Journal of Big Data

RESEARCH

Open Access

Evaluation of distributed stream processing frameworks for IoT applications in Smart Cities

Hamid Nasiri[*], Saeed Nasehi and Maziar Goudarzi

*Correspondence:
hnasiri@ce.sharif.edu
Department of Computer Engineering, Sharif University of Technology, Azadi Avenue, Tehran, Iran

Abstract

The widespread growth of Big Data and the evolution of Internet of Things (IoT) technologies enable cities to obtain valuable intelligence from a large amount of real-time produced data. In a Smart City, various IoT devices generate streams of data continuously which need to be analyzed within a short period of time; using some Big Data technique. Distributed stream processing frameworks (DSPFs) have the capacity to handle real-time data processing for Smart Cities. In this paper, we examine the applicability of employing distributed stream processing frameworks at the data processing layer of Smart City and appraising the current state of their adoption and maturity among the IoT applications. Our experiments focus on evaluating the performance of three DSPFs, namely Apache Storm, Apache Spark Streaming, and Apache Flink. According to our obtained results, choosing a proper framework at the data analytics layer of a Smart City requires enough knowledge about the characteristics of target applications. Finally, we conclude each of the frameworks studied here have their advantages and disadvantages. Our experiments show Storm and Flink have very similar performance, and Spark Streaming, has much higher latency, while it provides higher throughput.

Keywords: Distributed stream processing, Smart City, IoT applications, Latency, Throughput

Introduction

Large number of embedded devices, massive volumes of data, users and applications are driving the digital world to move faster than ever. To be competitive in today's digital economy companies have to process large volumes of dynamically changing data at real-time. There are many industries from health-care, e-commerce, insurance and telecommunications with various use cases such as DNA sequencing, capturing customer insights, real-time offers, high-frequency trading, and real-time intrusion detection that have taken the use of Big Data analytics into account to make critical decisions that impact their business [1].

On the other hand, the Internet of Things (IoT) is becoming the primary grounds for data mining and Big Data analytics [2]. With the rapid growth of IoT and its use cases in different domains such as Smart City, Mobile e-Health and Smart Grid, streaming applications are driving a new wave of data revolutions. In most IoT applications the resulting analytics give some feedbacks to the system to improve it [3]. Compared to the other Big Data domains, there is a low-latency cycle between system responses which makes it necessary to process events in real-time, to derive acceptable responsiveness. In all of these domains, one of the most fundamental challenge is to explore the large volumes of data and extract useful information for future actions. In particular, this real-time exploration has to be done at massive scales.

Nowadays generated data in IoT era have several characteristics that put them in the class of Big Data [4]. Specially, in a Smart City generated data normally has the following characteristics:

- *Large volumes of data* the amount of real-time generated data by different applications in a Smart City can be in order of zetabytes.
- *Heterogeneous data sources* in Smart Cities, the data sources are diverse. For example, there are many sensors data, RFID data, cameras data, human generated data, and so on.
- *Heterogeneous data types* collected data by different devices are different in format, packet size, required precision and arrival time.

Many distributed Big Data platforms are designed to provide scalable processing on commodity clusters. Apache Hadoop [5] is one of the most popular frameworks for batch processing, which uses MapReduce [6] programming model. There are several large-scale computing architectures customized for batch processing [7]; however, they are not suitable for stream processing; because in the MapReduce paradigm all input data need to be stored on a distributed file system (like HDFS) before start to process. To address the large-scale real-time processing problem, some distributed frameworks such as Apache Storm [8], Spark Streaming [9] and Apache Flink [10] have been emerged. They process a continuous stream of messages on distributed resources with low-latency and high throughput. These recent Big Data platforms are becoming one of the most essential components of IoT ecosystems.

Smart City

It is expected to about 68% of the population to live in cities by 2050 [11]. On the other hand, using technology to improve citizen's lives makes a city into a smart one. In other words, Smart Cities offer their citizens a high quality of life by improving processes and reducing environmental footprint with technology [12]. These cities capture data using sensors, and use Big Data analytics techniques to improve environmental, financial, and social aspects of urban life. According to [13] Smart City technology, spending reached $80 billion in 2016, and is predicted to grow to $135 billion by 2021.

An intelligent combination of technological advances such as IoT, Cloud Computing, Smart Grid and Smart Building allows tracking huge amounts of information; this combination creates an intelligent system known as Smart City. A Smart City uses IoT sensors and many other technologies to connect components through a city to derive data and improve the lives of citizens and visitors. For example, in a Smart City, buildings have interconnected electrical, cooling, and heating systems monitored by an intelligent management system and a Smart Grid can be used to control on-demand generation of electricity.

System architecture for Smart City

The backbone of a Smart City is Internet of Things and the IoT itself consists of different areas such as embedded systems, ubiquitous computing and Big Data analytics. Normally, the system architecture for IoT has five layers including: Devices, Gateways, Access network, Data processing and Application [4]. So the system structure that integrates Big Data into Smart City [12] can be considered as Fig. 1. Layers in this architecture are similar to IoT system architecture layers and can be divided as follow:

1. Device layer: Set of sensors, RFID, cameras, and other devices that continuously capture physical measures such as temperature, humidity, light, vibration, location, movement, etc.
2. Data collection layer: Continuous streams of unstructured data which are being captured by device layer, will be transferred to the data nodes. In the data collection layer, all of the collected data by the data nodes will be aggregated. Aggregation of these raw data allows offering usable services.
3. Data processing layer: In this layer, the aggregated data are processed using batch or stream distributed processing engines such as Hadoop [14], Spark [15], Storm [16], and Flink [17].

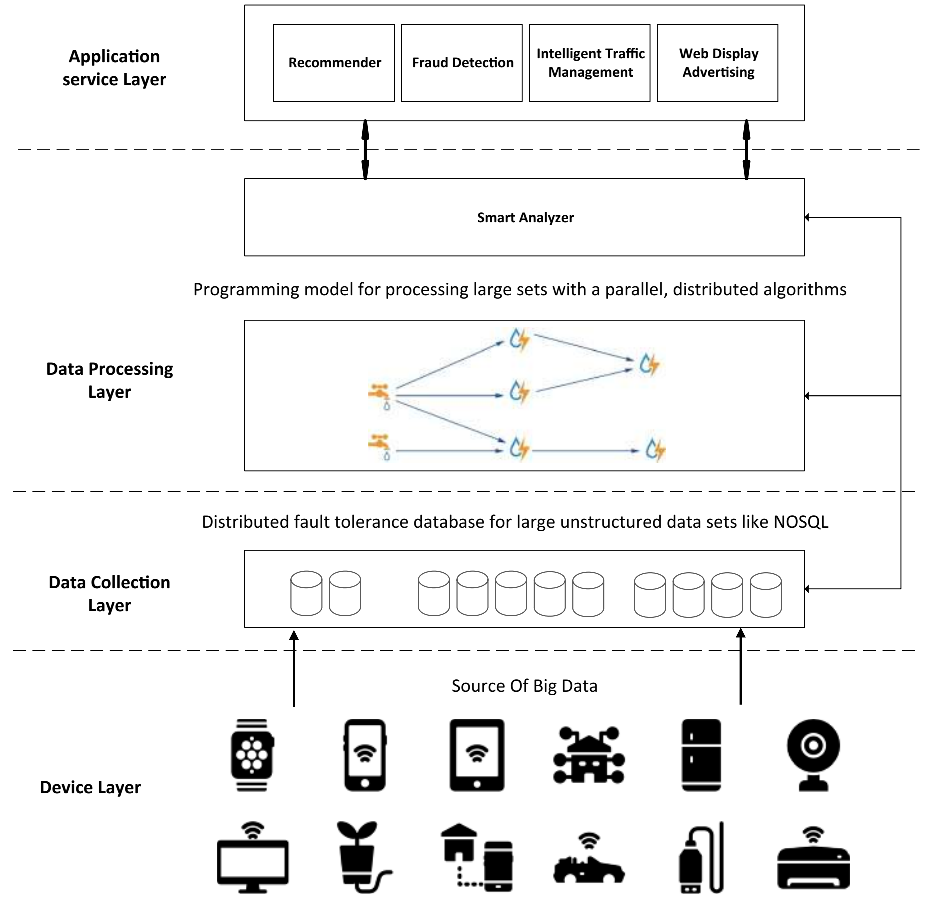

Application service Layer
Recommender
Fraud Detection
Intelligent Traffic Management
Web Display Advertising
Smart Analyzer
Programming model for processing large sets with a parallel, distributed algorithms
Data Processing Layer
Distributed fault tolerance database for large unstructured data sets like NOSQL
Data Collection Layer
Source Of Big Data
Device Layer

Fig. 1 System architecture of Smart Cities

4. Application service layer: Here many applications such as intelligent traffic management, water and electricity monitoring, disaster discovery, fraud detection and web display analysis are provided. In this layer, people and machines directly interact with each other.

In the third layer of this architecture the main requirements for Big Data processing are needed. So, choosing efficient Big Data tools and frameworks has important role to utilize the processing resources.

