Multimedia Systems (2018) 24:391–406
https://doi.org/10.1007/s00530-017-0566-5

REGULAR PAPER

A general framework for real-time analysis of massive multimedia streams

Ilaria Bartolini[1] · Marco Patella[1]

Received: 18 November 2016 / Accepted: 24 August 2017 / Published online: 8 September 2017

Abstract Big Data platforms provide opportunities for the management and analysis of large quantities of information, but the services they provide are often too raw, since they focus on issues of fault-tolerance, increased parallelism, and so on. An additional software layer is, therefore, needed to effectively use such architectures for advanced applications in several important real-world domains, such as scientific and health care sensors, user-generated data, supply chain systems and financial companies, to name a few. In this paper, we present RAM^{3}S, a framework for the real-time analysis of massive multimedia streams, where data come from multiple data sources (such as sensors and cameras) that are widely located on the territory, with the final goal to discovery new and hidden information from the output of data sources as they occur, thus with very limited latency. We apply RAM^{3}S to the use case of automatic detection of "suspect" people from several concurrent video streams, and instantiate it on top of three different open source engines for the analysis of streaming Big Data (i.e., Apache Spark, Apache Storm, and Apache Flink). The effectiveness and scalability of RAM^{3}S instantiation is experimentally evaluated on real data, also comparing the performance of the three considered Big Data platforms. Such comparison is performed both on a cluster of physical machines in our datalab and on the Google Cloud Platform.

Communicated by M. Wang.

✉ Ilaria Bartolini
ilaria.bartolini@unibo.it

Marco Patella
marco.patella@unibo.it

[1] DISI-Alma Mater Studiorum, Università di Bologna, Bologna, Italy

Keywords Multimedia streams · Big Data analysis · Face detection/recognition

1 Introduction

Recently, the world has witnessed an increased call for using data technologies for many aspects of everyday life. Indeed, the huge amount of information produced by modern society could help in several important tasks of public life such as security, medicine, environment, and smarter use of cities, but is rarely exploited due to aspects of privacy, ethics, and availability of appropriate technology, among the others. The emerging Big Data paradigm provides opportunities for the management and analysis of such large quantities of information, but the services such systems provide are often too raw, since they focus on issues of fault-tolerance, increased parallelism, and so on.

To help bridging the technological gap between facilities provided by Big Data analysis platforms and advanced applications, in this paper we focus on the real-time analysis of massive multimedia streams, where data come from multiple data sources (such as sensors and cameras) that are widely located on the territory. The goal of such analysis is the discovery of new and hidden information from the output of data sources as they occur, thus with very limited latency. A strong technological support is, therefore, needed to analyze the information generated by the growing presence of multimedia sensors. Such technologies have to meet strict requirements: rapid processing, high accuracy, and minimal human intervention. In this context, Big Data techniques can help this automated analysis, e.g., to enable corrective actions or to signal a state of security alarm for citizens. Indeed, the real-time analysis of massive multimedia streams has recently been cited as one of the most important research challenges for disaster prevention and management [23]:

> “The research challenges here are to develop new architectures for real time processing of massive volumes of heterogeneous streaming data”.

1.1 Scenario of application

Article 3 of the Universal Declaration of Human Rights guarantees, along with right of life and liberty, the security of person. In the last decades, however, governments have struggled to be effective in protecting this fundamental right, due to a growing number of threats, such as terrorist attacks, intrusions in private/public infrastructures, vandalism and disaster prevention [14]. Improving the security of people should be one of the most important priorities of any government.

Researchers could help improve the security of citizens through analysis of massive multimedia streams along three different directions [9]:

1. proposing new methods for the analysis of one particular media,
2. applying existing methods in innovative ways, or
3. discussing architectural issues for the application of methods in real environments.

We focus here on the latter subject, presenting RAM^{3}S (Real-time Analysis of Massive MultiMedia Streams), i.e., a software framework for the automated real-time analysis of multimedia streams, and its application on the performance comparison of three different open source engines for the analysis of streaming Big Data. This represents a smart way to “use” huge amounts of information produced by modern society to contribute to the safety of citizens. In particular, we focus on the study of face detection for the automatic identification of “suspect” people: this can be useful, for example, in counter-terrorism, protection against espionage, intelligent control of the territory, or smart investigations. Although we instantiated RAM^{3}S for a very specific task, it is independent of the particular application at hand, thus demonstrating its wide applicability. Other applications include the recognition of suspicious behavior from videos [20], human actions [28] or gesture [21], audio events [18], and so on.

Figure 1 shows the general operation of a stream analysis system: data coming from the stream are “compared” with those stored in a knowledge base, which have been obtained from a previous training phase to represent interesting patterns to be searched in the data stream. Whenever the data in the stream are deemed relevant, for example, because they are sufficiently similar to one of the data stored in the knowledge base, this is signaled by the system (this might be used to raise an alarm or to feedback the system with appropriate instructions).

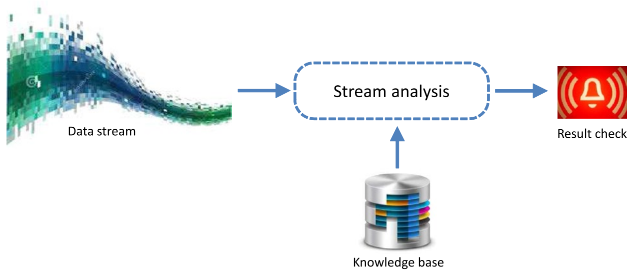


Fig. 1 Analysis of a data stream

Although the problem of stream analysis is not novel per se [3, 10], its application in the presence of several multimedia (MM) streams is questionable, due to the very nature of MM data, which are complex, heterogeneous, and of large size. This makes the analysis of MM streams computationally expensive, so that, when deployed on a single centralized system, computing power becomes a bottleneck. Moreover, the size of the knowledge base could also prevent its storage on a single computation node.

The “traditional” solution is to address such issues with faster, larger, better hardware, that is to scale up the underlying system. The shortcomings of this type of scalability are due to the inherent limitations of a single centralized computer (without mentioning costs).

Another (orthogonal) possibility is to scale out the underlying system, by adding extra machines to share the computational burden. However, this introduces issues of distributing the complex computation among nodes, resource contention, etc., which should be solved to effectively scale out the system. To increase the throughput of the overall system by way of parallel computation, we suppose that the time complexity of the data stream analysis technique is linear in the size of the incoming data stream. Clearly, a super-linear complexity would prevent real-time analysis, while sub-linearity would make the whole problem of scalability quite trivial.

1.2 Contribution

To allow efficient analysis of massive multimedia streams in real scenarios, we advocate scaling out the underlying system using platforms for Big Data management. The use of already established techniques for the analysis of Big Data has the clear advantage of making stream analysis techniques applicable in a distributed environment, shifting the focus away from low-level issues of fault-tolerance, replication, data storage/transfer, and so on. This is quite similar to the use of established techniques for data management, such as relational DBs, for traditional data processing applications. On the other hand, to make their traditional (centralized) analysis techniques work with any Big Data platform, researchers have to delve into the details specific to the platform, so as to understand how to interface their software to the provided services: this is clearly a neither easy nor cheap task.

The RAM^{3}S framework we propose here facilitates the implementation, on top of existing platforms for management of Big Data streams, of any data stream analysis technique working as in Fig. 1. In this way, researchers can avoid paying the additional cost of writing the ad hoc software layer required for interfacing their technique to the underlying platform. In details:

– First, we propose the software architecture of the RAM^{3}S framework. RAM^{3}S includes specific modules devoted to the complex task of interfacing with three different open source platforms representing recent trends for management of Big Data streams, namely Apache Spark, Apache Storm, and Apache Flink.
– Second, we show how RAM^{3}S can be easily instantiated for the real use case considered in this paper, i.e., automatic detection of "suspect" faces from several concurrent video streams.
– Finally, the applicability of RAM^{3}S to real domains/applications is demonstrated by experimental evaluating the effectiveness and scalability of the developed instance based on real data, including a comparison of the performance of the three considered Big Data platforms. Such comparison is performed both on a cluster of physical machines in our lab and on the Google Cloud Platform.

To the best of our knowledge, this is the first attempt to tackle, from a general perspective, the analysis of massive multimedia streams. The thesis we purport here is the following: by exploiting the RAM^{3}S framework together with a platform for Big Data, any traditional (centralized) system for data stream analysis can be effectively scaled out with little effort. The use of the proposed framework can thus effectively allow the application to huge data volumes of algorithms for multimedia stream analysis, which were originally conceived for a centralized scenario only, and whose extension to a distributed context is not immediate.

