Aziz *et al. J Big Data* (2019) 6:78
https://doi.org/10.1186/s40537-019-0240-1

Journal of Big Data

RESEARCH Open Access

Leveraging resource management for efficient performance of Apache Spark

Khadija Aziz*†, Dounia Zaidouni† and Mostafa Bellafkih†

*Correspondence:
k.aziz@inpt.ac.ma
†Khadija Aziz, Dounia Zaidouni and Mostafa Bellafkih contributed equally to this work
STRS Laboratory, National Institute of Posts and Telecommunications, Rabat, Morocco

Abstract

Apache Spark is one of the most widely used open source processing framework for big data, it allows to process large datasets in parallel using a large number of nodes. Often, applications of this framework use resource management systems like YARN, which provide jobs a specific amount of resources for their execution. In addition, a distributed file system such as HDFS stores the data that is to be analyzed by the framework. This design allows sharing cluster resources effectively by running jobs on a single-node cluster or multi-nodes cluster infrastructure. Thus, one challenging issue is to realize effective resource management of these large cluster infrastructures in order to run distributed data analytics in an economically viable way. In this study, we use the Machine Learning library (MLlib) of Spark to implement different machine learning algorithms, then we manage the resources (CPU, memory, and Disk) in order to assess the performance of Apache Spark. In this paper, we present a review of various works that focus on resource management and data processing in Big Data platforms. Furthermore, we perform a scalability analysis using Spark. We analyze the speedup and processing time. We deduce that from a certain number of nodes in the cluster, it is no longer necessary to add additional nodes to improve the speedup and the processing Time. Then, we investigate the tuning of the resource allocation in Spark. We showed that it is not only by allocating all the available resources we get better performance but it depends on how to tune the resource allocation. We propose new managed parameters and we show that they give better total processing time than the default parameters used by Spark. Finally, we study the Persistence of Resilient Distributed Datasets (RDDs) in Spark using machine learning algorithms. We show that one storage level gives the best execution time among all tested storage levels.

Keywords: Resource management, Performance, Tuning, Distributed data processing, Machine learning algorithms, Apache Spark, MLlib

Introduction

Many applications generate and handle very large volumes of data, like social networking, cloud applications, public web sites, search engines, scientific simulations, data warehouse and so on. These large volumes of data are coming from a variety of sources and are both unstructured and structured data. In order to transform efficiently this massive data of various types into valuable information and meaningful knowledge, we need large-scale cluster infrastructures. In this context, one challenging problem is to realize an effective resource management of these large-scale cluster infrastructures in order to run distributed data analytics.

Resource management techniques are very important to manage cluster infrastructures. They allow us to share cluster resources among multiple applications. Each application can reserve multiple containers and usually each container is a reservation for a number of cores and an amount of memory. They usually follow the master-slaves architecture [1], with a central master responsible for scheduling and orchestration, while slaves host application containers.

This research is a follow-up work to our previous studies. In [2], we showed how Spark performance decreased when using large data, in fact, Spark pays for more memory consumption. Moreover, we proposed to increase memory driver, to improve performances for computing data. Therefore we used RDDs so as to optimize processing time and storage space according to our needs, this method improved performance and decreased the execution time. In [3] we focused on machine learning algorithms adapted for Big Data processing in distributed systems. We evaluated MLlib and Mahout though several experiments by increasing data up to 10 GB, in order to compare the behaviors of MLlib and Mahout according to data size on three different algorithms. As obtained in this study, MLlib is much faster than Mahout but we noted that when data is extremely large, the memory is not enough to store newly results, then, Spark's MLlib crashes. In this study, we deal with the problem of resource management using the framework Apache Spark [4]. We chose this framework because it is the most powerful open source project in Big Data with more than 1200 developers that have contributed to Spark [5]. It is built to perform sophisticated analysis and it is designed as a computing platform for fast processing, general-purpose, and easy to use [6]. It allows to process a huge quantity of data coming from several sources. Apache Spark extends the MapReduce [7] paradigm and uses it on other levels [8]. It makes very fast processing compared to traditional engines because it supports in-memory computing across DAG (directed acyclic graph) [9]. Apache Spark relies on Hadoop ecosystem (MapReduce, YARN and HDFS). MapReduce has been useful and it began as a general batch processing system [10], but in most cases, it takes a long time to run jobs especially when it comes to processing huge quantities of data [11, 12]. The learning curve to writing a MapReduce job is also difficult as it takes specific programming knowledge and the know-how. MapReduce is not designed for iterative processes [13].

In this paper, we perform simulations using different number of nodes in the cluster and different data size. We analyzed the speedup and the processing time in order to assess the scalability of Spark. Then, we study the tuning of resource allocation in Spark. We present some recommendations for tuning and for a specific configuration, we propose new managed parameters and we show that these managed parameters give a better total processing time than the default parameters. Finally, we studied the RDDs' Persistence in Spark. We discuss the types of persistence in spark and we explain the different storage levels of RDDs' persistence. We conduct several simulations using different machine learning algorithms and using different storage levels.

The rest of the paper is structured as follows: the second section presents related work. The third section gives a brief overview of Apache Spark and MLlib. While the fourth section, details our experimental setup and the fifth section explain our experimental evaluation and results. Our conclusions and future works are drawn in the final section.

Related work

In this section, we present a review of literature and we show some works that have focused on resource management and data processing in Big Data plateforms.

Resource management mechanisms

Starfish [14] is a self-tuning system for big data analytics, it is built on top Hadoop in order to provide a good performance automatically, the authors proposed a cost-based optimization method to provide automatic configuration for users, they used what-if engine to estimate job performance with different parameters and configuration without executing the job. In [15], the authors proposed a new approach for tuning configuration parameters of spark based on machine learning, this method is composed of binary classification and multi-classification in order to tune the configuration parameters automatically, they chose a random sample from the parameter space, then they generated a parameter list of 500 records for each type of workload, the experimental results showed that a Decision Tree (C5.0) provides good accuracy and computational efficiency. In [16], the researchers introduced a system architecture that gathers several mechanisms for adaptive resource management, they stretched out Apache Hadoop's scheduling and data placement to enhance resource usage and job runtime for periodic jobs, then they developed a Hadoop submission tool that allows users to allocate resources for specific target runtime.