The goal of this paper is to examine applicability of employing distributed stream processing frameworks at the data processing layer of Smart Cities and appraising the current state of their adoption and maturity among the IoT applications. Regardless to where the data processing is happening (edge or center of processing layer) our experimental results can be taken into account to determine the best framework depending on the processing requirements of streaming applications. The main contribution of our work is providing a fair comparison among three well-known DSPFs in terms of performance, throughput, and scalability.

In comparison with our previously published conference paper [18], we have added complementary information to "Programming model of stream processing" section and studied more frameworks in "Distributed stream processing frameworks" section. The discussed frameworks are also explained with more details and some figures are used to better show their architectures. A new section is added to cover the prior works that evaluate stream processing systems from different aspects. Moreover, to show the powerful cooperation of DSPFs and message brokers (which makes the real-time streaming processing of Big Data possible) we have added some information about the integration of DSPFs with message brokers.

The evaluations are extended and we have also taken into account the resource utilization (including CPU and network usage) in our experiments. Another real-world application "Training application" is chosen among several benchmark application and is developed for all intended frameworks. We have also added a new section namely "Results and discussion" to discuss the experimental results in more details. Here, all discussions relay on our observations from two different benchmark applications, and we have tried to cover enough considerations. At the end of this section we have provided a recommendation on what kind of frameworks may be followed for certain IoT workloads.

The rest of this paper is organized as follows. "Related work" section presents the related works in this topic. "Programming model of stream processing" section explains some primary information about the stream processing models. "Distributed stream processing frameworks" section shortly introduces the most popular distributed stream processing frameworks. The next section describes the technological infrastructure and the application benchmarks used in this research process. "Evaluation" section presents the experiment scenarios and reports the obtained results, highlighting the performed benchmarks and the needed resources, in terms of performance metrics and resource usage. The obtained results are discussed in "Results and discussion" section. Finally, "Conclusion and future works" section presents the main conclusions and applicability of DSPFs in Smart Cities.

Related work

To realize the mechanism, performance and efficiency of different distributed stream processing frameworks many survey papers are published. They study the characteristics of these Big Data platforms and compare them in different conditions. In some papers [19–23] authors present a conceptual overview on several stream data processing systems such as Apache Storm, Apache Spark, etc. These works discuss general information of these frameworks such as mechanisms, programming languages, and some characteristics such as fault-tolerance; whereas no experimental evaluation is provided. So, there is no metric to assess the efficiency of such frameworks in case of IoT applications.

Authors in [24] consider some important aspects of Big Data processing frameworks such as scalability, fault tolerance, and availability and use k-means as the case study of their evaluations. However, they do not consider specific features of IoT applications such as throughput, latency and resource utilization. Paper [25] presents some information about Big Data analytics besides of some important open issues in this area. Lack of execution of some benchmark applications and hence not to pay attention to important metrics such as throughput and latency are the weaknesses of this work; whereas these parameters are taken into account in our work.

Researchers in [26] provide a valuable overview on Big Data processing frameworks and their efficiency in Smart City domain. They categorize Big Data processing frameworks according to their programming model, type of data sources, and supported programming languages. They have focused on hardware aspects of the processing cluster while some important considerations of Big Data processing in IoT applications (such as throughput and processing latency) are not covered.

Paper [27] provides some information about the Big Data processing frameworks namely Apache Hadoop, Apache Spark and Apache Flink. To have quantitative evaluation, it uses some well-known applications such as Terasort, WordCount, Grep, PageRank and K-means and provide comparative information about the performance of these frameworks in terms of latency. The first drawback of this work is the different programming model of evaluated frameworks which dramatically effects the definition of comparison metrics. In addition, in such a comparative works latency and throughput can not be reported separately; while no throughput is reported in the experiments. Moreover, other important metrics of large-scale distributed systems like resource usage efficiency are not discussed. The data sources are not clarified and the adaptation of applications and their input data for both batch and stream processing in different frameworks are not mentioned.

Programming model of stream processing

The most important aspect of a processing system probably is the programming model, because it defines the future limitations, costs and available operations. In stream processing, one of the fundamental parts of a programming model is how each pieces of new data is processed when it arrives; or later as a part of a set of new data. That distinction divides stream processing into two categories: native and micro-batch.

In native streaming model, all incoming tuples are processed as they arrive. It means that data processing happens as individual pieces before storing data in a storage media

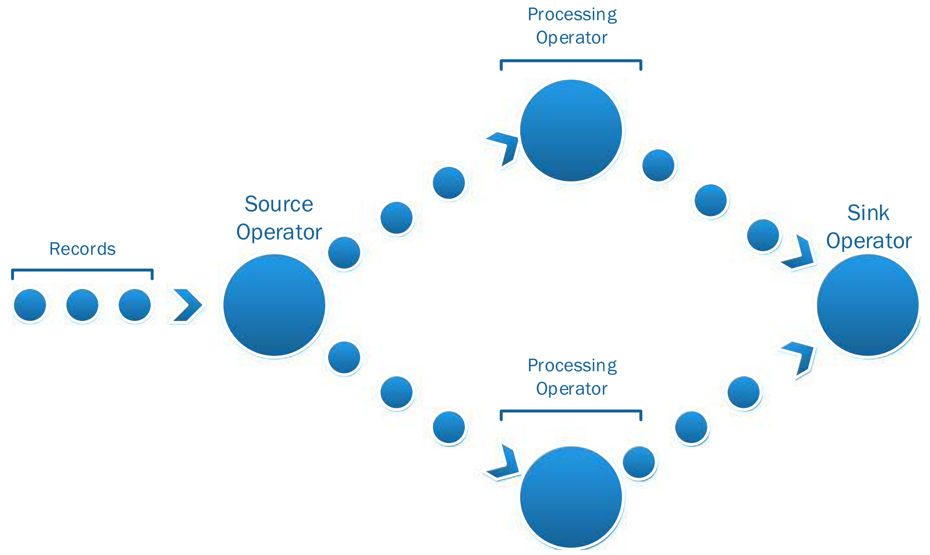


Fig. 2 Native stream processing model

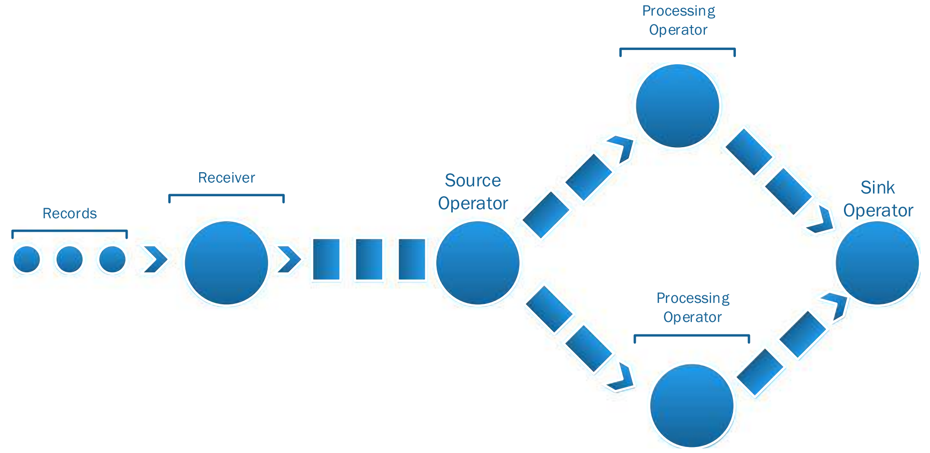


Fig. 3 Micro-batch stream processing model

rather than being processed a batch at a time. Figure 2 shows the data flow of native streaming model.

Second approach for stream processing is micro-batching. In this method, collections of tuples that are called short batches are created from incoming tuples and go through the system. These batches are created according to a scheduled time interval (e.g. every 10 min) or one or more triggered conditions (e.g. process each batch if it has more or equal than 100 KB of data). Batch processing can be useful when it is not important to have the most real time and up to date data. In Fig. 3, we can see the mechanism of micro-batch processing model.

Both methods have their own pros and cons. The great advantage of native streaming is its expressiveness. Because it takes stream as it is, it is not limited by any unnatural abstraction. Also in comparison with micro-batching systems, achievable execution time in native streaming is always much better; because the tuples are processed immediately upon arrival. But on the other hand, native streaming systems have usually lower throughput. Furthermore, fault tolerance and load balancing are more expensive, in native streaming rather than micro-batch systems [20, 28].

In micro-batching systems, splitting data streams into micro-batches reduces system's costs [29]. Some operations, such as state management, are much harder to implement, because system has to deal with whole batch [30]. Finally, it is good to mentioned that micro-batching systems can be built on a native streaming. In the following section, we will have quick look at some well-known streaming processing systems.

Distributed stream processing frameworks

Apache Storm

Apache Storm is the most popular and widely adopted open-source distributed real-time computational platform introduced by Twitter [16]. Similar to what Hadoop does for batch processing, Apache Storm performs on unbounded streams of data in a reliable manner. A Storm cluster consists of two types of nodes: master node and worker node.

Master node runs a daemon called Nimbus, which is the central component of Apache Storm. The responsibility of Nimbus is distributing codes and assigning tasks to the worker nodes. It also monitors health of the cluster by listening to the produced heartbeats by worker nodes and re-assigns failed tasks if needed. Worker nodes do the actual execution of the streaming application. Each worker node runs a daemon called Supervisor, which works by Nimbus and starts and stops the worker processes when required.

