Comparing Performances of Big Data Stream Processing Platforms with RAM^{3}S (extended abstract)

Ilaria Bartolini and Marco Patella

DISI - Alma Mater Studiorum, Università di Bologna
{ilaria.bartolini,marco.patella}@unibo.it

Abstract. Nowadays, Big Data platforms allow the analysis of massive data streams in an efficient way. However, the services they provide are often too raw, thus the implementation of advanced real-world applications requires a non-negligible effort for interfacing with such services. This also complicates the task of choosing which one of the many available alternatives is the most appropriate for the application at hand. In this paper, we present a comparative study of the three major open-source Big Data platforms for stream processing, as performed by using our novel RAM^{3}S framework. Although the results we present are specific for our use case (recognition of suspect people from massive video streams), the generality of the RAM^{3}S framework allows both considering such results as valid for similar applications and implementing different use cases on top of Big Data platforms with very limited effort.

1 Introduction

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, etc.

In order 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 (like sensors, cameras, etc.) 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.

To this end, we exploit our RAM^{3}S (Real-time Analysis of Massive Multi-Media Streams) framework for the automated real-time analysis of multimedia streams to compare the performance of three different open source engines for the analysis of streaming Big Data. 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.[1]

Although the problem of stream analysis is not novel per se [5,2], its application in 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.

In order to allow efficient analysis of massive multimedia streams in real scenarios, we advocate scaling out the underlying system by using platforms for Big Data management. Since we deal with online applications, we are interested in analyzing the data as soon as these are available, in order to exploit their "freshness". Therefore, we concentrate on platforms for Big Data analytics following the *stream processing* paradigm, as opposed to systems for *batch processing* [6]. While in the latter, whose main representative is Apache Hadoop (`hadoop.apache.org`), data are first stored on a physical support (usually on a Distributed File System, *DFS*) and then analyzed, for the case of stream processing data are 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.

2 Big Data Analysis Platforms

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

Apache Spark Streaming Apache Spark [10] (`spark.apache.org`) is an open source platform, originally developed at the University of California, Berkeley's AMPLab. 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

[1] 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 which is however out of the scope of this paper.

operations that can be performed on a RDD: a *transformation* is any operation that produces a new RDD, built from the original one; on the other hand, an *action* is any operation that produce a single value or writes an output on disk. 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.

The Streaming version of Spark introduces receiver components, in charge of receiving data from multiple sources, like Apache Kafka, 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. 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: A *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; in a *driver* 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.

Apache Storm Apache Storm (`storm.apache.org`) is an open source platform, 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*s are nodes that generate the stream of tuples, binding to an external data source (like a TCP/IP socket or a message queue), while *bolt*s are 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. The overall topology, therefore, specifies the computation pipeline. A key concept in Storm is the *acknowledgment* of input data, in order 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.

Apache Flink Apache Flink (`flink.apache.org`) is an open source platform, 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 dis-

tribution 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* [3], 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 DFS) at fixed time intervals. The use of the snapshot algorithm provides Flink with a "exactly-once" semantics, 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.

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.

3 The RAM^{3}S Framework

The RAM^{3}S framework provides 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.

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. 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.

3.1 Applying the Framework to the Use Case

Our scenario of suspect people recognition is illustrated in Figure 1. 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). The `Analyzer` receives a flow of images, and performs the face detection task first, by exploiting an Haar cascade detector [8], followed by face recognition, by way of an eigenfaces algorithm [7]: both techniques are provided by the OpenIMAJ library (`openimaj.org`).