Apache Spark and in-memory processing

Since its appearance in [4], Apache Spark quickly becomes the most popular complement of Apache Hadoop [17]. In [18], the authors described the weaknesses of MapReduce which are related to its performance limits. The authors identified a list of problems related to the processing of Big Data with MapReduce, for example: MapReduce consumes very high communication, it makes a selective access to input data, and it is wasteful processing. Despite the success that has had MapReduce, it remains always limited to analyze a huge amount of data. Gu et al. [19] evaluated the performance of Hadoop and Apache Spark. In their study, they show that Spark is very consuming memory, and Spark is efficient than Hadoop when there is enough memory to do an iterative treatment. On the other hand, if there is not enough memory, Hadoop is better than Spark to store the intermediate results that are recently created. Singh and Reddy [13] talked about the size of data to be processed. They said that Spark and Hadoop can analyze a large amount of data, but Hadoop remains too slow for iterative tasks, so if users need to optimize cluster performance, Spark is more appropriate in this case. Spark keeps data in memory while computing what makes it 40 times faster than Hadoop and it can be used interactively to query large datasets at sub-second latency [20].

Data processing in Apache Spark

The authors [21] treated a distributed Newton method for solving logistic regression (LR) and linear support vector machines (SVM) then they implemented these methods on Spark instead of MapReduce because this last one is insufficient on iterative

algorithms; Moreover, they studied several implementation problems that affect the running time. After conducting through empirical investigations, they proved that the distributed Newton method was powerful for LR and linear SVM with fault tolerance provided by Spark. The authors [22] implemented Java RMI (remote method invocation) technique to let users to leverage Local GPU (graphics processing unit), Remote GPU and no GPU. Moreover, they evaluated the performances of HeteroSpark [22] using three machine learning algorithms including logistic regression, K-means, and Word2Vec, and they compared it with the original Spark, In fact the results showed that HeteroSpark is the most efficient. Maillo et al. [23] presented a new K-nearest neighbor (KNN) based on Apache Spark, they implemented multiple 'Reduces' to speed up the computing. Furthermore, they compared Apache Spark and Apache Hadoop using datasets up to 10 million instances to show the advantages of the new KNN based on Apache Spark.

Performance evaluation of Apache Spark through machine learning techniques

Smart-MLlib [24] is a new Machine Learning library, it is build on c++ and MPI (message passing interface) based on middle-ware system,, it allows to implement machine learning from a Scala program using application programming interface (API) of Spark. To assess the performances of Smart-MLlib, the authors used different machine learning algorithms such as linear regression, K-means, support vector machines, and Gaussian mixture models. As a result of this implementation, they proved that their new Smart-MLlib library scaled well than Spark's MLlib for each evaluation. Applying machine learning on a large and complex dataset requires a considerable number of physical resources to process this data, in [25], the authors explored Apache Spark MLlib version 2.0 as an open-source, distributed, scalable, and platform independent Machine Learning library, and they performed different real-world machine learning experiments to evaluate the qualitative and quantitative attributes of the platform. Alternating direction method of multipliers (ADMM) [26], it is a method used to solve a generic convex problem for most machine learning algorithms, this solution helps to transform the problem to an iterative system of linear equations, the authors implemented ADMM in Apache Spark and they compared this solution with MLlib then they showed that ADMM solution is like an alternative to MLlib for big-data problems, this approach has the added advantage of machine learning algorithms.

Our work differs from previous studies in the literature. In this study, we used different techniques to study and improve the performance of Big Data processing. In general, users try to improve performance by increasing the memory, the number of cores, and the number of nodes randomly to get better performances, but sometimes this technique could mitigate them, therefore it is necessary to find optimal resource management to improve performances. So for this paper, we rated the scalability using various data size and number of nodes, and we evaluated MLlib through several experiments by choosing different datasets to examine the behavior of MLlib and study the processing time using machine learning algorithms. Then, we allocated to executors all the available cores on each node to improve the computation time, but this method mitigated the performance instead of improving them, to remedy this problem, we proposed to adjust the parameters in an appropriate manner. In consequence, we suggest that performance

improvement has to be done in a more informed way about the effects of each parameter including memory, number of executors, and cores to get better performance. Finally, we used RDD persistence to decrease the computation time and make the whole cluster more efficient and fast.

Background

In this section, we give a brief overview of Apache Spark and MLlib.

Apache Spark

Apache Spark is an open source Big Data processing framework, it is designed for fast computing and easy to use. It is based on MapReduce paradigm and it uses it to a whole other level. Spark runs on Hadoop clusters such as Apache YARN [27], Apache Mesos [28], and standalone [29] with its own scheduler. Spark provides parallel distributed processing, fault tolerance on commodity hardware, and scalability [30]. Spark adds to the concept with cached in-memory distributed computing, low latency, high-level APIs and stack of high-level tools. MapReduce was designed as a fault tolerant framework that ran on commodity systems, Spark comes with the same framework to run data processing on commodity systems as well as using a fault tolerant framework. Apache Spark provides API (application programming interface) in different programming languages including Scala, Java, Python and R [31]. This framework permit to write the concept of data transformations and machine learning algorithms using the parallelism technique. It is also possible to write computations that are fast for different distributed storage systems for example HBase and HDFS.