The configuration of worker nodes determines how many slots they can provide for the cluster; so each worker node, may run one or more worker processes, depending on its number of slots [8]. Since Apache Storm is not able to manage its cluster state, it relies on Apache Zookeeper [31] for this purpose. Zookeeper simplifies communication between Nimbus and Supervisors with the help of message acknowledgments, processing status, etc. Figure 4 represents the architecture view of a Storm cluster.

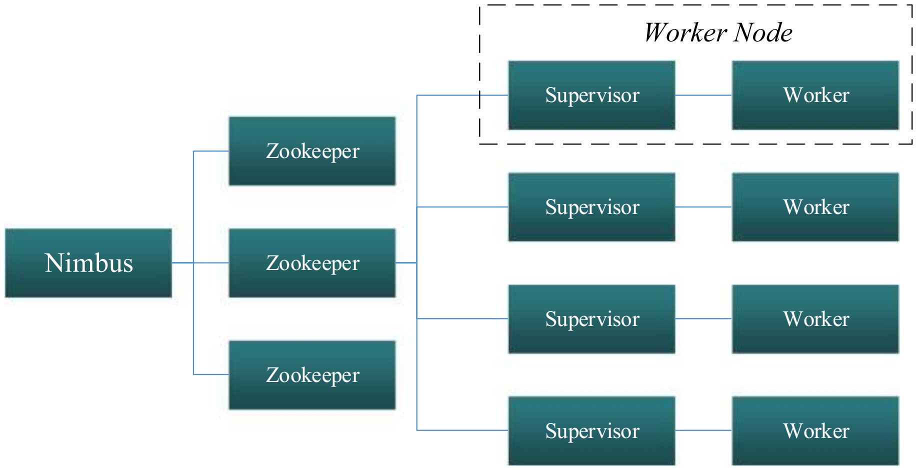


Fig. 4 High level architecture of Apache Storm

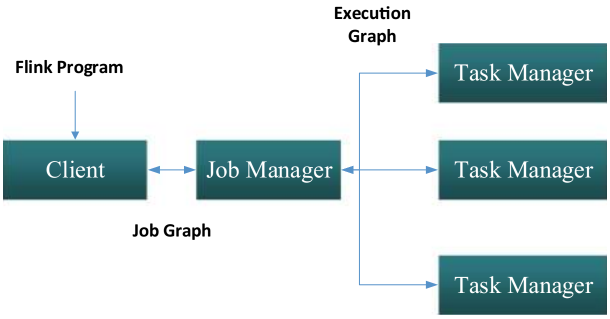


Fig. 5 High level architecture of Apache Flink

Apache Flink

Apache Flink is an open-source streaming platform, which provides capability to run real-time data processing pipelines in a fault-tolerant way at a scale of millions of tuples per second [17]. Flink is based on native stream processing rather than processing micro-batches. Flink processes the user-defined functions code through the system stack. It has master-slave architecture, consists of a Job Manager and one or more Task Manager(s) [10].

The duty of the Job Manager is coordination of all the computations in the Flink system, while the Task Managers are being used as workers and execute parts of the parallel programs. Figure 5 shows the architecture of Apache Flink which is completely clear to the programmers and they just need to know how to work with the API to write programs [17]. Flink is famous for its ability to compute common operations such as hashing, very efficiently.

Apache Spark

Apache Spark is a widely used, highly flexible engine for batch-mode and stream data processing that is well developed for scalable performance at high volumes [15]. To maximize the performance of Big Data analytics applications, Spark supports in-memory processing, but it also can perform disk-based processing when data sets are too large to fit into the available memory. The architecture of Apache Spark is based on the following components [29]:

- Spark Driver
- Cluster Manager
- Executors

A Spark application takes data from a collection of sources (HDFS, NoSQL and relational DBs, etc.), then applies a set of transformations on them, and finally executes an action that generates meaningful results. The Spark Driver, which is the master node in a Spark cluster, converts the application into a set of tasks to be executed by a set of Executors. Once the Spark Driver has converted the application into a set of tasks, it passes them to the Cluster Manager for distribution.

The purpose of the Cluster Manager is to understand where the intended data resides and distribute tasks to the most appropriate server in the cluster. Each server in the cluster has an Executor that receives tasks from the Cluster Manager, executes them, and then returns the results back to the Cluster Manager. It is then the Cluster Manager's responsibility to combine the results from all Executors and make a response for the Spark Driver. In Fig. 6 the cluster view of Apache Spark is shown.

Apache Heron

Twitter open sourced Heron is re-imagined Storm with emphasis on higher scalability and better debug ability [32]. The goals of developing Heron are handling petabytes of data, improving developer productivity, simplifying debugging and providing better efficiency [33]. As it is shown in Fig. 7, Heron consists of three key components:

- Topology Master: it is responsible for managing a topology from its submitting moment until it is killed.

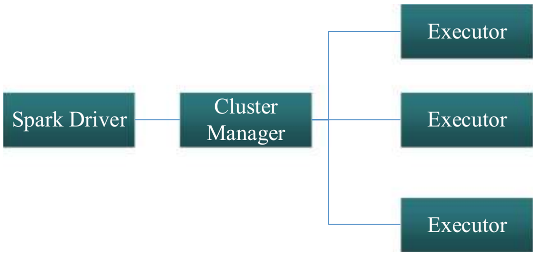


Fig. 6 High level architecture of Apache Spark

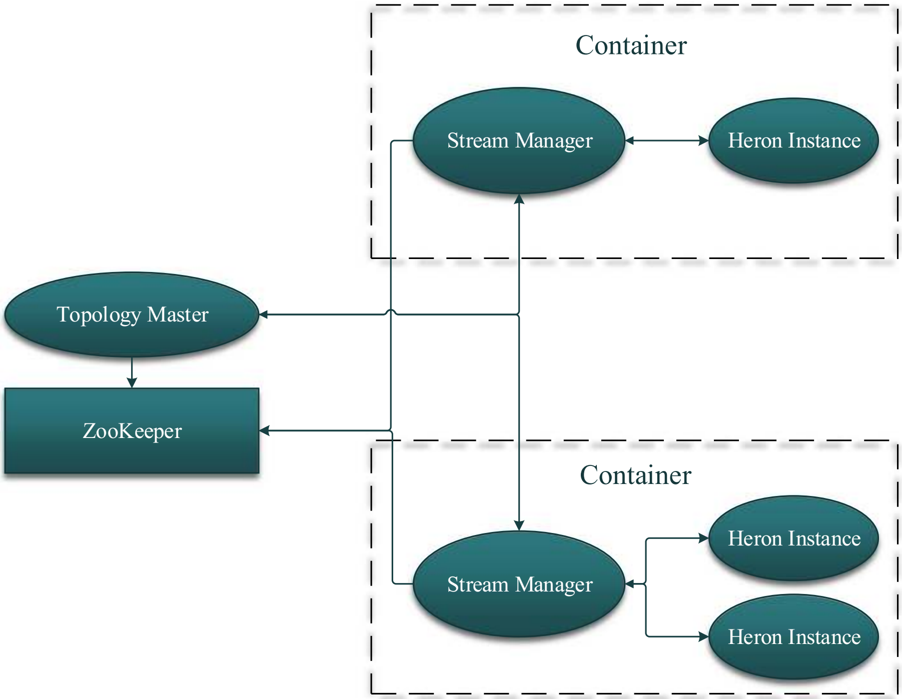


Fig. 7 High level architecture of Apache Heron

- Stream Manager: its duty is managing the routing of tuples between topology components.
- Heron Instance: it is a process that executes a single task and allows for easy debugging and profiling.

Samza

Apache Samza is formed by combination of Apache Kafka [34] and YARN [35] to perform computation over data streams. The main goals of Apache Samza are having better fault tolerance, processor isolation, security, and resource management [36]. Input streams of tuples are decomposed and partitioned so that data flow graph is created. Each graph contains multiple streams and jobs, which let user to partition the streams and parallelize execution of operators across a cluster of machines [37]. Figure 8 shows the high level architecture of Apache Samza.

Akka

Akka is an advanced toolkit and message-driven runtime based on the Actor Model that helps development teams to build the right foundation for successful micro-services architectures and streaming data pipelines [38]. In Akka, the communication between services uses messaging primitives that optimized for CPU utilization, low latency, high throughput and scalability.

Akka Streams is a streaming dataflow abstraction on top of Akka Actors, giving developers a better way to define their workflows. As a founding implementation of the Reactive Streams specification, Akka Streams adds benefits of back-pressure and type-safety to the development of streaming applications [39]. Akka Streams and the underlying Akka Actor model are ideal for low-latency processing of data streams.

