Information Fusion 42 (2018) 51–61

Contents lists available at ScienceDirect

Information Fusion

journal homepage: www.elsevier.com/locate/inffus

Big Data: Tutorial and guidelines on information and process fusion for analytics algorithms with MapReduce

Sergio Ramírez-Gallego*,[a], Alberto Fernández[a], Salvador García[a], Min Chen[b], Francisco Herrera[a]

[a] Department of Computer Science and Artificial Intelligence, University of Granada, Granada, Spain
[b] School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China

ARTICLE INFO

Keywords:
Big Data Analytics
MapReduce
Information fusion
Spark
Machine learning

ABSTRACT

We live in a world were data are generated from a myriad of sources, and it is really cheap to collect and storage such data. However, the real benefit is not related to the data itself, but with the algorithms that are capable of processing such data in a tolerable elapse time, and to extract valuable knowledge from it. Therefore, the use of Big Data Analytics tools provide very significant advantages to both industry and academia. The MapReduce programming framework can be stressed as the main paradigm related with such tools. It is mainly identified by carrying out a distributed execution for the sake of providing a high degree of scalability, together with a fault-tolerant scheme.

In every MapReduce algorithm, first local models are learned with a subset of the original data within the so-called Map tasks. Then, the Reduce task is devoted to fuse the partial outputs generated by each Map. The ways of designing such fusion of information/models may have a strong impact in the quality of the final system. In this work, we will enumerate and analyze two alternative methodologies that may be found both in the specialized literature and in standard Machine Learning libraries for Big Data. Our main objective is to provide an introduction of the characteristics of these methodologies, as well as giving some guidelines for the design of novel algorithms in this field of research. Finally, a short experimental study will allow us to contrast the scalability issues for each type of process fusion in MapReduce for Big Data Analytics.

1. Introduction

Big Data Analytics is nowadays one of the most significant and profitable areas of development in Data Science [1–6]. One of the main reasons of its success is related with the Internet-of-Things (IoT), the Web 2.0 and the social networks, and all the myriad of data from different sources that can be collected and processed [6–8]. In this sense, corporations that are able to extract valuable knowledge from large volumes of data in a reasonable time, may obtain significant advantages over their competitors [9,10]. Researchers from academia are also aware of the interest in developing robust and accurate models for Data Mining in Big Data applications [11,12]. There is a clear growing rate in the number of research studies [13,14], and the trend is not expected to change in the short future.

However, even years after the boom of Big Data, there is still a misleading definition for the concept itself [15]. We must stress that the topic of Big Data is strongly linked with the scalability issue [16]. Those models developed in this context must be able to adapt dynamically the data growth, as well as being fault tolerant to be reliable in case of time consuming operations. In order to fulfill these requirements, a change in the traditional technology and framework for carrying out the learning process is mandatory [17].

MapReduce (MR) has established as a de-facto solution that comprises all the previous capabilities [18–20]. It is basically an execution environment which lays over a distributed file system [21]. By means of two simple functions, Map and Reduce, any implementation can be automatically parallelized in a transparent way for the programmer, also supporting by default the aforementioned fault-tolerant scheme.

- The Map function is devoted to divide the computation into different subparts, each one related to a partial set of the data.
- The Reduce function needs to fuse the local outputs into a single final model.

Whereas the procedure to be included in the Map task is, most times, straightforward to determine, the hitch comes when deciding how to carry out the models' fusion within the Reduce task. At this point, the design depends on many factors, namely whether the

* Corresponding author.
E-mail addresses: sramirez@decsai.ugr.es (S. Ramírez-Gallego), alberto@decsai.ugr.es (A. Fernández), salvagl@decsai.ugr.es (S. García), minchen2012@hust.edu.cn (M. Chen), herrera@decsai.ugr.es (F. Herrera).

http://dx.doi.org/10.1016/j.inffus.2017.10.001
Received 26 September 2017; Accepted 3 October 2017
Available online 10 October 2017

submodels are different and independent among them, or they have a nexus for being able to join them directly.

In this paper, we aim at analyzing the different alternatives on process fusion for Big Data Analytics models under the MR framework. To do so, we propose a brief taxonomy distinguishing two types of approaches.

1. Direct fusion of models: approximate methods. We refer to those that carry out a direct fusion of partial models via an ensemble system [22].
2. Exact fusion for scalable models: distributed data and models' partition. In this case, they are those designs that carry out a global distribution of both data and sub-models (the prior types mentioned just considered data division), and iteratively build the final system.

We will carry out a practical study on scalability for each of the different fusion proposals with the sake of contrasting how time and accuracy performance vary as resources increase. In order to provide a better understanding of each type of implementation, we will present some case studies of these algorithms from both well-known Machine Learning libraries such as Mahout [23–25] (from Apache Hadoop [26,27]) and MLlib [28] (from Apache Spark [29,30]).

In order to address all these objectives, this paper is organized as follows. First, Section 2.1 presents an introduction on the MR programming framework, also stressing some alternatives for Big Data processing. Section 3 includes an overview on those technologies currently available to address Big Data problems from a distributed perspective. Section 4 presents the core of this paper, analyzing the different design options for developing Big Data Analytics algorithms regarding how the partial data and models are fused. Then, we show a case study in Section 5 to contrast the capabilities regarding scalability of the different approaches previously introduced. In Section 6 we present a discussion on the findings obtained in this research, as well as several guidelines for future study on the topic. Finally, Section 7 summarizes and concludes this paper.

2. MapReduce as information and process fusion

The rapid growth and influx of data from private and public sectors, and the novel opportunities derived from the IoT [31], have popularized the notion of "Big data [3,4,11]". This scenario has led to the development of custom paradigms for distributed processing that are able to extract significant value and insight in different areas such as Bioinformatics [32], health care [33], social mining [34,35], and so on.

Although focused on standard processing, distributed paradigms have also been widely utilized for fusing information [36,37] (ontologies and genetic data) and learning models [38,39] (trees and fuzzy rules). This section describes in detail these paradigms, paying more attention to the most widespread paradigm in the market: MR. Furthermore, several examples on how MR is applied as a fusion process are given here.

2.1. MapReduce programming model

The MR execution environment [18] is the most common paradigm used in the distributed processing scenario. Being a privative tool, its open source counterpart, known as Hadoop, has been traditionally used in academia research [27]. It has been designed to allow distributed computations in a transparent way for the programmer, also providing fault tolerance, automatic data partition and management, and automatic job/resource scheduling. To take advantage of this scheme, any algorithm must be divided into two main stages: Map and Reduce. The first one is devoted to split the data for processing, whereas the second collects and aggregates the results.

Additionally, the MR model is defined with respect to an essential data structure: the (key,value) pair. The processed data, the intermediate and final results work in terms of (key,value) pairs. To summarize its procedure, Fig. 1 illustrates a typical MR program with its *Map* and *Reduce* steps.

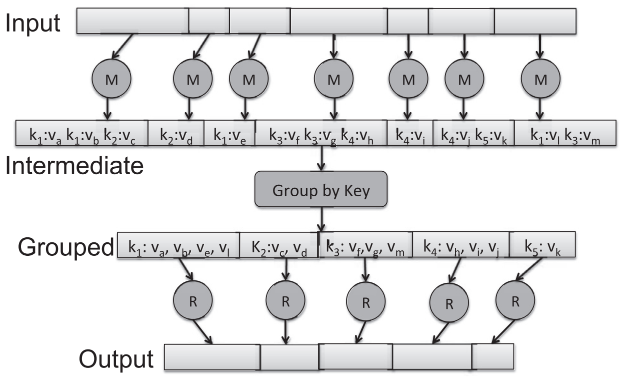
Input
M
Intermediate
k_1:v_a k_1:v_b k_2:v_c
k_2:v_d
k_1:v_e
k_3:v_f k_3:v_g k_4:v_h
k_4:v_i
k_4:v_j k_5:v_k
k_1:v_l k_3:v_m
Group by Key
Grouped
k_1: v_a, v_b, v_e, v_l
K_2:v_c, v_d
k_3: v_f,v_g, v_m
k_4: v_h, v_i, v_j
k_5: v_k
R
Output

Fig. 1. The MapReduce programming model. *k* elements represent the keys in the pairs, whereas *v* the values.

The MR scheme can be described as follows.

- Map function first reads data and transforms records into a key-value format. Transformations in this phase may apply any sequence of operations on each record before sending the tuples across the network.
- Output keys are then shuffled and grouped by key value so that coincident keys are grouped together to form a list of values. Keys are then partitioned and sent to the Reducers according to some key-based scheme previously defined.
- Finally, the Reducers perform some kind of fusion on the lists to eventually generate a single value for each pair. As a further optimization, the reducer is also used as a combiner on the map outputs. This improvement reduces the total amount of data sent across the network by combining each word generated in the Map phase into a single pair.

