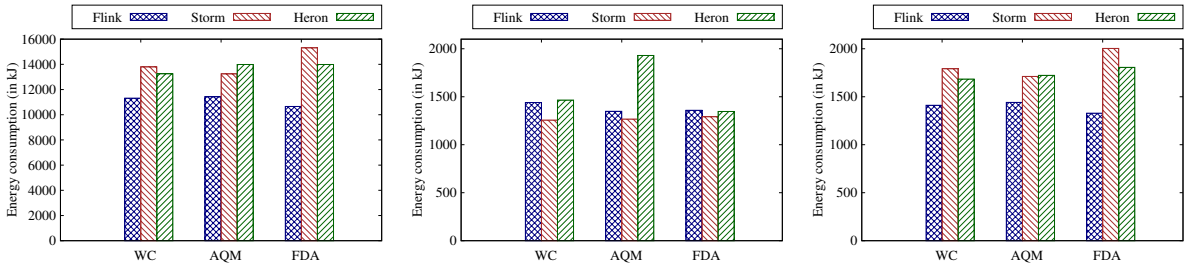
Definition 2. The number of emitted tuples/records is the number of tuples/records emitted both by the spouts/sources and the bolts/intermediate operators.

Definition 3. The number of processed tuples/records is the sum of the tuples/records emitted from the spouts/sources and the bolts/intermediate operators and executed by the final operators.

The tuple/record statistics are extremely related to the operator parallelism but are not totally controlled by it. In Figure 3(a), for WC, we observe that Flink emits and processes more records than Storm, whereas, in AQM (Figure 3(b)), Flink emits less but processes more than Storm. This implies that although Storm emits more tuples, it fails to process them at the same rate and eventually, Flink outperforms Storm in FDA application (Figure 3(c)). Heron, on the other hand, has always outperformed Storm and Flink because of its already-high operator parallelism. In FDA, we can again observe that Flink emits more than Heron but ends up processing less records. Heron tends to handle emitted tuples at a rate slower than Flink, but eventually guarantees a greater magnitude of processed tuples. In case of stream processing, if tuples are emitted too fast than they can be handled, they start buffering up in input and processing queues and in most of the cases, when the queue is exhausted, the tuples are dropped.

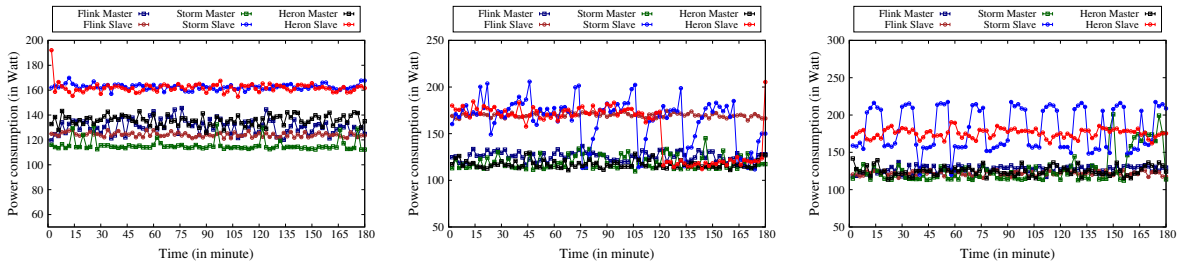
We discuss about tuple throughput in Figure 4. Although, an indication of the throughput per minute directly follows from Figure 3, the throughput patterns in WC exhibit a stable nature compared to AQM and FDA, where we observe more fluctuations of the throughput over time. The primary reason here again is that WC simply involves flow of streams whereas, for the other two applications, the stream processing and analytics make the difference in the throughput. The variation of time spent differently towards fetching streams from the input queue and processing them for the output directly have an impact on the fluctuation of the throughput.