Integration with message brokers

DSPFs and message brokers naturally complement each other, and their powerful cooperation enables real-time streaming processing of Big Data. In most streaming applications, message brokers such as Apache Kafka, RabbitMQ, ActiveMQ, and Kestrel, prepare data streams instead of connecting to the data source, directly. They employs a pull-based consumption model that allows an application to consume data at its own rate and rewind the consumption whenever needed. They usually provide integrated distributed support and can scale out [40]. Furthermore, the message brokers allow data

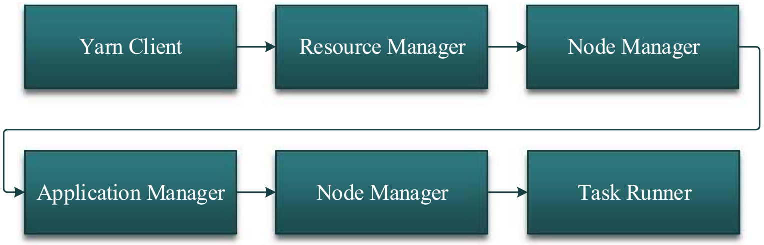


Fig. 8 High level architecture of Apache Samza

replication to make it available for several systems and persists data coming from various sources.

Methods/experimental

In all experiments the reported values for all metrics are measured under the same conditions and a specific input rate. For each run 5 min of warm up execution is considered before the metrics were captured from the Web UI of each framework; because the throughput of the streaming application that is being evaluated should have stabilized and converged. According to our experiments after 5 min all applications have stable state in the target frameworks. All of the benchmarks were performed using default settings for Storm, Flink, and Spark Streaming and we have avoided configuration tweaks to tune them.

Experimental setup

For our experimental evaluation, we use a cluster with 8 homogeneous worker nodes. Table 1 shows the specification of our master and worker nodes. Master node runs the job manager and its corresponding demons, and manages all employed worker nodes. Depending on our experiment we use 1, 2, 4, or 8 worker nodes. In all experiments we have used Apache Storm 0.10.0, Apache Flink 1.2.0 and Spark 2.2.1 on top of Ubuntu 16.04 operating system.

Moreover, since Apache Storm supports two message processing guarantee semantics, we have run the benchmark applications under both at least once and at most once (No-Ack) semantics. At most once semantics happens when the acknowledgment mechanism is disabled.

Application benchmarks

In our experiments two different streaming applications are used to make sure that our evaluation is reliable. These applications are visually depicted in Figs. 9, and 10. The *Advertising* application is a sample IoT streaming program from Yahoo Streaming Benchmarks [41]. In this program each incoming tuple is processed by the following operators respectively.

Table 1 System characteristics

Parameter	Master node	Worker node
Processor	Core i7	Xeon E5-2650
Number of cores	4	2
Memory	8 GB	4 GB
Operating system	Ubuntu 16.04	Ubuntu 16.04

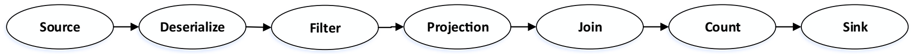


Fig. 9 Layout of Advertising application

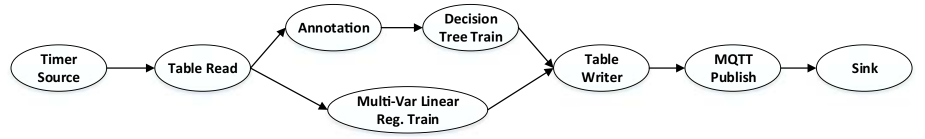


Fig. 10 Layout of Model Training application

- Source: This component reads the tuples from Kafka message broker and prepares them as standard data units according to the data processing model of intended framework.
- Deserialize: Divides the input JSON string to some meaningful fields.
- Filter: Filters out irrelevant tuples based on their type.
- Projection: Remove unnecessary fields.
- Join: Joins tuples by a specific field with its associated information of another field.
- Count: Take a windowed count of tuples per joined field and store them.

The second application is based on the *Model Training* application benchmark from RIoTBench [42]. This application consists of some micro benchmark tasks which are mainly developed for Apache Storm. We have partially changed it to be able to work off-line. It uses a timer to periodically trigger a model training run as follows:

- Timer Source: This component simulates the model training trigger.
- Table Read: At each run it fetches data from the stored table, available since the last run.
- Multi-Var Linear Reg. train: It uses the fetched data to train a linear regression model.
- Annotation and Decision Tree: Fetched tuples from the Table-Read operator are also annotated to allow a decision tree classifier to be trained.
- Table Write: Stores the trained model files.
- MQTT Publish: Publishes all updates to the database records to the MQTT broker.

Both *Advertising* and *Model Training* applications, are written for Storm, Flink and Spark Streaming according to the operators of their dataflow graph. The programming model of both Storm and Flink is based on directed acyclic graph (DAG) so the structure of the applications for these frameworks is similar. But Spark Streaming is a modified version of Apache Spark and its programming model is something between batch and stream processing, called micro-batch.

Evaluation

As explained in "Distributed stream processing frameworks" section there are many distributed stream processing frameworks which can be employed as the data analytics layer of IoT applications in Smart Cities [18]. However, Apache Storm [8], Apache Spark Streaming [29], and Apache Flink [10] are the most popular DSPFs for real-time processing [21]. So, we evaluate these three open source and community driven frameworks in terms of performance and scalability, using two streaming applications.

The most commonly-used quantitative performance measures to evaluate efficiency of DSPFs, are latency and throughput [19]. In our experiments, we consider these metrics to compare performance and scalability of intended DSPFs. Considering each streaming application as a dataflow graph, we use the following definitions for latency and throughput. We also measure CPU and Network utilization, to realize how efficient each DSPF works.

Latency: For a tuple that is consumed by an application graph, latency is the time it took for that tuple to be processed by intermediate vertexes and transferred (including the network and queuing time) between them. This latency is known as end-to-end latency. As the end-to-end latency of each tuple may vary depending on the tuple size and type, resource allocation, and input rate, we consider the average latency in our evaluations.

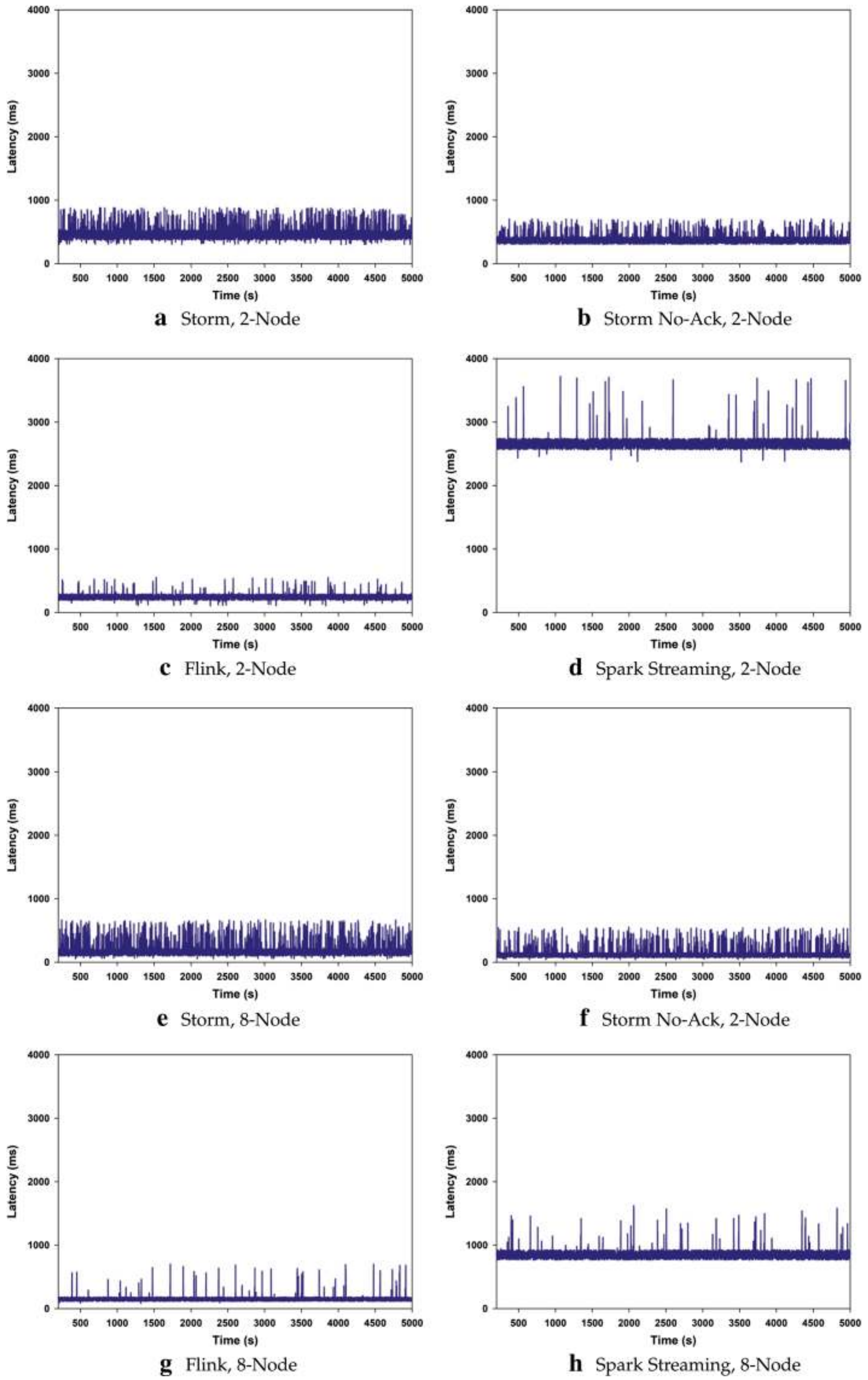
Throughput: Depending on the operation of each vertex of the application graph, for each incoming tuple, different number of output tuples may be emitted. Normally the aggregated output tuple rate of all vertexes is considered as the overall throughput. But, when we are talking about different frameworks, we face different implementation of the application graph. So, we cannot use the overall throughput as a unique metric. In our evaluation, we consider the total number of received tuples by the sink operator of the application at 1 s as the processing throughput (all implementations of the applications have a sink operator which collects the output tuples).