Apart from considering MR as a processing paradigm, this scheme (concretely, the Reduce stage) can also be seen as a fusion process that allows to blend partial models and information schemes into a final fused outcome. Fusion of models in MR is typically performed following some sort of ensemble strategy that combine multiple hypothesis through voting or attachment. Also other proposals exist that go beyond ensemble learning, and offers as outcome a single coalesced model. For example, logistic regression in Spark is composed by several subgradients that are locally computed and eventually aggregated (more examples will be given in Section 4). From another perspective, we may refer to aggregation of partial information collected within different maps. In this case, the fusion process will be more dependent from the input domain. The amount of scenarios that can be found in this context is highly diverse. Use cases for MR in the literature range from fusion of ontologies [36] to the composition of fuzzy rules [39], among others.

Word Count comes to be one of the most widespread examples to illustrate the intrinsic information fusion process behind MR. WordCount is intended to count the number of occurrences per word in a set of input text files. Each mapper reads a set of blocks formed by lines, and splits them into words. It then emits a key-value pair with the word as key, and 1 as value. Afterwards, each reducer sums the scores for each word, and outputs a single key-value pair with the word and sum.

Consider the phrase 'knock, knock, who is there?". A single mapper would receive and split this sentence as words, and then, it would form the initial pairs as: (knock,1), (knock,1), (who,1), (is,1), (there,1). In reducers, the keys are grouped together and the count values for identical keys (words) are added. In this case only one pair of similar keys 'knock' would be aggregated so that the output pairs would be as

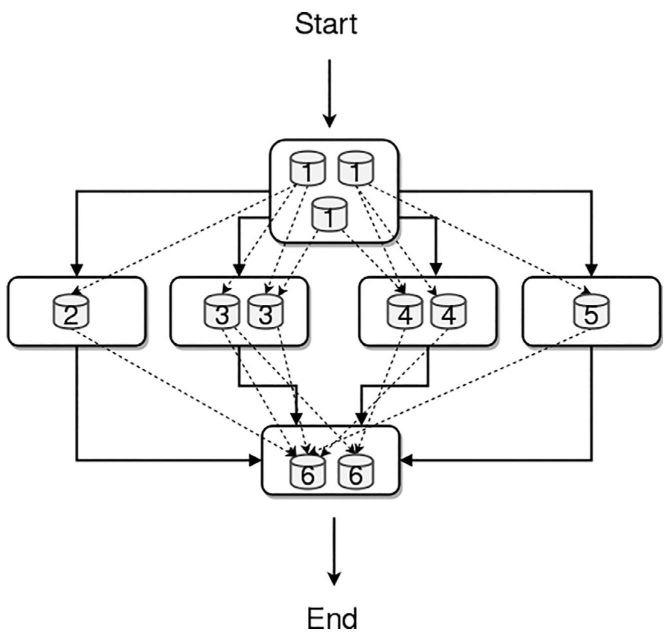

Fig. 2. Direct Acyclic Graph Parallel Processing. Squares represent the tasks to process and the nodes in the graph, arrows connecting nodes represent the data flow between nodes and the vertexes in the graph, dashed lines represent the dependencies between data blocks (cylinders).

follows: (knock,2), (who,1), (is,1), (there,1). As result, the user would receive the final number of hits for each word.

2.2. Alternative distributed processing paradigms

Although MR is the most popular paradigm to tackle distributed processing, there exist other modern distributed frameworks, conceived prior to the dawn of MR, that offer other alternatives for information and model fusion. Most of them follows a Single Instruction Multiple Datasets (SIMD) [40] scheme to execute the same sequence of instructions simultaneously on a distributed set of data partitions. In this section we focus on two paradigms (graph and bulk processing) relevant for current Big Data platforms.

2.2.1. Directed Acyclic Graph Parallel processing

All distributed frameworks based on Directed Acyclic Graph (DAG) [41], like Spark, organize their jobs by splitting them into a smaller set of atomic tasks. In this model vertexes correspond with parallel tasks, whereas edges are associated with exchange of information. As shown in Fig. 2, vertexes can have multiple connections between inputs and outputs, which imply that the same task can be run in different data and the same data in different partitions. Data flows are physically supported by shared memory, pipes, or disks. Instructions are duplicated and sent from the master to the slave nodes for a parallel execution. Notice that MR can be deemed as an specific implementation of DAG-based processing, with only two functions as vertexes.

2.2.2. Bulk Synchronous Parallel processing

Bulk Synchronous Parallel (BSP) [42] systems are formed by a series of connected supersteps, implemented as directed graphs. In this scheme input data is the starting point. From here to the end, a set of Supersteps are applied on partitioned data in order to obtain the final output. As mentioned before, each Superstep correspond with an independent graph associated with a subtask to be solved. Once all compounding subtasks end, bulk synchronization of all outputs is committed. At this point vertexes may send messages to the next Superstep, or receive some from previous steps, and also to modify its state and outgoing edges. Fig. 3 shows a toy example for BSP processing with two Supersteps and one synchronization barrier.

3. Big Data technologies for analytics

Nowadays, the volume of data currently managed by our storage systems have surpassed the processing capacity of traditional systems [11], and this applies to Data Mining as well. Distributed computing has been widely used by experts and practitioners before the advent of Big Data to boost up sequential solutions in medium-size data. Nevertheless, for most of current massive problems, a distributed approach becomes mandatory nowadays since no batch architecture is able to address such magnitudes.

Beyond High Performance Computing solutions, new large-scale processing platforms are intended to bring closer distributed processing to practitioners and experts by hiding the technical nuances derived from these environments. Novel and complex designs are required to create and maintain these platforms, which generalizes the utilization of distributed computing for standard users.

As a result of the fast evolving of Big Data environment, a myriad of tools, paradigms and techniques have surged to tackle different use cases in industry and science. However, because of this large number of alternatives, it is often difficult for practitioners and experts to analyze and select the right tool for each goal.

In this section we present and analyze three well-known alternatives for distributed processing belonging to the “Apache Hadoop ecosystem.” The objective is providing the necessary knowledge that helps users to decide which alternative better fits their requirements. We also outline the software libraries that gives support to the distributed learning task in these platforms, being summarized in Table 1.

3.1. Apache Hadoop

Undoubtedly Hadoop MapReduce may be deemed as the primary platform in the Big Data space. After the presentation of MR by Google designers [43], Hadoop MapReduce was grown by the community, and became the most used and powerful open-source implementation of MR. Nowadays leading companies such as Yahoo has scaled from 100-node Hadoop clusters to 42K nodes and hundreds of petabytes [44] thanks to the outstanding performance of Hadoop.

The main idea behind Hadoop was to create a common framework which can process large-scale data on a cluster of commodity hardware, without incurring in a high cost in developing (in contrast to HPC solutions) and execution time. Hadoop MapReduce was originally composed by two elements: the first one was a distributed storage system called Hadoop Distributed File System (HDFS), whereas the second one was a data processing framework that allows to run MR-like jobs. Apart from these goals, Hadoop implements primitives to address cluster scalability, failure recovery, and resource scheduling, among others.

But Hadoop is today more than a single technology, but a complete software stack and ecosystem formed by several top-level components that address diverse purposes. For instance, Apache Giraph for graph processing or Apache Hive for data warehousing. The common factor is that all of them rely on Hadoop, and are tightly linked to this technology. Some projects are actually Apache top-level projects [45], whereas others are continuously evolving or being created.

HDFS [46] can be deemed as the main module of Apache Hadoop. It supports distributed storage for large-scale data through the use of distributed files, which themselves are composed by fixed-size data blocks. These blocks or partitions are equally distributed among the data nodes in order to balance as much as possible the overall disk usage in the cluster. HDFS also allows replication of blocks across different nodes and racks. In HDFS, the first block is ensured to be placed in the same processing node, whereas the other two replicas are sent to different racks to prevent abrupt ends due to inter-rack issues.

HDFS was thought to work with several storage formats. It offers several APIs to read/write registers. Some relevant APIs are:

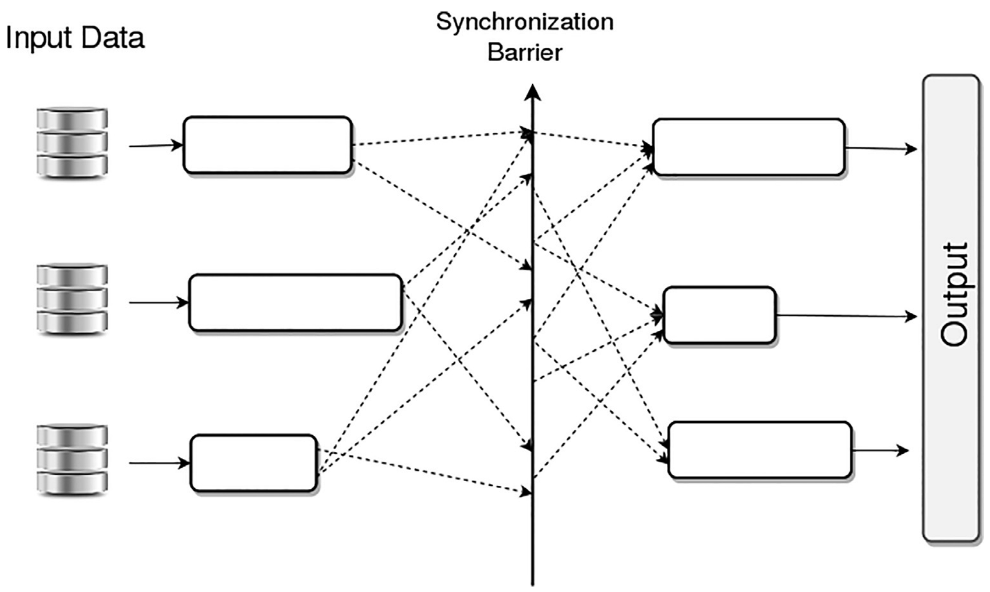

Fig. 3. Bulk Synchronous Parallel processing. Subtasks in each Superstep are depicted as rectangles with variable height (task duration), and data flows as dashed lines. Synchronization barrier acts as a time proxy between stages.

