Progress in Artificial Intelligence (2020) 9:239–261
https://doi.org/10.1007/s13748-020-00210-6

REGULAR PAPER

An analysis of technological frameworks for data streams

Fernando Puentes[1] · María Dolores Pérez-Godoy[1] · Pedro González[1] · María José Del Jesus[1]

Received: 28 May 2019 / Accepted: 30 May 2020 / Published online: 28 June 2020
© Springer-Verlag GmbH Germany, part of Springer Nature 2020

Abstract
Real-time data analysis is becoming increasingly important in Big Data environments for addressing data stream issues. To this end, several technological frameworks have been developed, both open-source and proprietary, for the analysis of streaming data. This paper analyzes some open-source technological frameworks available for data streams, detailing their main characteristics. The objective is to facilitate decisions on which framework to use, meeting the needs of data mining methods for data streams. In this sense, there are important factors affecting the choice about which framework is most suitable for this purpose. Some of these factors are the existence of data mining libraries, the available documentation, the maturity of the platform, fault tolerance and processing guarantees, among others. Another decisive factor when choosing a data stream framework is its performance. For this reason, two comparisons have been made: a performance and latency comparison between Spark Streaming, Spark Structured Streaming, Storm, Flink and Samza following the Yahoo Streaming Benchmark methodology, and a comparison between Spark Streaming and Flink with a clustering algorithm for data streaming called streaming K-means.

Keywords Data streaming · Big data · Big data frameworks · Technological frameworks · Data stream engines

1 Introduction

Data mining has traditionally focused on the analysis of static stored datasets, in which all data are available, their order is not important, and they are processed using the *batch mode* [1]. However, every day there are more and more sources that generate huge amounts of data on a continuous basis, in a big data context. In order to process these data, it is necessary to use the *streaming mode* [2], in which data are processed as soon as they reach the system, without being stored.

This work has been supported by the Ministry of Economy and Competitiveness Under the Project TIN2015-68454-R and PID2019-107793GB-I00.

✉ Fernando Puentes
fpuentes@ujaen.es

María Dolores Pérez-Godoy
lperez@ujaen.es

Pedro González
pglez@ujaen.es

María José Del Jesus
mjjesus@ujaen.es

[1] Universidad de Jaen, Jaén, Spain

A data stream is a continuous flow of data generated from one or more sources that arrives online at the system [3]. Today, it is essential to use specific frameworks that facilitate the processing of data streams, so many of these frameworks have appeared. Figure 1 summarizes the necessary phases for the analysis of data streams [4]:

(a) Data ingestion. In this first phase, a framework is used to collect data generated continuously by one or more data sources. It is also necessary that the data are in the appropriate format so that they can be used in the next phase; thus, it is necessary to perform pre-processing tasks, such as structure modifications, filtering columns, and elimination of anomalous values.
(b) Data processing. Once the data are in the right format, the processing phase is launched. In this one, data mining algorithms are used to analyze the resulting stream and to extract knowledge. To improve performance, it is essential to choose a framework that allows distributed processing. The framework that best suits the requirements and needs for the problem to be solved should be chosen from among the available technological frameworks.
(c) Analysis and evaluation. This last phase is optional and is responsible for displaying the results obtained in the previous phase using some type of real-time visualization framework.

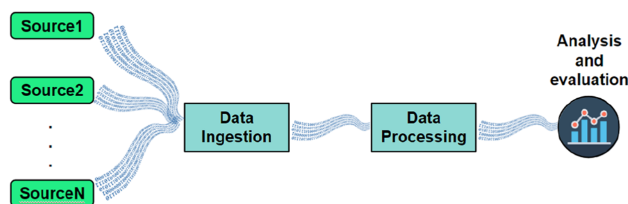


Fig. 1 General structure of stream analysis

Most of the available frameworks focus on one of these phases, although there are some that cover two or even all three phases. This paper analyzes technological frameworks for both data ingestion and data processing, since they are the most determinant phases when processing data streams in streaming mode.

An important aspect of this type of technological frameworks is to consider whether the framework is open-source or proprietary. There are currently a wide variety of proprietary frameworks that offer advantages such as increased support and the ability to implement solutions very quickly. However, nowadays, open-source frameworks are increasingly used [5], because they allow the creation of more efficient solutions without implementation restrictions, and facilitate the access to the entire research community. Therefore, this paper mainly focuses on open-source frameworks, even though the main features of some of the most commonly used proprietary frameworks are described.

There is another important aspect to bear in mind with regard to technological frameworks for data streams. When analyzing the data of a data stream, it is not only important to ingest the data to the system but also to extract knowledge, usually by applying a machine learning algorithm. For this reason, the availability of machine learning libraries is an important factor when choosing a framework for the data processing phase. Therefore, this work analyzes which machine learning libraries can be used by different data stream frameworks, and the list of algorithms implemented in each of these libraries.

A final aspect to consider when choosing a processing framework, both in batch and in streaming mode, is the performance of the framework. For this reason, this paper includes a performance comparison between the different technological frameworks. This is achieved by comparing the performance of technological frameworks following the methodology of the Yahoo Streaming Benchmark [6], but also with a specific comparison of machine leaning algorithms for data streams.

In summary, the objective of this paper is to analyze open-source technological frameworks that perform the ingestion and processing phases, providing information that facilitates the choice of the most appropriate framework to be used depending on the objective to be achieved. The paper is organized as follows: In Sect. 2, important concepts regarding processing data streams are introduced. Section 3 describes some important properties of data stream technological frameworks, and Sect. 4 shows several comparisons of these frameworks. In Sect. 5, the most important technological frameworks for data ingestion phase are analyzed, while in Sect. 6 the technological frameworks for the processing phase are analyzed. Section 7 analyzes machine learning libraries to be used with the data stream frameworks, showing a list of the algorithms available in each library. Sections 8 and 9 introduce performance comparisons of both the different technological frameworks and machine learning algorithms. Finally, Sect. 10 describes the conclusions obtained.

2 Processing data streams: basic concepts

A data stream is a continuous flow of data generated from one or more sources that arrives online at the system [3]. For a dataset to be considered as a data stream, it must contain the following features:

- Data are continuously generated, usually at high speed.
- Distribution of data may change over time. This fact, known as *concept drift*, is detailed later. It is important to detect these drifts in order to adapt the models to them as soon as possible [7].
- The size of the stream is theoretically infinite, so it cannot be completely stored in memory for processing. These data are usually discarded once processed.
- The order in which the data arrive at the system cannot be controlled, and must be maintained for processing.

Therefore, each instance in the stream is only processed once, and then, discarded or archived. But there exist different ways to process the data stream:

1. Processing each instance as soon as it reaches the system, one by one.
2. Grouping data in chunks and processing each chunk of data, using a data structure called window.

When a window is used, only data within the window are considered. There are two basic types of windows [8]:

- *Sequence-based window* It stores a number of elements. Two subtypes of sequence-based windows can be found, as shown in Fig. 2:
    - *Landmark Window* It is a window of variable size, which stores all the elements of a stream, from the first to the last. Figure 2a shows how it works. When

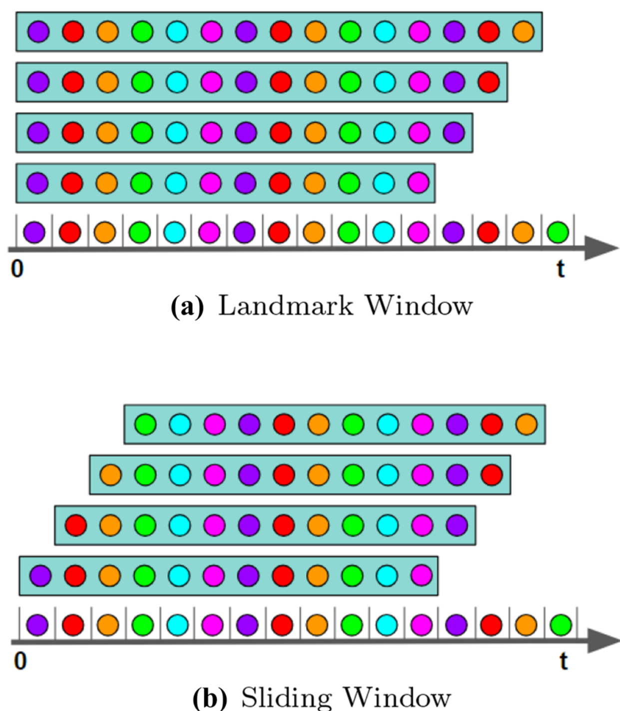


Fig. 2 Sequence-based windows in data stream processing

new data arrive, the size of the window is increased to include them. Old data are not discarded.
   - *Sliding Window* In this case, the window has a fixed size, so both the size and the sliding of the window must be specified. This type of sequence-based window stores the number of specified elements, and slides when it is full. This window acts like a FIFO (first in, first out) data structure in which new data replace old data, as shown in Fig. 2b.

- *Time-based window* In this type of window, time duration and sliding must be defined. It stores all the data arriving at the system within the specified period, and updates its content each sliding time. Figure 3 shows an example of a time window with 3 s of time duration and 2 s of time slide. Let us assume that one example may or may not arrive at the system each second. It stores data from the previous 3 s, processes all the data in this window and slides 2 s.

Two scenarios can be found when dealing with data streams [9]: the stationary scenario, where all data are generated by a single probability distribution; and the non-stationary one, where the distribution generating data can change over time, producing the phenomenon known as *concept drift* [7]. This concept is defined next in a more formal way. Let us suppose that a stream is a sequence of states S = $\{S_1, S_2, \ldots, S_n, \ldots\}$ where S_i is generated by a distribution D_i. In a stationary context, this sequence of states is generated by the same distribution, so $D_{i+1} = D_i$. In a non-stationary context, a concept drift can occur, thus changing the distribution that generates data. When a concept drift appears on a stream, the current model needs to be updated according to the new concept [10]. The evidence of the occurrence of a concept drift is somehow reflected in the data of the stream [2].