MapReduce

In 2004, the MapReduce paradigm was invented by Google for large data processing, this model allows parallel and distributed computing of huge quantity of data through a cluster. In fact, it aims to process data by partitioning and aggregation of intermediate results. The basic idea of the MapReduce paradigm is to process data in parallel, while data splitting, distribution, synchronization and fault tolerance are handled automatically by the MapReduce framework. MapReduce has main functions: map() to split out and distribute the different data partitions, it is done by a node that distributes the data to other nodes; and reduce() to collect the computed results in parallel and merge them into a single global result.

Resilient Distributed Dataset

Resilient Distributed Dataset [32, 33] or RDD is the main component in Apache Spark, it is a distributed collection of elements parallelized into the cluster. The most instructions of processing data consist of performing operations in RDDs. There are two types of RDD operations. Transformations and Actions. Transformations do not return any value and create a new RDD based on an existing one, nothing is evaluated during the transformation statements. When actions are called, they return a value from an RDD.

In the midst of RDDs' strengths, it recovers lost data after each failure. Overall, data is partitioned over worker nodes. Partitioning is done automatically three times by Spark, however it is possible to control the number of partitions that can be created.

Spark architecture and processing data

Apache Spark runs applications independently through its architecture [34] in the cluster as shown in Fig. 1, these applications are combined by SparkContext Driver program, then Spark connects to several types of Cluster Managers (such as YARN, Mesos) to allocate resources between applications to run on a Cluster, when it is connected, Spark acquires executors on the Cluster's Worker Nodes, to perform calculations and store data for the application, besides Spark sends the application code passed to SparkContext to the executors, finally, SparkContext sends tasks to executors to execute.

Apache Spark processes data through several steps. firstly, a RDD is created by parallelizing the data in the driver program or by loading the data from the external storage system such as HDFS or HBase; then, the results of RDDs are saved to be apply to the data. Once a new action is called, the whole RDD must be recalculated, intermediate results are saved in memory; finally, the output is returned to the driver.

MLlib of Apache Spark

MLlib (Machine Learning library) is distributed machine learning library of Spark [35]. MLlib aims to large-scale learning settings that benefit from data-parallelism or model-parallelism to store and operate on data or models [36]. It gathers parallel algorithms that run better on clusters. There are other classic machine learning algorithms that are not integrated because they were not designed for parallel processing [37].

Apache Spark MLlib aims to make machine learning scalable and useful, it includes different tools for example [38]: machine learning algorithms (including Classification, Clustering, and Collaborative Filtering). Pipelines (for evaluating and tuning ML

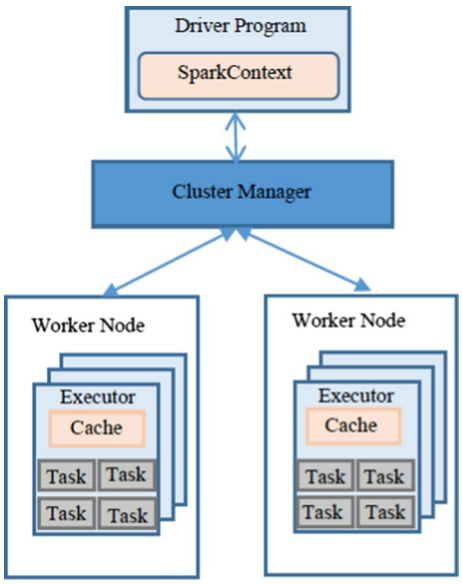

Fig. 1 Apache Spark architecture. Apache Spark runs applications independently through its architecture in the cluster, these applications are combined by SparkContext Driver program, then Spark connects to several types of Cluster Managers to allocate resources between applications to run on a Cluster, when it is connected, Spark acquires executors on the cluster nodes, to perform calculations and store data for the application, besides Spark sends the application code passed to SparkContext to the executors, then, SparkContext sends tasks to executors in order to be executed

Pipelines), Persistence (for saving and loading algorithms and pipelines) and Utilities (for data management and optimization algorithms).

Experimental setup

Methodology

After the installation of Apache spark, we save input data in HDFS (Hadoop distributed file system), since Spark can read from any Hadoop input. In this study, we chose different data size to perform our simulations using a different number of nodes in the cluster and different data size. Then, we use the Machine Learning library (MLlib) of Spark to implement different machine learning algorithms including K-means, GMM, LR, and SVM. We perform a scalability analysis using Spark. We analyze the speedup and processing time. Besides, We investigate the tuning of the resource allocation in Spark. We show that it is not only by allocating all the available resources we get better performance but it depends on how to tune the resource allocation. We propose new managed parameters and we show that these managed parameters give a better total processing time than the default parameters used by Spark. Then we manage the resources (CPU, memory, and Disk) in order to assess the performance of Apache Spark. Moreover, we study the RDDs' Persistence in Spark using machine learning algorithms. After each processing experimental, spark saves the results in HDFS. Figure 2 summarizes the methodology of our approach.

Description of machine learning algorithms

This section describes in detail the four algorithms used in our experimental study.

K-means

K-means clustering algorithm is an iterative algorithm and it is an example of unsupervised machine learning problem. It aims to classify dataset using several clusters or K clusters fixed before starting the procedure of clustering. The k-means algorithm aims to minimize the objective function given by Eq. (1):

$$J = \sum_{k=1}^{K} \sum_{i=1}^{m} \|x_i - c_k\|^2 \tag{1}$$

- K: The number of clusters.
- m: The number of data examples.
- x_i: The data example i.
- c_k: The centroid for cluster k.
- $\|x_i - c_k\|^2$: The distance function.

K-means algorithm proceeds as follows: (1) Select K random examples as starting centers, (2) Find all examples closest to each center, (3) Find the center or the mean of each cluster, (4) If the center changed, iterate again (repeat the steps (2) and (3) until to find the closest center).