1.3 Roadmap

We first introduce works related to ours (Sect. 2) and the background on face detection and recognition (Sect. 3); then we describe the three open source platforms for Big Data analysis investigated in the paper (Sect. 4). Afterwards, we present the architecture of the proposed RAM^{3}S framework (Sect. 5) and investigate its performance through experiments on real data (Sect. 6). Finally we conclude the paper, outlining directions for future works (Sect. 7).

2 Related work

In the last decade, we have witnessed a proliferation of data production, due to technology advances and the wide availability of cheap data-generation apparels, such as scientific sensors and mobile devices. Analyzing such plethora of data to derive information has become, therefore, a major research problem that has fostered a new investigation field, named Big Data Analytics [13]. In a famous research report [16], it is stated that Big Data have challenging characteristics of volume (amount of data), velocity (data production rate), and variety (heterogeneity of data format and source). The main components of a Big Data analytics system are [19]:

1. technologies for data analysis (such as machine learning and data mining methods),
2. Big Data technologies (such as cloud computing infrastructures), and
3. visualization tools (such as graphs, charts, and dashboards).

Here we focus on making data analysis (in particular, MM stream analysis) techniques effectively available on top of state-of-the-art Big Data technologies.

Mining data streams concerns the extraction of information from a continuous data flow [10]. This is particularly challenging in the presence of multimedia data, because of their high production rate and huge volume. Most of the research in the field has been focused on providing effective and efficient methods for the analysis of data streams, e.g., see [3, 9, 20]. Only few recent papers have tackled the high computing requirements of analyzing MM streams using existing platforms for distributed processing: for example, in [30] a method for outlier detection on MM data streams is presented; however, the use of Apache Storm is only advocated to deal with the fact that data streams are physically distributed over a network, not as a mean to improve performance of the underlying machine learning algorithm through parallel computation.

Finally, we highlight here differences between our context and those of distributed data mining and continuous queries over data stream management systems.

– Distributed data mining (DDM) [29] concerns the application of data mining techniques in a distributed environment, where both data and computing resources are scattered over a communication network. In this light, the typical approach distributes computation at computing nodes, where only local data are analyzed, and then aggregates local models to obtain a global model. As we will see in the following, in our scenario the model is already trained (possibly by exploiting DDM techniques), and we are distributing the use of the model (which is supposed to be computationally expensive) on the incoming MM data streams.

– Data stream management systems (DSMSs) are programs designed to manage uninterrupted streams and offer facilities for executing continuous queries (CQs) [5, 22] that are permanently executed against incoming data. Processing of CQs in DSMSs is based on operators forming an algebra similar to the relational algebra and on cost-based optimization techniques similar to those exploited in database management systems. The emphasis here is on obtaining certain quality-of-service guarantees in the presence of multiple CQs running concurrently over a number of incoming streams (possibly considering also the presence of parallel computing resources). We consider an alternative scenario, where the whole analysis of a single data stream can be performed by a single computing node, while parallel computing is delegated to the underlying Big Data platform (with guarantees of reliability, fault-tolerance, and so on).

3 Background

The typical architecture for a system for face recognition from video streams is composed by four main steps to be performed in sequence: frame splitting, face detection, face recognition, and result check. Frame splitting consists of separating into frames the video coming from the camera(s), producing a sequence of images. During face detection, each image of the sequence is analyzed to check whether it contains a face. In case a face is discovered, the recognition phase compares it against a number of known faces, to retrieve the known face most similar to the discovered face. In case the similarity between the discovered face and its most similar known face is sufficiently high, the face is considered as correctly recognized, otherwise it is regarded as an unknown face. For the purpose of suspect identification, whenever a discovered face is sufficiently similar to one of the faces in the face DB, an alarm is raised.

In the following, we describe the most difficult tasks of face detection and recognition, presenting the state-of-the-art techniques that have been implemented on top of the RAM^{3}S framework for our use case. We emphasize again the fact that, although we have instantiated RAM^3 S with the most popular techniques, it is generally enough to encompass other (different) algorithms, possibly with higher accuracy or efficiency, with very limited programming effort. Indeed, a possible application of RAM^{3}S is the comparison (on the available data at hand) of alternative face detection/recognition algorithms or of different settings for the implemented algorithms (this is exactly what is shown in the experimental Sect. 6).

3.1 Face detection

Face detection is defined as the task of identifying human faces in a digital image. This has many applications, ranging from human–computer interfaces to photography. One of the most used algorithms, thanks to its robustness to scale and location and quickness, is the one provided by [25], exploiting a cascade of Haar detectors to quickly localize typical facial features (eyes, mouth, bridge of nose) which are usually darker than the surrounding facial areas. The image is scanned using a sliding window of fixed size and the cascade of Haar detectors (each detector is simpler but less accurate than the following one in the cascade) allows to quickly discard windows that cannot contain faces, since a window reaches the last, most accurate, detector only if all previous (simpler) detectors did not discard it.

3.2 Face recognition

Face recognition is the task of identifying or verifying a person identity from a digital image. The most common approach to face recognition is the comparison of selected facial features from the image with those contained in a face database. The face DB, therefore, contains the features of all known subjects (these are extracted during the training phase, by exploiting different images for each subject): during the test phase, when a face image is submitted to the system, its features are extracted and compared against those contained in the face DB. This is clearly a problem of classification, where categories include known faces and the null hypothesis. One of the most used classifiers is the 1-NN classifier, so that the comparison between features of two faces amounts to a (dis-)similarity value and the input face is considered recognized if the closest known face (lowest dissimilarity value) has a dissimilarity not exceeding a predefined threshold δ, otherwise it is considered unknown (null hypothesis).

One of the most popular recognition algorithms exploits principal component analysis using eigenfaces [24]. Eigenfaces are the eigenvectors derived from the covariance matrix of the probability distribution over the high-dimensional vector space of face images. During the training phase, all DB images are considered as vectors, by concatenating the pixels in each image. The covariance matrix of so-obtained vectors is then computed and its eigenvalues and eigenvectors are calculated. Finally, only the k (with *k* usually in the range [100, 150]) principal components (i.e., the ones corresponding to the *k* highest eigenvalues) are kept. Each face image can be, therefore, projected on the eigenfaces matrix, to produce a vector of *k* components. Finally, similarity between two faces can be computed as (one minus) the normalized Euclidean distance between projected vectors.

4 Big Data analysis platforms

In this section, we describe the alternatives available for the analysis of Big Data. This is necessary to grasp common characteristics present in such systems that will be exploited to guarantee the generality of the proposed framework.

Platforms for Big Data analytics can be broadly classified into two macro-classes [13]:

- Batch processing: according to this paradigm, data are first stored on a physical support (usually on a distributed file system, DFS) and then analyzed. MapReduce [8] is the de facto reference model for the processing of batch data. Following the MapReduce idea, data are first divided into chunks that are processed in parallel during the *Map* phase, producing intermediate results. Such results are then aggregated, again in parallel, during the *Reduce* phase to compute the final result. Examples of open source platforms following the batch paradigm include Apache Hadoop[1] and Apache Spark.[2]
- Stream processing: this paradigm analyzes the data as soon as these are available, to exploit their "freshness". Data are, therefore, not stored and the efficiency of the system depends on the amount of data processed, keeping low latency, at the second, or even millisecond, level. This is necessary to deal with online applications. Examples of open source platforms following the streaming paradigm include Apache Storm[3] and Apache Flink.[4]

The need for general-purpose Big Data architectures has also created some hybrid platforms. Examples include Spark Streaming that exploits micro-batching to emulate stream batching; this way, streaming data are accumulated for the micro-batch duration and batch analysis can be performed on such data. This clearly leads to a latency not lower than the micro-batch duration. On the other hand, the pipelined architecture of Flink also allows the execution of batch analysis. Figure 2 summarizes the dichotomy between batch and streaming processing, highlighting the hybrid behavior of Spark and Flink.

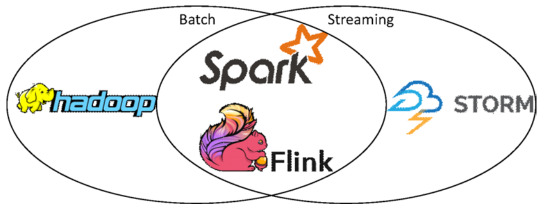


Fig. 2 Paradigms for Big Data analysis

In the following, we will describe the three major open-source platforms for Big Data streaming analysis: Spark Streaming, Storm, and Flink.