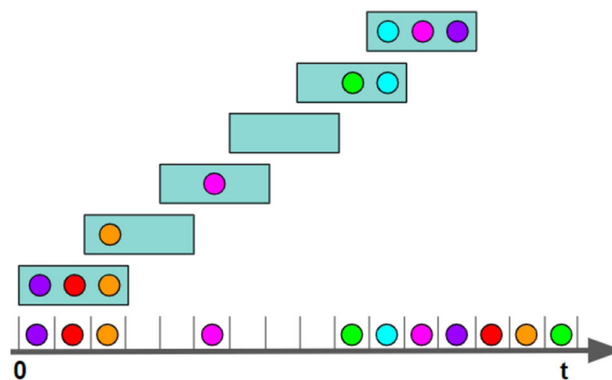
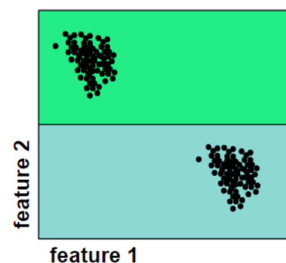
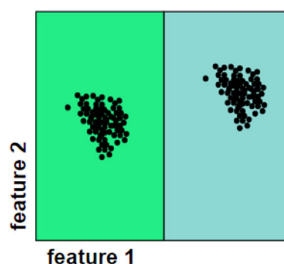


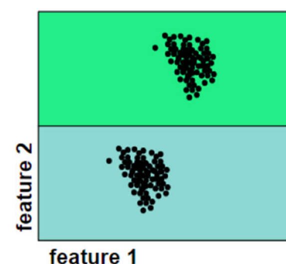
Fig. 3 Time-based window in data stream processing



(a) Initial distribution



(b) Real concept drift



(c) Virtual concept drift

Fig. 4 Two main types of concept drift in data stream mining

Two main types of concept drift can be identified according to its influence on the decision boundaries of a model, as shown in Fig. 4 [10]:

- *Virtual concept drift* The concept drift does not affect decision boundaries of the model, so the model cannot detect it.
- *Real concept drift* The concept drift can be detected because it affects decision boundaries of the model, and the model can be updated to adapt to the new concept.

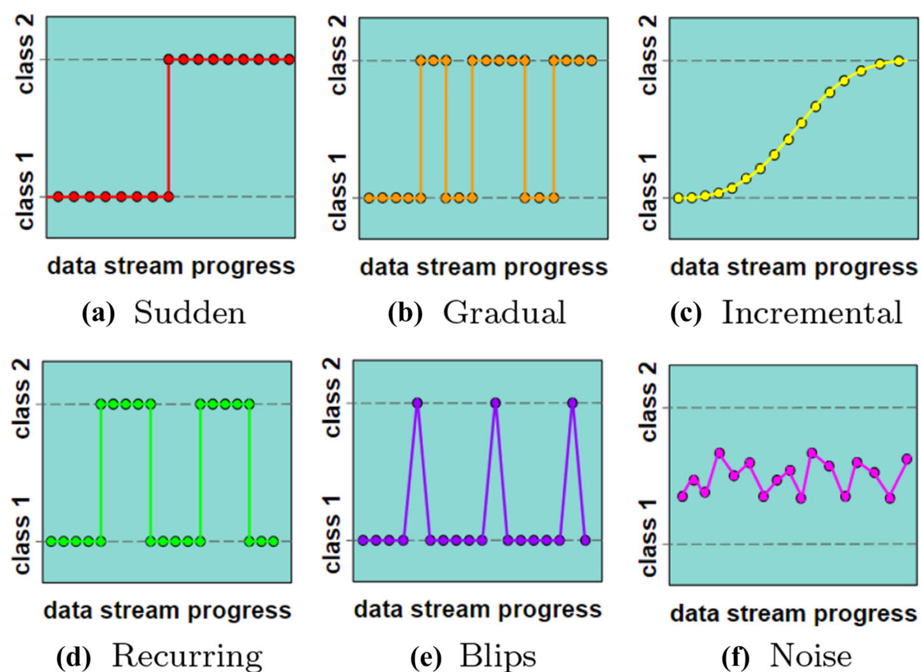


Fig. 5 Types of real concept drift in data stream mining

In addition to the above, different types of concept drift can be identified, but according to the ratio of changes, as can be seen in Fig. 5 [10]:

- *Sudden* The old concept is rapidly replaced by the new concept.
- *Gradual* The current concept is a mixture of both the old and the new concept. Gradually, the new concept replaces the old one until it disappears and only the new concept remains.
- *Incremental* The new concept slowly replaces the old concept.
- *Recurring* The current concept is replaced by a concept that already appeared in the past.
- *Blips* The current concept is stable except in random situations in which a concept drift appears.
- *Noise* The concept has insignificant variations that should be filtered out.
- *Mixed* This is a combination of two or more concept drift types.

In this context, it is also important to consider that machine learning algorithms dealing with data streams should take concept drift into account. To do so, these algorithms need to include an adaptation method to allow the model to be updated. Two types of adaptation methods can be distinguished [8]:

- *Blind methods* The model is updated when a certain specified time passes or a number of data arrive at the system. No detection method is used to detect the concept drift. In this case, a fixed-size window is usually implemented, updating the model with new data periodically.
- *Informed methods* A detection method is used for the concept drift detection. Here, the model is only updated when a concept drift is detected. In this case, it is usual to use a drift detection method based on the error of the model, which can be fully or partially updated.

There is another key factor for the streaming algorithms: the forgetting mechanism. A forgetting mechanism is used to specify how quickly old data should be discarded in order to generate new models. Two types of forgetting mechanism can be considered [8]:

- *Abrupt forgetting (or partial memory)* Here, a type of window is used to store a set of data, and only data inside the window is used to generate the model.
- *Gradual forgetting (or full memory)* In this approach all data are taken into account to update the model, but more importance is given to new data. To do this, a weight is associated to each instance according to its age, so that new data have more weight than the old ones.

3 Data streaming technological framework properties

Nowadays, many big data processing technological frameworks for data stream analysis facilitate the implementation of streaming algorithms. The most important open-source

technological frameworks for data streams are summarized in Table 1. In addition, Table 2 shows a list of proprietary streaming frameworks collected by Gartner [5,11].

Before describing the frameworks, and in order to be able to make a comparison, it is necessary to introduce some features that will determine their behavior and performance.

- The first of these features is the one known as *processing mode*. There are two processing modes: *traditional mode* (or *batch mode*) and *streaming mode*. In the traditional mode, a set of stored and static data that are analyzed as a whole is used, while in the streaming mode data reach the system over time, being analyzed as they arrive. Within the streaming processing mode, data can be analyzed *one by one* or in small groups called *micro-batches*. Each micro-batch is a time window that stores data from a given time interval.
- The next feature to consider is the *processing guarantee*. This refers to the number of times a instance can be processed. Although in theory each instance is processed only once in the data stream context, in practice this could not occur because data have to be sent between machines and problems can appear, producing unprocessed or duplicated data. There are three types of processing guarantees:
  - *At most once* Data can be processed once or not at all. Unprocessed data can appear.
  - *At least once* Data can be processed one or more times. There will never be any unprocessed data, but duplicated data can appear.
  - *Exactly once* Data will be processed only once. This type is the best processing guarantee that a processing framework can have. It does not produce duplicate or unprocessed data.

Table 1 Open-source technological frameworks for data streams

Frameworks
Apache Apex
Apache Beam
Apache Flink
Apache Flume
Apache Gearpump
Apache Ignite
Apache Kafka
Apache Nifi
Apache S4
Apache Samza
Apache Spark
Apache Storm
Elastic Logstash

Table 2 Proprietary technological frameworks for data streams

Frameworks
Alooma
Attunity
Amazon Kinesis
Google Cloud Dataflow
Confluent Platform
dA Platform
Unified Analytics Platform
Data Stream Manager
Data Beaming
Esper Enterprise Edition
EVAM
Interstage Big Data Complex Event Processing Server
Hitachi Streaming Data Platform (HSDP)
Hortonworks DataFlow (HDF)
IBM Streams
StreamAnalytix
Big Data Streaming
Azure Stream Analytics
Nexla
Oracle Stream Analytics
Spring Cloud Data Flow (SCDF)
RB Light
SAP Event Stream Processor
APPAMA Streaming Analytics
s-Server
Streamlio
Striim
Talend Data Streams
TIBCO BusinessEvents
VIA Analytics Platform
WSO2 Complex Event Processor (WSO2 CEP)
3Forge AMI
ActiveViam In-Memory Analytical Platform
First Derivatives kdb+
Inetco Systems Analytics
Interana
Logtrust
Maana Knowledge Platform
OneMarketData OneTick
Unscrambl Brain
Splunk Enterprise; Cloud
Amazon QuickSight
Arcadia Data Enterprise
Datameer Enterprise
Information Builders WebFocus
Microsoft Power BI
RapidMiner Platform
TIBCO Spotfire
Zoomdata Server; Fusion

– Finally, it is necessary to consider the possibility of using *stateless* or *stateful processing*. In stateless processing, a node processes each instance without depending on previous results and does not store any state after processing it. On the contrary, with stateful processing the state of the node is modified after processing the data, on the basis of previous results.

4 Previous data stream technological frameworks reviews

Over the last years, some comparisons between some of the most popular processing frameworks have been made, both for the batch and streaming modes:

– In a recent paper [12], the performance of Spark and Flink with different algorithms for batch processing is shown. According to the results of the experiments, this work shows that Spark performs better than Flink in batch processing.
– In [13] a comparison between Spark, Storm and Samza is shown. This comparison consists in a system that obtains data from speed sensors, filters invalid speed values (between 0 and 90) and stores correct data, measuring different metrics like throughput and system resource usage. In this comparison, Spark shows better throughput than Storm and Samza.
– Another methodology for comparing data stream technological frameworks, called StreamBench, was implemented in [14,15]. This benchmark evaluates the throughput and latency of technological frameworks by running different programs and performing operations over the stream. In [15] seven programs are run in order to compare Spark and Storm. The results show that Spark has a better throughput than Storm, but Storm has lower latency in non-complex processing scenarios. This is because Storm processes data one by one (real streaming), while Spark processes data in micro-batches. In [14] only three processing programs are used to compare Spark, Storm and Flink. In this case, Flink obtained the best throughput and latency results.
– In other studies [6,16–18] a streaming processing benchmark called Yahoo Streaming Benchmark is used, which evaluates the latency and throughput of technological frameworks. This benchmark was developed by Yahoo Inc. [6] to compare the performance of Storm with respect to Spark and Flink. In [17] the same comparison was performed with Spark and Storm. Another comparison with this benchmark is [18], but this adds an analysis over the usage of system resources. In a more recent study [6], this benchmark was implemented with more streaming frameworks (Spark, Storm, Flink and Apex), using the current versions.

Recently, a review of technological frameworks in both batch and streaming mode was carried out [19]. According to this study, the most widely used processing framework is Apache Spark. Studies analyzing the use of big data frameworks and big data analytic techniques in different application areas are described below. In [20] a machine learning approach called *SCARFF*, which deals with imbalance, non-stationary and feedback latency is presented. This algorithm was designed for credit card fraud detection and implemented with Kafka and Spark. In [21] a real-time remote health status prediction system is developed using Spark. This system extracts user tweets from Twitter with their health attributes, predicts user's health status, and the user is directly messaged at that instant to take appropriate action. In [22] scalable distributed clustering algorithms based on the micro-clustering paradigm are developed using Storm. In addition, two distributed architectures for executing the algorithms in parallel are implemented. In [23] an efficient algorithm for mining maximal frequent patterns over dynamic data streams is implemented using Spark.

5 Frameworks for data ingestion

Before processing the data, it is usually necessary to ingest them, as they are often generated by different simultaneous sources, so it is necessary to unify them in order to have a single input channel for the processing. The objective of this section is to analyze and compare different frameworks for performing these tasks, highlighting their main characteristics. In particular, this section analyzes Apache Kafka, Flume and Nifi.