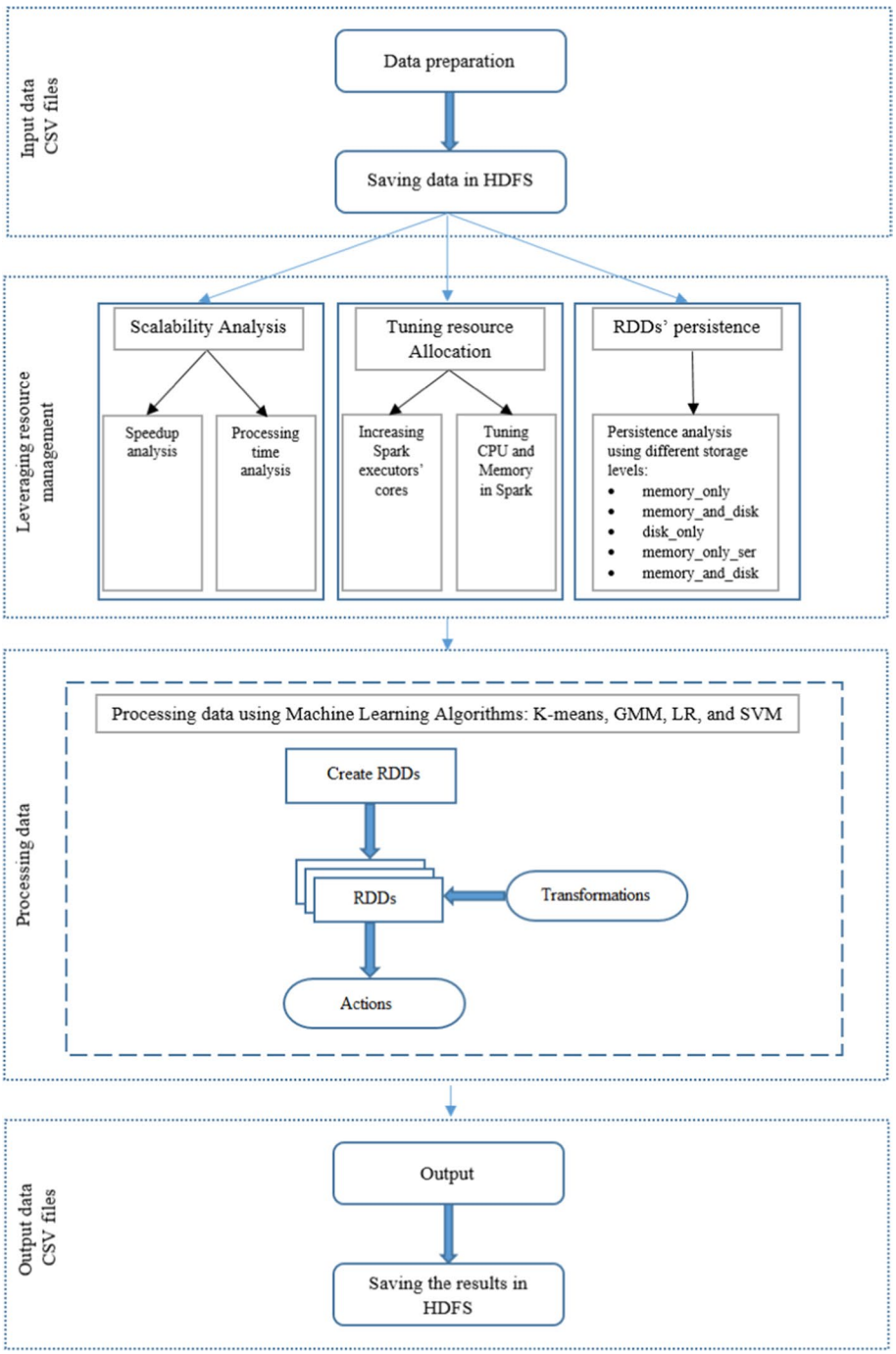

Fig. 2 Flow chart of our methodology. This figure represents the flow chart to show the alignment of all steps of this study

Algorithm 1 Scala source code that uses K-means Algorithm

```
val input = sc.textFile(input.csv)      //load data
val pInput= input.map(d ⇒ vectors.dense(d.split(' ').map(_.toDouble))).cache()      //parse data

val c = 2      //two classes
val i= 16      //number of iterations
val cluster = KMeans.train(pInput,c,i)

cluster.save(sc,'output/kmeansModel')      //save the model
val model = KMeansModel.load(sc,'output/kmeansModel')      //load the model
```

Gaussian mixture model

Gaussian mixture model (GMM) is a linear combination of several Gaussian components and it is an example of unsupervised machine learning problem. It is particularly used in cases where the data in studies cannot be modeled by a simple Gaussian. In other words, if the data structure is composed of several groups, it must be represented by a Gaussian mixture model rather than a simple Gaussian distribution.The model is constructed by the following Eq. (2):

$$g(x, \Phi) = \sum_{k=1}^{K} P_k f(x, \theta_k) \tag{2}$$

- K: The number of Gaussians we want to model.
- $\theta_k = (\mu_k, \sum_k)$: The parameter of each normal distribution.
- $P_1, \ldots, P_K$: The proportions of different groups or the mixture weight such that: $\sum_{k=1}^{K} P_k = 1$.
- $\Phi = (P_1, \ldots, P_K, \theta_1, \ldots, \theta_K)$: It is mixture parameter.
- $f(x, \theta_K)$: It is the normal multi-dimension distribution parameterized by θ_K.

GMM algorithm is done using the following steps: (1) Select the initial K Gaussians, (2) Determine the responsibility of each Gaussian to every data point, (3) Update the Gaussian weights, means, and covariance matrices, (4) iterate again (repeat the steps (2) and (3) until convergence).