In our experiments, Apache Kafka [34] is used to produce desired data streams, so we can see the behavior of the target DSPF in the real world conditions. There are some classes and API to integrate Kafka with DSPFs. The classes are used to track the Kafka brokers dynamically by maintaining the details in ZooKeeper, or statically set the Kafka brokers and its details. Provided API can be used to define configuration settings, fetching the messages from kafka topic and emitting them into DSPF as tuples.

As the data partitioning of message brokers mostly is not chosen wisely, may the raw data is partitioned in a different way than the streaming system requires it. To handle this issue some technique are used, but trades some overhead and performance lost. So, when we are benchmarking a distributed stream processing system, the data exchange between the message broker and the streaming system may become the performance bottleneck. We ran the Kafka in a separated machine and took enough instances from the spout to get sure that the message broker has no effect in our experiment results and never becomes bottleneck.

Latency evaluation

To compare the performance of intended DSPFs in terms of latency, we measure end-to-end latency for each tuple, and calculate the average latency in desired time slices. For a particular input rate (*Advertising*: 400 k tuple/s, *Model Training*: 1 k tuple/s) we have executed the applications by different number of worker nodes. In Figs. 11 and 12 we can see the latency results obtained by Storm, Storm No-Ack, Flink and Spark Streaming respectively; using 2 and 8 worker nodes.



a Storm, 2-Node **b** Storm No-Ack, 2-Node

c Flink, 2-Node **d** Spark Streaming, 2-Node

e Storm, 8-Node **f** Storm No-Ack, 2-Node

g Flink, 8-Node **h** Spark Streaming, 8-Node

Fig. 11 Latency of intended frameworks with different number of worker nodes for Advertising application

Throughput evaluation

According to the mentioned above definition for throughput, we consider the total number of received tuples by the sink operator at 1 s as the processing throughput, to have a fair comparison among different frameworks in terms of throughput.

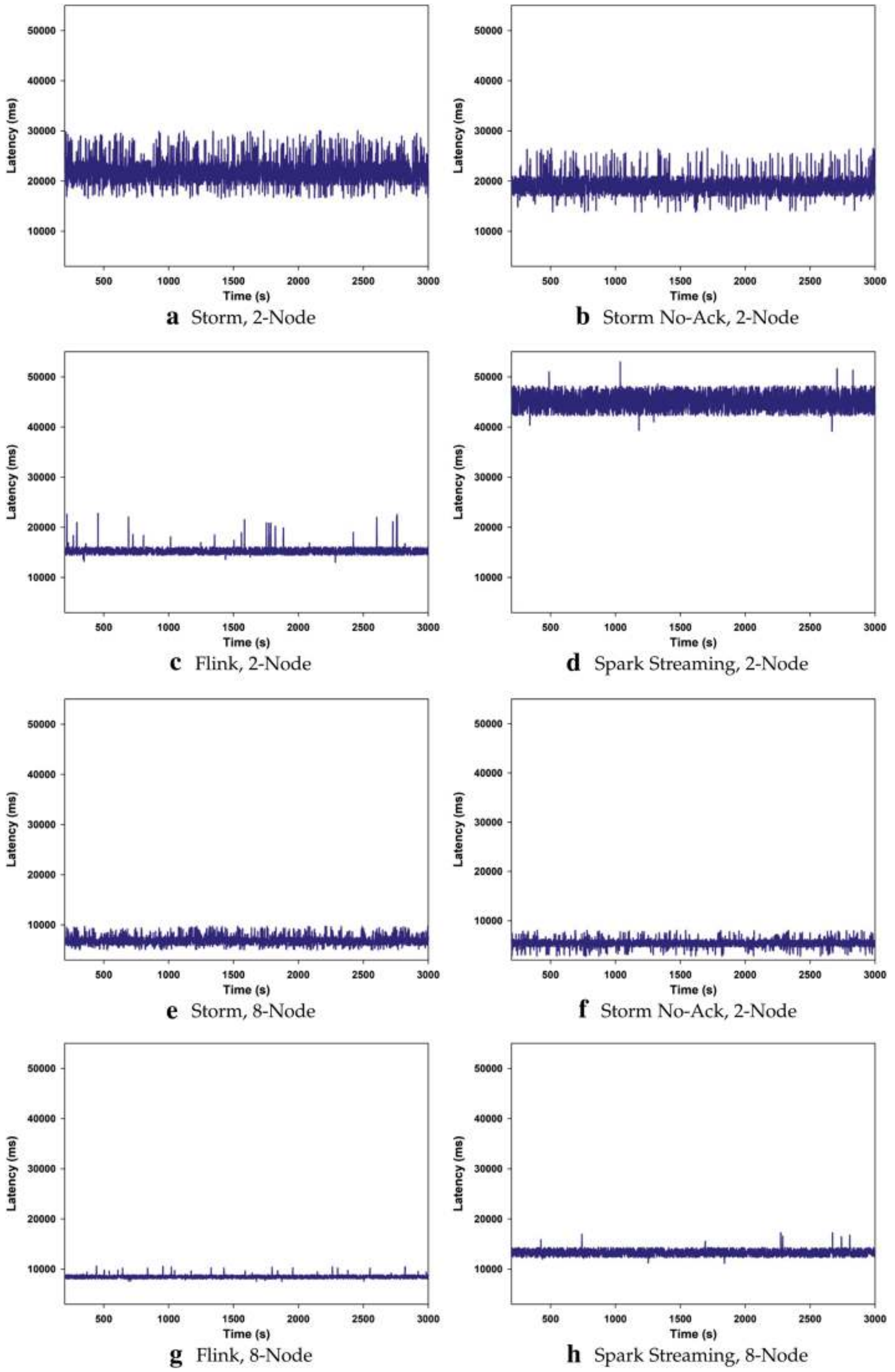
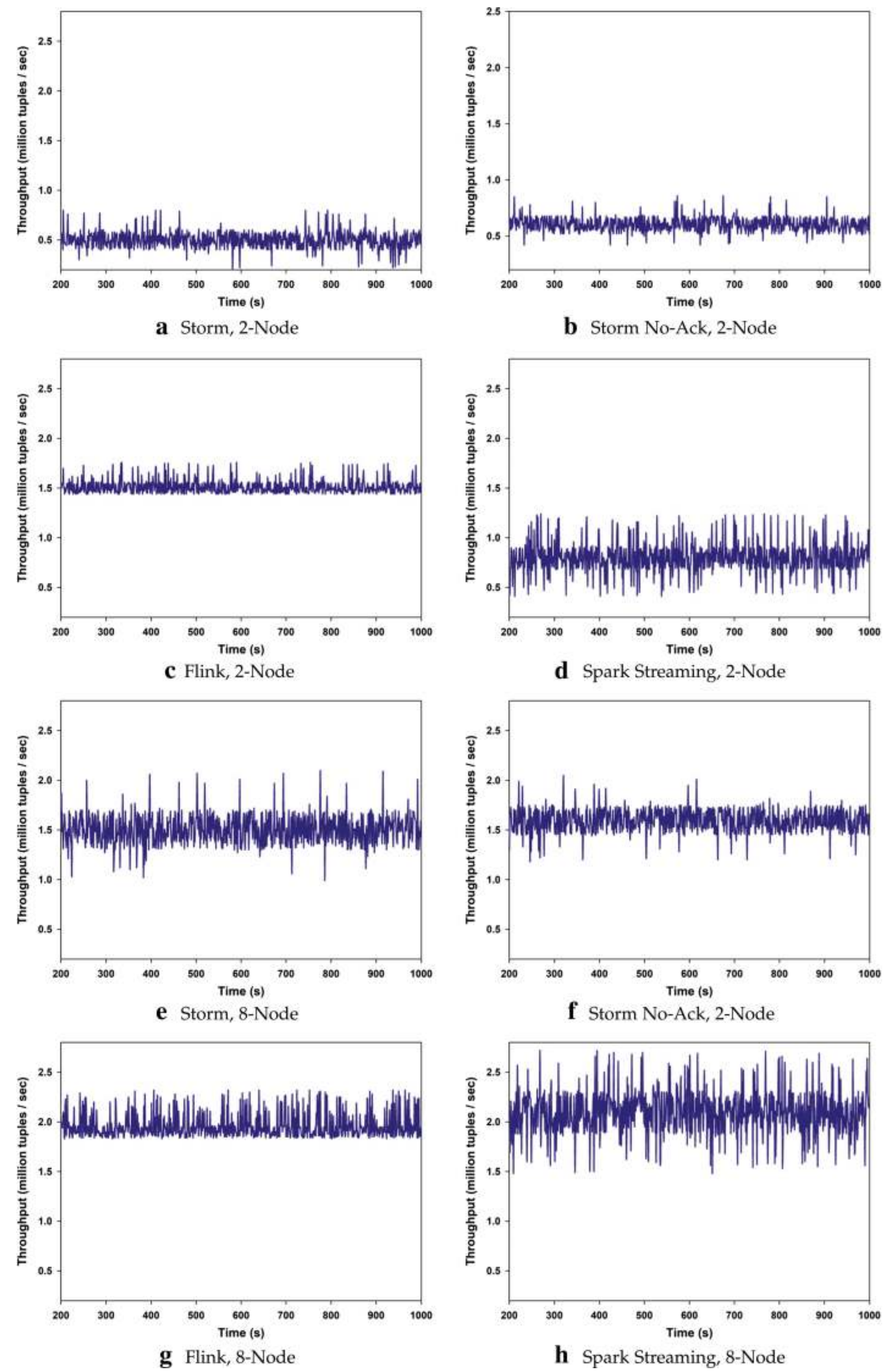