4.1 Apache Spark Streaming

Apache Spark [27] is an open source platform written in Java, Scala, and Python, originally developed at the University of California, Berkeley's AMPLab to overcome the limits of Hadoop. In particular, Spark avoids the constraint imposed by Hadoop that both input and output of the *Map* and *Reduce* phases should be stored on the DFS. The main idea of Spark is that of storing data into a so-called resilient distributed dataset (RDD), which represents a read-only fault-tolerant collection of (Python, Java, or Scala) objects partitioned across a set of machines that can be stored in main memory. RDDs are immutable and their operations are lazy. The "lineage" of every RDD, i.e., the sequence of operations that produced it, is kept so that it can be rebuilt in case of failures, thus guaranteeing fault-tolerance. Every Spark application is, therefore, a sequence of operations on such RDDs, with the goal of producing the desired final result. There are two types of operations that can be performed on a RDD:

- Transformation: every operation that produces a new RDD, built from the original one; for example, a *Map* operation on a RDD can produce a new RDD.
- Action: every operation that produces a single value or writes an output on disk. For example, a *Reduce* operation combining elements to produce a result at the driver program.

The kernel of Spark is Spark Core, providing functionalities for distributed task dispatching, scheduling, and I/O. The program invokes the operations on Spark Core exploiting a functional programming model. Spark Core then schedules the function's execution in parallel on the cluster. This assumes the form of a directed acyclic graph (DAG), where nodes are operations on RDDs performed on a computing node and arcs represent dependencies among RDDs.

1 http://hadoop.apache.org.
2 http://spark.apache.org.
3 http://storm.apache.org.
4 http://flink.apache.org.

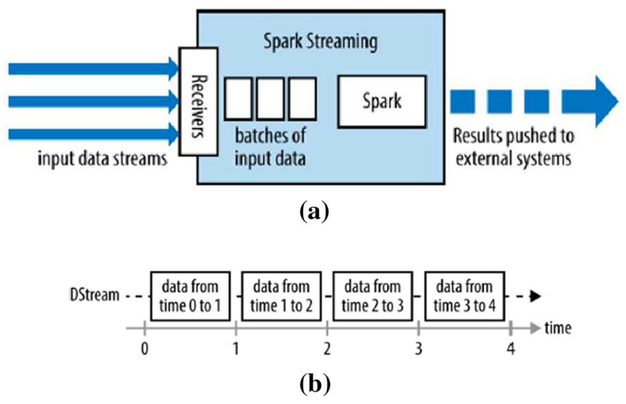


Fig. 3 Spark Streaming micro-batching (**a**) and D-stream (**b**)

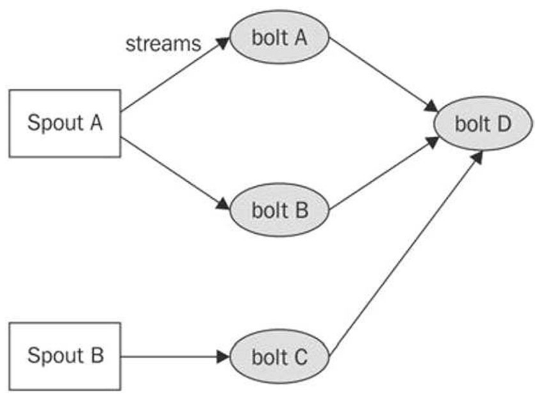


Fig. 4 A sample Storm topology

The streaming version of Spark introduces receiver components, in charge of receiving data from multiple sources, such as Apache Kafka,[5] Twitter, or TCP/IP sockets. Real-time processing is made possible by introducing a new object, the D-stream, a sequence of RDDs, sampled every *n* time units (see Fig. 3). Every D-stream can be now introduced in the Spark architecture as a "regular" RDD to be elaborated, thus realizing micro-batching.

A Spark Streaming application is built by two main components:

Receiver: every receiver contains the data acquisition logic, defining the `OnStart()` and `OnStop()` methods for initializing and terminating the data input; every receiver occupies a core in the architecture.

Driver: here the program specifies the receiver(s) creating the D-streams and the sequence of operations transforming the data, terminated with an action storing the final result.

4.2 Apache Storm

Apache Storm is an open source platform written in Clojure and Java, originally developed by Backtype and afterwards acquired by Twitter. The basic element that is the subject of computation is the *tuple*, a serializable user-defined type. The Storm architecture is composed of two types of nodes:

Spout: nodes that generate the stream of tuples, binding to an external data source (like a TCP/IP socket or a message queue).

Bolt: nodes that perform stream analysis; ideally, every bolt node should perform a transformation of limited complexity, so that the complete computation is performed by the coordination of several bolt nodes. Every bolt is characterized by the `prepare()` and the `execute(Tuple)` methods; the former is used when the bolt is started, while the latter is invoked every time a `Tuple` object is ready to be processed.

A Storm application simply defines a *topology* of spout and bolt nodes in the form of a DAG, where nodes are connected by way of streams of tuples flowing from one node to the other (see Fig. 4). The overall topology, therefore, specifies the computation pipeline.

A key concept in Storm is the acknowledgment of input data, to provide fault-tolerance, so that if a node emits a tuple but this is not acknowledged, this can be re-processed, e.g., forwarding it to another node. This amounts to an "at-least-once" semantics, guaranteeing that every datum is always processed. This allows low latency, but does not ensure the uniqueness of the final result.

4.3 Apache Flink

Apache Flink is an open source platform written in Java and Scala, originally started as a German research project [1], that was later transformed into an Apache top-level project. Its core is a distributed streaming dataflow that accepts programs in form of a graph (*JobGraph*) of activities consuming and producing data. Each JobGraph can be executed according to a single distribution option, chosen among the several available for Flink (e.g., single JVM, YARN, or cloud). This allows Flink to perform both batch and stream processing, by simply turning data buffering on and off.

Data processing by Flink is based on the snapshot algorithm [4], using marker messages to save the state of the whole distributed system without data loss and duplication. Flink saves the topology state in main memory (or on HDFS) at fixed time intervals. The use of the snapshot algorithm provides Flink with a "exactly-once" semantics,

[5] http://kafka.apache.org.

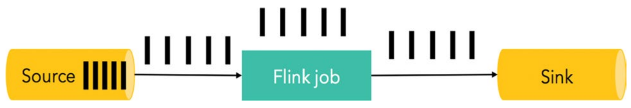


Fig. 5 Flink data flow

guaranteeing that every datum is processed at least one time and no more than that. Clearly, this allows both a low latency and a low overhead, and also maintains the original flow of data. Finally, Flink has a "natural" data flow control, since it uses fixed-length data queues for passing data between topology nodes (see Fig. 5).

The main difference between Storm and Flink is that, in the former, the graph topology is defined by the programmer, while the latter does not have this requirement, since each node of the JobGraph can be deployed on every computing node of the architecture.

5 The RAM^{3}S framework

The objective of our RAM^{3}S framework is to provide a general infrastructure for the analysis of multimedia streams on top of Big Data platforms. By exploiting RAM^{3}S, researchers can apply their multimedia analysis algorithms to huge data streams, effectively scaling out methods that were originally created for a centralized scenario, without incurring the overhead of understanding issues peculiar to distributed systems. To be both general and effective, we designed RAM^{3}S as a "middleware" software layer between the underlying Big Data platforms and the top data stream application. RAM^{3}S exposes a simple interface towards data stream analysis, as seen in Fig. 1: this allows an easy transition from the already available (centralized) software to the scaled out solution. On the other hand, RAM^{3}S interacts with the bottom world by way of modules implementing the (complex) specific components requested by any of the Big Data platforms presented in Sect. 4.

Our assumption is that the incoming multimedia stream can be represented as a sequence of individual multimedia objects (`MMObjects`): the analysis of each `MMObject` instance, continuously repeated over time, produces the result. The goal of this reductionist approach is not to be an all-inclusive representation of multimedia stream analysis, but (1) to be general enough to encompass the vast majority of stream analysis algorithms and (2) to be simple enough to make interfacing with the underlying Big Data platform as easy as possible.

5.1 General description

The core of RAM^{3}S consists of two interfaces that, when appropriately instantiated, can act as a general stream analyzer: the `Analyzer` and the `Receiver` (see below).

```
Receiver
   start(): void
   stop(): void
```

```
Analyzer
   Analyzer(KnowledgeBase)
   analyze(MMObject): boolean
```

The `Analyzer` is the component in charge of the analysis of individual multimedia objects. In particular, every multimedia object is compared to the underlying KB to see whether an alarm should be generated or not. On the other hand, the `Receiver` should break up a single incoming multimedia stream into individual objects that can be analyzed one by one. Of course, the type of "individual multimedia object" depends on the application at hand: for example, in our face detection scenario, the object is a single image, extracted as a single frame of a camera video (see Sect. 5.2); in a scenario where we have to detect and classify sounds for acoustic surveillance, the object is a single audio stream which is sufficiently discernable from the background noise [15, 18].

The interface of the `Receiver` component is very simple: only the `start()` and the `stop()` methods should be defined, specifying how the data stream acquisition is initiated and terminated. The output of the `Receiver` component is a sequence of `MMObject` instances.