Algorithm 2 Scala source code that uses GMM Algorithm

```
val input = sc.textFile(input.csv)
val pInput = data.map(s ⇒ Vectors.dense(d.trim.split(' ').map(_.toDouble))).cache()

val gmm = new GaussianMixture().setK(2).run(pInput)

gmm.save(sc, "output/gmmModel")
val model = GaussianMixtureModel.load(sc,"output/gmmModel")
```

Logistic regression

Logistic regression (LR) is an iterative algorithm for binary classification and it is an example of supervised learning problem. LR attributes observations to a collection of categories. It is a special case of Generalized Linear models that predicts the probability of the outcomes. Logistic regression is based on the following main hypothesis (3):

$$f(x) = \sigma(W_T x) \tag{3}$$

- σ: The logistic function $\sigma(z) = \dfrac{1}{1+\exp(-z)}$.
- W: The weights vector.
- x: The data example.

LR is done using the following iterative process: (1) Choose W at random, (2) Compute the gradient $\frac{\sigma}{W}$, (3) Update each W_j by adding a function of W so as to improve its value, (4) Repeat step (2) and (3) until to find the best value of W.

Algorithm 3 Scala source code that uses Logistic Regression Algorithm

```
val input = MLUtils.loadLibSVMFile(sc, "input.csv")

val split = data.randomSplit(Array(0.7, 0.3), seed = 1234) //70%for train and 30% for test
val train = split(0).cache()
val test = split(1)

val model = new LogisticRegressionWithLBFGS().setNumClasses(10).run(train)

model.save(sc, "outputLRModel")
val sameModel = LogisticRegressionModel.load(sc,"outputLRModel")
```

Support vector machine

Support vector machine (SVM) is a supervised machine learning algorithm, it is used for classification problems and it can sometimes be used for regression problems. SVM algorithm aims to find a hyper plane in N-dimensional space (N is the number of features) so as to classify the data points.

To have an idea about the mathematical formula of SVM, we take the example inspired by Schôlkof [39] who puts the principle of SVM into practice. In this example, the nonlinearly separable data in R^2 become linearly separable in R^3 due to the transformation ψ defined by the Eq. (4):

$$\begin{aligned} &\psi : R^2 \Rightarrow R^3 \\ &(x_1, x_2) \Rightarrow (x_1^2, \sqrt{2}x_1x_2, x_2^2) \end{aligned} \tag{4}$$

SVM algorithm works as follows: (1) Select the initial weights; (2) For each point, accumulate the partial derivatives of the hinge-loss function; (3) Update the weights with applying the gradient-descent rules by taking into consideration the partial derivatives found in the step (2), the number of points and the iteration number; (4) iterate again (repeat the steps (2) and (3) until convergence).

Algorithm 4 Scala source code that uses SVM Algorithm

```
val input = MLUtils.loadLibSVMFile(sc, "input.csv")

val split = input.randomSplit(Array(0.7, 0.3), seed = 1234) //70%for train and 30% for test
val training = split(0).cache()
val test = split(1)

val model = SVM.train(training, 50)

model.save(sc, "output/svmModel")
val sameModel = SVMModel.load(sc, "output/svmModel")
```

Datasets chosen in the experimental study

The datasets we investigated are the HIGGS dataset and SUSY dataset found at the UCI Machine Learning Repository [40].

Higgs dataset contains 28 features and 11,000,000 examples that describe the difference between a signal process which produces Higgs bosons and a background process which does not [41]. SUSY datasets contains 18 features and 5,000,000 examples that describe the difference between a signal process which produces supersymmetric particles and a background process which does not [41].

Environment

The experimental studies are performed on Google Cloud Platform (GCP) which is a cloud computing platform provided by Google, it provides the same infrastructure that Google uses internally. This platform offers a wide range of products and services. In this work, we used Cloud Dataproc service [42], which is fully managed for the use of Apache Spark and Apache Hadoop clusters.

We chose several clusters to perform various experimental studies, each cluster has different environment, Table 1 shows the characteristics of each environment, every cluster has different memory and the number of cores, for primary disk, we used 100GB size and SSD persistent disk type.

In the cluster, we have Apache Spark version 2.3 and Apache Hadoop version 2.9. The Master node is equipped with the YARN Resource Manager, HDFS NameNode, and all job drivers. Each Worker node contains a YARN NodeManager and a HDFS DataNode. In this study, we use HDFS of Apache Hadoop to save the input data and the results. Figure 5 shows the cluster and Table 2 summarizes all the information about software configuration.

Results and discussion

Scalability analysis

Analysis of the speedup

In this part, we rated the scalability of four algorithms. We set the dataset size to 1 GB, 2 GB, 4 GB and 6 GB in CSV format; at the same time, we increased the number of nodes, all the nodes have 15 GB in memory per node and 4 cores per node.

The speedup is one of the main performance characteristics of distributed systems. It is usually defined as the ratio of application's processing time on a single node to the processing time, of the same workload, on a system composed on n nodes. The notion of speedup was established by Amdahl's law, which was particularly focused on parallel processing. The speedup can be used more generally to show the effect on performance after any resource enhancement. The speedup is calculated as follows:

Table 1 Environments

Environment	Memory (GB)	Cores per node	Nodes
Env1	2	2	4
Env2	10	4	8
Env3	15	4	12

Table 2 Software configuration

Software name	Version
Spark	2.3
Virtualization platform	KVM
OS	Debian GNU/Linux 9
Kernel	Linux 4.9.0-8-amd64
Architecture	x86-64

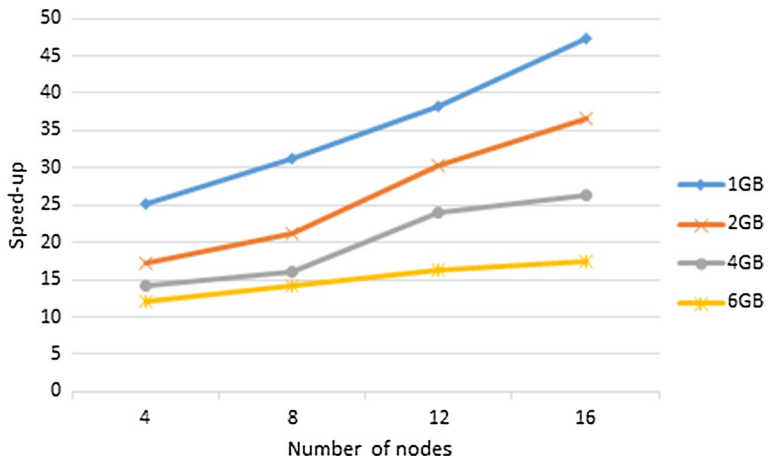

Fig. 3 Scalability of GMM. This figure presents the scalability characteristics using GMM algorithm and different number of nodes (4, 8, 12 and 16) and different data size (1 GB, 2 GB, 4 GB and 6 GB)

$$speedup = T_1/T_n,$$

T_1: is the processing time with one single node, and T_n: is the execution time with n nodes.