5.1 Apache Kafka

Apache Kafka[1] is a distributed streaming framework that allows data to be ingested from different sources with a *publication/subscription* methodology [24] (the *producer* publishes the data and the *consumer* subscribes to receive them). To do this, the Kafka cluster must define a set of identifiers for buffers where publications and subscriptions are made, named *topics*.

In Fig. 6 a general scheme of the Kafka methodology can be seen. Each producer is an application that can publish messages on one or more topics. On the other hand, each consumer is another application that can subscribe to one or more topics in order to receive data published under

[1] https://kafka.apache.org/.

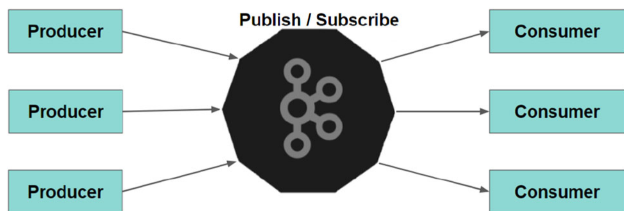


Fig. 6 General scheme of Apache Kafka

these topics. Kafka has a dynamic structure, so new producers/consumers can appear and existing producers/consumers can stop without restarting the Kafka system.

In Kafka the data are called *Records*, which are key-value pairs in text format, where the key is usually some type of timestamp and the value is a string in a specific format (usually JSON). For each topic, the Kafka cluster maintains an ordered and immutable sequence of Records, called *partition*, where each publication is maintained for a validity period, called *retention period*, and deleted when this period ends. Each partition has a sequential numeric identifier, called *offset*, which uniquely identifies each record within a partition. The consumer is responsible for controlling the offset value himself, so different consumers may be receiving data from different buffer positions (normally the offset of all consumers advances in a similar way).

In addition to ingesting data from different sources, Kafka also allows pre-processing. There are two options for carrying out pre-processing in Kafka: at the producer or by implementing a consumer-producer. The first option would consist in a program that transforms the data before sending them to the processing system; the second would be an application that listens to a topic, collects everything that comes to that topic to transform it and finally sends the result to a different topic.

Some points to highlight about Kafka are:

– It is integrated into all the processing frameworks which we will analyze later in Sect. 6.
– It is necessary to implement both a producer and a consumer program. If a streaming processing framework has compatibility with Kafka, the consumer is usually implemented in it. For this reason, in most situations only producers have to be implemented.
– Extensive documentation and information regarding many solved problems are available for developers, which accelerates the development of data ingestion solutions and error correction.
– It allows data replication. Data are stored in partitions that can be replicated to provide fault tolerance.
– It allows data distribution, so these partitions can be distributed over different servers of the cluster. Each partition has a leader server and zero or more follower servers.

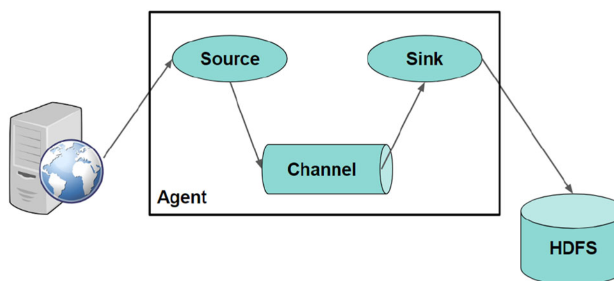


Fig. 7 General scheme of Apache Flume

– It has no native monitoring utility, but there are some third-party applications that do this, such as Burrow from LinkedIn.
– Kafka allows many consumers to obtain data at different reading speeds, thanks to the concept of offset.
– Kafka also allows to perform processing tasks, although its performance, in this case, is worse than the processing frameworks seen in Sect. 6.

Kafka allows to pre-process, replicate and distribute the data. In addition, it has a large amount of documentation and its publish/subscribe policy gives it a more general purpose than the rest of the ingestion frameworks analyzed, since it does not have a pre-established number of producers and consumers. For this reason, new producers and consumers interacting with Kafka may appear during the execution.

5.2 Apache Flume

Apache Flume[2] is a distributed service that efficiently ingests, aggregates and moves data streams. It has a simple and flexible architecture based on streaming data flows. It is based on an agent composed of 3 components (Fig. 7) [25]: *Source*, *Channel* and *Sink*.

An agent can contain one or more Sources, which ingest data from an external source (such as a web server) and store them in one or more Channels with a specific format. When a new event arrives at a Source, it is stored in the corresponding Channel. A Channel is a passive buffer that stores the events received from the Source until they are consumed by the Sink. This is necessary because, in general, the reading and writing speed are different, so it acts as a temporary data buffer. Flume has different types of Channels, depending on the type of storage used. Afterwards, the Sink is responsible for removing an event from the Channel and storing it in the destination file system. Source and Sink are executed asynchronously with events organized in Channels, so that until a instance is not stored in the destination file system it

[2] https://flume.apache.org/.

is not deleted from the Channel. A Sink can send events to one or more destinations.

These components are configured in a file containing the parameters corresponding to each one. Each component must have a name, a type and a set of specific properties depending on the type.

Flume can be easily integrated with Apache Hadoop. For this reason, the main use of Flume is to move large amounts of data from one or more source systems to an HDFS system, although it can be another target system, such as HBase.

This framework also allows data pre-processing. There are two options for pre-processing in Flume: at the Source or at the Sink. In the first case, the Source obtains data from the external system, performs the pre-processing and sends the result to the Channel. In the second case, the Sink obtains data from the Channel, pre-processes them and sends the result to the destination system.

The characteristics of Flume are summarized below:

- Among the technological frameworks analyzed in Sect. 6, only Flume is integrated into Spark Streaming.
- In Kafka new producers and consumers may appear during execution without the need to specify it to the system. This does not happen in Flume, since it is necessary to preset the origin and destination sources and these will not change during the execution. This feature makes Kafka a more general purpose framework than Flume.
- It can be used with Kafka as a Channel (*Flafka*), so it would be no longer necessary to implement the producer and the consumer.
- Although in Fig. 7 only one element of each type appears, an agent can have multiple elements of each type.
- Flume allows to use file-based Channels and memory-based Channels. Fault tolerance is provided with file-based Channels, since the data are stored on disk and can be recovered in the case of error. This does not happen with memory-based Channels.
- Only one Sink can be associated with a Channel.

Flume allows data pre-processing, but does not allow data replication. It has compatibility only with Spark Streaming and is not as general-purpose as Kafka. Different types of Channels can be used, both memory-based and file-based, providing fault tolerance only in the latter. For these reasons, it is usually employed to transport large amounts of data from one system to another.

5.3 Apache Nifi

Apache Nifi[3] is a technological framework that allows data to migrate from one file system to another. Data can

[3] https://nifi.apache.org/.

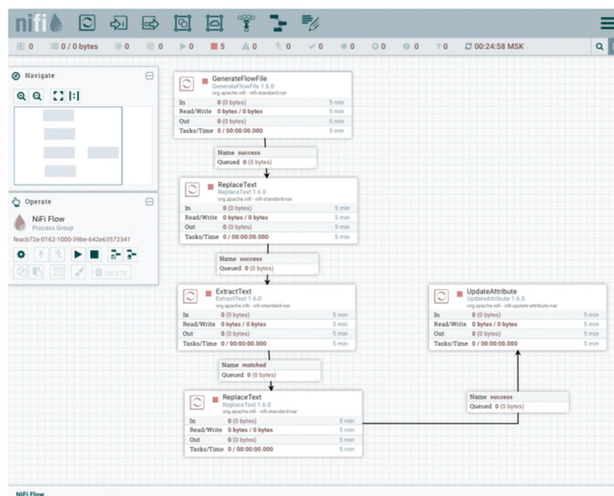


Fig. 8 General scheme of Apache Nifi

be processed by a sequence of operations to transform them before being sent to the destination. It has a web user interface to facilitate real-time visualization and the creation/configuration of nodes called *processor*, which performs operations on the data [26]. In Fig. 8 an example of an application in the web user interface is shown.

The pre-processing is carried out by the processors, which allows creating, deleting, modifying or inspecting data (called *FlowFiles*), and includes a content and a set of attributes that act as metadata. When defining the application, the data will be modified by different processors. Each processor will perform a different operation to pre-process the data and finally send them to the destination. Nifi has many implemented processors, including Flume and Kafka processors. The connection between processors is established by a queue.

The main features of Nifi are:

- Among the technological frameworks analyzed in Sect. 6, only Nifi is integrated into Flink.
- It has a web interface that facilitates definition of the application and the configuration of each of its elements.
- It allows transformations on the data, such as the conversion of data/formats, the delegation of functionalities and union and data broadcasting operations.
- It can be used with Kafka, so that it is not necessary to implement the producer and the consumer.
- It allows users to monitor the flow status in real time, identifying possible errors.
- It allows users to send data to multiple destinations at the same time.
- Nifi does not allow data replication. If a node goes down, the flow can be directed to another node, but the data in the queue of the failing node is maintained until the node goes up again.

Table 3 Summary of characteristics of ingestion frameworks

Characteristics	Frameworks		
	Kafka	Flume	Nifi
Analyzed version	2.1.0 (11/2018)	1.9.0 (01/2019)	1.8.0 (10/2018)
Pre-processing	Yes	Yes	Yes
Processing	Yes*	No	No
Delivery guarantee	At least one	At least one	At least one
Fault tolerance	Yes	Yes*	Yes
Programming languages	Java	Java	Java
Monitoring utility	No*	No	Yes
Platforms	Windows Linux MacOS	Windows Linux MacOS	Windows Linux MacOS
Documentation	Very large	Large	Large
Data replication	Yes	No	No
Data abstraction	Record	Event	FlowFile
Processing frameworks integration	All	Spark streaming	Flink

See information in the corresponding section describing the framework

- Like Flume, Nifi is responsible for obtaining data from the source. This is a difference with respect to Apache Kafka, where the producer is responsible for depositing the data. This makes Kafka more general-purpose.

Nifi allows data pre-processing but does not allow data replication. It only has compatibility with Flink and it is not as general-purpose as Kafka. It has a graphical interface that facilitates the definition of the application and allows monitoring of the flow in real time, so it is usually used when it is necessary to move data from one site to another and monitor the data streams.

5.4 Summary of frameworks for data ingestion

In this section three technological frameworks for data ingestion have been analyzed, providing their properties. In Table 3 a summary with the characteristics of the different data ingestion frameworks explained in this paper is shown. All these frameworks are multi-platform and allow data pre-processing, but only Kafka can carry out processing tasks, albeit at low performance (it is better to perform processing with a processing framework). The three technological frameworks have the same delivery guarantee, at least once, so data can have duplicates. With respect to fault tolerance, this is supported by the three technological frameworks, but in Flume is it only supported when using a file channel. Kafka and Flume do not have a monitoring utility, but Kafka has external third-party utilities which perform this function, like Burrow of LinkedIn. Only Kafka supports data replication and has a dynamic structure, so it is more dynamic and general-purpose than other ingestion frameworks.