The `Analyzer` component includes only a constructor and a single `analyze()` method. The constructor has a single parameter, the KB which will be used for the analysis of individual objects. On the other hand, the `analyze()` method takes as input a single `MMObject` instance and outputs a boolean, indicating whether the comparison of the input `MMObject` "matches" the KB or not.

When cascaded, the `Receiver` and the `Analyzer` components implement the generic functionalities of a stream analyzer, where the continuous flow of input data produces a sequence of activations/de-activations of the alert. This is illustrated in Fig. 6, which mimics the description of data stream analysis in Fig. 1.

Encapsulating the complexity of a system for stream analysis to two interfaces has two advantages:

1. standardizing the interface of the `Receiver` and the `Analyzer` allows the generalization of several stream analysis techniques;
2. as we will show in Sect. 5.3, the interface to Big Data platforms is greatly simplified, since the `Receiver` and the `Analyzer` already mimic the operations of components such as spouts and bolts.

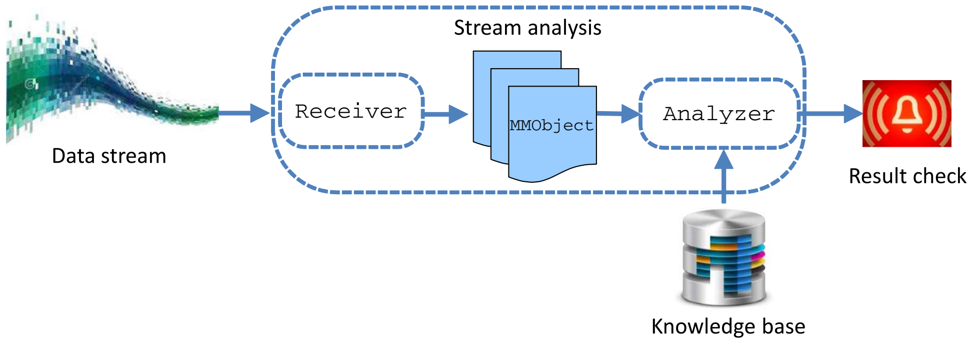


Fig. 6 General view of the two main RAM^{3}S classes

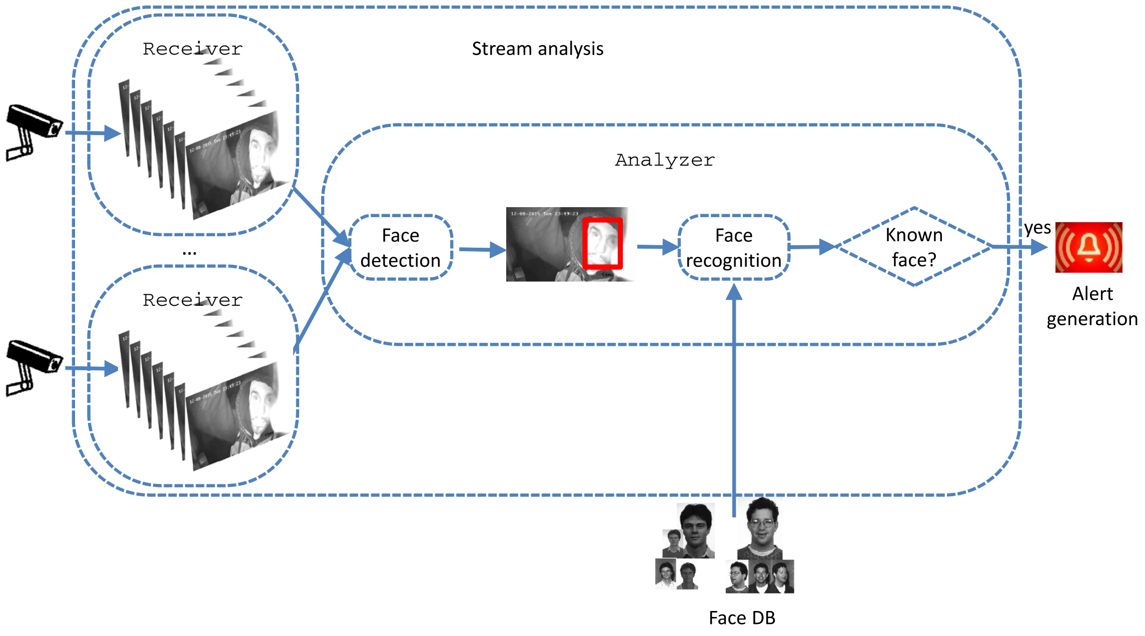


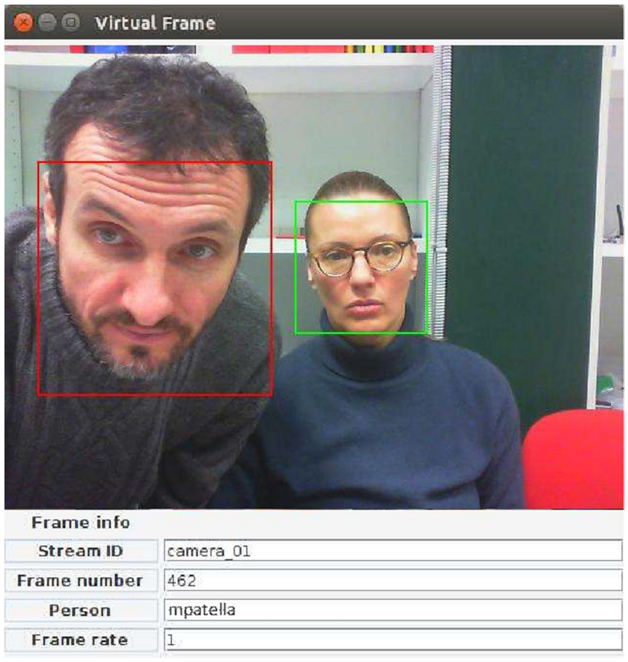
Fig. 7 Instantiation of the RAM^{3}S framework classes for suspect face recognition

5.2 Applying the framework to the use case

Our scenario of suspect people recognition is illustrated in Fig. 7. Each camera sends its video flow to a single `Receiver` that performs the task of frame splitting and generates a stream of still images (instances of the `MMObject` interface). Hence, the `Analyzer` receives a flow of images and performs the face detection/recognition tasks. In the `Analyzer`, we exploit the Haar cascade detector and the eigenfaces algorithm provided by the OpenIMAJ library.[6] In all cases, the face recognition module has been trained in advance, so that the KB consists of the eigenfaces matrix M and of the k-dimensional vectors of known faces (as projected on M). The `Analyzer::analyze()` method first applies the Haar cascade detector to find potential faces in the input image; then, every discovered face is projected on the M matrix and the so-obtained k-dimensional vector is compared to the ones in the KB to compute its dissimilarity with its nearest neighbor: if this is not higher than the system threshold δ, then the output of the method is true, otherwise (no detected face is sufficiently similar to its nearest neighbor in KB) it is false.

The combination of the `Receiver` and the `Analyzer` components transforms an input multimedia data stream into a sequence of boolean values in output (alarm on/off). Clearly, this result can be expanded in case the application at hand requires a more complex output. As an example, we show the GUI of our suspect face detection application in Fig. 8. The output visualization of every camera (identified by the Stream ID field) reports, for any detected face, the position of that face within the image with a box which is colored in red (alarm) if the face has been recognized, and in green (all clear) otherwise; finally, for any recognized face,

6 http://openimaj.org.



Frame info

Stream ID	camera_01
Frame number	462
Person	mpatella
Frame rate	1

Fig. 8 The GUI of the suspect face detection application: the left person is correctly recognized (as "mpatella", present in the face DB), the right person is correctly recognized as unknown (not present in the face DB)

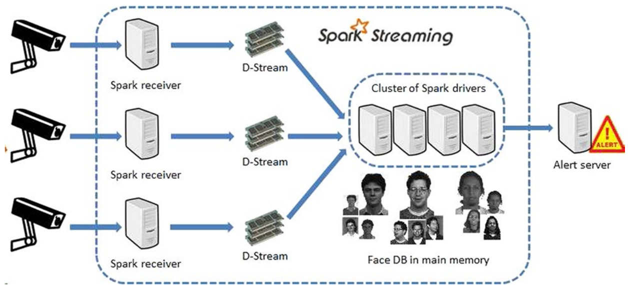


Fig. 9 Architecture for the Apache Spark platform

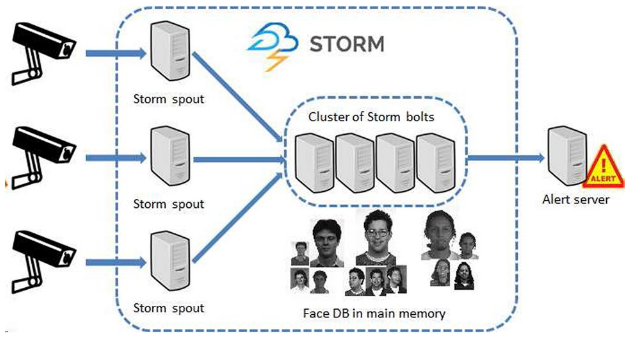


Fig. 10 Architecture for the Apache Storm platform

the name associated with the known face in the face DB ("mpatella" in the illustrated case) is also shown.