Table 1
Analytics tools for each Big Data platform.

Big Data distributed platforms	Analytics tools
Hadoop	Mahout
Spark	MLlib
Flink	FlinkML

InputFormat (to read customizable registers), or RecordWriter (to write record-shaped elements). Users can also developed their own storage format, and to compress data according to their requirements. Persistence in Hadoop is mainly performed in disk. However, there are some novel advances to optimize persistence by introducing some memory usage. For instance, in Apache Hadoop version 3.0 was introduced the option of memory usage as temporary storage.

Although **MR** [43] is the native processing solution in Apache Hadoop, today it supports multiple alternatives with different processing schemes. All these solutions have in common that use a set of data nodes to run tasks on the local data blocks, and one master node (or more) to coordinate these tasks. For instance, **Apache Tez** [47] is a processing engine that transforms processing jobs into direct acyclic graphs (DAGs). Thanks to Tez, users can run any arbitrary complex DAG of jobs in HDFS. Tez thus efficiently solves interactive and iterative processes, like those present in machine learning processes. Its most relevant contribution is that Tez translate any complex job to a single MR phase. Furthermore, it does not need to store intermediate files and reuses idle resources, which highly boost the overall performance.

Hadoop MapReduce evolves to a more general component, called **Yet Another Resource Negotiator (YARN)** [48], which provides extra management and maintenance services relied to other components in the past. YARN also acts as a facade for different types of distributed processing engines based on HDFS, such as Spark, Flink or Storm. In short, YARN was intended as a generic purpose system that separates the responsibilities of resource management (performed by YARN), and running management (performed by top-level applications).

Among the full set of advantages claimed by YARN, we can highlight its capacity to run several application on the same cluster without the necessity of moving data. In fact, YARN allows reusing resources across alike applications in parallel, which improves the overall usage of resources.

3.1.1. Apache Mahout

Since the magnitude of learning problems has been growing exponentially, data scientists demands rapid tools that efficiently extract knowledge from large-scale data. This problem has been solved by MR and other platforms by providing scalable algorithms and miscellaneous utilities in form of machine learning libraries. These libraries are compatible with the main Hadoop engine, and use as input the data stored in the storage components.

Apache Mahout [49] was the main contribution from Apache Hadoop to this field. Although it can be deemed as mainly obsolete nowadays, Mahout is considered as the first attempt to fill the gap of scalable machine learning support for Big Data. Mahout comprises several algorithms for plenty of tasks, such as: classification, clustering, pattern-mining, etc. Among a long list of golden algorithms in Mahout, we can highlight Random Forest or Naïve Bayes.

The most recent version (0.13.0) provides three new major features: novel support for Apache Spark and Flink, a vector math experimentation for R, and GPU support based on large matrix multiplications. Although Mahout was originally designed for Hadoop, some algorithms have been implemented on Spark as a consequence of the latter one's popularity. Mahout is also able to run on top of Flink, being only compatible for static processing though.

3.2. Spark

Apache Spark Framework [50] was born in 2010 with the publication of Resilient Distributed Datasets (RDD) structures [30], the keystone behind Spark. Although Spark has a close relationship with Hadoop Ecosystem, it provides specific support for every step in the Big Data stack, such as its own processing engine, and machine learning library.

Apache Spark [51] is defined as a distributed computing platform which can process large volume data sets in memory with a very fast response time due to its memory-intensive scheme. It was originally thought to tackle problems deemed as unsuitable for previous disk-based engines like Hadoop. Continued use of disk is replaced in Spark by memory-based operators that efficiently deal with iterative and interactive problems (prone to multiple I/O operations).

As stated previously, the heart of Spark is formed by **RDDs**, which transparently controls how data are distributed and transformed across the cluster. Users just need to define some high-level functions that will be applied and managed by RDDs. These elements are created whenever data are read from any source, or as a result of a transformation.

RDDs consist of a collection of data partitions distributed across several data nodes. A wide range of operations are provided for transforming RDDs, such as: filtering, grouping, set operations, among others. Furthermore RDDs are also highly versatile as they allows users to customize partitioning for an optimized data placement, or to preserve data in several formats and contexts.

In Spark, fault tolerance is solved by annotating operations in a structure called lineage. Spark transformations annotated in the lineage are only performed whenever a trigger I/O operations appears in the log. In case of failure, Spark re-computes the affected brach in the lineage log. Although replication is normally skipped, Spark allows to spill data in local disk in case the memory capacity is not sufficient.

Spark developers provided another high-level abstraction, called **DataFrames**, which introduces the concept of formal schema in RDDs. DataFrames are distributed and structured collections of data organized by named columns. They can be seen as a table in a relational database or a dataframe in R, or Python (Pandas). As a plus, relational query plans built by DataFrames are optimized by the Spark's Catalyst optimizer throughout the previously defined schema. Also thanks to the scheme, Spark is able to understand data and remove costly Java serialization actions.

A compromise between structure awareness and the optimization benefits of Catalyst is achieved by the novel Dataset API. **Datasets** are strongly typed collections of objects connected to a relational schema. Among the benefits of Datasets, we can find compile-time type safety, which means applications can be sanitized before running. Furthermore, Datasets provide encoders for free to directly convert JVM objects to the binary tabular Tungsten format. These efficient in-memory format improves memory usage, and allows to directly apply operations on serialized data. Datasets are intended to be the single interface in future Spark for handling data.

3.2.1. MLlib

MLlib project [28] was born in 2012 as an extra component of Spark. It was released and open-sourced in 2013 under the Apache 2.0 license. From its inception, the number of contributions and people involved in the project have been growing steadily. Apart from official API, Spark provides a community package index [52] (Spark Packages) to assemble all open source algorithms that work with MLlib.

MLlib is a Spark library geared towards offering distributed machine learning support to Spark engine. This library includes several out-of-the-box algorithms for alike tasks, such as: classification, clustering, regression, recommendation, even data preprocessing. Apart from distributed implementations of standard algorithms, MLlib offers:

- Common Utilities: for distributed linear algebra, statistical analysis, internal format for model export, data generators, etc.
- Algorithmic optimizations: from the long list of optimizations included, we can highlight some: decisions trees, which borrow some ideas from PLANET project [53] (parallelized learning both within trees and across them); or generalized linear models, which benefit from employing fast C++++-based linear algebra for internal computations.
- Pipeline API: as the learning process in large-scale datasets is tedious and expensive, MLlib includes an internal package (*spark.ml*) that provides an uniform high-level API to create complex multi-stage pipelines that connect several and alike components (preprocessing, learning, evaluation, etc.). *spark.ml* allows model selection or hyper-parameter tuning, and different validations strategies like k-fold cross validation.
- Spark integration: MLlib is perfectly integrated with other Spark components. Spark GraphX has several graph-based implementations in MLlib, like LDA. Likewise, several algorithms for online learning are available in Spark Streaming, such as online k-Means. In any case, most of component in the Spark stack are prepared to effortlessly cooperate with MLlib.

3.3. Flink

Apache Flink [54] is a distributed processing component focused on streaming processing, which was designed to solve problems derived from micro-batch models (Spark Streaming). Flink also supports batch data processing with programming abstractions in Java and Scala, though it is treated as a special case of streaming processing. In Flink, every job is implemented as a stream computation, and every task is executed as cyclic data flow with several iterations.

Flink provides two operators for iterations [55], namely, standard and delta iterator. In standard iterator, Flink only works with a single partial solution, whereas delta iterator utilizes two worksets: the next entry set to process and the solution set. Among the set of advantages provided by iterators is the reduction of data to be computed and sent between nodes [56]. According to the authors, new iterators are specially designed to tackle machine learning and data mining problems.

Apart from iterators, Flink leverages from an optimizer that analyzes the code and the data access conflicts to reorder operators and create semantically equivalent execution plans [57,58]. Physical optimization is then applied on plans to boost data transport and operators' execution on nodes. Finally, the optimizer selects the most resource-efficient plan, regarding network and storage.

Furthermore, Flink provides a complex fault tolerance mechanism to consistently recover the state of data streaming applications. This mechanism is generating consistent snapshots of the distributed data stream and operator state. In case of failure, the system can fall back to these snapshots.