Figures 3, 4, 5 and 6 present the scalability characteristics using different number of nodes (4, 8, 12 and 16) and different data size (1 GB, 2 GB, 4 GB and 6 GB). All figures for the four algorithms show that for small size of dataset (specially 1 GB size), we can see an almost linear speedup improvement with the addition of the new nodes in the cluster. However, for Figs. 3 and 6, for large size of dataset (specially 6 GB size), we note that the speedup curve becomes asymptotic, the speedup does not improve linearly with the addition of nodes.

So, from this scalability analysis, we can deduce that from a certain number of nodes in the cluster, it is no longer necessary to add additional nodes to improve the speedup. Thus, we must investigate other techniques in order to enhance the speedup and to improve the execution performance of Spark applications.

Analysis of the processing time

In this part, we analyze the processing time using different environments (Env1, Env2 and Env3) and using different datasets (HIGGS and SUSY) described above in CSV format. These environments have the same characteristics that are described in Table 2.

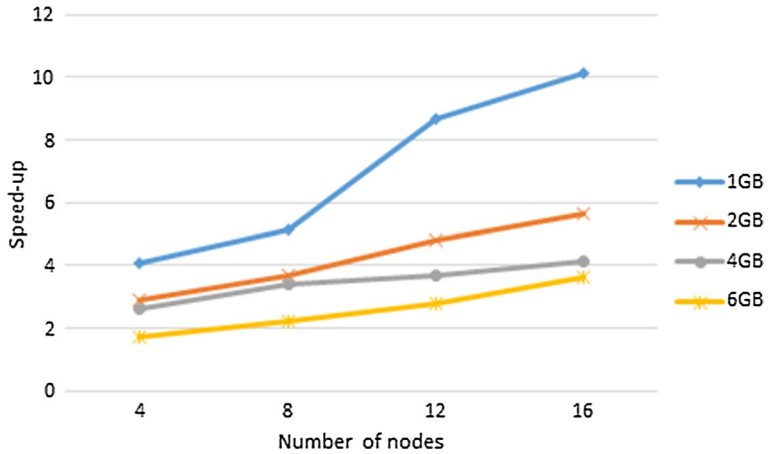

Fig. 4 Scalability of K-means. This figure presents the scalability characteristics using K-means algorithm and different number of nodes (4, 8, 12 and 16) and different data size (1 GB, 2 GB, 4 GB and 6 GB)

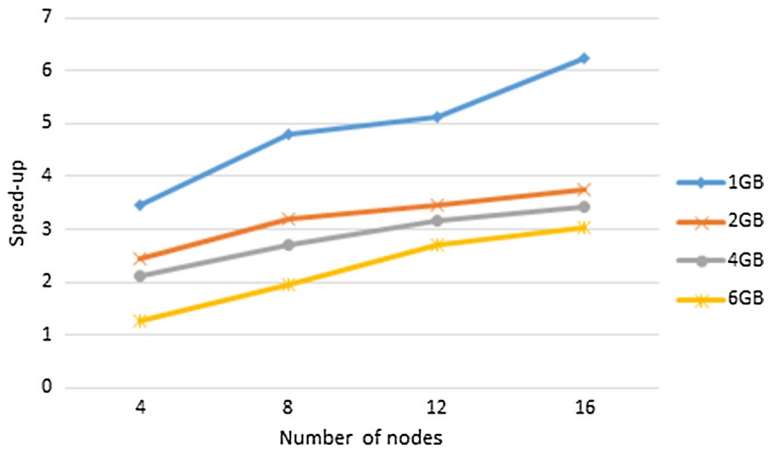

Fig. 5 Scalability of LR. This figure presents the scalability characteristics using LR algorithm and different number of nodes (4, 8, 12 and 16) and different data size (1 GB, 2 GB, 4 GB and 6 GB)

The trends in Figs. 7, 8, 9, and 10 are clear. All figures show that when we use HIGGs dataset (the biggest dataset) the total processing time of spark is slower than when we use SUSY dataset (the smallest dataset). The figures also show that the total processing time decreased when increasing the number of nodes. We also noted that for all tested algorithms except GMM algorithm, the difference in total processing time is not significant between Env2 (8 nodes) and Env3 (12 nodes). Thus, an important conclusion could be drawn from these simulations which is the fact that increasing the number of nodes will obviously contribute to reduce the total processing time but from a certain number of nodes, the value of the execution time will stagnate and the total processing time will not continue to decrease. This remark leads us to study (in the following section of this paper) the importance of tuning resource allocation in order to run data in an economically viable way. Another important remark in Fig. 7 is that when we use GMM algorithm using Env1