5.3 Interface to Big Data platforms

The software framework described so far (and its application to the use case) is directly applicable to a centralized framework, by instantiating a `Receiver` for each input data stream and a single `Analyzer`, collecting outputs of the `Receivers`. In the following, we show how the two RAM^{3}S interfaces have been appropriately extended to work as components of the distributed Big Data platforms. We suppose that the KB (the face DB) can be contained in main memory on a single node; if this is not the case, we envision two possible workarounds:

1. the `Analyzer` is replicated on multiple nodes, each considering a partition of the original KB, or
2. the KB is stored on secondary storage and the `Analyzer` exploits appropriate access methods [2, 7] to avoid a sequential scan of the KB.

5.3.1 Apache Spark

The architecture for the Apache Spark framework is depicted in Fig. 9. In detail, every camera sends its video stream to a `Receiver`, corresponding to a single Spark receiver: the output of the receiver is buffered to build the D-stream. The buffer consists of *n* `MMObject` instances, where *n* is a system parameter. Every time the buffer is full, a D-stream is completed and it is sent to the cluster of Spark drivers. The `OnStart()` and `OnStop()` methods of the Spark receiver correspond to the `start()` and `stop()` methods of the `Receiver` interface.

Any Spark driver includes an `Analyzer`. The logic of Spark drivers should iterate the `analyze()` method for all the *n* `MMObject` instances included in the input D-stream. The output of the Spark driver is the boolean "or" of the *n* boolean results of the `analyze()` method. Finally, this alarm is sent to the alert server.

5.3.2 Apache Storm

The topology for the Apache Storm platform is shown in Fig. 10. As before, every camera sends its video stream to a `Receiver`, encapsulated into a Storm spout. Now, however, every `MMObject` generated by the spout is immediately sent to the cluster of Storm bolts.

Storm bolts correspond to single `Analyzer` objects. The `prepare()` method is equivalent to the `Analyzer` constructor, while `execute()` matches the `analyze()` method, which is executed for any received `MMObject`. Any time the `Analyzer` receives a `MMObject`, it has to

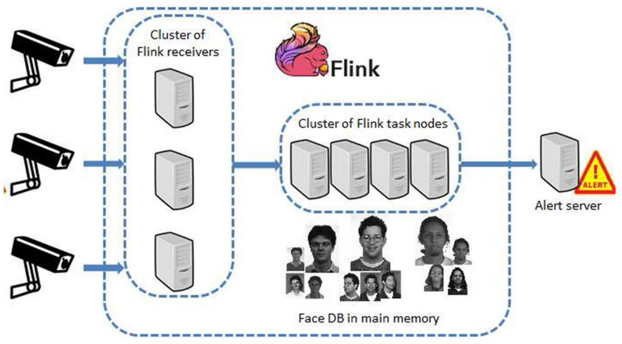


Fig. 11 Architecture for the Apache Flink platform

acknowledge the emitting Receiver to respect the Storm semantics.

With respect to the Spark solution, we have the advantage of a real stream processing architecture: latency is likely to be reduced, since we do not have to wait for the completion of a D-stream to begin the analysis of any single `MMObject`.

5.3.3 Apache Flink

Finally, Fig. 11 illustrates the architecture for the Apache Flink platform. Here, the configuration is apparently rather similar to the Storm case: cameras send their video streams to a cluster of Flink receivers, each including a single `Receiver` which sends individual instances of `MMObject` to the cluster of Flink task nodes. A single Flink task node consists of an `Analyzer`, executing the `analyze()` method for any received `MMObject`, finally sending an alarm to the alert server.

With respect to the Storm architecture, although for ease of presentation Fig. 11 shows Flink receivers separated from Flink task nodes, we remind that this difference is only virtual, since each activity of the Flink JobGraph can be executed on any computing node, i.e., there are no "receiver" nodes and "task" nodes, but any single computing node can host either a receiver or a task node at any given moment. This gives to the Flink architecture high resiliency and versatility, since it can easily accommodate node faults (for both receivers and task nodes) and addition/deletion of input data streams (which would require modification of the topology in Storm).

6 Experiments

As a proof of the usability of the presented framework, we performed a series of experiments on our use case to prove the accuracy of the implemented `Analyzer` and to compare the efficiency of the used Big Data platforms. The goal of these experiments is to show that RAM^{3}S allows, with very limited programming effort, to effectively build a working system that can be effectively exploited, for example, to choose the optimal settings for the situation at hand. In particular, we look for an answer to the following questions:

– What is the accuracy of the implemented face detection/recognition module?
– How do the available architectures scale when more nodes are added to the computing cluster?
– How do the available architectures compare against each other in terms of latency and throughput?

We start with a description of our experimental environment (Sect. 6.1), then we present results on the `Analyzer` accuracy (Sect. 6.2) and on the scalability of the overall system (Sect. 6.3). We conclude with a summary of lessons learned in Sect. 6.4.

6.1 Experimental environment

The open source platforms for real-time data analysis described in Sect. 4 have been implemented both in a local environment with a cluster of 12 PCs and on the Google Cloud Platform.[7] The first example, realized in our datalab,[8] mimics the case where streams from surveillance cameras are processed locally by a commodity cluster of low-end computers, while the latter represents the scenario where all video streams are transmitted to a center of massively parallel computers. While it is expected that the cloud environment attains superior performance, this could represent an impracticable solution if this type of hardware is not available internally: in such case, recurring to an external service (such as Google and Amazon) could raise privacy issues, since data have to be moved to a non-proprietary infrastructure.

For tests on the local cluster, we used (up to) 12 PCs with a Pentium 4 CPU @ 2.8 GHz, equipped with 1GB of RAM, interconnected with a 100 Mbps Ethernet network. For tests on the cloud architecture, we used (up to) 128 machines of type n1-standard-1 with 1 vCPU (2.75 GCEU) and 3.75 GB of RAM. All code has been written in Java. Since our goal was to prove the generality of the framework, all platforms have been used with default settings, focusing on writing simple and understandable code, without optimizing it to the extreme.

Our real dataset consists of the YouTube Faces Dataset (YTFD) [26], including 3425 videos of 1595 different people, with videos having different resolutions (480×360 is

[7] http://cloud.google.com.
[8] http://www-db.disi.unibo.it/research/datalab/.

Table 1 Accuracy of face detection

	Face detected?		
	Yes	No	Total
Face present?			
Yes	68.9%	1.1%	70%
No	13.1%	16.9%	30%
Total	82%	18%	100%

the most common size) and a total of 621,126 frames containing at least a face (on average, 181.3 frames/video).[9]

6.2 Accuracy of the face detection/recognition module

The first set of experiments aims at evaluating the effectiveness of the `Analyzer` component in the considered use case, as implemented using the algorithms described in Sect. 3. We are interested in the following measures for evaluating face detection/recognition, as a binary classifier:

$$\text{Recall:}\quad R = \frac{P}{P+\bar{N}}$$

$$\text{Specificity:}\quad S = \frac{N}{N+\bar{P}}$$

$$\text{Precision:}\quad P = \frac{P}{P+\bar{P}}$$

$$\text{Accuracy:}\quad A = \frac{P+N}{P+N+\bar{P}+\bar{N}}$$

where P is the number of true positives, N is the number of true negatives, $\bar{P}$ is the number of false positives, and $\bar{N}$ is the number of false negatives.

Our first test aimed at evaluating the accuracy of the face detector alone. To this end, we added to the original faces dataset about 21,000 images not containing faces, drawn from the German Traffic Sign Detection Benchmark (GTSDB) [12] (900 images[10]) and the Swedish Traffic Signs Dataset (STSD) [17] (20,000 images).[11] From this combined dataset, we randomly chose (without repetition) 35,000 images containing faces and 15,000 not containing faces and performed the face detection on such 50,000 images. Results, averaged over 5 different detection runs for a total of 250,000 images, are depicted in Table 1.