FlinkML is aimed at providing a set of scalable ML algorithms and an intuitive API to Flink users. Until now, FlinkML provides few alternatives for some fields in machine learning: SVM with CoCoA [59], or Multiple Linear regression for supervised learning, k-NN join for unsupervised learning, scalers and polynomial features for preprocessing, Alternating Least Squares for recommendation, and other utilities for validation and outlier selection, among others. FlinkML also allows users to build complex analysis pipelines via chaining operations (like in MLlib). FlinkML pipelines are inspired by the design introduced by sklearn in [60].

3.4. Comparison among Big Data alternatives (Hadoop, Spark and Flink)

Main divergence between Big Data frameworks rests on its design philosophy, and how they respond to different data formats such as data streams. Starting from Hadoop, we can directly assert that Hadoop is essentially batch-oriented due to its intensive disk usage. In contrast, Flink is a native streaming technology originally designed to work with memory-based streams. Although Apache Spark was not originally designed for static problems, it provides a micro-batching strategy capable of easily processing streaming data. Spark micro-batching however may be deteriorated when it faces low latency requirements.

Unlike Hadoop MapReduce, Spark and Flink both offer native support for in-memory persistence and iterative processing. Spark allows users to persist data in memory, and load them in several occasions. Regarding the execution engine, Spark relies on an acyclic graph planner formed by vertices and edges, which can be seen as a strict generalization of MapReduce. In counterpart, Flink utilizes a thoroughly iterative processing scheme created from the scratch, which is based on cyclic data flows (a single iteration per schedule).

Moving to optimization matters, Spark and Flink both provide mechanisms to analyze, control and optimize user code so that the best execution plan for each program is obtained. Spark mainly exploits new SQL-based DataFrame API and the Tungsten engine for optimization, whereas Flink did that as first citizen. Manual optimization can also be carried out by controlling how data are partitioned, or transmitted across the network.

Finally, Apache Spark and Flink offer plenty of alternatives to coders specially attached with a given programming language. Namely, Spark offers ad-hoc APIs for R, Java or Python, and a native support for the Scala language, whereas Apache Flink only focus on Java Virtual Machine, providing full APIs for Scala and Java. In contrast, Hadoop only supports Java. For further comparison insights, please refer to García-Gil et al. [61].

4. Big Data analytics as a process fusion

"Synchronizing flags" in every distributed algorithmic approach is undoubtedly the most challenging part during the design process. This issue is not an exception in the MR scheme. In this particular case, developers must take two significant decisions. On the one hand, the components selected for the key-value representation in both the Map and Reduce input and outputs. On the other hand, how the list of values are aggregated in the Reduce step. In this section, we want to analyze in detail this characteristic of the MR programming environment by introducing a taxonomy to organize current state-of-the-art approaches regarding how partial models from the Map task are aggregated. We will determine how this fact may also affects the actual scalability of the whole system.

Specifically, we distinguish among two different type of models according to the fusion strategy implemented. First, those that produce an approximate model by applying a direct process fusion on partial submodels (Section 4.1). Second, those that distribute both data and models to iteratively build a final exact system (Section 4.2).

Besides this standard categorization, some further considerations must be taken into account in order to properly classify large-scale learning algorithms. Among others, we must distinguish between:

- Whether the model (or required statistics) is evaluated (or are computed) on all the distributed partitions (global), or local submodels are independently yielded by each task and then fused (local).
- Whether algorithms only consist of a single stage (1-step) or several iterations/stages are required (multistage).
- Whether a master node guides the model construction process (guided) or it is fully decoupled (unguided).

Table 2 enumerates and categorizes the distributed methods described below according to the previous taxonomy. Note that although some categories appear more frequently with others, all categories are independent.

4.1. Direct fusion of models: approximate methods

Roughly, approximate distributed models are those that emulate the learning behavior of sequential algorithms, yet generating a completely different and non-exact solution. Most of them follows an ensemble paradigm, which is straightforwardly parallelizable in the MR framework following the rule of thumb: one submodel per mapper. The compounding model will be eventually generated by directly joining all submodels in the reduce phase [71]. Note that the MR approach reminds of the traditional *bootstrap aggregating* (bagging) approach; yet in this case with no replacement.

The premise for this type of methodologies is simple yet effective. Each Map task works independently on its chunk of data. Depending on the user's requirements, each Map function is devoted to build one or more models. For example, a pool of 100 classifiers to be obtained from 5 Maps will result on 20 classifiers per Map, each one of them built from a 20%% of the original dataset. The "key" is shared by all values given as output from the Map. This fact makes the logic of the Reduce phase to stress for its simplicity. Every single classifier is directly joined into the final system. In the following, we shortly describe some algorithms.

- One of the first approaches to build an approximate ensemble with MapReduce was the Random Forest for Hadoop [72,73], whose implementation can be found at Mahout-MapReduce Machine Learning library [23,24]. The algorithm consist of two MR phases: the first one is devoted to the development of the model (where model fusion occurs), and the second one is focused on class prediction once the model is built.
  In the first phase, each Map creates several random trees (a subset of the forest) using the partitions given as input [74]. The reduce phase simply concatenates all the trees forming the final forest of random trees. The second task replicates the forest across the nodes, and launches the mappers on the test set partitions. Mappers predict the class for each record assigned by using the final model. Finally, reducers concatenate predictions in a single file.
- Another interesting approaches come from the boosting perspective [75], which are AdaBoost.PL, LogitBoost.PL and MultiBoost.PL [64]. These include an ensemble of ternary classifiers [76], or an ensemble of ensembles. In this case, the reduce phase is much more complex than applying a simple voting or coalescing scheme. Concretely, the whole boosting procedure, i.e. T iterations, is carried out within each Map, so M ensembles of T classifiers are obtained, being M the number of maps. Then, the T classifiers are arranged according to their score, i.e. the weighted error, and then emitted to the Reducers using their index as "key" and the model as value. Finally, each Reduce process takes all classifiers with the same index (key) and compute an average weight for this ensemble. At classification time, each "sub-ensemble" provides a vote for its class, whose value is exactly the aforementioned weighted average score.
- Prototype Reduction [65] is also a another significant example that provides a further complex and effective reduce process. First, each Map process works with a different chunk of data by applying any available reduction procedure [77]. Then, selected prototypes are fed to a single reducer that is devoted to eliminate redundant ones. This implies a clear fusion of data in order to obtain a final model, in this case by merging similar examples.
- Fuzzy Rule Based Classification Systems [78] are probably one of the clearest type of models in this category. The pioneer implementation was based on the Chi et al. approach [62,63]. Each Map task comprises a complete fuzzy learning procedure to obtain an independent rule base. To do so, all examples from the input data chunk are iterated deriving a single rule per example, while merging those with the same antecedent and consequent. Then, rule weights are computed based on a fuzzy confidence value from the input examples, also allowing to determine the output class. We must acknowledge that at this stage every single rule base might be used for classification purposes. However, the system can be further enhanced by combining all partial models for every Map, as stated in the beginning of this section. To do so, the Map writes as key-value pair the antecedent and consequent (including class and rule weight) respectively. Reducers then merges those rules with the same antecedent (key) by averaging the values of the rule weights

Table 2
Categorization of distributed models for large-scale machine learning. Pseudonym and reference for each method are provided.

Method	Fusion tactic	Model scope	Model phases	Model guidance
Mahout-RF [23]	approximate	local	1-step	unguided
FRBS [62,63]	approximate	local	1-step	unguided
Boost.PL [64]	approximate	local	1-step	unguided
Spark-PR [65]	approximate	local	1-step	unguided
Spark-Trees [66]	exact	global	multistage	guided
Spark-Gradient [67]	exact	global	multistage	unguided
Spark-kMeans [68]	exact	global	multistage	guided
Spark-kNN [69]	exact	global	1-step	unguided
IFSF [70]	exact	local	multistage	guided

for the same class consequents, and then the class with the highest rule weight is finally maintained.

There are two main advantages for this type of design. On the one hand, we have stressed its direct synergy with the MR programming model, easing the programmer implementation. On the other hand, we must refer to the degree of diversity of those models obtained from different Maps, being a crucial characteristic for the success of these types of methods [79].

In spite of the previous goodness, there is one significant problem: there is a limit in the scalability degree. It must be acknowledge that there is a strong relationship between the number of Maps and the required number of models. In other words, we cannot decrease the total running time by adding a larger number of Maps, as this may result in “void” functions with a data input but no output.

One final drawback that must be stressed is related to the inner concept of ensemble. Being composed of a “large” set of individual models, two facts must be taken into account. On the one hand, since all of them are considered for the classification step, it is hard to understand the actual reason for the labeling of the queried data. On the other hand, the robustness of the whole ensemble system somehow depends on the individual quality of its components, as well as their diversity. Both properties are not easy to hold, and impose a restriction in the building of the ensemble, for both standard and Big Data applications.

Throughout this section we have presented a short overview on those MR methods that are based on fusion of partial models. As such, they allow to reinforce the abilities of those individual systems, leading to possibly more robust approaches. However, two issues must be taken into account:

1. The key-value representation to link the Map and Reduce processes. It must be very carefully selected as it has a strong influence in the design of the Map function. Although the procedure is usually implemented to be independent among Maps, a common point should be given in order to allow the final combination.
2. The fusion function in the Reducer. It has a clear dependence on the previous item, so that these must be designed as a whole to ensure a robust workflow.