For the data ingestion phase Kafka is the most widely used by the research community, because the system offers many advantages over other ingestion frameworks, like data replication, and a more dynamic structure. Being the most used technological framework, it has compatibility with all the processing frameworks seen in Sect. 6 and, for this reason, has much more documentation than the remaining ingestion frameworks.

In order to install these technological frameworks it is necessary to have Java version 8 or higher installed, since they only allow Java to be used as a programming language. In addition, these three ingestion frameworks are multi-platform, so they can be installed in Windows, Linux or MacOS. Additionally, Kafka needs to use Zookeeper and Flume needs the Hadoop Distributed File System (HDFS). ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.

6 Frameworks for data processing

When data are ingested into the system, it is necessary to use a processing framework to process the data and discover knowledge within them by applying a machine learning algorithm. In this section, Apache Spark (streaming mode and structured streaming mode), Storm, Flink, Samza, Apex and Beam are analyzed and compared, highlighting their main characteristics.

6.1 Apache Spark

Apache Spark is an open-source unified analytics engine for big data processing that allows distributed processing to split computing over different machines and reduce the execution

Fig. 9 General scheme of Apache Spark Streaming

time when the amount of data is enormous. Spark has different built-in modules that offer some algorithms and utilities to facilitate new implementations:

- *Streaming* This allows to process data streams that can arrive from different types of sources.
- *SQL* It is a module for structured data processing. The interfaces provided by Spark SQL have more information about both the data and the computation being performed. This additional information allows to perform extra optimizations.
- *MLLIB* This is the machine learning library from Spark that includes several types of machine learning algorithms and data structures and utilities to make practical machine learning scalable and easy.
- *GraphX* This module is a component for graphs and graph-parallel computation.

In the next two subsubsections, streaming modes used in this paper are explained along with their main features.

6.1.1 Apache Spark Streaming

Apache Spark has different operating modes. One of these modes is Apache Spark Streaming[4], an extension of Spark's core that allows scalable, high-performance and fault-tolerant processing of data streams. Data arrive through one or more input streams that can be processed using some transformations implemented by Spark Streaming. Later, processed data can be transferred to a file system or a database.

Spark Streaming works as follows [27] (Fig. 9 shows the general scheme): the user defines a time interval, called *batch interval*. For example, let us suppose that the batch interval is 2 s. When the Spark Streaming application starts, all data received each 2 seconds is stored in a different RDD (data from seconds 0 to 2 on RDD0, data from seconds 2 to 4 on RDD1, etc). This sequence of RDDs is abstracted in another data structure called *Discretized Stream* or *DStream*. Afterwards, each RDD from DStream is processed by Spark Engine through an algorithm, generating an output RDD for each input RDD.

When programming in Spark Streaming, a read-only abstraction called *RDD (Resilient Distributed Dataset)* is used. RDD is a set of data partitioned between distributed nodes of a cluster that can be processed in parallel. It supports two types of operations: transformations and actions. When a transformation is performed, a new RDD is generated with the changes made and the original one remains unchanged. When an action is executed, the value of performing a calculation on an RDD is returned.

Some of the main features about Spark Streaming are:

- It allows users to perform data processing in traditional mode (static dataset or batch) or in streaming mode (an adaptation of the batch mode, in which the stream is divided into micro-batches), based on the RDD or DStream abstraction.
- It allows the use of time windows. A window size and a sliding (a multiple of the batch interval) have to be defined. This is a window that contains some RDDs and processes all data from these RDDs as a whole.
- The latency (time that data have to wait from when they arrive in the system until they are processed) is greater than in other technological frameworks, since it does not perform continuous processing of the data. Spark Streaming always has to wait for the time of a micro-batch.
- There are several compatible machine learning libraries, such as MLLIB, with many algorithms already implemented. These algorithms are shown in Sect. 7.

Spark Streaming is useful in those cases where it is necessary to process data in batch mode or streaming mode, using micro-batches. In addition, it is notable the existence of machine learning libraries, its extensive documentation and its active community.

6.1.2 Apache spark structured streaming

Another Spark mode is Apache Spark Structured Streaming[5], built on the Spark SQL engine, which allows users to process a stream in a similar way to the batch mode over static data. Structured Streaming allows users to make queries that are resolved incrementally, updating the final result which is stored in an output table according to the *output mode* used, which specifies how the output table will be updated [28]. There are three output modes in Spark Structured Streaming:

- Append mode: This is the default mode, in which only the new tuples that have appeared since the last update are added to the output table. This mode is only supported by those queries where the existing tuples in the output table are not modified (only new tuples are added).

[4] https://Spark.apache.org/streaming/.

[5] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.

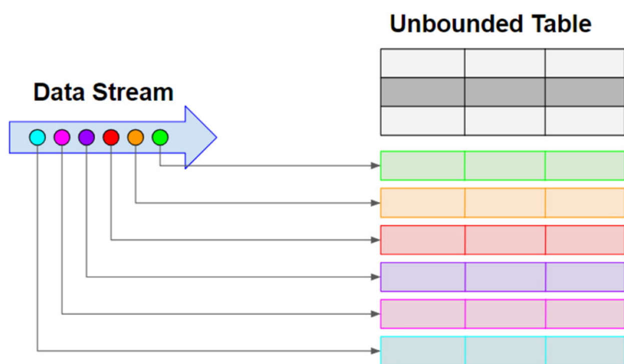


Fig. 10 Data abstraction of structured streaming

– Complete mode: All the results are added to the output table in each update. This mode is supported by aggregation queries.
– Update mode: It only updates those tuples whose value has been modified since the last update.

In the streaming mode there is an unlimited input table to which data are added as they arrive at the stream (Fig. 10). On this table, the incremental queries are realized. These queries process the data in micro-batches as low as 100 milliseconds, although from the version 2.3 of Spark a new mode, called *Continuous processing*, has been introduced. It allows latencies below 1 millisecond with processing guarantees of at least once. However, it is an experimental mode still in development.

The abstraction used when programming in Spark Structured Streaming is *Dataset*/*DataFrame*. A Dataset is a distributed data collection, which provides the benefits of RDDs along with those of Spark SQL and that can be manipulated using transformations. A DataFrame is a Dataset organized in columns, similar to a table in the relational database model.

Some points to highlight about Spark Structured Streaming are:

– It allows processing in batch mode and streaming mode, on the basis on the Dataset/DataFrame abstraction.
– Depending on the type of query made, append mode, complete mode or update mode can be used.
– In this model, Spark is responsible for maintaining and updating the tables with the new data, while in other models the user is responsible.
– It allows the user to use time windows like Spark Streaming, with the difference that here the batch interval does not have to be specified. If the batch interval is not specified, Spark Structured Streaming performs the processing as soon as possible.
– Along with the time windows *watermarks* can be used, which set a limit for data arriving late, based on a timestamp field. If an instance arrives late and does not exceed the watermark, it will be accepted to be processed on its corresponding time instant, not on the arrival instant.
– No libraries with data mining algorithms are available for Spark Structured Streaming.

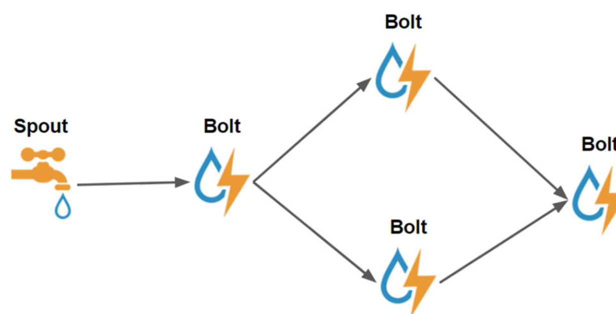


Fig. 11 General scheme of Apache Storm

Spark Structured Streaming is useful in those cases related to queries to databases, thanks to its incremental queries over the input data. In addition, the use of watermarks with time windows can be a factor for choosing this technological framework.

6.2 Apache storm

Apache Storm[6] is a real-time distributed computing system that processes the input data one by one. A stream in Storm is an unlimited sequence of data, in which each instance is a *Tuple* that contains an immutable ordered set of key-value pairs.

In Storm the flow will go through a series of nodes with each one performing a data processing task. An example of an application of Storm can be seen in Fig. 11. There are two types of nodes in a Storm application [29]:

– *Spout* This regulates the input data to the system. This node reads data from an external source and stores them in a queue, to introduce data in the system according to the desired input speed. A node can generate more than one stream.
– *Bolt* It performs operations over data. Optionally, additional tuples can be generated.

Storm has two operating modes: normal mode (processes the data one by one) and another mode that uses a higher-level abstraction called *Trident* (processes the data by micro-batches). This second mode offers processing guarantees of exactly once, although it increases the latency because the system has to wait to collect a micro-batch.

6 http://storm.apache.org/.

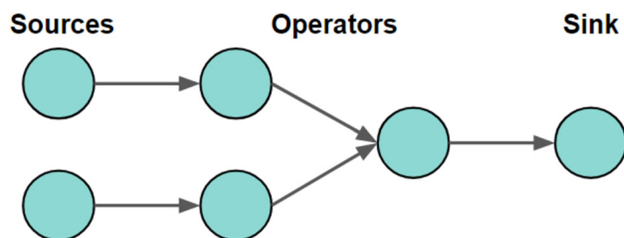


Fig. 12 General scheme of Apache Flink

The main features of Storm are:

- It only allows data processing in streaming mode, one by one or in micro-batches (with Trident).
- Storm does not guarantee that data will be processed in order.
- It allows the use of both time windows and watermarks.
- Storm executes nodes as tasks, which allows users to execute the function of a node in parallel in order to distribute the work using multiple machines. This is useful when there are some tasks in the application which take longer to compute.
- Storm does not have high-level functionalities implemented, so all the functionalities of the nodes have to be programmed, which can be a drawback.
- There is currently no library with data mining algorithms, although there is a library under development called SAMOA[7]. Current algorithms are included in Table 5.

Storm is useful in those cases where the tasks are distributed by nodes and the data have to be processed one by one with at least once processing guarantees.

6.3 Apache Flink

Apache Flink[8] is a processing framework for distributed data streaming applications that allows the user to process the data one by one or by micro-batches. It also allows processing in traditional (or batch) mode, with a static dataset.

In Flink (Fig. 12) there are 3 types of nodes [30]:

- *Source* It regulates the input data to the system, similar to Spout nodes in Storm.
- *Operator* It performs transformations on the input data and generates output data, similar to Bolt nodes in Storm.
- *Sink* It deposits the results in the destination.