Results show the very low percentage (about 1%) of false negatives. The higher number of incorrectly detected faces (over 40% of images not containing faces are incorrectly

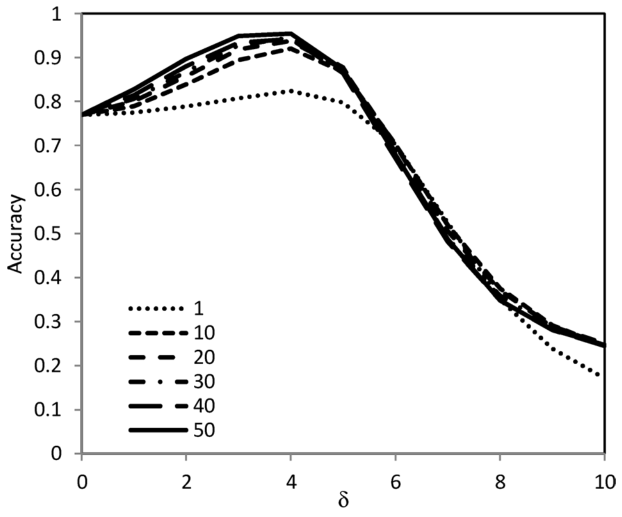


Fig. 12 Accuracy of face recognition when varying the number N of face examples

classified) is not very critical, because such false positives can be discarded by the subsequent phase of face recognition: indeed, it is very likely that an incorrectly detected face is sufficiently dissimilar to all faces in the KB.

The second test we performed was to evaluate the accuracy of face recognition alone. Here, the two parameters to be tested are:

1. the number of image examples for each known face, N, to be used for creating the knowledge base, and
2. the distance threshold, δ, to be used by the 1-NN classifier (if the distance between the query face and its nearest known face is lower than δ, the query face is considered recognized).

For this, we created the knowledge base using 200 people from YTFD (with N ranging from 1 to 55). Then, the test set was created using 10 images for each of the 200 known people and other 6670 images of unknown people, for a total of 8670 faces. Figure 12 shows the overall accuracy of face recognition. When $\delta = 0$, the accuracy obviously equals 6670/8670 = 76.9%. Increasing the value of δ, the number of false negatives decreases but the number of false positives increases, so we need to find a trade-off value for δ.

First of all, we note that the maximum accuracy, for any number of training images, is reached for a threshold value of 4. Moreover, the graph shows that increasing the number of images per person leads to growing values of the accuracy, for most values of the threshold value δ. However, such increment decreases as N grows. For example, the accuracy at $\delta = 4$ is 82.4% with $N = 1$, 93.8% for $N = 20$, 94.3% at $N = 40$, and 95.4% for $N = 50$. Further, by increasing N to the value of 55 (results are not included in Fig. 12 to avoid

[9] http://www.cs.tau.ac.il/~wolf/ytfaces/.

[10] http://benchmark.ini.rub.de.

[11] http://www.cvl.isy.liu.se/research/datasets/traffic-signs-dataset/.

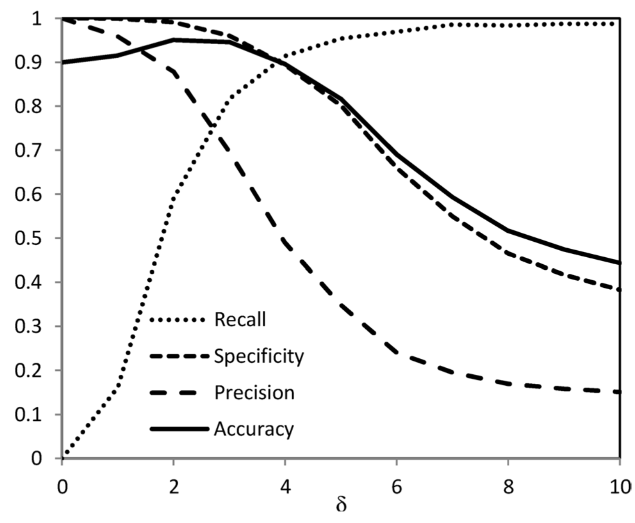


Fig. 13 Evaluation of the overall face detection/recognition classifier

cluttering the graphs), the accuracy only increases 0.2%. For this, in the following we will use a value of $N = 50$.

Finally, we evaluate the overall (combined) performance of face detection and recognition. For this, we built a new test set, composed of 40,000 images, with 20,000 images from the GTSDB/STSD datasets containing no faces, 16,000 images from the YTFD with faces not included in the training set, and 4000 images from YTFD with faces included in the training set, so as to simulate an environment where suspect faces are rather infrequent. For any given input image, possible outputs are:

True positive (P): a person included in the training set which is correctly recognized.

False negative ($\bar{N}$): a person included in the training set which is not recognized/detected.

True negative (N): an image containing no faces which is correctly not detected or a person not included in the training set which is correctly unrecognized.

False positive ($\bar{P}$): an image containing no faces or a person not included in the training set, which is, however, recognized as a known face.

Figure 13 plots the four evaluation measures for different values of the distance threshold δ.

Clearly, as δ increases, so does the recall, because the number of true positives increases as well. However, both the precision and the specificity decrease, because the number of false positives increases, too. For the dataset considered in our experiments, the best choice is a threshold value of $\delta = 3$, corresponding to a value of accuracy of 94.6% (only slightly lower than the maximum value of 95.1%), but with a recall of 81.7% and an overall precision of 69.6%.

6.3 Scalability of the overall architecture

The goal of our second set of experiments is to evaluate the efficiency of the proposed architectures, comparing them on a fair basis. Besides graphs for the three implemented architectures, figures also include the performance of a centralized system (where a single `Receiver` and `Analyzer` are instantiated) that does not exploit any Big Data platform: this aims to compare the performance of a traditional system with the ones obtainable by exploiting RAM^{3}S in a distributed computing scenario.

In this context, we are interested in the following measures:

- Sustainable input rate: the maximum frame rate such that the input buffer does not overflow.
- Latency: the (average) length of the time interval between the input of a frame and the system response; this is measured using the maximum input rate sustainable by an architecture composed of a single node.

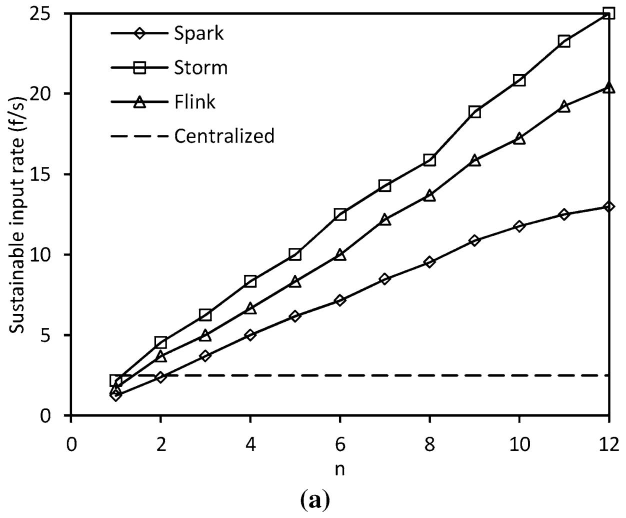
To be able to freely vary the input frame rate, in this set of experiments the incoming stream of video frames was simulated by feeding a FIFO queue (implemented with RabbitMQ[12]) with sequences of still images. The same machine running the FIFO queue acts as the alert server, where the result of the face detection/recognition is sent. Latency of the system is measured as the (average) time interval between an image exiting the FIFO queue and the corresponding alert signal received by the alert server.

Figure 14 shows the sustainable input rate obtained for the two scenarios. Results show that Apache Storm consistently outperforms the other two frameworks on both scenarios. We believe this behavior is due to the simplest at-least-once semantics of Storm (with respect to the exactly-once semantics of Flink) and to the fact that the Storm topology has to be explicitly defined by the programmer, while for Flink the topology is decided by the streaming optimizer; this superior flexibility of Flink comes at the cost of a slightly diminished efficiency.

On the other hand, Apache Spark exhibits the worst performance, although its results are quite similar to those of Apache Flink for the cloud scenario. This is largely expected, since Spark is not a real streaming engine and the management of D-streams is the main reason of its inferior input rates.

Comparing performance of the centralized solution with the ones based on Big Data platforms, the former attains a higher throughput than the latter in the case where a single machine is used, since the overhead of the

[12] http://www.rabbitmq.com.