Both points will determine the scalability and exactness of the output system. We refer to the changes in behavior when increasing the number of Maps. Concretely, the efforts must be mostly focused on the development of the partial models, regarding the influence of the lack of data and/or whether the knowledge acquired from different subsets of the problem space are able to be accurately merged.

4.2. Exact fusion for scalable models: distributed data and models’ partition

The previous scheme based on approximate models is presented as ideal for distributed learning since submodel creation is naturally parallelizable and applicable to most of standard algorithms. Nevertheless, in approximate fusion, models suffers from several deficiencies, such as: extra parameter configuration (e.g.: number of map tasks), or a subsequent drop on generalization due to a narrower view in submodel creation. Indeed, most of supervised methods demand to access to a large percentage of instances (usually, the whole set) during the training step. In this scenario submodel creation is not possible because of the strong dependency between data partitions. Throughout this section, we analyze those algorithms whose scheme were noted as “exact” in Table 2.

Decision trees (DT), as supervised algorithms, are an example of methods that do not naturally support partial construction in distributed processing. They are top-down algorithms that continuously evaluate splits by using statistics sketches computed on the entire dataset. In this case, all instances (and data partitions) are involved on the statistics computation at every step, except in the rare scenario where all feature values in a partition belongs to leaves nodes. Parallelization here is then considered as a much more complex task.

A possible solution to this problem is to utilize partitioned data to make decisions about how to construct a single exact model. For DTs, statistics for each split considered are collected from the datanodes, and used in the master node to decide which split is the best option for the current state of the model (guided model construction).

This scheme was originally implemented in the Parallel Learner for Assembling Numerous Ensemble Trees (PLANET) method [53]. In **PLANET**, master node controls the complete tree induction process, and launches the MR jobs that construct the nodes. It also maintains the tree model in memory, decides which split predicates will be evaluated, and eventually replicates the updated model across the nodes. Nodes to be evaluated are organized through several queues, one with nodes for MR-based evaluation and another for in-memory evaluation. Depending on the amount of instances involved in computations, nodes are pushed to one queue or the other. Here we focus on MR evaluation of nodes which is more common.

As tree construction proceeds, the master node retrieves nodes from the queue, and schedules MR jobs to evaluate splits. Inside these jobs, statistics for splits are computed in the map side by using the instances in the data partitions. Finally, the reduce side decides which split is more convenient for the tree model. Once a MR job ends, the master nodes updates the model with the nodes and the splits predicates selected, and updates the queues with new nodes at the periphery.

Notice that in this scheme, each MR job fetches the entire dataset in order to avoid determining which records are required by each node, and the extra communication that this step conveys. Instead, it performs a level-wise computation of splits so that the tree is constructed by following a breadth-first strategy. Thanks to this scheme, all nodes at a given level are expanded at the same step, and every instance is part of the input to some analyzed node.

PLANET project does not only offer an elegant solution for single tree induction, but it also extends the original idea to provide powerful exact algorithms based on bagging and boosting ensembles [80]. Concerning bagging, PLANET allows to build multiple trees by maintaining a single queue of nodes for the whole set of trees. By alternating evaluation of nodes from several trees, PLANET can build the random forest in a parallel way. During boosting, PLANET constructs weak learners sequentially as usual. Here training is only parallelized at the tree level. Residuals are easily computed since the model is sent to every MR job in full.

Based on PLANET, MLlib’ creators [28] developed two ensemble classifiers for exact fusion: one based on random forest of trees, and another based on gradient boosted trees [66]. Standard DTs are also adopted in MLlib, however, note that they are implemented as an singular case of random forest with a single tree and the full feature set. In general, all aforementioned tree-based algorithms have incorporated several optimizations [81] with respect to PLANET, intended to boost up the tree induction process. Major features introduced are describe below:

- *Efficient bootstrapping*: one of the major drawbacks in PLANET is that its does not allow bootstrapping in bagging. This fact was overcome with MLlib where each record has associated a vector that indicates the number of replicas of each instance in each tree. Note that this strategy reduces the total amount of memory used as replication is not performed.
- *Node tracking*: in order to keep simple its design, PLANET do not track the current position of instances in the tree as it evolves. Instead, every instance needs to be evaluated at every step by traversing the tree from the root. Although simpler, and less memory-consuming, this introduces an unfordable cost in time performance. MLlib solves this problem by assigning to each instance the node where it is currently stacked. It is clearly a better solution since CPU usage is an usual bottleneck for large-scale applications.

- *Enhanced group-wise training*: MLlib has improved the tree-level training scheme by extending the number of nodes to be evaluated to the memory available in the cluster for statistic aggregation. This means that several trees can evaluate complete levels of several trees at the same step, which extremely reduces the number of passes over the dataset.
- *Bin-wise computation*: Instead of directly implementing the best split computation strategy, MLlib proposes to categorize each feature value into a discrete bin. Bins are then exploited to easily compute aggregates for bins, and to calculate information gain.
- *Partition aggregation*: As bin-based categorization is known from the early stages, it is used by the learning process to reduce the number of key-value pairs to be sent across the network (less I/O usage). Each partition provides the aggregates in a single array for all the bins considered, which drastically simplify the scheme proposed by the original instance-wise map function.

Beyond tree learning, other scalable classifiers that generates exact models can be found in MLlib [67]. Non-linear classifiers such as support vector machines, logistic regression or neural networks rely on gradient descent optimization because of its great suitability for distributed computation, and thus these are implemented following a linear approach in MLlib library. All of them share the same objective function based on error minimization: $\min_{w\in\mathbb{R}^d} Q(w)$, which is solved by the aforementioned optimizer.

Without going into further details, gradient descent [80] aims at finding a local minimum of a convex function by iteratively moving towards the steepest descent, which corresponds with the negative of the derivative (gradient) of the function at a given point.

For those optimization problems whose objective function Q can be written as a sum of costs, a stochastic gradient descent (**SGD**) approach can be used. That is the case of supervised learning, where the loss and regularization parts can be decomposed in several single contributions (instance-wise) as shown below:

$$Q(w) := \lambda\, Reg(w) + \frac{1}{n}\sum_{i=1}^{n} Loss(w; x_i, y_i). \tag{1}$$

where λ represents the trade-off factor between regularization and loss, x the input vector, and y the output value.

In stochastic gradient descent, the exact result is approximated by locally computing the gradient at instance-level. In MLlib, SGD implementation computes (sub)gradients by partition (map phase) with an specific sampling fraction (mini-batches). If no sampling is applied, we get an exact gradient descent, otherwise, SGD scheme is performed at different levels. Partial results are then aggregated/sum to obtain the gradient contribution for each iteration (reduce phase).

In recent versions, the Limited-memory Broyden–Fletcher–Goldfarb–Shanno (**L-BFGS**) method [82] was introduced as another alternative for optimization in MLlib. Because of great benefits in performance, it is gaining increasing popularity in MLlib compared to mini-batch gradient descent. The L-BFGS primitive locally approximates the objective function as a quadratic by constructing the Hessian matrix underneath. The Hessian matrix is approximated by previous gradient evaluations, thus solving the vertical scalability issue present in gradient descent.

Beyond classification, some clustering algorithms have been adapted and included in MLlib. For instance, original k-Means is implemented in Spark by replicating and transforming centroids in each Spark stage. Concretely, each Map process is devoted to compute the nearest samples (from its input chunk of data) to each of the k centroids (initially set at random). The output key-value is the centroid “id” and the attributes of the nearest sample respectively. Then each Reducer recalculates the new centroids coordinates by averaging the values of those samples given as “list of values” for each key (centroid). The whole procedure is then iterated in order to refine the centroids, thus improving their robustness. It must be observed the importance of the Reduce step for the global combination of the partial information computed within each Map. k-Nearest Neighbor (kNN) classifier [69] has also some points in common with clustering, as it is based on distances among examples. In this case, the Map stage is devoted to obtain the k nearest training instances to each query input. To carry out an exact computation, the whole test set should be given as input to each Map, allowing to obtain as key-value the index of the test instance and a list of the k closest distances with their corresponding training class labels. Then, the Reducer only needs to find, among all M sets of k-nearest neighbors (with M the number of Maps) the one with lowest value to finally assign the output class. Compared to k-Means (which updates centroids), model guidance is more decoupled in k-NN where the lazy model (case-base) is distributed across workers.

The information-based feature selection framework (IFSF) proposed in [70] is an example of how submodels (importance score by feature) can be fused in data preprocessing. This multivariate algorithm measures redundancy and relevance between the input features and the output class in order to select a final reduce set of features. As a first step in IFSF, input data is vertically split to strictly separate computations between features. Then feature interactions are modeled by replicating the last selected feature and the class across the nodes so that the minimum amount of communication is performed. The map stage calculates the feature scores individually, and the reduce stage selects the best feature from the current set of unselected features. Notice that in this case, model's scope is local since input data is in column-format. This format allows Spark to compute feature scores independently in each partition. The generalization from the perspective of information theory is shown in a recent work in [83].