Lessons learnt: From the point of view of the stream throughput, clearly, Heron outstands both Flink and Storm for all the application types. Therefore, Heron is able to process and manage larger volume of streams for varied applications. However, as observed in Figure 4, especially for FDA application, Flink offers approximately 80% of Heron's throughput with an operator parallelism of almost 16% of that of Heron. Thus, the throughput per unit operator is considerably higher in Flink than Heron. Now, if each slave node of the cluster comprised of more than 24 TM slots, i.e., more than 24 CPU cores per node, Flink could have achieved the same throughput in a cluster of size less than 8. However, with this cluster setup, the scalability of the number of cores per slave node do not affect Heron's throughput as every Heron container is configured with 6 CPU cores and a maximum of 6 such containers were launched. In case of Heron, the RAM is of supreme importance and is the most significant resource for processing streams. Therefore, for an application that requires a high throughput but is supported by a small cluster with a high number of CPU cores, Flink



(a) Cluster (b) Master (c) Slave

Figure 5: Comparative analysis of energy consumption



(a) WC (b) AQM (c) FDA

Figure 6: Time series analysis of power consumption

can be a better choice of a stream processing platform, than Heron. Storm, on the other hand, can be a good choice of framework for an application with a very low rate of stream emission and a low expectation of throughput. As Storm can be very easily deployed and launched across clusters, it can serve low-throughput applications really well at low costs of maintenance, which will be further discussed in subsection VF.

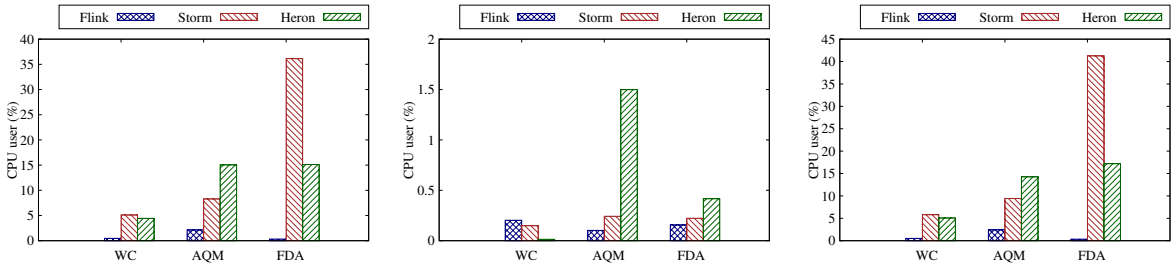
C. Energy Efficiency

In this subsection, we discuss about the energy consumption of the platforms. In Figure 5, the analyses have been separately provided for the cluster, the master node, and the slaves for all the applications and the values are provided by Kwapi tool [20] of Grid5000 testbed. Here, we can observe a general trend that the power consumption of the frameworks do not significantly vary across applications. From a cluster point of view, generally Storm consumes the maximum power in the cluster with the exception in AQM application (Figure 5(a)) in which the power consumption of Heron exceeds by approximately 7% to that of Storm. On the contrary, the power consumption of the master is the least in Storm, which directly implies that the nimbus offloads a significant amount of work to be done by the slaves. Flink is by par the most energy-efficient framework among the three. For both the master and the slaves, Storm is generally observed to be the least energy efficient followed by Heron.

We performed an analysis of the time-variation of power consumption of the frameworks across all the applications, as highlighted in Figure 6. The methodology followed for obtaining the results is explained in subsection VA. In all the applications, we observe that the plot for the power consumption of the Storm master (the nimbus) has periodic peaks because the zookeeper in the nimbus periodically receives heartbeats from all the slaves to check if they are alive. Additionally, it periodically checkpoints the state of each of the slaves and the processes running within it for issues concerning fault-tolerance and recovery. On the other hand, both Flink and Heron also performs periodic checkpointing, but we observe them to be more frequent and hence the peaks are small. For Storm, the interval between successive checkpointing is more and hence the peaks are higher owing to the larger volume of information to be checkpointed. Generally, the power consumption of a slave is significantly different from that of the master for all the platforms across all applications.