The core data structure of Flink is *DataStream*. When a data stream arrives at the system it is in DataStream state, and this state changes after each operation. Depending on the current state of the stream, a node could perform some transformations or others. The possible states are [14]:

- *DataStream* It is the core structure of Flink data stream API.
- *KeyedStream* It represents a data stream where elements are grouped by a key.
- *WindowedStream* It represents a data stream where elements are organized into groups by key, and each group is windowed.
- *ConnectedStream* It is a way to share states between two tuple-at-a-time operations. It can be thought of as executing two map (or flatMap) operations on the same object.

Flink is very powerful in terms of window usage, since it allows a great variety. Windows can be count-based or time-based. Within a window it is also possible to distinguish between events of different types. There are different types of windows already implemented, but Flink even allows users to implement a custom window if it is necessary.

The following points can be highlighted about Flink:

- It allows users to process data in streaming or in traditional mode. In addition, processing the data one by one or in micro-batches can be chosen.
- Like Storm, nodes are executed as tasks, which allows the execution of a node in parallel using multiple machines.
- Flink has an API that allows programming at a higher level, while in Storm all the functionality of the nodes has to be programmed. This is an important difference with respect to Storm.
- To provide fault-tolerance Flink uses *Checkpoints*, which give the summation of the flow at a specific moment. These checkpoints are taken with a frequency called checkpoint intervals.
- Flink implements a wide variety of windows, which also allow the use of watermarks to accept a delay in the input data. This is an advantage with respect to the other technological frameworks.

Flink is useful when data have to be processed one by one or in micro-batches, with exactly once processing guarantees, and both for traditional and streaming mode. Flink is very similar to Storm, but it provides features that have to be implemented in Storm. In addition, the possibility of using time-based windows together with watermarks or count-based windows can be an important factor for choosing this framework.

[7] https://samoa.incubator.apache.org.

[8] https://flink.apache.org/.

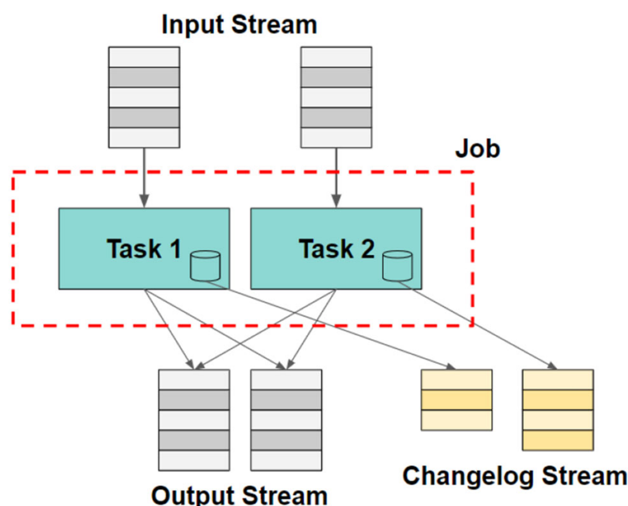


Fig. 13 General scheme of Apache Samza

6.4 Apache Samza

Apache Samza[9] is a distributed streaming processing framework that uses Kafka and *YARN* [31], which is the improved version of Apache Hadoop[10]. Samza offers many functionalities similar to Storm and processes data streams using predefined *Tasks*, which perform operations on the data stream. A Samza application is a data flow that consists of consumers who obtain data that are processed by a graph of *Jobs*, where each Job contains one or more Tasks. In Samza, each Job is an entity that can be deployed, started or stopped immediately [32]. An example with one Job that contains two Tasks can be seen in Fig. 13.

A stream can have multiple producers writing data in it and multiple consumers that read data from it. A stream can be divided into multiple partitions, so the number of consumers can be adjusted to the processing for scaling how these data are processed. Each partition is an immutable and ordered sequence of data, so when new data are written they are added to the end of a partition.

In Samza, each Task contains a key-value store used to store the state. Changes in this store are replicated to the other machines in the cluster to allow tasks to be recovered quickly in case of failure. It also offers the possibility of remotely storing the status.

Samza also presents the following characteristics:

- It guarantees that messages are processed in order.
- It uses single-thread processes.
- Each Task processes the data of an input stream partition.
- It provides fault-tolerance by *incremental checkpointing*, storing the status of each task to facilitate system recovery in case of failure.
- No library with data mining algorithms is available, although in the future it will be able to use SAMOA algorithms. Current algorithms are included in Table 5.

Samza is useful in cases where data have to be processed one by one. Samza has been implemented over Kafka, so the data will be stored in partitions and ensures that they will be processed in order.

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

[10] https://hadoop.apache.org/.

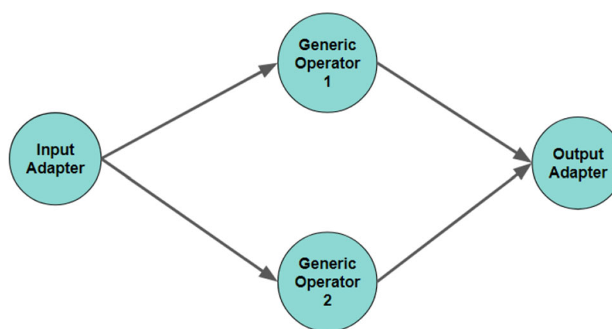


Fig. 14 General scheme of Apache Apex

6.5 Apache Apex

Apache Apex[11] is a native processing platform based on YARN [31]. Apex provides a simple API, which allows users to write generic and reusable code.

In Apex there are nodes called *Operators*, which take one or more input streams, perform a computation and emit one or more output streams. In Apex there are three types of nodes (Fig. 14), which have a similar functionality to those of the previous technological frameworks [33]: *Input Operator*, *Generic Operator* and *Output Operator*.

The core of the Apex platform is complemented by *Malhar*, a library of logical functions and connectors. It provides access to different file systems, message systems and databases. In addition, the library has some demos that illustrate the capabilities and characteristics of the operators.

Some Apex characteristics are presented below:

- The data are processed one by one, although it also supports micro-batch processing using time windows.
- It provides fault tolerance by the use of *checkpoints*. When a node goes down, its recovery is instantaneous, since it uses a *Heartbeat* mechanism. This mechanism checks periodically the state of nodes and tries to raise it in the case of detecting a fallen node.

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

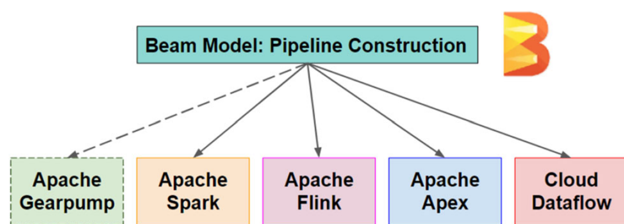


Fig. 15 General scheme of Apache Beam

- Malhar is a support library for application development. Developers have to implement any unimplemented functionality in order to use it.
- It does not provide a library with data mining algorithms, although in the future it will be able to use SAMOA algorithms. Current algorithms are included in Table 5.

Apex is useful when the data have to be processed one by one with exactly once processing guarantees, and reusable code will be used. It also has compatibility with the Malhar library, which can be an important factor in choosing this technological framework.

6.6 Apache Beam

Apache Beam[12] is a streaming processing framework that allows users to define a processing independent of the data processing system (called *runner*) used. It allows users to change the runner using the same implemented processing. The following runners can be used in Beam [34]: Apache Apex, Apache Flink, Apache Spark, Apache Gearpump and Google Cloud Dataflow. A general scheme of Beam can be seen in Fig. 15.

Some points to highlight regarding Apache Beam are:

- The processing guarantee depends on the runner used to process the data, since each one offers its own guarantees.
- Both Java and Python can be used as the programming language, although there is also a Scala API in a GitHub project[13].
- The latency varies depending on the runner used.

Beam is useful for testing the processing with different runners in order to see which one suits better.

6.7 Summary of frameworks for data processing

In this section, seven technological frameworks for data processing are analyzed, presenting their characteristics and the differences between them. In Table 4 a summary with the characteristics of the different data processing frameworks analyzed in this paper is shown. Each technological framework uses its own notation for the nodes, but the idea is similar (they have one node to introduce the data into the system and another one to perform operations and, optionally, send the results to a given destination).

All these technological frameworks have a monitoring utility to manage the system, offer fault tolerance and are multi-platform. With respect to the processing guarantee, Spark, Flink, Apex and Storm (with Trident) offer exactly once processing guarantees, and Storm (without Trident) and Samza offer at least once processing guarantees. There is a variety of programming languages to use in all these technological frameworks, except in Samza and Apex, which only allow programming in Java. When a framework processes data, Spark, Storm (with Trident) and Flink can use a micro-batch model, and Storm (without Trident), Flink, Samza, Apex and Beam can use a one-by-one model (true streaming). Only Flink and Storm allow the use of both models. Lastly, machine learning libraries are another important factor in a processing framework. Spark and Flink are the technological frameworks with more implemented algorithms of this paper, but Spark has more than Flink in both batch and streaming modes.

For the data processing phase, Spark is the most widely used by the research community because it allows stateless and stateful processing and has a very complete documentation, as well as a machine learning library (MLLIB) with a large number of implemented algorithms. In addition, Spark allows data processing in batch and streaming modes and has other developed components like Spark SQL that can be integrated in applications.

To be able to install these technological frameworks it is necessary to use one of the programming languages supported by each framework. This is shown in Table 4. However, a specific version of the different languages has to be used:

- Java: 8 or higher.
- Scala: 2.11 or higher.
- Python: 2.7/3.4 or higher.
- R: 3.1 or higher.
- GO: 1.10 or higher.

In addition, it is necessary to install Zookeeper and use one of the operating systems supported by each technological framework, as shown in Table 4.

7 Machine learning libraries

One of the most important factors when choosing a technological framework for data processing is the existence of

[12] https://beam.apache.org/.

[13] https://github.com/spotify/scio.