Fig. 12 Latency of intended frameworks with different number of worker nodes for Model Training application

Figures 13 and 14 represent the throughput of each framework over the time; using 2 and 8 worker nodes. In these experiments, we have executed the applications by different number of worker nodes with a particular input rate (*Advertising*: 2.8 M tuple/s,



a Storm, 2-Node

b Storm No-Ack, 2-Node

c Flink, 2-Node

d Spark Streaming, 2-Node

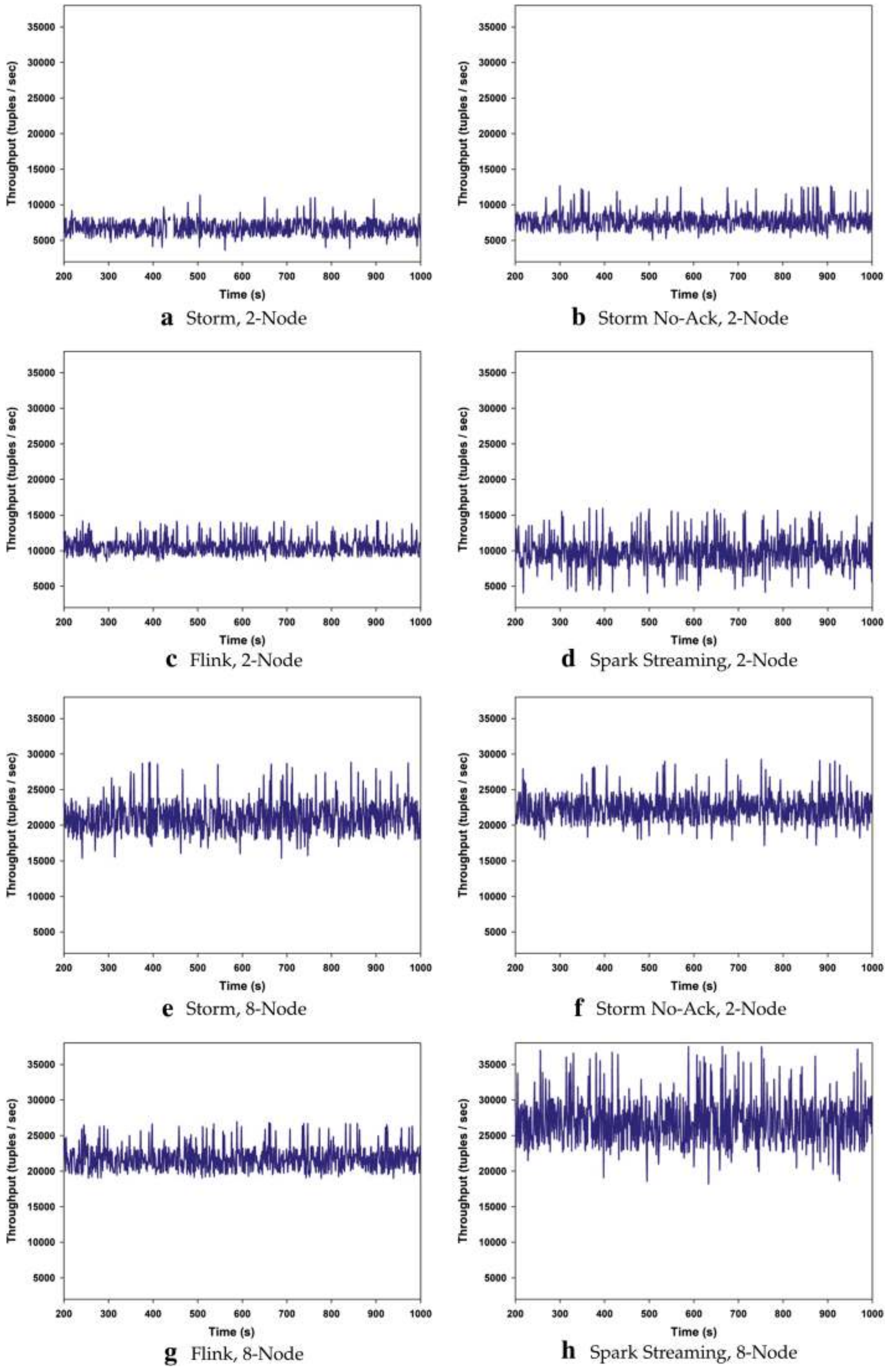
e Storm, 8-Node

f Storm No-Ack, 2-Node

g Flink, 8-Node

h Spark Streaming, 8-Node

Fig. 13 Throughput of intended frameworks with different number of worker nodes for Advertising application



a Storm, 2-Node

b Storm No-Ack, 2-Node

c Flink, 2-Node

d Spark Streaming, 2-Node

e Storm, 8-Node

f Storm No-Ack, 2-Node

g Flink, 8-Node

h Spark Streaming, 8-Node

Fig. 14 Throughput of intended frameworks with different number of worker nodes for Model Training application

Model Training: 14 k tuple/s). We have set these rates to fully utilize the worker nodes and hence realize how much throughput each frameworks provides at high load.

Considering these results, we see on smaller cluster Flink is the winner in terms of throughput and its throughput values have the least variance over the time. By scaling the cluster out via adding more worker nodes, the throughput of Storm and Spark Streaming increase with an almost linear ratio; however, throughput of Flink has little increment.

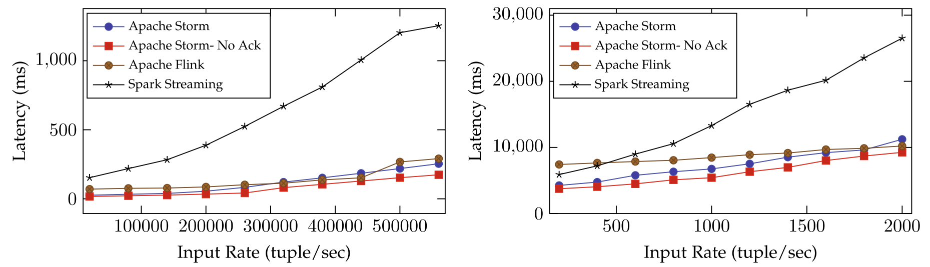
Load scalability

To appraise the ability for each framework to efficiently expand and contract its resource pool to accommodate heavier or lighter loads, we increase the input rate and measure the corresponding latency. For *Advertising* application the input rate is varied from 20,000 to 560,000 tuples/s, and for *Model Training* it is varied from 200 to 2000 tuples/s. For each input rate, benchmark applications are executed for 100 min and the end-to-end latencies are measured.

Figure 15 shows the latency behavior of different frameworks as the load increases, for both benchmarks. As we can see, for all frameworks the average end-to-end latency increases as the system load increases; however, the distance between Spark Streaming and other frameworks increases dramatically due to the essence of its programming model. As mentioned in "Programming model of stream processing" section Spark Streaming is not using a real streaming paradigm and relay on micro-batching mechanism. So, when the input rate is increased more tuples are buffered to form a micro-batch in the configured batching time slice, and hence the end-to-end latency of all of these buffered tuples is increased.

From the results in Fig. 15, we also observe that the average latency of Flink has the least dependency to the load because of its internal message handling mechanism and works well while the network is not over-utilized; and Storm shows a bit more reaction to load variances; while Spark Streaming performs worst among them due to the mentioned above reason.