An interesting observation here is the periodic occurrence of crests ad troughs for a Storm slave node, particularly for AQM and FDA application (Figures 6(b) and (c)). The crests reach around 210 Watt and each crest is of an average duration ranging from 3 to 15 minutes for AQM and FDA application, respectively. This is primarily because of periodic garbage collection cycles of Storm for computationally intensive applications. FDA being a statistical batch processing application with high rate of streams, the cycles are more frequent in FDA whereas, in AQM the cycles are asynchronous because the sensor data arrival rate is not very high. When a slave has a certain threshold of unwanted objects and processes, it schedules the operation for garbage collection.

Lessons learnt: Overall, Flink is the most energy-efficient stream processing framework, compared to Storm and Heron. Therefore, it is well-justified to be the appropri-



(a) Cluster (b) Master (c) Slave

Figure 7: Comparative analysis of CPU utilization

ate framework for green computing IoT applications. Also, for computationally-intensive applications with data-intensive streams, the heavy power consumption can be significantly reduced by the usage of Flink.

D. CPU Utilization

We now focus on the CPU utilization of the frameworks, as shown in Figure 7. In case of Flink, the CPU utilization is the least because of the constraint on the upper-bound of the number of the TM slots to be equal to the CPU cores of the slave nodes. Therefore, a single thread corresponding to a TM slot is scheduled and there is no scope of time-sharing of CPU cores. Therefore, the CPU utilization of a core is totally controlled by the single thread that is executed within it. This affects the overall CPU utilization across the cluster (Figure 7(a)) and across all applications. In case of Storm, we can observe a gradual increase of CPU utilization of the cluster with WC being the least as it has less data analytics (Figure 7(a)) and topped by FDA with heavy batch processing workload. For Heron, we observe that WC has the least CPU utilization, but for both AQM and FDA, the CPU utilization of the cluster is not too high and also does not significantly vary with the increase in workload for FDA application. For all the applications, we observe that the CPU utilization of the master node for all the frameworks is almost negligible and not more than 2% (Figure 7(b)). The CPU utilization of the slaves (Figure 7(c)) is very low and consistent for Flink because of the afore-mentioned reason of the limitation in the number of TM slots. For Storm, the CPU utilization is really high because of the repeated garbage collection, which is also explained through Figure 6(c). Although Heron shows a gradual increase in CPU utilization, the difference in the increase is negligible. It is imperative to mention here that, one cannot anyhow scale up the operators or the workload to increase the CPU utilization because it would directly affect the memory consumption of the cluster as well, which we discuss subsequently.

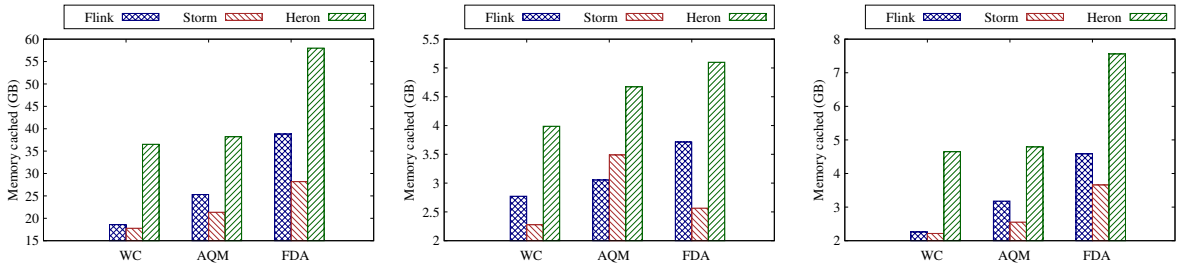
Lessons learnt: For computationally-intensive applications, especially applications demanding statistical inferences, Flink is a rational choice as the CPU utilization is really low. For light-weight applications or IoT applications, both Flink and Storm perform well with CPU utilization not more than 10%.