Table 4 Summary of characteristics of processing frameworks

Characteristics	Frameworks						
	Spark streaming	Spark structured streaming	Storm	Flink	Samza	Apex	Beam
Analyzed version	2.4.0 (11/2018)	2.4.0 (11/2018)	1.2.2 (06/2018)	1.7.1 (12/2018)	1.0.0 (11/2018)	3.7.0 (04/2018)	2.9.0 (12/2018)
Processing guarantee	Exactly once	Exactly once	At least one Exactly once	Exactly once	At least one	Exactly once	Depend*
Fault tolerance	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Programming languages	Java Scala Python R	Java Scala Python R	Java Scala Python Ruby	Java Scala Python	Java	Java	Java Scala* Python
Monitoring utility	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Platforms	Windows Linux MacOS	Windows Linux MacOS	Windows Linux	Windows Linux MacOS	Windows Linux	Windows Linux MacOS	Windows Linux MacOS
Documentation	Very large	Large	Large	Large	Little	Little	Large
In-flight modifications	No	No	Yes	No	No	Yes	No
Processing unit	Micro-batch	Micro-batch	One by one Micro-batch	One by one Micro-batch	One by one	One by one	One by one
Data abstraction	DStream	Dataset DataFrame	Tuple	DataStream	Message	Tuple	PCollection
Windowing	Time-based	Time-based	Time-based Count-based	Time-based Count-based	Time-based	Time-based	Time-based
Processing type	Stateless Stateful	Stateful	Stateless	Stateless Stateful	Stateless Stateful	Stateless Stateful	Stateless Stateful
Machine Learning library	MLLIB streamDM AmidstToolbox	No	SAMOA*	FlinkML SAMOA* AmidstToolbox	SAMOA*	No	No

*See information in the corresponding section describing the framework

machine learning libraries with implemented algorithms and methods that facilitate the use of the framework. Currently, there are not many data mining algorithms for streaming data implemented in these libraries. In this paper the following machine learning libraries are analyzed:

- MLLIB (MLL) [35] is the scalable machine learning library of Apache Spark that is included in the Spark project, so it is tested and updated with each new Spark version. It provides many useful algorithms and functions, such as statistical calculation and normalization.
- StreamDM (SDM) [36] is an open source library developed by the Huawei Noah's Ark Lab to perform streaming data mining using Apache Spark Streaming.
- FlinkML (FML) [30] is the machine learning library of Apache Flink, which provides scalable machine learning algorithms, an intuitive API and tools that help to minimize the use of these algorithms.
- SAMOA (SAM) [37] is a distributed streaming machine learning framework that contains an abstraction for programming distributed streaming machine learning algorithms. The idea of this framework is to implement distributed streaming machine learning algorithms without having to know the processing engine that executes them. It is currently in incubation.
- Amidst Toolbox (AMT) [38] is a scalable, probabilistic machine learning library developed in Java, with a special focus on data streaming. It is a library that can be used with Spark and Flink, but currently Spark's link is in development.

Table 5 shows the available algorithms implemented by the machine learning libraries analyzed in this paper. As can be seen, there are 20 algorithms in MLLIB, 9 in streamDM, 6 in FlinkML, 6 in SAMOA and 14 in Amidst Toolbox. Not all algorithms are for streaming processing. Currently, there are not many streaming algorithms implemented in these libraries, so more streaming algorithms must be implemented for technological frameworks.

Table 5 Available algorithms

Algorithms	MLL	SDM	FML	SAM	AMT
SVM	×	×	×		
Logistic regression	×	×			
Linear regression	×	×	×		
Streaming logistic regression	×				
Streaming linear regression	×				
DecisionTree	×				
RandomForest	×				
Gradient boosted trees	×				
NaiveBayes	×	×			×
Isotonic regression	×				
ALS	×		×		
K-means	×				
Gaussian mixture	×				×
PIC	×				
LDA	×				
Bisecting K-means	×				
Streaming K-means	×				
FPGrowth	×				
Association rules	×				
PrefixSpan	×				
Multilayer perceptron	×				
k-NN	×		×		
SGD			×		
SOS			×		
Perceptron		×			
CluStream		×			
StreamKM++		×			
Hoeffding decision trees		×		×	
OnlineBagging		×		×	
AMRules				×	
OnlineBoosting				×	
Adaptive bagging				×	
Distributed stream clustering				×	
TAN					×
AODE/HODE					×
Dynamic NB					×
Gaussian discriminant analysis					×
LCM					×
Bayesian linear regression					×
FactorAnalysis					×
DynamicLCM					×
HMM					×
KF					×
SwitchingKF					×

8 Comparison of technological frameworks for data processing

In this section the different technological frameworks for data stream processing analyzed in Sect. 6 are compared, with a streaming processing benchmark used in other studies [6,16–18] called *Yahoo Streaming Benchmark*, which evaluates *latency* and *throughput*. Latency is the time that an instance remains in the system from when it arrives until it is processed. For its part, throughput measures the number of instances that are processed per unit of time (usually seconds). In this paper the comparison is increased by adding

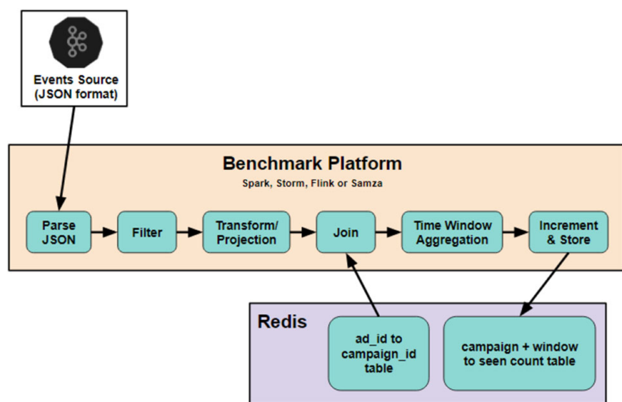


Fig. 16 General scheme of Yahoo Streaming Benchmark

Samza to the comparison and updating versions of technological frameworks: Spark (2.4.0), Storm (1.2.2), Flink (1.7.1) and Samza (1.0.0). In addition, a different configuration of nodes is used (this is detailed in the following subsection).

In this section each experimentation is detailed and then the results obtained from each experimentation are analyzed.

8.1 Experiment design

The Yahoo Streaming Benchmark [6], that was developed in 2016 by Yahoo Inc, is used for the comparison. The source code is available in [39]. It simulates a simple advertisement application that has 100 campaigns, 10 ads per campaign and uses a time window of 1 second. Ads arrive at the system and the objective is to count the number of relevant events of each campaign on each time window, measuring the latency of input data for different throughputs. Each ad has seven fields:

- user_id: Id of the user.
- page_id: Access page to ad.
- ad_id: Id of the ad.
- ad_type: Type of ad. This field can be banner, modal, sponsored-search, mail or mobile.
- event_type: Type of event, which can be view, click or purchase.
- event_time: Timestamp when event was generated.
- ip_address: IP address from access.

Figure 16 shows the flow of operations in this benchmark:

1. *Read Kafka events* Get events of the input stream.
2. *Extract the JSON information* Data arrive in JSON format, so in this step data are converted into the objects we are working with, extracting the information from the fields.
3. *Filter irrelevant events* Through the input, stream events with different values of *event_type* can be received. This step filters events and only those with *view* value are kept.
4. *Transform/Project fields* On each instance there are a series of values and fields. In this step the fields that interest us are projected. These are *ad_id* and *event_time*.
5. *Join* In Redis there is a table that includes the *campaign_id* that each *ad_id* has. In this step the data from the Redis table are obtained and joined to the events.
6. *Time window aggregation* In this benchmark time windows are used, so in this step the number of campaigns within each window are counted.
7. *Store results* Finally, the results obtained are sent to Redis.

In comparison [6] Spark Streaming (1.5.1), Storm (0.11.0) and Flink (0.10.1) were compared. In 2018 another comparison with Yahoo Streaming Benchmark [16] was performed. It makes the same comparison but updating the version of technological frameworks from the previous comparison: Spark Streaming (2.3.0), Storm (1.2.1) and Flink (1.5.0), and adding Spark Structured Streaming (2.3.0) and Kafka Streams (1.1.0). The source is available in [40].

In our benchmark, different executions are run in Spark Streaming (2.4.0), Spark Structured Streaming (2.4.0), Storm (1.2.2), Flink (1.7.1) and Samza (1.0.0) with different rates for each one, from 4000 to 88000 *Transactions Per Second* (TPS). The final event latency of each processed window is calculated as follows:

$$\begin{aligned} &\text{final_event_latency} \\ &= (\text{window_last_event_updated} - \text{window_start}) - \text{window_duration} \end{aligned} \tag{1}$$

In experiments, ten homogeneously configured nodes were used, each one with 64-bit Ubuntu 16.04 LTS as Operative System, Intel Core i5 750 processor at 2.67 GHz (4 cores per node), 4 GB of memory RAM at 1066 MHz and connections of 1 GB/s. All the nodes were connected through a gigabit Ethernet switch.

The architecture of the benchmark can be seen in Fig. 17, in which 2 nodes are for Kafka Producers, 1 node is for Apache Zookeeper, 1 node is for Kafka broker, 1 node is for Redis and 5 nodes are for Stream Processors. In each Stream Processor Spark, Storm, Flink and Samza were installed. Apex is not included in this benchmark because it uses software from DataTorrent and the web supporting this library has closed.

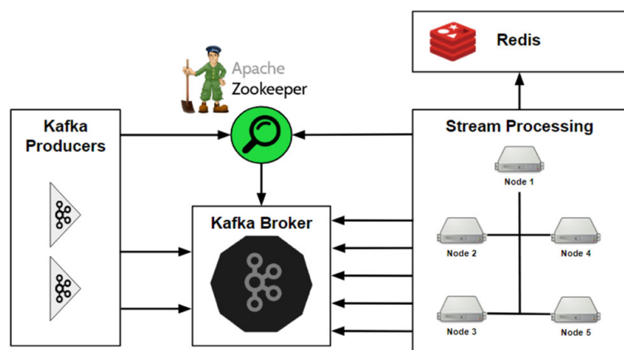


Fig. 17 Benchmark architecture for experiments

8.2 Analysis of results

Now, the results obtained are commented for each streaming processing framework. In these results, the latency obtained for each experiment can be seen. The objective of this experiment is to find a bottleneck in the latency, which would indicate that the technological framework receives more data than it can process by time unit.

Latency can be more or less constant if the processing time is lower than the time window, or increases linearly or even exponentially, because data have to wait for the processing to finish. An increment in latency indicates that the system receives more data than it can process. Depending on the amount of data received per second, latency increase linearly or exponentially. When latency increases exponentially, this indicates that there is a bottleneck. With these experiments, the latency of each streaming processing framework for each TPS is analyzed and we find out the TPS at which the bottleneck appears.

Table 6 shows a summary of the individual experiments carried out on each processing framework analyzed. The first row shows the TPS needed to produce a bottleneck (latency increases exponentially). The last two rows show values in seconds of the lowest and highest latency for each technological framework, respectively.

Each graph in Fig. 18 shows the evolution of latency by the percentage of completed examples for the different technological frameworks analyzed.

In Fig. 18a it can be seen that Spark Streaming has a latency under 10 s between 8 K TPS and 80 K TPS and that the latency increases linearly. In 88 K TPS there is a bottleneck, because the latency increases exponentially and reaches around 55 s. Thus, the results obtained show that Spark Streaming has a good performance bellow 80 K TPS.

The second streaming processing framework analyzed was Structured Streaming, whose results are shown in Fig. 18b. Between 8 K TPS and 80 K TPS, latency was under 24 s and for 88 K TPS it reached 33 s of latency. This indicates that a bottleneck appears after 80 K TPS, but the latency is lower than in Spark Streaming for 88 K TPS. For this reason, Spark Structured Streaming has better performance than Spark Streaming and Samza for higher TPS.