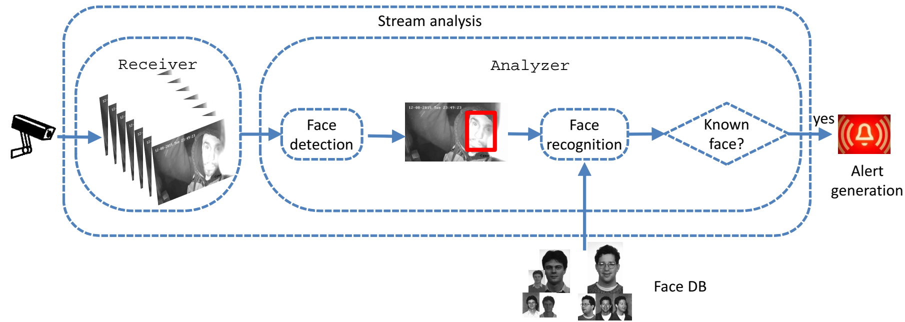

Fig. 1. Instantiation of the RAM^{3}S framework classes for suspect face recognition.

3.2 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 Figure 2, we show how RAM^{3}S has been appropriately extended in order to work as components of the distributed Big Data platforms.

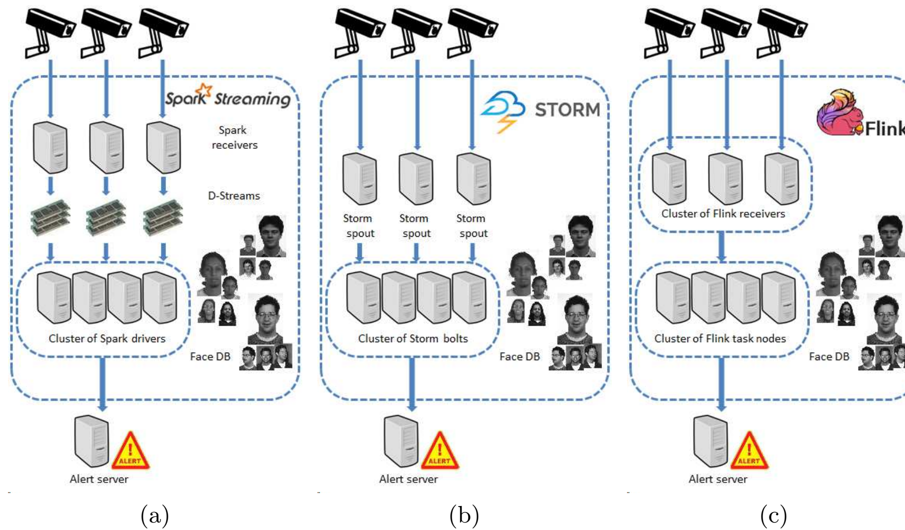

Fig. 2. Architecture for Big Data platforms: Spark (a), Storm (b), and Flink (c).

Apache Spark For Spark (Figure 2 (a)), 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.

Apache Storm For Storm (Figure 2 (b)), 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 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 the completion of a D-stream to begin the analysis of any single `MMObject`.

Apache Flink Finally, for Flink (Figure 2 (c)) 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 Figure 2 (c) 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).

4 Experiments

The goal of our experiments is to evaluate the efficiency of the proposed architectures, comparing them on a fair basis. The open source platforms for real-time data analysis described in Section 2 have been implemented both in a local environment with a cluster of 12 PCs and on the Google Cloud Platform (`cloud.google.com`). The first example, realized in our datalab (`www-db.disi.`

unibo.it/research/datalab/), 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.

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 100Mbps 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 3.75GB of RAM. Our real dataset for the face detection/recognition task consists of the YouTube Faces Dataset (YTFD) [9], including 3,425 videos of 1,595 different people, with videos having different resolutions (480x360 is the most common size) and a total of 621,126 frames containing at least a face (on average, 181.3 frames/video). In order to be able to freely vary the input frame rate, the incoming stream of video frames was implemented as a RabbitMQ queue fed with sequences of still images.

Figure 3 shows the sustainable input rate (the maximum frame rate such that the input buffer does not overflow) 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.