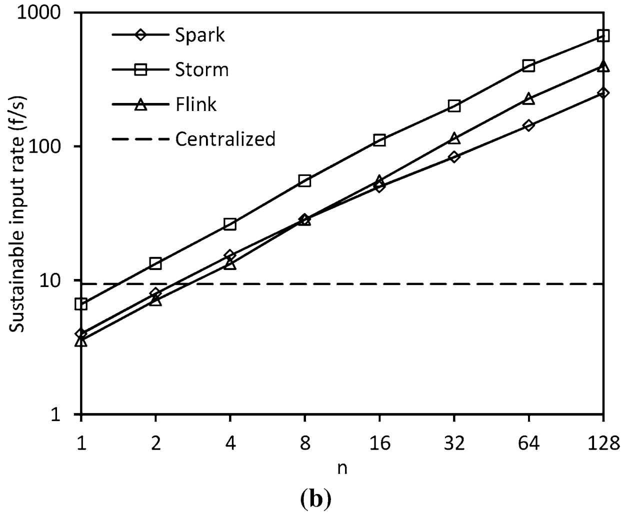
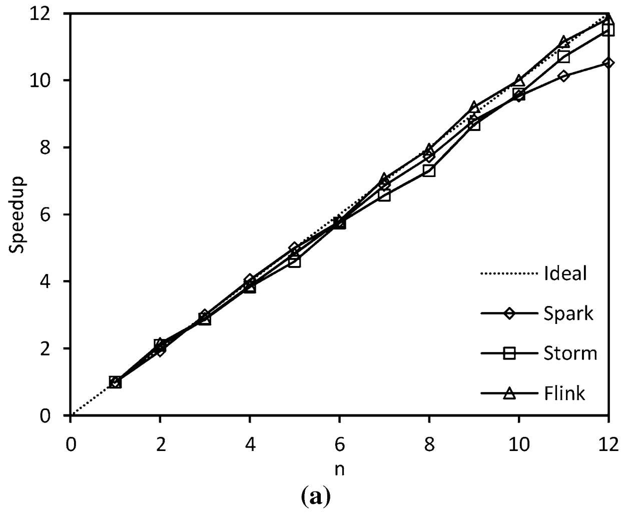


Fig. 14 Sustainable input rate for the local cluster (a) and cloud (b)



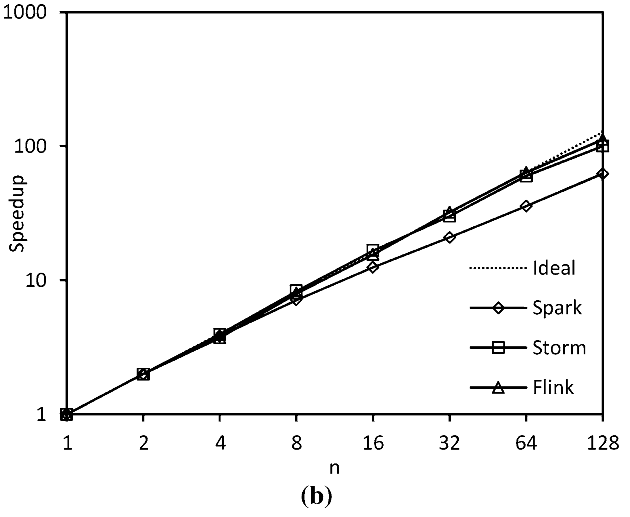


Fig. 15 Sustainable input rate speedup for the local cluster (**a**) and cloud (**b**)

underlying platform is absent in the "centralized" scenario. On the other hand, as the number of machines increases, the use of Big Data platforms helps reaching sustainable input rates not achievable in the centralized case.

In Fig. 15 we show the speedup in sustainable input rate for the two scenarios. The speedup is defined as:

$$S_n = \frac{\mathrm{IR}_n}{\mathrm{IR}_1} \tag{1}$$

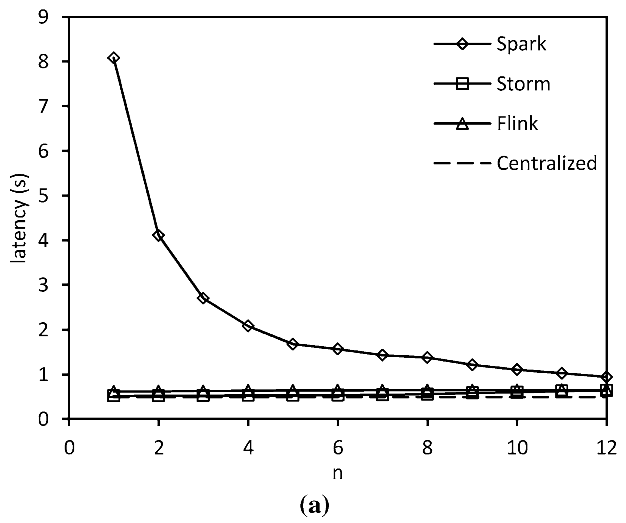
where IR_n is the measured sustainable input rate on $n = 1, 2, \ldots$ cluster nodes. All frameworks exhibits quasi-ideal speedup when the number of nodes is limited. When n grows, the system scalability begins to diverge from linearity: in both cases, the system that first exhibits such behavior is Apache Spark.

An explanation for the sub-linear scalability is the following. According to [11], the theoretical speedup of a system can be expressed as:

$$S_n = \frac{n}{1 + \sigma n + \kappa n(n-1)} \tag{2}$$

where the σ coefficient represents the degree of contention in the system (that grows linearly with the number of nodes), while the κ coefficient takes into account the incoherency in distributed data (growing quadratically, due to pairwise exchange of data between nodes). In our example, no coherence is needed, so $\kappa = 0$, but $\sigma > 0$ due to the bottlenecks of the face recognition/detection task and of the communication network limited speed.

Finally, Fig. 16 illustrates the latency for the three architectures on the local and the cloud scenarios. For Apache Spark, the delay is the average delay of images included



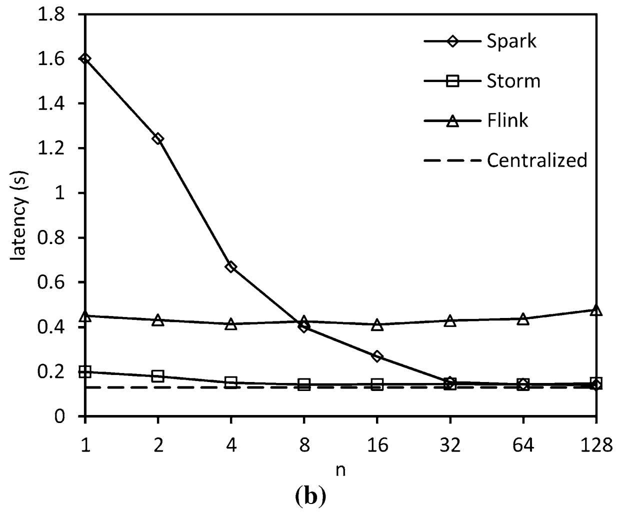


Fig. 16 Latency for the local cluster (**a**) and cloud (**b**)

in a single D-stream. Since a micro-batch period of 30 s was considered, this can be computed as 30/2 plus the average latency. Figure 16 only plots the average latency, omitting the 15 s from the total delay. Again, the figures include the graph for the "centralized" scenario, whose latency is clearly the lowest possible, since the whole underlying data platform is now absent. However, the graphs show that the latency obtained through the Storm and Flink platforms is always very close to the best (only about 20% higher). This demonstrates that the use of the RAM^{3}S framework does not introduce an excessive overhead.

Apache Storm consistently achieves the lowest latency, with Apache Flink always very close. For both streaming-native platforms, latency times are largely unaffected by the number of nodes: this same behavior is also common to the non-discounted latency of Apache Spark (not shown here to avoid cluttering of figures). The discounted latency of Spark, on the other hand, clearly takes advantage of the increased number of nodes, reaching latency values similar to Storm for high numbers of nodes.

6.4 Lessons learned

Commenting on the accuracy of the overall system, performance of the face detection/recognition module was consistent with results obtained for publicly available systems. Superior results could have been achieved using a more complicated face recognition module, at the expenses of higher latency and lower throughput: the implemented system is a nice trade-off between simplicity and performance.

Although faces in the YTFD are characterized by different backgrounds, orientation, expression, and lighting conditions, the quality of face images has to be evaluated as quite good: this clearly helps obtaining good results in face detection and recognition. To obtain a more robust system, a thorough experimentation should be carried out on a dataset of facial images including occlusions, poor lighting, use of (sun)glasses, (facial) hairstyle changes and multiple races; to the best of our knowledge, such dataset is not (publicly) available. For example, the NIST Face In Video Evaluation (FIVE) project[13] includes test data that are likely to exhibit above characteristics; yet, as stated in the project homepage:

> "None of the test data can be provided to participants. Instead prospective participants should leverage public domain and proprietary datasets as available".

The good news is that the face detection/recognition module implemented here could be easily replaced with a different, possibly better performing, analyzer with only limited effort, thanks to the generality of the proposed framework.