E. Memory Utilization

We first focus on the amount of memory cached by these frameworks across the different applications. From the point of views of the cluster, master and slaves, it can be clearly observed that Heron caches memory much more than Storm and Flink. This is the primary reason why Heron does not suffer badly from backpressure. Of course, a wrongly-designed topology would anyway lead to backpressure even in Heron, but, it has better mechanisms to handle it. Generally, the incoming tuples that cannot be handled immediately are constantly buffered and pulled from the cache memory to prevent tuple loss and guarantee accuracy of information. Although, we can see that the memory cached in the master node does not heavily differ across the platforms, but the memory cached within the slave nodes differ widely thereby affecting the amount of memory cached across the cluster.

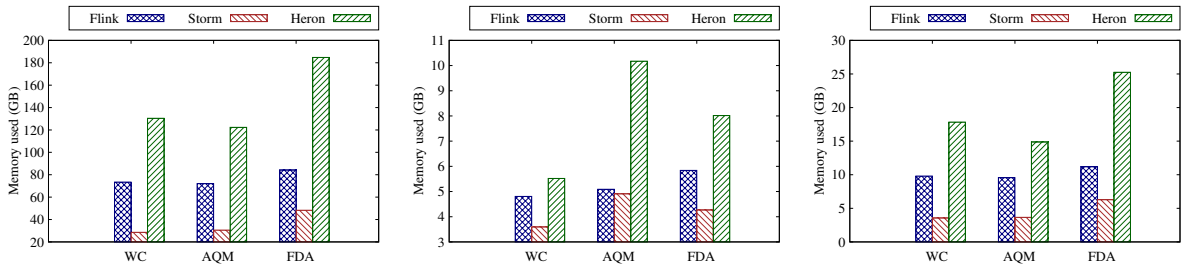
In WC application, the memory cached within a cluster (Figure 8(a)) for Storm and Flink do not vary significantly. However, it is still greater in Flink as earlier it was observed (Figure 3(a)) that Flink processed larger records of WC application compared to Storm. In AQM, although Storm emitted more tuples, but processed less due to backpressure which is also reflected by the reduced usage of cached memory by Storm, compared to Flink. This difference of memory cached is even higher for FDA application (Figure 8(a)) because Storm had significantly higher throughput of processed tuples, as shown in Figure 4(c). However, with the increase in the complexity of data processing across the three applications, we observe every framework to use larger amount of cached memory. The memory cached for the master and the slave node (Figures 8(b) and (c)) are very close for all the platforms, however, the value is larger for the slave node because of the execution of the core computing of the topology within slaves.

Having discussed about the cache memory utilization by the frameworks, we now focus on the overall memory used by these frameworks across different applications. We clearly observe again in Figure 9, that the memory consumption is the highest in Heron, compared to Flink and Storm. The main reason behind this is that, Heron is extremely resource-hungry compared to the other two platforms. Heron possesses a hierarchical architecture comprising of zookeeper at the bottom layer, followed by the resource manager, resource



(a) Cluster (b) Master (c) Slave

Figure 8: Comparative analysis of memory cached



(a) Cluster (b) Master (c) Slave

Figure 9: Comparative analysis of memory used

scheduler, distributed file system, and lastly, the stream processor. Consequently, it demands a very high threshold of the minimum resources required to launch any application thereby consuming more memory thank Flink and Storm. The memory consumption of Flink is higher than Storm because Flink contains an internal aggressive optimization mechanism for faster processing of records [22].

For both WC and AQM applications, as indicated in Figure 9(c), respectively, we observe that the difference in the memory used by the slaves for all the platforms is not very high. However, the memory used in FDA application is highest for all the frameworks because FDA being an application for statistical analysis, it involves very frequent garbage collection. Thus, the memory consumption for FDA application increases.