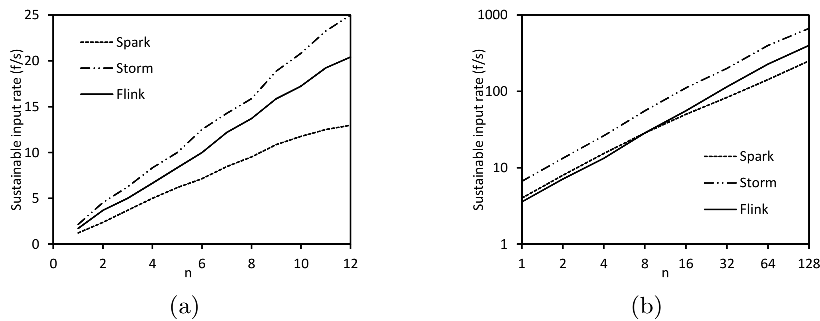

Fig. 3. Sustainable input rate when varying the number n of computing nodes for the local cluster (a) and cloud (b).

Figure 4 illustrates the latency for the three architectures on the local and the cloud scenarios. Latency of the system is the measured as the (average) time interval between an image exiting the FIFO queue and the corresponding alert signal received by the alert server, when using the maximum input rate sustainable by an architecture composed of a single node. For Apache Spark, the delay is the average delay of images included in a single D-Stream. Since

a micro-batch period of 30 seconds was considered, this can be computed as 30/2 plus the average latency. Figure 4 only plots the average latency, omitting the 15 seconds from the total delay. 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.

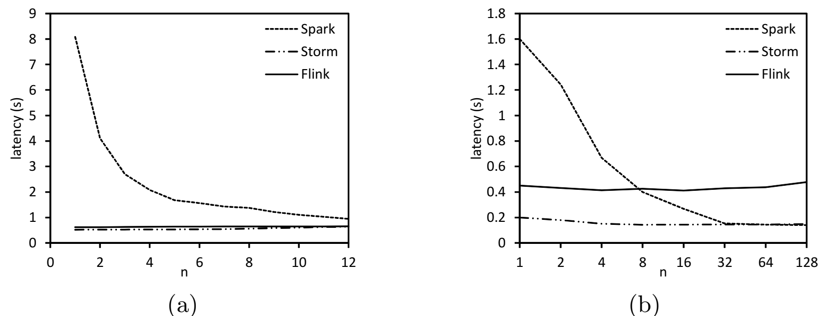

Fig. 4. Latency when varying n for the local cluster (a) and cloud (b).

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, 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 in latency: as already pointed out in Section 3.2, 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 [4].

References

1. A. Alexandrov, R. Bergmann, et al. The Stratosphere Platform for Big Data Analytics. In *VLDBJ*, 23(6), 2014.
2. A. Bifet, G. Holmes, et al. MOA: Massive Online Analysis. In *JMLR*, 99, 2010.
3. K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. In *TOCS*, 3(1), 1985.
4. S. Chintapalli, D. Dagit, et al. Benchmarking Streaming Computation Engines at Yahoo!. *Yahoo! Engineering blog*, 2015.
5. M.M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining Data Streams: A Review. In *SIGMOD Record*, 34(2), 2005.
6. H. Hu, Y. Wen, T.-S. Chua, and X. Li. Toward Scalable Systems for Big Data Analytics: A Technology Tutorial. In *IEEE Access*, 2, 2014.
7. M. Turk and A.P. Pentland. Face Recognition Using Eigenfaces. In *CVPR 1991*, Lahaina, HI.
8. P. Viola and M. Jones. Rapid Object Detection Using a Boosted Cascade of Simple Features. In *CVPR 2001*, Kauai, HI.
9. L. Wolf, T. Hassner, and I. Maoz. Face Recognition in Unconstrained Videos with Matched Background Similarity. In *CVPR 2011*, Colorado Springs, CO.
10. M. Zaharia, M. Chowdhury, et al. Spark: Cluster Computing with Working Sets. In *HotCloud '10*, Boston, MA.