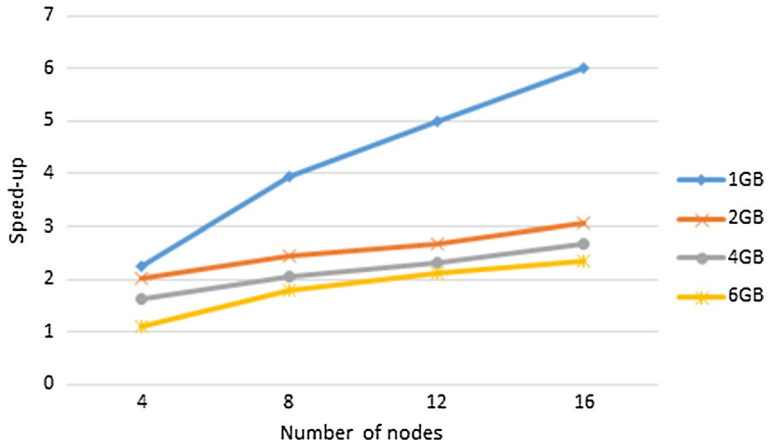

Fig. 6 Scalability of SVM. This figure presents the scalability characteristics using SVM algorithm and different number of nodes (4, 8, 12 and 16) and different data size (1 GB, 2 GB, 4 GB and 6 GB)

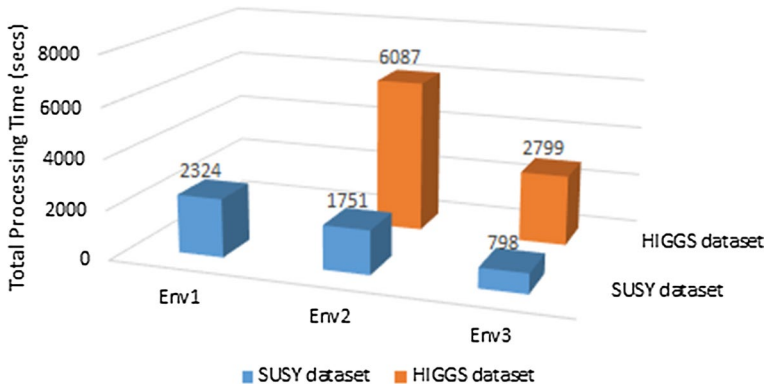

Fig. 7 Running time of GMM algorithm. This figure shows the running time of GMM algorithm using two datasets, when we use HIGGs dataset (the biggest dataset) the total processing time of spark is slower than when we use SUSY dataset (the smallest dataset)

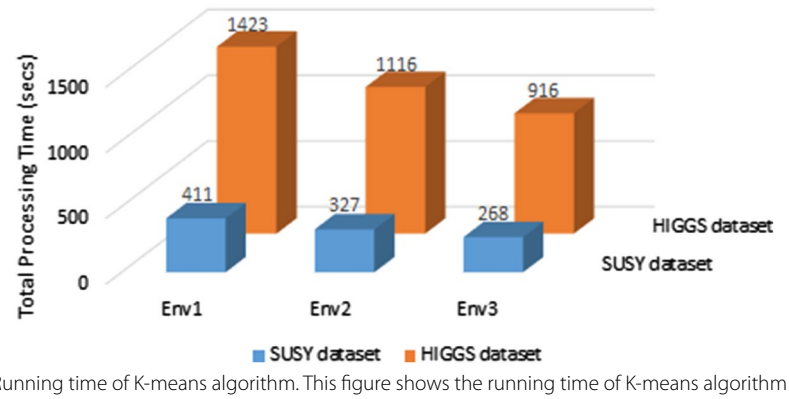

Fig. 8 Running time of K-means algorithm. This figure shows the running time of K-means algorithm using two datasets, when we use HIGGs dataset (the biggest dataset) the total processing time of spark is slower than when we use SUSY dataset (the smallest dataset)

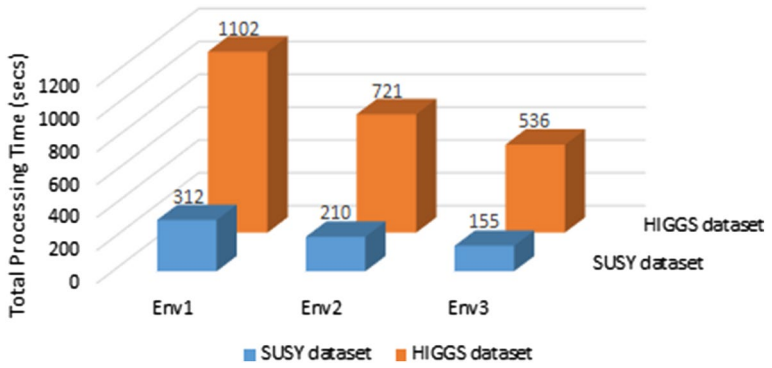

Fig. 9 Running time of LR algorithm. This figure shows the running time of LR algorithm using two datasets, when we use HIGGs dataset (the biggest dataset) the total processing time of spark is slower than when we use SUSY dataset (the smallest dataset)

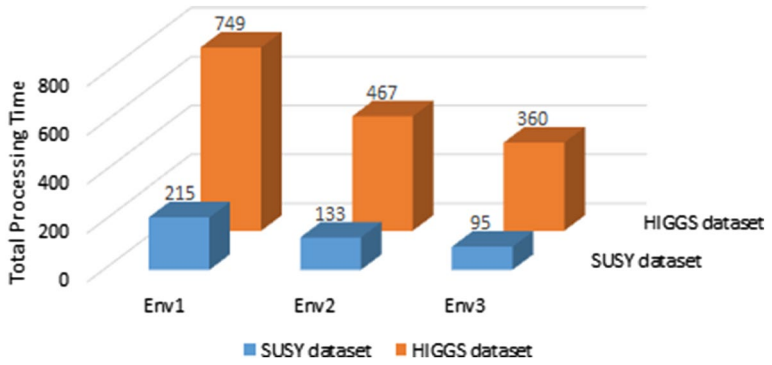

Fig. 10 Running time of SVM algorithm. This figure shows the running time of SVM algorithm using two datasets, when we use HIGGs dataset (the biggest dataset) the total processing time of spark is slower than when we use SUSY dataset (the smallest dataset)

and using the biggest dataset (HIGGS), Spark becomes too slow and it crashes. We can explain that by the fact that Spark involves in-memory based storage for RDDs, therefore, the Spark's memory crashed because there is not enough space to save the intermediate results.