For each input rate, the 99th percentile latency for a tuple to be completely processed by the DSPFs is illustrated in Fig. 16.



a Advertising application **b** Model training application

Fig. 15 Latency comparison over different rate

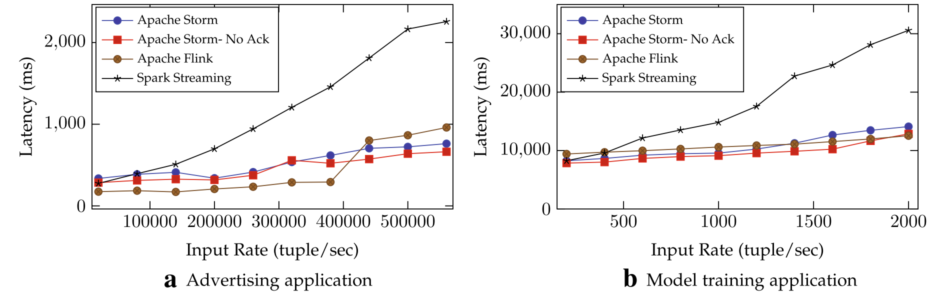


Fig. 16 99th percentile latency comparison over different rate

Horizontal scaling

To examine the ability of the appraised frameworks to scale out, we increased the cluster size by adding more worker nodes from 1 to 8. Figure 17 shows scale out effect on the latency of *Advertising* and *Model Training* applications, respectively. Also, Fig. 18 represents the scalability of *Advertising* and *Model Training* applications in terms of throughput.

Looking at these graphs we can see Flink has the worst scalability and its latency and throughput have a few improvements when the number of worker nodes is increased. In large-scale clusters, Storm and Spark Streaming beat Flink in terms of latency and throughput. Both Storm and Spark Streaming have near linear behavior in terms of scalability but Storm scales even better than Spark Streaming and behaves almost linear, specially when no acking mechanism is applied.

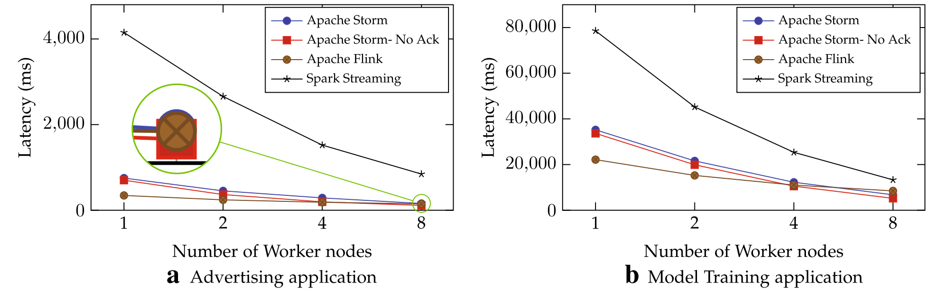


Fig. 17 Scalability comparison in terms of latency

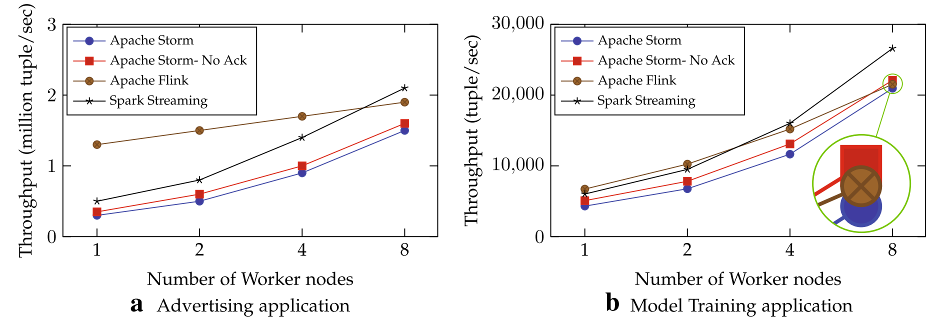


Fig. 18 Scalability comparison in terms of throughput

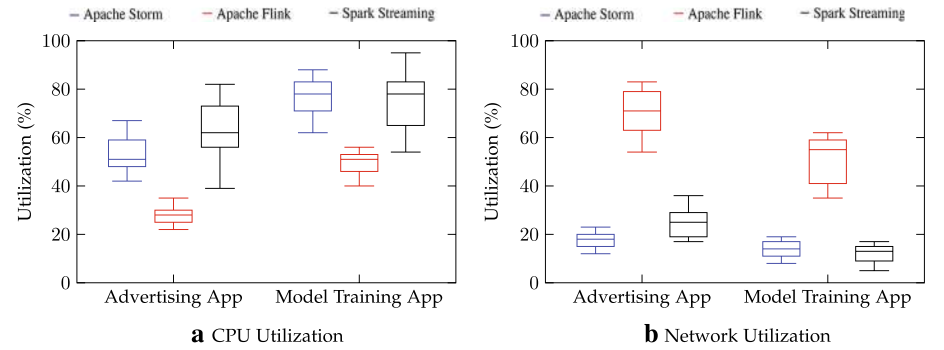


Fig. 19 Resource usage for both Advertising and Model Training applications running on 8 worker nodes

Resource utilization

In IoT domain, applications are too resource intensive and efficient use of distributed resources has vital role. To make better sense about efficiency of intended frameworks we calculate average CPU usage and Network utilization in all experiments. The average resource utilization of eight worker nodes related to experiments in "Latency evaluation" section (latency evaluation of different frameworks) are shown in Fig. 19. Using this information, we know which resource may become performance bottleneck in each framework.

Results and discussion

Load scalability: Throughout the results from "Load scalability" section, we observe that (1) Storm and Flink consistently outperforms Spark Streaming in terms of latency. (2) Increasing the input rate leads to more significant improvement of them. In Spark Streaming, multiple tuples are processed in a micro-batch, which leads to higher throughput but it cause deterioration of the end-to-end latency for individual tuples since they have to wait for a little bit before being batched by the Spark Streaming component.

Resource utilization: By placing resource utilization of all frameworks under scrutiny, we realized that Flink is more network intensive than other frameworks while its CPU usage is less than both Storm and Spark Streaming. As we can see in Fig. 15 Flink has stable performance while the input rate is increasing, but at a certain rate the network gets bottleneck and its latency becomes worse than both Storm and Storm no-Ack.

Ability to scale out: According to our observations when the cluster has a few number of worker nodes Flink provides the lowest latency and hence the best performance in both applications. The reasoning behind this excellence is the message passing mechanism of Flink which trades more network usage at a benefit of better CPU utilization. For small-scale cluster the extra CPU power that is saved by efficient message passing is consumed by the processing components and hence, the latency is decreased. By increasing the number of worker nodes Storm latency is reduced almost linearly, but Flink latency decrement is lighter and does not scale as expected; insofar as, for a cluster of 8 worker nodes Storm and Flink result almost equal performance. In the larger clusters we have more inter-node communications and the network utilization becomes more valuable parameters. Although, the massage passing of Flink benefits from better CPU utilization but it worsens the network utilization which increases the processing latency. Spark Streaming, always has much higher latency, but its latency scales better than Flink by increasing size of the cluster, however not as well as Storm.

From these results, we conclude that while the network resources do not get bottleneck Flink provides more stable response time and its 99th percentile latency value says it is better solution for real-time applications. On the other hand, Spark Streaming latency strongly depends on the input rate. It would not be good choice for application with variable data load like network monitoring, but a good selection to achieve high-throughput when latency is not as important as throughput. With acking disabled, Storm has better performance and provides more reliable response time at high throughput. However, in this conditions the ability to handle failures is disabled.

From the application perspective it is ideal to predict the certain behavior of streaming applications on different frameworks according to their category. Using these information, we can determine which framework better suits each category of applications. Table 2 represents a categorization of most popular streaming workloads. An ETL application is a process consists of cleaning or correcting the data, retrieving data from the sources, transforming data into a usable format and finally transmitting data to the consumers. An executing machine learning algorithm on streaming data is called Stream ML. A CEP application combines data from multiple sources to identify events or patterns. An IVP application is a process consists of some operations, performing on an image or a frame of an incoming video.

As our experiments show there are some trade-off to choose the proper framework for each category of applications. For example, an ETL application may has the least latency on Storm but Spark Streaming provides more processing throughput. In another case may Flink process a single tuple much faster than Storm but when the data arrival rate or cluster size are changed, Storm passes Flink.

So, regardless to the category of applications there are several parameters such as data arrival rate, cluster size, tuple size and data type [43] which severely effect different performance metrics. Further, giving a recommendation for a framework per each category can vary depending on the desired metric. Nevertheless, we made a recommendation for a specific framework per each category in Table 2 based on the main characteristics of the applications in each category. In this table, for each category we have specified which framework provides the best results in general terms.

Overall, each of the frameworks studied here have their advantages and disadvantages. Our experiments show Storm and Flink have very similar performance, and Spark Streaming, has much higher latency, while it provides higher throughput.

Table 2 Framework recommendation for different streaming application categories

Category	Description	Recommended framework
ETL	Extract, transform and load	Storm, Spark Streaming
Stream ML	Stream machine learning	Spark Streaming
CEP	Complex event processing	Flink
IVP	Image and video processing	Storm

However, Flink behaves very well at small-scale clusters but it has poor scalability and loses the competition on the large-scale clusters.

Conclusion and future works

Collections of large amount of IoT devices and objects are producing huge amount of data in Smart Cities which requires being processed immediately. Big Data analytics tools have the capacity to handle large volumes of data generated from IoT devices that create a continuous stream of information. There are plenty of Big Data processing platforms which are designed for special purposes. At the age of IoT and Smart Cities it is interesting to compare the behavior of available distributed stream processing frameworks and examine the applicability of employing them to process high volume of data generated in Smart Cities.