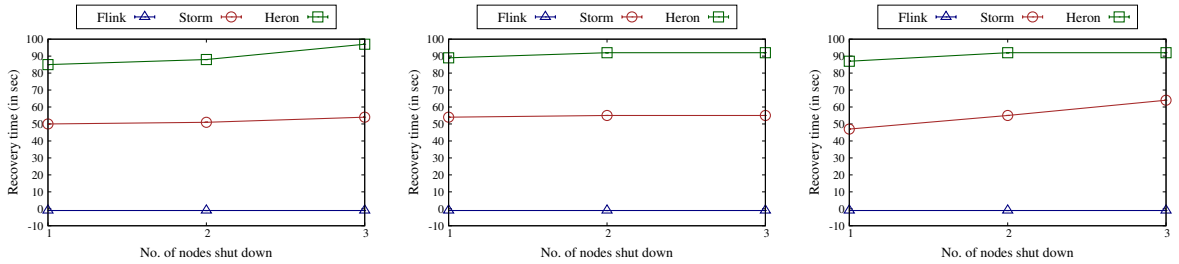
Lessons learnt: The trends of the plots of both memory cached and overall memory used, in Figures 8 and 9, indicate that Heron is a memory-hungry framework. The memory cached in Storm and Flink do not vary heavily, however, the results still indicate that Storm can be a wise choice of stream processing engine for clusters with low memory specifications. For non-batch processing applications, the memory consumption of Flink do not differ significantly and hence, in order to schedule a statistical application within a Flink cluster, it is important to check the memory offerings of the cluster. This also holds true for Heron, however, we observe that the memory consumption of the Topology Master is less than the average memory consumption of a slave node. Therefore, in a Heron-cluster comprising of nodes with heterogeneous resource capacity, the node with the minimum available memory may be selected as the Topology Master.

F. Fault Tolerance

As mentioned earlier in the methodology, the metric of fault-tolerance of a platform for a particular application is considered as the recovery time or the time interval after a slave node is killed till the time the respective responsibilities of the node are successfully delegated to another slave node.

Definition 4. For a slave node which is running t TM slots in a Flink-cluster, or w worker processes in a Storm-cluster, or c containers in a Heron-cluster, the recovery time is respectively defined and computed as the interval of time after the node is shut down till the time a new set of t threads are scheduled in the newly created TM slots or w new worker processes are spawned and launched in some other slave node(s), or c new Heron containers are created and scheduled in some other slave node(s).

The recovery time of the platforms is studied across the applications by forcefully killing 1, 2, and 3 slave nodes at a time, as shown in Figure 10. First of all, it is important to mention that, Flink does not support a sudden or abrupt shutdown of a slave node (or a TM) during an application uptime. The application simply terminates after throwing an error mentioning the sudden death of a TM. Hence, the recovery time of Flink, in terms of the death of a slave node, is indicated along the -1 value along the negative y axis of Figure 10. Now, comparing the fault-tolerance of Storm and Heron, we observe that the time taken to restore the normal operations of the topology does not vary significantly across applications for both Storm and Heron. Overall, Storm is highly fault-



(a) WC (b) AQM (c) FDA

Figure 10: Comparative analysis of fault tolerance

tolerant compared to Heron as it stabilizes the situation in approximately atleast $30-40$ seconds faster than that of Heron.

Lessons learnt: Abrupt termination of a TM in Flink leads to termination of the topology as well. This suggests that Flink may not be a wise choice for applications demanding high resiliency. Clearly, Storm serves better than Heron in terms of recovering quickly from the event of sudden termination of a slave node. But, nimbus is one of the major limitations of Storm as it is considered to be the single point of failure [7], which is resolved in Heron. However Heron, takes a higher latency to recover from node failures.

VI. Conclusion

In this work, we focus on a thorough comparative study and analyses Flink, Storm, and Heron. Earlier research works have compared streaming platforms from different perspectives, however, none of the works holistically covered all the parameters ranging from tuple statistics to resource consumption to fault-tolerance. This work focuses to present a wide range of comparative analyses for all the three stream processing platforms and investigates the energy-efficiency of the frameworks as well. The performance evaluation of this work includes a wide range of applications experimented on the platforms. Further, the work contributes by throwing insights to cloud end-users or providers about the choice of a framework for a given application within a specific physical infrastructure setting.