Implementation of the `Receiver` and `Analyzer` for the suspect face recognition has been extremely simple and concise, thanks to the use of public libraries for the complex tasks of face detection and recognition. This clearly helps in shifting the focus from issues related to data analysis and distributed computation to the ones specific to data management, which are more interesting for the purpose of our research. For example, one of the difficulties arisen in our scenario was the transfer of `MMObjects` (still images) between nodes of the architecture: indeed, in our first implementation, the output of any `Receiver` was a sequence of bitmap images whose size can be computed as $480 \times 360 = 172,800$ pixels with 32 bits per pixel, i.e., 675 KB. This would limit the maximum frame rate in the local scenario to 18.5 f/s, since nodes are connected to a 100 Mbps network, and $0.675 \times 18.5 \times 8 \approx 100$ Mbps. This was solved by avoiding de-compression of the image at the

[13] http://www.nist.gov/programs-projects/face-video-evaluation-five.

receiver, which allows increasing the frame rate of at least one order of magnitude.

In the implemented scenario, the bottleneck of the system consists of the face detection phase: about 90% of latency time is spent in detecting faces in each image, while the tasks of frame splitting, image decompression, and face recognition (composed of projection of image on the eigenvectors matrix and computation of distance between feature vectors) are relatively quick.

Comparing performance of the three Big Data platforms, Spark attains the lower throughput and highest latency, clearly due to its micro-batch design: when looking at its good latency performance on the cloud environment (see Fig. 16b), we should remember that the time for building the D-stream is not considered there. Comparing Storm and Flink, the former attains slightly better results, both in terms of throughput and latency: as already pointed out in Sect. 5.3.3, Flink is likely to be superior to Storm when issues related to fault-tolerance and correctness of the final result are considered; this clearly comes at the expenses of slightly inferior results when only performance is an issue. Similar considerations, albeit for a slightly different scenario (counting JSON events from Apache Kafka), were recently obtained in [6].

7 Conclusions

We have proposed RAM^{3}S, a framework for the real-time analysis of massive multimedia streams, demonstrating its generality; we applied it on top of three state-of-the-art platforms for processing of Big Data streams.

To show its general applicability, we have instantiated RAM^{3}S to the case of suspect people identification, with the help of existing libraries for the complex task of face detection/recognition.

By exploiting RAM^{3}S, information from already existing sources (sensors, cameras, microphones, and so on) can be effectively exploited to improve the safety of citizens, thus providing a smarter use of cities and territory, without introducing new complex infrastructures in the environment.

We are in the process of using RAM^{3}S for a number of other real-world applications, in the fields of bioinformatics and neurology. We are also interested in considering the enforcement of service guarantees over RAM^{3}S, inspired by the works on continuous queries over data streams [22]: this would require the definition of a cost model for the `Receiver` and `Analyzer` components and the automatic request for more (or less) computing nodes for maintaining a minimum input rate or maximum latency.

Acknowledgements The authors thank the datalab students that participated in the software development of the RAM^{3}S framework.

References

1. Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J.-C., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M.J., Schelter, S., Höger, M., Tzoumas, K., Warneke, D.: The stratosphere platform for big data analytics. VLDB J. **23**(6), 939–964 (2014)
2. Bartolini, I., Ciaccia, P., Patella, M.: Query Processing Issues in Region-Based Image Databases. Knowledge and Information Systems **25**(2), 389–420 (2010)
3. Bifet, A., Holmes, G., Kirkby, R., Pfahringer, B.: MOA: massive online analysis. J. Mach. Learn. Res. **99**, 1601–1604 (2010)
4. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. **3**(1), 63–75 (1985)
5. Chang, S.K., Zhao, L., Guirguis, S., Kulkarni, R.: A computation-oriented multimedia data streams model for content-based information retrieval. Multimed. Tools Appl. **46**(2), 399–423 (2010)
6. Chintapalli, S., Dagit, D., Evans, B., Farivar, R., Graves, T., Holderbaugh, M., Liu, Z., Nusbaum, K., Patil, K., Jerry Peng, B., Poulosky, P.: Benchmarking streaming computation engines at Yahoo!. Yahoo! Engineering blog (2015). http://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at
7. Ciaccia, P., Patella, M., Zezula, P.: M-tree: an efficient access method for similarity search in metric spaces. In: Proceedings of the 23rd VLDB International Conference, pp. 426–435 (1997)
8. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
9. Dziech, A., Leszczuk, M., Baran, R.: Guest editorial: intelligent audio-visual observation systems for urban environments. Multimed. Tools Appl. **75**(17), 10397–10405 (2016)
10. Gaber, M.M., Zaslavsky, A., Krishnaswamy, S.: Mining data streams: a review. SIGMOD Rec. **34**(2), 18–26 (2005)
11. Gunther, N.J., Puglia, P., Tomasette, K.: Hadoop superlinear scalability. Commun. ACM **58**(4), 46–55 (2015)
12. Houben, S., Stallkamp, J., Salmen, J., Schlipsing, M., Igel, C.: Detection of traffic signs in real-world images: the german traffic sign detection benchmark. In: Proceedings of the 2013 International Joint Conference on Neural Networks (IJCNN 2013), pp. 1–8 (2013)
13. Hu, H., Wen, Y., Chua, T.-S., Li, X.: Toward scalable systems for big data analytics: a technology tutorial. IEEE Access **2**, 652–687 (2014)
14. Indect Project Homepage. http://www.indect-project.eu/. Accessed 07 Sept 2017
15. Kotus, J., Łopatka, K., Czyżewski, A., Bogdanis, G.: Processing of acoustical data in a multimodal bank operating room surveillance system. Multimed. Tools Appl. **75**(17), 10787–10805 (2016)
16. Laney, D.: 3D data management: controlling data volume, velocity and variety. META Group (Gartner) research report (2001). http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf
17. Larsson, F., Felsberg, M.: Using fourier descriptors and spatial models for traffic sign recognition. In: Proceedings of the 17th Scandinavian Conference on Image Analysis (SCIA 2011), pp. 238–249 (2011)
18. Łopatka, K., Kotus, J., Czyżewski, A.: Detection, classification and localization of acoustic events in the presence of background noise for acoustic surveillance of hazardous situations. Multimed. Tools Appl. **75**(17), 10407–10439 (2016)
19. Manyika, J., Chui, M., Bughin, J., Brown, B., Dobbs, R., Roxburgh, C., Byers, A.H.: Big data: the next frontier for innovation, competition, and productivity. McKinsey Global Institute report (2011). http://www.mckinsey.com/Insights/MGI/Research/

Technology_and_Innovation/Big_data_The_next_frontier_for_innovation. Accessed 07 Sept 2017

20. Mu, C., Xie, J., Yan, W., Liu, T., Li, P.: A fast recognition algorithm for suspicious behavior in high definition videos. Multimed. Syst. **22**(3), 275–285 (2016)
21. Roh, M.-C., Lee, S.-W.: Human gesture recognition using a simplified dynamic bayesian network. Multimed. Syst. **21**(6), 557–568 (2015)
22. Sharaf, M.A., Chrysanthis, P.K., Labrinidis, A., Pruhs, K.: Algorithms and metrics for processing multiple heterogeneous continuous queries. ACM Trans. Database Syst. **33**(1), 5:1–5:44 (2008)
23. Tang, M., Pongpaichet, S., Jain, R.: Research challenges in developing multimedia systems for managing emergency situations. In: Proceedings of the 2016 ACM on Multimedia Conference (MM '16), pp. 938–947 (2016)
24. Turk, M., Pentland, A. P.: Face recognition using eigenfaces. In: Proceedings of the 1991 Conference on Computer Vision and Pattern Recognition (CVPR 1991), pp. 586–591 (1991)
25. Viola, P., Jones, M.: Rapid object detection using a boosted cascade of simple features. In: Proceedings of the 2001 Conference on Computer Vision and Pattern Recognition (CVPR 2001), pp. 511–518 (2001)
26. Wolf, L., Hassner, T., Maoz, I.: Face recognition in unconstrained videos with matched background similarity. In: Proceedings of the 2011 Conference on Computer Vision and Pattern Recognition (CVPR 2011), pp. 529–534 (2011)
27. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud '10), p. 10 (2010)
28. Zhang, J., Han, Y., Jiang, J.: Tucker decomposition-based tensor learning for human action recognition. Multimed. Syst. **22**(3), 343–353 (2016)
29. Zeng, L., Li, L., Duan, L., Lu, K., Shi, Z., Wang, M., Wu, W., Luo, P.: Distributed data mining: a survey. Inf. Technol. Manag. **13**(4), 403–409 (2012)
30. Zheng, Z., Jeong, H.-Y., Huang, T., Shu, J.: KDE based outlier detection on distributed data streams in multimedia network. Multimed. Tools Appl. **76**(17), 18027–18045 (2016)