5. Practical study on scalability

In this section we present a brief experimental study on the scalability of different fusion proposals. We start by presenting Higgs dataset, the large-scale dataset used as reference in our experiments (Section 5.1). Then, the random forest version for Mahout-Hadoop will serve us to illustrate the most elementary fusion process: the approximate strategy based on direct fusion (Section 5.2). Finally, Section 5.3 provides outcomes for exact models, represented by Spark's k-Means and Random Forest.

Experiments have been launched in a 12-node cluster and an extra master node with the following features per node: 6 CPU cores (Intel(R) Core(TM) i7-4930K CPU at 3.40 GHz), 64GB RAM, 2TB HDD, Gigabit Ethernet network connection. Hadoop 2.6.0-cdh5.10.0 and Mahout 0.9 were installed in all the nodes. 4GB was set as maximum for memory in map and reduce processes.

5.1. Higgs boson data

Higgs boson dataset [84] has been elected to measure the performance impact of distinct fusion schemes on standard learning methods. This dataset was uploaded to the UCI repository in 2014 [85] as a result of Higgs's discovery in 2012, and the complex experiments performed at the Large Hadron Collider at CERN.

In this dataset, particle data generated by Monte Carlo simulations aim at distinguishing between a signal process generated by Higgs bosons, and a background process with the identical decay products but distinct kinematic attributes (binary problem). Although experts have confirmed its decay into two tau particles, the signals are rather small and buried in background noise.

From the total of 28 numerical features, the first 21 are kinematic feature measured by the particle sensors in the accelerator. The last features are high-level functions derived by physicists to improve the discriminate power of features. Finally, 11 millions of instances were collected for the entire dataset.

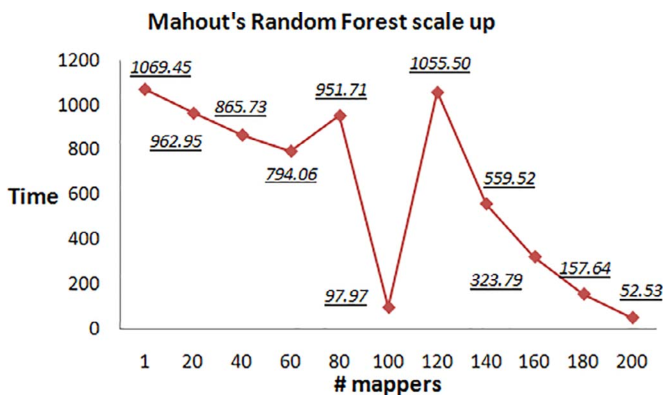

Fig. 4. Average time (3 runnings) obtained by Mahout's Random Forest (200 trees) varying the number of mappers. Label values are displayed close to the points.

5.2. Approximate models: random forest on Mahout-Hadoop

In order to test the scalability behavior of Mahout's Random Forest (200 trees), we have taken runtime measurements for different map configurations, from 1 mapper to 200. Fig. 4 depicts how time performance varies as resources augment.

The plot depicts an expected downtrend as more CPU power is available. However, some rare spikes are present around the 100-cores value. It seems that Mahout's Random Forest prefers an amount of cores divisible between the number of trees, in this case, only 100 and 200. The algorithm shows some rare deficiencies when dealing with values close to 100 (e.g.: 80 and 120) which occur during the tree construction in the map phase. This strange behavior may be explained by some design failures derived from the preliminary design implemented in Mahout's Random Forest.

Concerning accuracy, an increment of map partitions is followed by a drop on this measure as reflected in Fig. 5. This is mainly due to subtrees usually have a narrower view of input space when more map partitions are present or, in other words, the learning stage for the trees suffers from the problem of lack of data [86].

5.3. Approximate and exact fusion: k-Means and random forest on ML-Spark

As introduced in Section 3.4, Spark is mainly formed by exact implementations. Concretely, k-Means and Random Forest are two great illustrative examples in MLlib. Scalability experiments on k-Means and Random Forest have been performed in order to assess their scalability power. For these experiments, Apache Spark 1.6.2 is used as reference platform, and the Higgs dataset is previously partitioned into 216 partitions (= number of cores). Default parameter values are set for both algorithms.

Notice that scalability evaluation implemented here is slightly different than that of used in Hadoop. In contrast to manual partitioning in Hadoop, Spark allows users to specify the amount of executor cores available for each program running. This feature is much more interesting since it allows to tune resource allocation without altering the original partitioning scheme.

Fig. 6 depicts how the k-Means implementation scales-up by augmenting CPU power. From 1 to 120, the method shows an expected downtrend as more resources are provided, being the minimum stuck at 120 (9.18). The remaining values follow a smooth uptrend until 200 cores.

Fig. 7 illustrates the same study for Random Forest. The results shows a similar trend to that of the case of k-Means, leading to the same conclusions. The global minimum can also be found in 120 cores. The only noticeable difference is the lower time cost held by Random Forest compared with the clustering technique.

In both cases (kNN and Random Forest), we observe a rapid reduce of time from 1 to 20 cores, and a barely stable behavior after 20 cores. A possible explanation for this is the trade-off between the penalty associated with the distributed computation versus the increment due to the parallelization itself. This issue is mainly shown in the case of those datasets with an average size with respect to the computing capacity. Furthermore, we observe that in these experiments the absolute elapsed time is low (about ten seconds), implying a threshold for a larger efficiency improvement.

In addition to the former, we observe that no real scale-up happens from 120 cores. To understand this behavior, we must cite the Spark's authors suggestion [67] for partition tuning, which recommends to prepare 2–4× partitions per core. A partition rate below 2× implies that some CPU cores may become idle as the processing granularity

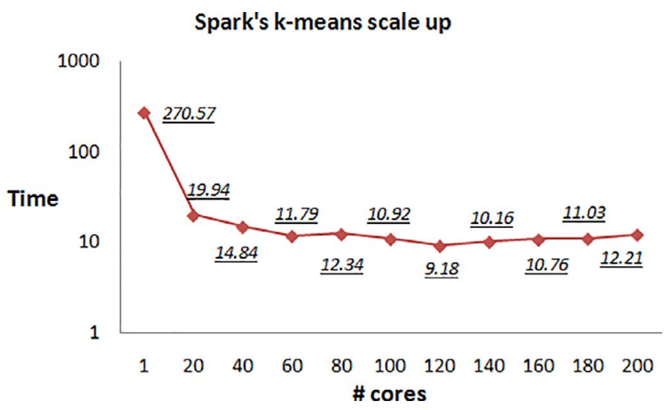

Fig. 6. Average time (3 runnings) obtained by Spark's k-Means varying the number of cores. Label values are displayed close to the points. Y-axis in 10-log scale.

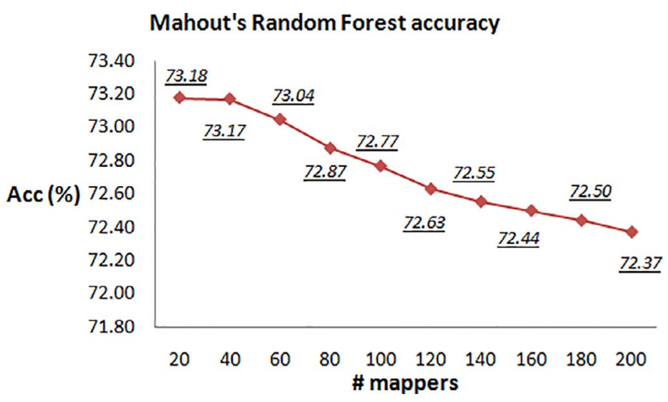

Fig. 5. Average accuracy (5-fold) obtained by Mahout's Random Forest (200 trees) varying the number of mappers. Label values are displayed close to the points.

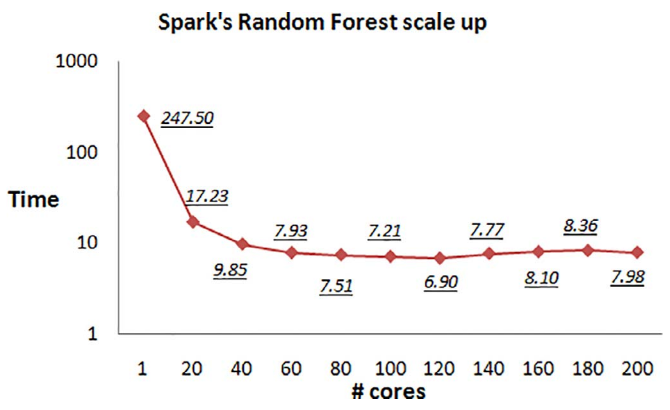

Fig. 7. Average time (3 runnings) obtained by Spark's Random Forest varying the number of cores. Label values are displayed close to the points. Y-axis in 10-log scale.

becomes small. Imagine the 1× case where the most rapid thread must wait the slowest one. A rate above 4× implies a oversized granularity where time usage is mainly dominated by startup overheads and short processes. In our study the closest value to a 2× partition rate (120-cores) is ranked as the best option.