References

[1] M. Mozaffari, W. Saad, M. Bennis, and M. Debbah, "Mobile unmanned aerial vehicles (uavs) for energy-efficient internet of things communications," *IEEE Transactions on Wireless Communications*, vol. 16, no. 11, pp. 7574–7589, Nov 2017.

[2] S. Dong, S. Duan, Q. Yang, J. Zhang, G. Li, and R. Tao, "Mems-based smart gas metering for internet of things," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1296–1303, Oct 2017.

[3] M. Gorawski, A. Gorawska, and K. Pasterak, *A Survey of Data Stream Processing Tools*. Cham: Springer Intl. Publ., 2014, pp. 295–303.

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

[5] P. Carbone, A. Katsifodimos, . Kth, S. Sweden, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," vol. 38, Jan 2015.

[6] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *Proc. of the 2014 ACM SIGMOD Intl. Conf. on Mngmnt of Data*, ser. SIGMOD '14, 2014.

[7] 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 *Proc. of the 2015 ACM SIGMOD Intl. Conf. on Mngmnt of Data*, ser. SIGMOD, 2015, pp. 239–250.

[8] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *SIGMOD Rec.*, vol. 34, no. 4, pp. 42–47, Dec. 2005.

[9] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski, "Chromium: A stream-processing framework for interactive rendering on clusters," in *Proc. of the* 29th *Conf. on Comp. Graphics and Interactive Techniques*, ser. SIGGRAPH, 2002.

[10] F. J. Cangialosi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The Design of the Borealis Stream Processing Engine," in *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

[11] Z. Nabi, E. Bouillet, A. Bainbridge, and C. Thomas, "Of Streams and Storms," IBM Research Dublin and IBM Software Group Europe, Tech. Rep., April 2014.

[12] J. Samosir, M. Indrawan-Santiago, and P. D. Haghighi, "An evaluation of data stream processing systems for data driven applications," *Procedia Computer Science*, vol. 80, pp. 439 – 449, 2016, international Conference on Computational Science 2016.

[13] S. Perera, A. Perera, and K. Hakimzadeh, "Reproducible experiments for comparing apache flink and apache spark on public clouds," *CoRR*, vol. abs/1610.04493, 2016.

[14] https://community.hortonworks.com/questions/106314/spark-vs-flink-vs storm.html.

[15] C. Michael. (2017) Open Source Stream Processing: Flink vs Spark vs Storm vs Kafka. [Online]. Available: https://www.bizety.com/2017/06/05/open-source-stream-processing-flink-vs-spark-vs-storm-vs-kafka/

[16] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 1789–1792.

[17] M. A. Lopez, A. G. P. Lobato, and O. C. M. B. Duarte, "A performance comparison of open-source stream processing platforms," in *2016 IEEE Global Communications Conference (GLOBECOM)*, Dec 2016, pp. 1–6.

[18] https://www.epa.gov/.

[19] R. Bolze, F. Cappello, E. Caron, M. Dayd, F. Desprez, E. Jeannot, Y. Jgou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche, "Grid5000: A large scale and highly reconfigurable experimental grid testbed," *Intl J. of High Performance Computing Applications*, pp. 481–494, Nov 2006.

[20] https://www.grid5000.fr/mediawiki/index.php/Measurements tutorial.

[21] https://www.grid5000.fr/mediawiki/index.php/Measurements-tutorial/Measurements-using Ganglia.

[22] D. M. Fernandez. (2016) Comparing Hadoop, MapReduce, Spark, Flink, and Storm. [Online]. Available: http://www.metistream.com/comparing-hadoop-mapreduce-spark-flink-storm/