In Fig. 18c it can be seen that latencies of Storm are much higher than other streaming processing frameworks in all TPS. This is the technological framework in which the bottleneck has appeared earliest. From the results, obtained it can be deduced that Storm is not good for the streaming problem proposed in Yahoo Streaming Benchmark.

For Flink the best latencies of the benchmark for all TPS was obtained. In Fig. 18d it can be seen that between 8 K TPS and 80 K TPS the latency has a more or less linear increase, with latencies of between 4 s and around 5 s. At 88 K TPS the bottleneck appears, but the latency is of around 10 s, the lowest latency of all streaming processing frameworks.

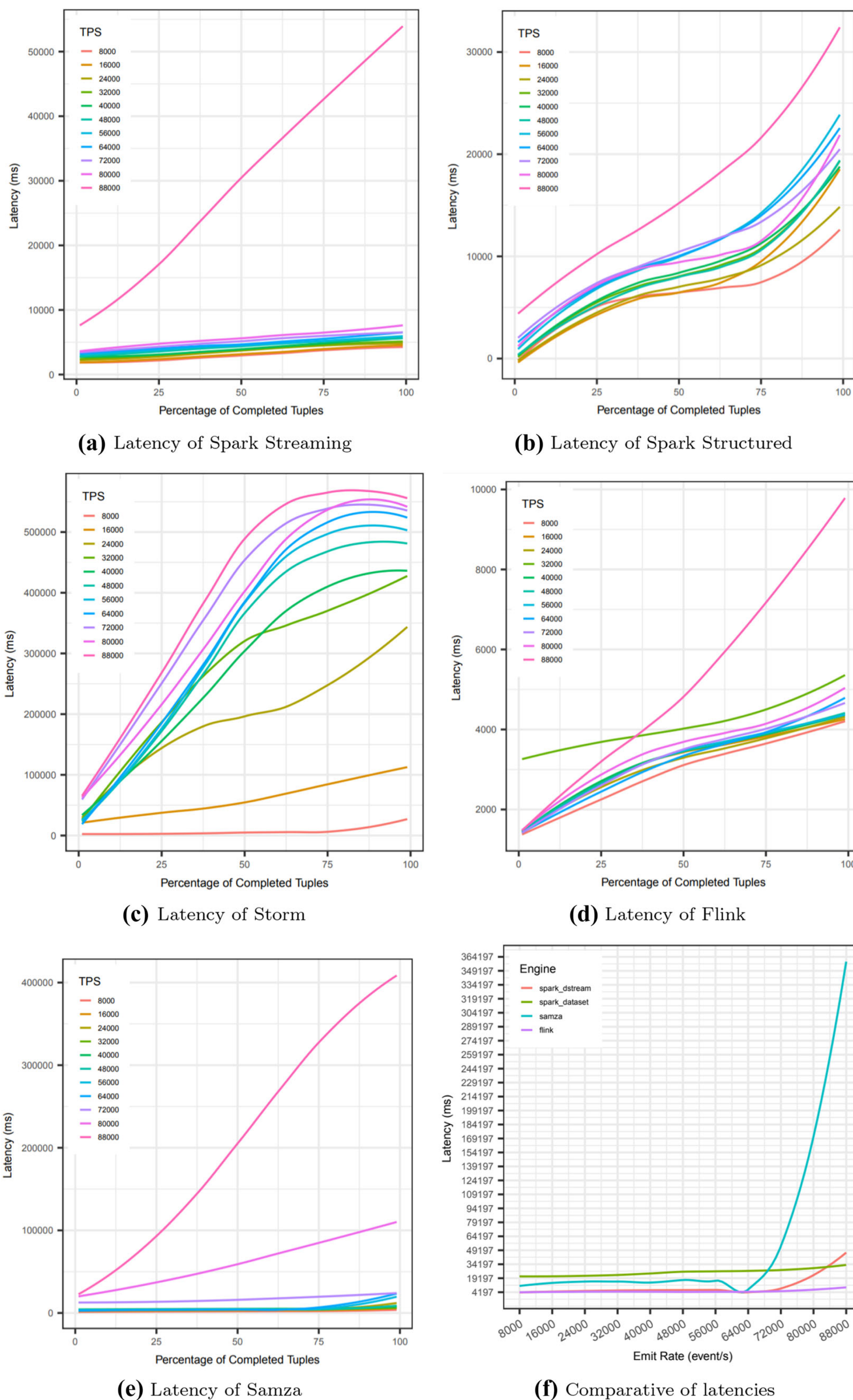
The results for Samza are shown in Fig. 18e, where it can be observed that the latency is under 10 s from 8 K TPS to 48 K TPS. Between 48 K TPS and 72 K TPS the latency reaches around 20 s, and for TPS greater than 72 K, latency increases exponentially, reaching 400 s in the case of 88 K TPS. These results show that Samza has a good performance when TPS is low.

Lastly, the results of Spark Streaming (spark_dstream), Spark Structured Streaming (spark_dataset), Flink and Samza are compared in Fig. 18f, using the same scale on axis to better evaluate the performance. Storm is not included in this comparison because a very low performance was obtained with respect to the other streaming processing frameworks.

Spark Streaming has a better latency than Spark Structured Streaming for 80 K TPS or lower, but for more than 80 K TPS Spark Structured Streaming maintains a good performance while Spark Streaming increases exponentially. Samza has a good performance until 64 K TPS, but after this latency has an exponential increase for high TPS. Flink's latency is more or less constant at around 5 s for all TPS, until higher TPS, where it starts to increase faster, but always slower than the other streaming processing frameworks.

Table 6 Summary of experiments

	Spark streaming	Spark structured streaming	Storm	Flink	Samza
Bottleneck (TPS)	88 K	88 K	16 K	88 K	72 K
Lowest latency (S)	4	12.5	25	4	4
Highest latency (S)	58	32.5	550	10	400



(a) Latency of Spark Streaming

(b) Latency of Spark Structured

(c) Latency of Storm

(d) Latency of Flink

(e) Latency of Samza

(f) Comparative of latencies

Fig. 18 Benchmark results

According to our results Flink has a better behavior in this benchmark than the other technological frameworks for streaming processing, because it has the lowest latency for all TPS.

9 Comparison of machine learning algorithms for data stream

In this section, a new comparison with a machine learning algorithm over some technological frameworks for data stream processing seen in Sect. 6 is performed.

In the experiments, Apache Kafka was used for data ingestion because, as was shown in Sect. 5, it has compatibility with all the technological frameworks seen in this paper and is the most used in the literature. For their part, Apache Spark and Apache Flink were used for data processing, because Flink had the best performance in the previous comparison and Spark was the second. Although Spark Structured Streaming mode was the second best framework, Spark Streaming mode was used for experiments, because Spark Structured Streaming does not have a machine learning library with streaming algorithms, but Spark Streaming has MLLIB, which includes some streaming algorithms.

In the experiments, the training time of the processing frameworks was measured using a machine learning algorithm from MLLIB called *streaming K-means*.

Flink has no implementation for streaming K-means, so in this study a new implementation of this algorithm for Flink was performed. This implementation for Flink is the same algorithm as Spark, but uses sequential windows instead of time windows. In Spark, this algorithm updates the model at each batch interval time, but in Flink the model is updated every certain number of examples. The Flink implementation has an update frequency, so at each certain number of instances, the model is updated. It is updated in the same form in both implementations.

Four datasets with different sizes were used to train the algorithm. There were two real and two synthetic datasets. The main characteristics of these datasets are described below:

- The Covtype dataset is a real dataset that contains 581,012 instances and 54 integer attributes and models the forest cover type prediction problem from cartographic variables.
- The Power dataset is another real dataset that contains 2,049,280 instances (after removing instances with missing values) and 7 floating-point attributes and measures electric power consumption in one household at a one-minute sampling rate over a period of almost four years.
- The SEA dataset is a synthetic dataset generated by MOA [41] that contains 5,000,000 instances and 3 floating-point attributes, where only two are relevant attributes: $x_i \in [0, 10]$, where $i = 1, 2, 3$. The target concept is $x_1 + x_2 \leq \beta$, where $\beta \in \{7, 8, 9, 9.5\}$. We have used $\beta = 8$.
- The RBF dataset is another synthetic dataset generated by MOA [41] that contains 10,000,000 instances and 20 floating-point attributes. This generator first creates a fixed number of random centroids and, for each instance to generate, it selects a centroid and generates an instance inside this centroid.

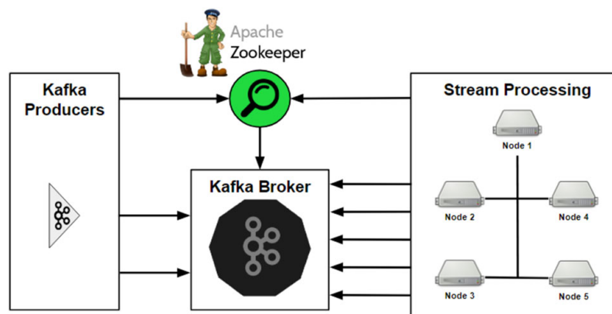


Fig. 19 Machine learning benchmark architecture

For the experiments, a cluster configuration with eight nodes was used. In one node, Apache Zookeeper was used to state management of the other nodes in the cluster. In another node, a Kafka producer was implemented to send data to Kafka broker. Apache Kafka was installed in another node to receive messages from the source and store them. Finally, five nodes with Apache Spark or Flink (one master and four slaves) were used to process input data stream with the streaming K-means algorithm. In Fig. 19 a scheme of the system used is shown. To run each experiment, first a cluster with the number of desired slaves (1, 2, 3 or 4) was created. Once the cluster was running, the Spark or Flink submit command was executed to send the jar file to master and a wait of around 20 s was needed for the streaming application to run in all slaves. After that, Kafka producer was run to send one dataset to a topic and the streaming application processed these data.

Table 7 shows the mean in seconds of 10 executions for Spark and Flink. In the different experiments, the number of worker nodes was changed, increasing this number one by one from 1 to 4. The results show that Flink has a better performance in this comparison than Spark in Power, SEA and RBF datasets, using more or less half the time. Training time decreases with the increment on the number of workers in the cluster. For example, let us suppose that with one node the training time is 40 seconds. If the number of workers increases to 2, this time will be 20 s and, with 4 workers, it will be 10 s. But as can be seen in Table 7, some values did not decrease as they should. This is because the send time

Table 7 Results of streaming K-means

Dataset	Workers	Spark streaming time (s)	Flink time (s)
Covtype	1	**13.6313**	18.2095
	2	**8.4619**	10.6988
	3	**6.3911**	7.9011
	4	**6.1384**	6.4233
Power	1	21.4361	**12.7472**
	2	11.8503	**8.5622**
	3	7.6643	**7.442**
	4	7.1309	**6.758**
SEA	1	59.6979	**29.172**
	2	32.1117	**22.8102**
	3	23.7461	**19.6678**
	4	20.7679	**18.7681**
RBF	1	676.6879	**320.9581**
	2	360.5873	**248.321**
	3	269.9693	**197.9395**
	4	237.5595	**185.5117**

Bold value is the best obtained time (in seconds) for each experiment

Table 8 Results of streaming K-means on covtype dataset

Dataset	Workers	Spark Streaming time (s)	Flink time (s)
Covtype	1	**13.6313**	18.2095
	2	**8.4619**	10.6988
	3	**6.3911**	7.9011
	4	**6.1384**	6.4233
Covtype2	1	**113.1811**	130.3124
	2	**60.5999**	68.94
	3	**43.8494**	47.64275
	4	38.3538	**37.88**
Covtype3	1	177.081	**134.2934**
	2	96.6267	**71.6498**
	3	73.1975	**53.5453**
	4	62.7564	**50.6911**

Bold value is the best obtained time (in seconds) for each experiment

limits the processing time. The Covtype and Power datasets do not show a good decrease because there are few data. In SEA dataset this decrease can be seen in Spark, and in the RBF dataset both Spark and Flink show this decrease.