Finally, we must point out that in this case an study on accuracy is not necessary for k-Means and Random Forest since both models provides the same predictive solution regardless of the partition scheme used. For exact fusion models, this claim is always true because all the methods implement the same intrinsic strategy, however, approximate models may vary their output depending on the division scheme followed.

6. Discussion and guidelines

The core objective in this paper was to acknowledge different paradigms and strategies implemented by modern large-scale learning algorithms. Their behavior was also analyzed into detail. To do so, we selected the level of discrepancy between the distributed models and their corresponding single-machine versions. We considered it as the most remarkable aspect to categorize the existing solutions for large-scale Machine Learning.

From this perspective, we identified two unlike groups: (1) approximative fusion of models (one submodel per partition, eventually fused), and (2) exact fusion for scalable models (compounding model with the same output as the sequential version). Other relevant aspects considered in this model categorization were their scope (local vs. global), iteration nature (1-step vs. multistage), or possible guidance by a master thread (guided vs. unguided).

Approximate models present some advantages with respect to approximate fusers such as being usually faster, especially as the number of partitions is increased. Furthermore, any existing model can be embedded into such scheme, focusing the efforts of the design into the Reduce stage. The main drawback of approximate solutions is the loss of accuracy as the number of partitions increases, since there is a clear lack of data for training. On the contrary, exact models are expected to achieve greater accuracy, yet they require more effort in their design (in order to meet the correctness condition).

Regarding these issues, we may provide some tips or suggestions in the development of distributed analytics models for Big Data. These can be considered for researchers in order to go one step further on the topic.

- The main point is to focus on the development and adoption of global and exact parallel techniques in MapReduce, Spark and/or Flink technologies. One clear example is the PLANET methodology, which allowed decision trees to have a global learning scope. This way, a robust an scalable approach that is independent on the number of processes / data partitions can be obtained.
- There is a necessity in developing more theoretical studies to facilitate the migration of current Machine Learning models towards Big Data. This way, a direct connection between a certain learning methodology and its distributed design can be established.
- Approximate models based on the traditional MapReduce scheme may still offer many interesting opportunities for research. In particular, the case of ensemble learning is of extreme importance in this scenario. The coordination among the different submodels must be a priority for a thorough learning of the problem space, and therefore to boost the performance of the final system.
- Finally, a smart use on the distributed operators for Scala is mandatory in order to implement robust and scalable solutions for the fusion process from a practical point of view.

7. Concluding remarks

In this paper, we have focused on the context of Big Data analytics and, in particular, on the design of Machine Learning algorithms following the MR programming model where the processes fusion is the core in the design. This distributed paradigm is based on parallelizing the computation among nodes, each of which is devoted to a subset of the main data. Then, the local learned models must be somehow fused in order to output a single approach.

We have proposed a taxonomy of Big Data distributed models for Machine Learning based on both the fusion tactic and the model scope, distinguishing between two main categories. On the one hand, approximate methods that carry out a direct fusion of models. On the other hand, those that provide an exact fusion of models. To obtain well founded conclusions about these different types of methodologies, we have carried out an experimental study to contrast the scalability of the different schemes. Our results have determined the higher quality of those algorithms based on exact fusion of models. In addition, we have observed the best option to a 2× partition rate for Spark-based Machine Learning implementations.

Finally, we have carried out a discussion on the main findings extracted throughout this research work. Specifically, we have analyzed with relative detail the strong and weak points for both types of the fusion models. This have allowed us to provide several guidelines for future study on the topic, namely to strengthen the development of process fusion searching for a robust design of exact parallel techniques for analytics.

Acknowledgments

This work have been partially supported by FEDER funds; the Spanish Ministry of Science and Technology under projects TIN2014-57251-P and TIN2015-68454-R; and the Foundation BBVA project 75/2016 BigDaPTOOLS.

References

[1] H. Chen, R.H.L. Chiang, V.C. Storey, Business intelligence and analytics: from big data to big impact, MIS Q. 36 (4) (2012) 1165–1188.
[2] M. Minelli, M. Chambers, A. Dhiraj, Big Data, Big Analytics: Emerging Business Intelligence and Analytic Trends for Today's Businesses, 1st ed., Wiley, 2013.
[3] A. Fernández, S. Río, V. López, A. Bawakid, M.J. del Jesus, J. Benítez, F. Herrera, Big data with cloud computing: an insight on the computing environment, mapreduce and programming framework, WIREs Data Min. Knowl. Discovery 4 (5) (2014) 380–409.
[4] C.P. Chen, C.-Y. Zhang, Data-intensive applications, challenges, techniques and technologies: a survey on big data, Inf. Sci. 275 (2014) 314–347.
[5] K. Hwang, M. Chen, Big Data Analytics for Cloud, IoT and Cognitive Computing, 1st ed., Wiley, 2017.
[6] D. Davis, BIG DATA and DATA ANALYTICS: The Beginner's Guide to Understanding the Analytical World. CreateSpace Independent Publishing Platform, 2017.
[7] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, M. Ayyash, Internet of things: a survey on enabling technologies, protocols, and applications, IEEE Commun. Surv. Tutorials 17 (4) (2015) 2347–2376.
[8] G.B. Orgaz, J.J. Jung, D. Camacho, Social big data: recent achievements and new challenges, Inf. Fus. 28 (2016) 45–59.
[9] D. Larson, V. Chang, A review and future direction of agile, business intelligence, analytics and data science, Int. J. Inf. Manage. 36 (5) (2016) 700–710.
[10] T.-M. Choi, H.K. Chan, X. Yue, Recent development in big data analytics for business operations and risk management, IEEE Trans. Cybern. 47 (1) (2017) 81–92.
[11] X. Wu, X. Zhu, G.-Q. Wu, W. Ding, Data mining with big data, Knowl. Data Eng. IEEE Trans. 26 (1) (2014) 97–107.
[12] B. Wixom, T. Ariyachandra, D. Douglas, M. Goul, B. Gupta, L. Iyer, U. Kulkarni, B.J.G. Mooney, G. Phillips-Wren, O. Turetken, The current state of business intelligence in academia: the arrival of big data, Commun. Assoc. Inf. Syst. 34 (1) (2014) 1–13.
[13] A. Abbasi, S. Sarker, R.H.L. Chiang, Big data research in information systems: toward an inclusive research agenda, J. Assoc. Inf. Syst. 17 (2) (2016) 1–32.
[14] M. Chen, Welcome to the new interdisciplinary journal combining big data and cognitive computing, Big Data Cognit. Comput. 1 (1) (2016). 1–1.
[15] A. Gandomi, M. Haider, Beyond the hype: big data concepts, methods, and analytics, Int. J. Inf. Manage. 35 (2) (2015) 137–144.
[16] H. Hu, Y. Wen, T.. Chua, X. Li, Toward scalable systems for big data analytics: a technology tutorial, IEEE Access 2 (2014) 652–687.
[17] M. Chen, S. Mao, Y. Zhang, V.C.M. Leung, Big Data - Related Technologies, Challenges and Future Prospects, Springer briefs in computer science, Springer, 2014.
[18] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters,