In this paper we made a deep comparison between three most popular frameworks, namely Apache Storm, Apache Flink, and Spark Streaming to show that which platform is suitable for what kind of streaming application. In our experiments we focused on evaluating the performance of intended frameworks in terms of latency, and throughput. We also evaluated the scalability (in terms of data arrival rate and the number of cluster nodes), and resource utilization of these frameworks using two benchmark applications from real world.

In addition to further consideration for resource utilization, we like to take into account the processing guarantees and fault tolerance. We also like to include other stream processing frameworks like Apache Heron and Apache Samza.

Abbreviations

IoT: Internet of Things; DSPF: distributed stream processing framework; RFID: Radio Frequency IDentification; HDFS: Hadoop Distributed File System; API: application programming interface; DB: database; JSON: JavaScript Object Notation; DAG: directed acyclic graph; ETL: extract, transform, load; CEP: Complex Event Processing; ML: machine learning; IVP: Image and Video Processing.

Acknowledgements

The authors thanks reviewers for valuable feedback on an early draft of this manuscript.

Authors' contributions

HN performed the primary work and analysis of this manuscript and designed the experiments. HN and SN performed experiments and analyzed the observed results and co-wrote the paper. MG supervised the research. All authors read and approved the final manuscript.

Funding

This work is partially supported by the Iran National Science Foundation (INSF), under Grant Number 96015834.

Availability of data and materials

The data used in this paper are publicly online available at [41] and [44]. The link for the same is mentioned in the "References" section.

Competing interests

The authors declare that they have no competing interests.

Received: 20 February 2019 Accepted: 3 June 2019
Published online: 11 June 2019

References

1. Agarwal S. 2016 state of fast data and streaming applications survey. https://www.opsclarity.com/2016-state-fast-data-streaming-applications-survey/. Accessed 12 Oct 2017.
2. Díaz M, Martín C, Rubio B. State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing. J Netw Comput Appl. 2016;67:99–117.

3. Zhu C, Zhou H, Leung VC, Wang K, Zhang Y, Yang LT. Toward big data in green city. IEEE Commun Mag. 2017;55(11):14–8.
4. Chen F, Deng P, Wan J, Zhang D, Vasilakos AV, Rong X. Data mining for the internet of things: literature review and challenges. Int J Distrib Sens Netw. 2015;11(8):431047.
5. Guo Y, Rao J, Jiang C, Zhou X. Moving hadoop into the cloud with flexible slot management and speculative execution. IEEE Trans Parallel Distrib Syst. 2017;3:798–812.
6. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. Commun ACM. 2008;51(1):107–13.
7. Goudarzi M. Heterogeneous architectures for big data batch processing in mapreduce paradigm. IEEE Trans Big Data. 2017. https://doi.org/10.1109/TBDATA.2017.2736557.
8. Toshniwal A, Taneja S, Shukla A, Ramasamy K, Patel JM, Kulkarni S, Jackson J, Gade K, Fu M, Donham J, et al. Storm@ twitter. In: Proceedings of the 2014 ACM SIGMOD international conference on management of data. New York: ACM; 2014. p. 147–56.
9. Zaharia M, Das T, Li H, Shenker S, Stoica I. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. HotCloud. 2012;12:10.
10. Katsifodimos A, Schelter S. Apache flink: stream analytics at scale. In: 2016 IEEE international conference on cloud engineering workshop (IC2EW). New York: IEEE; 2016. p. 193.
11. Wilmoth J. 2018 revision of the world urbanization prospects. https://population.un.org/wup/Publications/Files/WUP2018-PressRelease.pdf. Accessed 02 Mar 2019.
12. Hashem IAT, Chang V, Anuar NB, Adewole K, Yaqoob I, Gani A, Ahmed E, Chiroma H. The role of big data in smart city. Int J Inf Manag. 2016;36(5):748–58.
13. Shirer M, Rold SD. Worldwide semiannual smart cities spending guide. https://www.idc.com/getdoc.jsp?containerId=prUS43576718. Accessed 11 Feb 2018.
14. Apache hadoop. https://hadoop.apache.org/. Accessed 02 June 2018.
15. Apache spark: Lightning-fast unified analytics engine. https://spark.apache.org/. Accessed 02 June 2018.
16. Apache storm. http://storm.apache.org/. Accessed 02 June 2018.
17. Apache flink: Stateful computations over data streams. https://flink.apache.org. Accessed 02 June 2018.
18. Nasiri H, Nasehi S, Goudarzi M. A survey of distributed stream processing systems for smart city data analytics. In: Proceedings of the international conference on smart cities and internet of things. New York: ACM; 2018. p. 12.
19. Hesse G, Lorenz M. Conceptual survey on data stream processing systems. In: 2015 IEEE 21st international conference on parallel and distributed systems (ICPADS). New York: IEEE; 2015. p. 797–802.
20. Singh MP, Hoque MA, Tarkoma S. A survey of systems for massive stream analytics; 2016. arXiv preprint arXiv:1605.09021.
21. Kamburugamuve S, Fox G, Leake D, Qiu J. Survey of distributed stream processing for large stream sources. 2013. https://scholar.google.com/scholar?hl=en%26as_sdt=0%2C5%26q=Survey+of+distributed+stream+processing+for+large+stream+sources%26btnG=.
22. Kamburugamuve S, Fox G. Survey of distributed stream processing. Bloomington: Indiana University; 2016.
23. Pääkkönen P, Pakkala D. Reference architecture and classification of technologies, products and services for big data systems. Big Data Res. 2015;2(4):166–86.
24. Singh D, Reddy CK. A survey on platforms for big data analytics. J Big Data. 2015;2(1):8.
25. Tsai C-W, Lai C-F, Chao H-C, Vasilakos AV. Big data analytics: a survey. J Big Data. 2015;2(1):21.
26. Inoubli W, Aridhi S, Mezni H, Maddouri M, Nguifo EM. An experimental survey on big data frameworks. Fut Gener Comput Syst. 2018;86:546–64.
27. Veiga J, Expósito RR, Pardo XC, Taboada GL, Tourifio J. Performance evaluation of big data frameworks for large-scale data analytics. In: 2016 IEEE international conference on Big Data (Big Data). New York: IEEE; 2016. p. 424–31.
28. Hirzel M, Soulé R, Schneider S, Gedik B, Grimm R. A catalog of stream processing optimizations. ACM Comput Surv CSUR. 2014;46(4):46–50.
29. Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ, et al. Apache spark: a unified engine for big data processing. Commun ACM. 2016;59(11):56–65.
30. Oliver AC. Storm or spark: choose your real-time weapon. http://www.infoworld.com/article/2854894/application-development/spark-and-storm-for-real-time-computation.html. Accessed 01 Feb 2018.
31. Hunt P, Konar M, Junqueira FP, Reed, B. Zookeeper: wait-free coordination for internet-scale systems. In: USENIX annual technical conference, vol. 8, Boston, MA, USA; 2010.
32. Introduction to heron. https://streaml.io/blog/intro-to-heron. Accessed 10 Apr 2018.
33. Kulkarni S, Bhagat N, Fu M, Kedigehalli V, Kellogg C, Mittal S, Patel JM, Ramasamy K, Taneja S. Twitter heron: stream processing at scale. In: Proceedings of the 2015 ACM SIGMOD international conference on management of data. New York: ACM; 2015. p. 239–50.
34. Apache kafka: a distributed streaming paltform. http://kafka.apache.org/. Accessed 02 June 2018.
35. Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S et al. Apache hadoop yarn: Yet another resource negotiator. In: Proceedings of the 4th annual symposium on cloud computing. New York: ACM; 2013. p. 5.
36. Apache samza: A distributed stream processing framework. http://samza.apache.org/. Accessed 11 Aug 2018.
37. Gorawski M, Gorawska A, Pasterak K. A survey of data stream processing tools. In: Czachórski T, Gelenbe E, Lent R, editors. Information sciences and systems 2014. Cham: Springer; 2014. p. 295–303.
38. Kejariwal A, Kulkarni S, Ramasamy K. Real time analytics: algorithms and systems. Proc VLDB Endow. 2015;8(12):2040–1.
39. Zapletal P. Comparison of apache stream processing frameworks. Cakesolutions. http://www.cakesolutions.net/teamblogs/comparison-of-apache-streamprocessing-frameworks-part-1. Accessed 12 Feb 2018.
40. Kreps J, Narkhede N, Rao J et al. Kafka: A distributed messaging system for log processing. In: Proceedings of the NetDB; 2011. p. 1–7.
41. Yehuda G. Yahoo streaming benchmarks. https://github.com/yahoo/streaming-benchmarks. Accessed 08 Oct 2017.

42. Shukla A, Chaturvedi S, Simmhan Y. Riotbench: an iot benchmark for distributed stream processing systems. Concurr Comput Pract Exp. 2017;29(21):4257.
43. Ahmadvand H, Goudarzi M, Foroutan F. Gapprox: using gallup approach for approximation in big data processing. J Big Data. 2019;6(1):20.
44. Brian D, Dan W. New york city taxi trip data. https://databank.illinois.edu/datasets/IDB-9610843. Accessed 12 Apr 2018.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.