For Covtype, Spark performs better than Flink, so another experiment with this dataset was performed in order to learn the reason why. In this second experiment, another two datasets were used. One dataset (called Covtype2) is the same dataset as Covtype, but has eight times its content. This dataset starts with a copy of Covtype and, when the data send ends, another copy is sent, up to eight times. The resulting dataset contains 4,648,096 instances and 54 integer attributes. The other dataset (called Covtype3) is the same dataset as Covtype2, but zeros are replaced by a decimal value. This dataset contains 4,648,096 instances and 54 integer and real attributes.

The results of the second experiment can be seen in Table 8. The experiment was the same as the previous one, with the mean of 10 executions. In Covtype2, Spark performed better than Flink in all experiments except when the number of workers was 4, so the reason why Spark is better is not the number of instances in the dataset. When seeing data from Covtype, it can be observed that it contains many zeros, so in the third dataset zeros were replaced with a decimal value. In Covtype3, Flink performed better than Spark in all experiments, so this shows that Spark uses a more efficient data structure. Spark uses a data structure called *Vector* instead of *Array*, which is more efficient in processing instances with a lot of zeros in their attributes. Our implementation of streaming K-means for Flink uses Array, so it is less efficient than Spark in these cases.

Both Spark and Flink scaled well according to the number of workers in the cluster but, in general, Flink showed a better performance with respect to Spark, except over datasets with many zeros, as the previous comparison of Sect. 8 shows. These results show that the implementation details and data structures chosen are important factors to take in account in order to guarantee good efficiency of algorithms.

10 Conclusions

There are many open source and proprietary streaming technological frameworks that can be used for distributed machine learning. This paper analyzes some technological frameworks for data ingestion and data processing phases.

For data ingestion, Kafka, Flume and Nifi were analyzed. Each one has their strengths and weaknesses, but the analysis shows that the best of them is Apache Kafka, because it is a technological framework widely used by the research community that offers advantages over the other ingestion frameworks, like the concept of topic, data replication, general purpose and the high compatibility with the processing frameworks.

For data processing, Spark Streaming, Spark Structured Streaming, Storm, Flink, Samza, Apex and Beam were analyzed. The main characteristics about each one were shown and two comparisons were performed to compare the performance of technological frameworks. In the first comparison, the latency of data in a cluster of 10 nodes was analyzed for Spark Streaming, Spark Structured Streaming, Storm (without Trident), Flink and Samza in a benchmark called Yahoo Streaming Benchmark. The results of this comparison show that Flink has better behavior than the other streaming processing frameworks for this benchmark. The second comparison evaluates the performance of Spark Streaming

and Flink with a streaming machine learning algorithm running in a cluster of eight nodes. Spark Streaming has the streaming K-means algorithm implemented in MLLIB, but Flink does not have a version of this algorithm implemented, so a new version of this algorithm with sequential windows was performed for this comparison. In this case the training time of the streaming K-means algorithm was measured over 4 datasets of different sizes, and the results of this comparison show that Flink is faster than Spark in datasets that do not contain a lot of zeros in attributes of data, because Spark uses a data structure called Vector instead of Array. The performance of a processing framework depends on its implementation, its data structures and the problem to solve, among other reasons.

With respect to machine learning algorithms for the streaming mode, at this moment there are not many streaming algorithms implemented in these libraries. Spark is the technological framework that has more streaming algorithms implemented of all the analyzed frameworks analyzed. For this reason, the implementation of more open-source libraries for machine learning on data stream is necessary.

References

1. Han, J., Kamber, M., Pei, J.: Data Mining Concepts and Techniques, 3rd edn. Morgan Kaufmann, Burlington (2011)
2. Gama, J.: Knowledge Discovery from Data Streams. Chapman and Hall/CRC, Boca Raton (2010)
3. Aggarwal, C.: Data Streams: Models and Algorithms, vol. 31. Springer, New York (2007)
4. Gürcan, F., Berigel, M.: Real-time processing of big data streams: Lifecycle, tools, tasks, and challenges. In: 2018 2nd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT), pp. 1-6. IEEE (2018)
5. Heudecker, N., Schulte, W. R.: Market guide for event stream processing. id:g00332885. Gartner (2018)
6. Chintapalli, S., Dagit, D., Evans, B., Farivar, R., Graves, T., Holderbaugh, M., Liu, Z., Nusbaum, K., Patil, K., Peng, B. J. et al.: Benchmarking streaming computation engines: storm, flink and spark streaming. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1789-1792. IEEE (2016)
7. Khamassi, I., Sayed-Mouchaweh, M., Hammami, M., Ghédira, K.: Discussion and review on evolving data streams and concept drift adapting. Evol. Syst. **9**(1), 1–23 (2018)
8. Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., Bouchachia, A.: A survey on concept drift adaptation. ACM Comput. Surv. (CSUR) **46**(4), 44 (2014)
9. Krawczyk, B., Minku, L.L., Gama, J., Stefanowski, J., Woźniak, M.: Ensemble learning for data stream analysis: A survey. Inf. Fusion **37**, 132–156 (2017)
10. Ramírez-Gallego, S., Krawczyk, B., García, S., Woźniak, M., Herrera, F.: A survey on data preprocessing for data stream mining: current status and future directions. Neurocomputing **239**, 39–57 (2017)
11. Schulte, W. R., Heudecker, N.: Technology insight for event stream processing. id:g00334449. Gartner (2017)
12. García-Gil, D., Ramírez-Gallego, S., García, S., Herrera, F.: A comparison on scalability for batch big data processing on apache spark and apache flink. Big Data Anal. **2**(1), 1 (2017)
13. Samosir, J., Indrawan-Santiago, M., Haghighi, P.D.: An evaluation of data stream processing systems for data driven applications. Proc. Comput. Sci. **80**, 439–449 (2016)
14. Wang, Y.: Stream Processing Systems Benchmark: Streambench (2016)
15. Lu, R., Wu, G., Xie, B., Hu, J.: Stream bench: Towards benchmarking modern distributed stream computing frameworks. In: 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, pp. 69-78. IEEE (2014)
16. Shahverdi, E.: Comparative Evaluation for the Performance of Big Stream Processing Systems. PhD thesis, University of Tartu (2018)
17. Dasgupta, T.: Evaluation of Two Major Data Stream Processing Technologies. PhD thesis, University of Edinburgh (2016)
18. Karakaya, Z., Yazici, A., Alayyoub, M.: A Comparison of Stream Processing Frameworks. In: 2017 International Conference on Computer and Applications (ICCA), pp. 1-12. IEEE (2017)
19. Mohamed, A., Najafabadi, M. K., Wah, Y. B., Zaman, E. A. K., Maskat, R.: The state of the art and taxonomy of big data analytics: view from new big data framework. In: Artificial Intelligence Review, pp. 1-49 (2019)
20. Carcillo, F., Dal Pozzolo, A., Le Borgne, Y.-A., Caelen, O., Mazzer, Y., Bontempi, G.: Scarff: a scalable framework for streaming credit card fraud detection with spark. Inf. Fusion **41**, 182–194 (2018)
21. Nair, L.R., Shetty, S.D., Shetty, S.D.: Applying spark based machine learning model on streaming big data for health status prediction. Comput. Electr. Eng. **65**, 393–399 (2018)
22. Karunaratne, P., Karunasekera, S., Harwood, A.: Distributed stream clustering using micro-clusters on apache storm. J. Parallel Distrib. Comput. **108**, 74–84 (2017)
23. Karim, M.R., Cochez, M., Beyan, O.D., Ahmed, C.F., Decker, S.: Mining maximal frequent patterns in transactional databases and dynamic data streams: a spark-based approach. Inf. Sci. **432**, 278–300 (2018)
24. Apache Kafka Project: https://kafka.apache.org/documentation/ (2018). Accessed 10 Jan 2019
25. Apache Flume Project: https://flume.apache.org/documentation.html (2018). Accessed 10 Jan 2019
26. Apache Nifi Project: https://nifi.apache.org/docs.html (2018). Accessed 10 Jan 2019
27. Apache Spark Streaming Project: https://spark.apache.org/docs/latest/streaming-programming-guide.html (2018). Accessed 10 Jan 2019
28. Apache Spark Structured Streaming Project: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html (2018). Accessed 10 Jan 2019
29. Apache Storm Project: http://storm.apache.org/releases/1.2.2/index.html (2018). Accessed 10 Jan 2019
30. Apache Flink Project: https://ci.apache.org/projects/flink/flink-docs-release-1.7/ (2018). Accessed 10 Jan 2019
31. Apache Hadoop YARN Project: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html (2018). Accessed 10 Jan 2019
32. Apache Samza Project: http://samza.apache.org/learn/documentation/1.0.0/core-concepts/core-concepts.html (2018). Accessed 10 Jan 2019
33. Apache Apex Project: https://apex.apache.org/docs.html (2018). Accessed 10 Jan 2019
34. Apache Beam Project: https://beam.apache.org/documentation/ (2018). Accessed 10 Jan 2019
35. MLLIB Project: https://spark.apache.org/docs/latest/mllib-guide.html (2018). Accessed 10 Jan 2019
36. StreamDM Project: http://huawei-noah.github.io/streamDM (2018). Accessed 10 Jan 2019

37. Apache SAMOA Project: https://samoa.incubator.apache.org/documentation/Home.html (2018). Accessed 10 Jan 2019
38. Amidst Toolbox Project: http://www.amidsttoolbox.com/documentation/ (2018). Accessed 10 Jan 2019
39. Yahoo Streaming Benchmark. https://github.com/yahoo/streaming-benchmarks (2018). Accessed 10 Jan 2019
40. Yahoo Streaming Benchmark Source: https://github.com/elkhan-shahverdi/streaming-benchmarks (2018). Accessed 10 Jan 2019
41. Bifet, A., Holmes, G., Kirkby, R., Pfahringer, B.: Moa: Massive online analysis. J. Mach. Learn. Res. **11**(May), 1601–1604 (2010)

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