Commun. ACM 51 (1) (2008) 107–113.
[19] J. Dean, S. Ghemawat, MapReduce: a flexible data processing tool, Commun. ACM 53 (1) (2010) 72–77.
[20] K.H. Lee, Y.J. Lee, H. Choi, Y.D. Chung, B. Moon, Parallel data processing with mapreduce: a survey, SIGMOD Record 40 (4) (2011) 11–20.
[21] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST2010, (2010).
[22] B. Krawczyk, L.L. Minku, J. Gama, J. Stefanowski, M. Woniak, Ensemble learning for data stream analysis: a survey, Inf. Fus. 37 (2017) 132–156.
[23] S. Owen, R. Anil, T. Dunning, E. Friedman, Mahout in Action, 1st ed., Manning Publications Co., 2011.
[24] The Apache Software Foundation, Mahout, an open source project which includes scalable machine learning algorithms, (2017).
[25] D. Lyubimov, A. Palumbo, Apache Mahout: Beyond MapReduce, 1st ed., CreateSpace Independent, 2016.
[26] C. Lam, Hadoop in Action, 1st ed., Manning, 2011.
[27] T. White, Hadoop: The Definitive Guide, 4th ed., O'Reilly Media, 2015.
[28] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M.J. Franklin, R. Zadeh, M. Zaharia, A. Talwalkar, Mllib: machine learning in apache spark, J. Mach. Learn. Res. 17 (34) (2016) 1–7.
[29] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, HotCloud 2010, (2010), pp. 1–7.
[30] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing, Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, (2012). 2–2.
[31] C.-W. Tsai, C.-F. Lai, M.-C. Chiang, L.T. Yang, Data mining for internet of things: a survey, IEEE Commun. Surv. Tut. 16 (1) (2014) 77–97.
[32] D. Galpert, S. Río, F. Herrera, E. Ancede-Gallardo, A. Antunes, G. Agero-Chapin, An effective big data supervised imbalanced classification approach for ortholog detection in related yeast species, Biomed. Res. Int. 2015 (2015) 748681:1–748681:12.
[33] M. Chen, Y. Hao, K. Hwang, L. Wang, L. Wang, Disease prediction by machine learning over big data from healthcare communities. IEEE Access 5 (2017) 8869–8879.
[34] J.A. Balazs, J.D. Velásquez, Opinion mining and information fusion: a survey, Inf. Fus. 27 (2016) 95–110.
[35] S. Sun, C. Luo, J. Chen, A review of natural language processing techniques for opinion mining systems, Inf. Fus. 36 (2017) 10–25.
[36] Q. Zhang, B. Wu, J. Yang, Parallelization of ontology construction and fusion based on mapreduce, 2014 IEEE 3rd International Conference on Cloud Computing and Intelligence Systems, (2014), pp. 439–443.
[37] J. Meng, R. Li, J. Zhang, Parallel information fusion method for microarray data analysis, 2015 IEEE International Conference on Big Data (Big Data), (2015), pp. 1539–1544.
[38] S. del Río, V. López, J. Benítez, F. Herrera, On the use of mapreduce for imbalanced big data using random forest, Inf. Sci. (Ny) 285 (2014) 112–137.
[39] S. del Río, V. López, J. Benítez, F. Herrera, A mapreduce approach to address big data classification problems based on the fusion of linguistic fuzzy rules, Int. J. Comput. Intell. Syst. 8 (3) (2015) 422–437.
[40] M. Sung, SIMD parallel processing michael sung 6.911: architectures anonymous, 2000.
[41] A.J. Zaman Khan RZ, Use of DAG in distributed parallel computing, Int. J. Appl. Innov. Eng. Manage. 2 (11) (2013) 81–85.
[42] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111.
[43] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, In Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI), (2004), pp. 137–150.
[44] D. Harris, The history of Hadoop: from 4 nodes to the future of data, 2013, (https://gigaom.com/2013/03/04/the-history-of-hadoop-from-4-nodes-to-the-future-of-data/).
[45] A.S. Foundation, Apache project directory, 2017, (https://projects.apache.org/).
[46] H.D.F. System, Hadoop distributed file system, 2017, (https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html).
[47] A. Tez, Apache tez, 2017, (https://tez.apache.org/).
[48] A. YARN, Apache YARN, 2017, (https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html).
[49] A. Mahout, Apache mahout, 2017, (https://mahout.apache.org/).
[50] A. Spark, Apache spark: lightning-fast cluster computing, 2017, (http://spark.apache.org/).
[51] M. Hamstra, H. Karau, M. Zaharia, A. Konwinski, P. Wendell, Learning Spark: Lightning-Fast Big Data Analytics, O'Reilly Media, Incorporated, 2015.
[52] S. Packages, 3rd party spark packages, 2017, (https://spark-packages.org/).
[53] B. P, J.S. Herbach, S. Basu, R.J. Bayardo, G. Inc, PLANET: massively parallel learning of tree ensembles with mapreduce, PVLDB (2009) 1426–1437.
[54] A. Flink, Apache flink, 2017, (http://flink.apache.org/).
[55] Apache Flink Project, Peeking into Apache flink's engine room, 2017, (https://flink.apache.org/news/2015/03/13/peeking-into-Apache-Flinks-Engine-Room.html).
[56] S. Ewen, K. Tzoumas, M. Kaufmann, V. Markl, Spinning fast iterative data flows, PVLDB 5 (11) (2012) 1268–1279.
[57] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinlnder, M.J. Sax, S. Schelter, M. Hger, K. Tzoumas, D. Warneke, The stratosphere platform for big data analytics, Int. J. Very Large Databases 23 (6) (2014) 939–964.
[58] F. Hueske, M. Peters, M. Sax, A. Rheinlnder, R. Bergmann, A. Krettek, K. Tzoumas, Opening the black boxes in data flow optimization, PVLDB 5 (2012) 1256–1267.
[59] M. Jaggi, V. Smith, M. Takác, J. Terhorst, S. Krishnan, T. Hofmann, M.I. Jordan, Communication-efficient distributed dual coordinate ascent, CoRR abs/1409.1458 (2014).
[60] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, G. Varoquaux, API design for machine learning software: experiences from the scikit-learn project, ECML PKDD Workshop: Languages for Data Mining and Machine Learning, (2013), pp. 108–122.
[61] D. García-Gil, S. Ramírez-Gallego, S. García, F. Herrera, A comparison on scalability for batch big data processing on apache spark and apache flink, Big Data Anal. 2 (1) (2017) 1, http://dx.doi.org/10.1186/s41044-016-0020-2.
[62] S. del Río, V. López, J.M. Benítez, F. Herrera, A mapreduce approach to address big data classification problems based on the fusion of linguistic fuzzy rules. Int. J. Comput. Intell. Syst. 8 (3) (2015) 422–437.
[63] V. López, S. Río, J.M. Benítez, F. Herrera, Cost-sensitive linguistic fuzzy rule based classification systems under the MapReduce framework for imbalanced big data, Fuzzy Sets Syst. 258 (2015) 5–38.
[64] I. Palit, C.K. Reddy, Scalable and parallel boosting with mapreduce, IEEE Trans. Knowl. Data Eng. 24 (10) (2012) 1904–1916.
[65] I. Triguero, D. Peralta, J. Bacardit, S. García, F. Herrera, Mrpr: a mapreduce solution for prototype reduction in big data classification. Neurocomputing 150 (2015) 331–345.
[66] D. Blog, Random forests and boosting in MLlib, 2017, (https://databricks.com/blog/2015/01/21/random-forests-and-boosting-in-mllib.html).
[67] A. Spark, Machine learning library (MLlib) guide, 2017, (https://spark.apache.org/docs/latest/ml-guide.html).
[68] W. Zhao, H. Ma, Q. He, Parallel k-means clustering based on mapreduce, CloudCom 2009, Lecture notes in computer science 5931 (2009), pp. 674–679.
[69] J. Maillo, S. Ramírez-Gallego, I. Triguero, F. Herrera, Knn-is: an iterative spark-based design of the k-nearest neighbors classifier for big data. Knowl. Based Syst. 117 (2017) 3–15.
[70] S. Ramírez-Gallego, I. Lastra, D. Martínez-Rego, V. Bolón-Canedo, J.M. Benítez, F. Herrera, A. Alonso-Betanzos, Fast-mrmr: fast minimum redundancy maximum relevance algorithm for high-dimensional big data. Int. J. Intell. Syst. 32 (2) (2017) 134–152.
[71] R. Polikar, Ensemble based systems in decision making, IEEE Circuits Syst. Mag. 6 (3) (2006) 21–45.
[72] J. Assuncao, P. Fernandes, L. Lopes, S. Normey, Distributed stochastic aware random forests - efficient data mining for big data, Proceedings - 2013 IEEE International Congress on Big Data, BigData 2013, (2013), pp. 425–426.
[73] Y. Wang, W. Goh, L. Wong, G. Montana, Random forests on hadoop for genome-wide association studies of multivariate neuroimaging phenotypes, BMC Bioinfor. 14 (Suppl 16) (2013).
[74] L. Rokach, Decision forest: twenty years of research, Inf. Fus. 27 (2016) 111–125.
[75] R.E. Schapire, A brief introduction to boosting, IJCAI, (1999), pp. 1401–1406.
[76] R.E. Schapire, Y. Singer, Improved boosting algorithms using confidence-rated predictions, Mach. Learn. 37 (3) (1999) 297–336.
[77] I. Triguero, J. Derrac, S. García, F. Herrera, A taxonomy and experimental study on prototype generation for nearest neighbor classification. IEEE Trans. Syst. Man Cybern. Part C 42 (1) (2012) 86–100.
[78] A. Fernandez, C. Carmona, M. del Jesus, F. Herrera, A view on fuzzy systems for big data: progress and opportunities, Int. J. Comput. Intell. Systems 9 (1) (2016) 69–80.
[79] L.I. Kuncheva, Diversity in multiple classifier systems, Inf. Fus. 6 (1) (2005) 3–4.
[80] T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition, 2nd ed., Springer series in statistics, Springer, 2011.
[81] D. Blog, Scalable decision trees in MLlib, 2017, (https://databricks.com/blog/2014/09/29/scalable-decision-trees-in-mllib.html).
[82] D.C. Liu, J. Nocedal, On the limited memory BFGS method for large scale optimization, Math. Program. 45 (3) (1989) 503–528.
[83] S. Ramírez-Gallego, H.M. no Talín, D. Martínez-Rego, V. Bolón-Canedo, J. Benítez, A. Alonso-Betanzos, F. Herrera, An information theoretic feature selection framework for big data under apache spark, IEEE Trans. Syst. Man Cybern. in press (2017), http://dx.doi.org/10.1109/TSMC.2017.2670926.
[84] P. Baldi, P. Sadowski, D. Whiteson, Searching for exotic particles in high-Energy physics with deep learning, Nat. Commun. 5 (2014) 4308.
[85] M. Lichman, UCI machine learning repository, 2013.
[86] A. Fernandez, S. Rio, N. Chawla, F. Herrera, An insight into imbalanced big data classification: outcomes and challenges, Complex Intell. Syst. 3 (2) (2017) 105–120.