Tuning resource allocation

Increasing Spark executors' cores

Each Spark executor when submitting an application has the same fixed default value of cores (one single core). It is possible to change this value with the flag '- -executor-cores' when invoking 'spark-submit' or 'spark-shell' from the command line, or even by adjusting 'spark.executor.cores' property in 'spark-defaults.conf' file or on a 'Spark-Conf object'.

The number of cores controls the number of tasks that an executor can run at the same time. For example, set '– –executor-cores 6' signifies that each executor will run a maximum of six tasks simultaneously. To some extent, when we increase the number of

cores, we will get better performances as a result of more parallelism and more tasks that will be executed simultaneously.

In this section, we allocate to executors all the available cores. In our simulation, we use Higgs Dataset in CSV format and environment 3 (described above) with 12 nodes and each node has 4 cores. In this experiment, we consider Fat executors which means one executor per node and we allocate the four available cores for each executor.

Figure 11 shows the results of this approach. We tried to improve the performances, but instead of that, the performances have been mitigated. The total processing time has greatly increased for all tested algorithms. We can conclude that the performance deteriorates because of the following reasons:

- The ApplicationMaster, which is a non-executor container, requires CPU that must be accounted for.
- When we run spark application, there will be several daemons that will run in the background (Hadoop and YARN daemons). So, we must leave at least 1 core per node for these daemons.
- When the number of cores is too high, the spark application spends more time and this is because of switching between threads, consequently, it required more computing and eventually more processing time.
- The HDFS client has difficulty processing many concurrent threads. Increasing number of cores per executor can lead to bad HDFS I/O throughput.

So, the result of this experiment is that we must avoid allocating 100% of available cores to executors and we must leave CPU resources for the ApplicationMaster and for Hadoop and YARN daemons.

Some recommendations for tuning resource allocation

- The ApplicationMaster would need at least a memory $\geq$ 1 GB and 1 Executor [43].

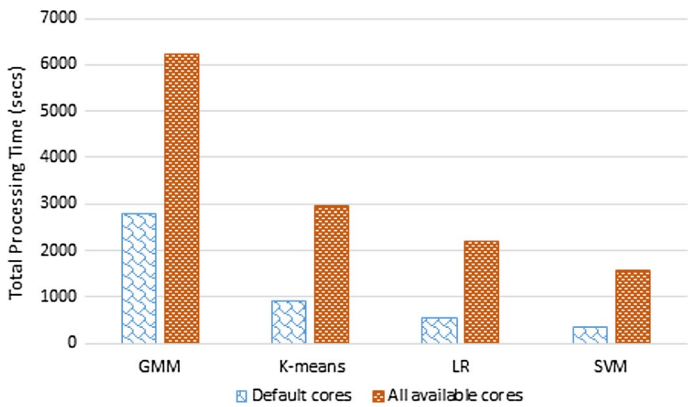

Fig. 11 Processing time before and after increasing the number of cores. This figure shows the processing time before and after increasing the number of cores, in this step we tried to improve the performances by allocating all the available cores, but instead of that, the performances have been mitigated. The total processing time has greatly increased for all tested algorithms

- In case of using YARN for cluster manager, it is necessary to leave at least 1 core per node for YARN's daemons [43].
- it is better to use 64 GB as an upper limit for one executor because when using a big memory for executors could produce garbage collection delays [43].
- For HDFS client, it has trouble processing several concurrent threads. In preference, one must keep the number of cores per executor below than five [43].

Tuning CPU and memory in Spark

The two important resources that Spark manages are CPU and memory. Disk and network I/O also affect Spark performance as well, but Apache Spark does not manage efficiently these resources. It is important to equilibrate the use of RAM, number of cores, and other parameters so that processing is not strained by any one of these cluster resources.

In Table 3, we mentioned the default values configured automatically by Spark, such as:

- – –driver-memory: the memory needed by the ApplicationMaster to run the driver, it is set by default to 1 GB.
- – –num-executors: the number of executors per node. It is set by default to 'allocated dynamically', which means Spark uses dynamic allocation to adds and removes executors dynamically.
- – –executor-memory: the memory assigned to each executor. It is set by default to 1 GB.
- – –executor-cores: the number of cores per executor (which control the concurrent tasks that an executor can run) is set by default to 1.

In order to reduce processing time and improve Spark's performance, we proposed to tune these parameters taking into account the above recommendation using Higgs dataset.

We consider the following configuration cluster: **12** nodes, **5** cores per node and **15** GB RAM per node.

- First of all, we leave 1 core and 4 GB per node for Hadoop and YARN daemons, So:
  - Num cores available per node $= 5-1 = 4$

Table 3 Tuning Spark parameters

Property name	Default value	Configured value
– –driver-memory	1 GB	4 GB
– –num-executors	allocated dynamically	2
– –executor-memory	1 GB	5 GB
– –executor-cores	1	2

  - Available RAM per node = 15−4 = 11 GB

- The total available cores in the cluster is $4 \times 12 = 48$ cores.
- In order to avoid using tiny executors (as seen in the above recommendations), we chose 2 cores per executor. So:
  - Number of available executors = (total cores/num-cores-per-executor) = 48/2 = 24 executors.
- We leave 1 executor for ApplicationMaster, So:
  – –num-executors = 23 executors.
- Number of executors per node: $23/12 \simeq 2$
  So, we have 2 executors on all nodes except one node which has one executor because we leave one executor for the ApplicationMaster.
- Memory per executor = 11 GB/2 = 5.5 GB
- We need to count the spark.yarn.executor.memoryOverhead which is the memory requested to YARN for each executor, it is calculated by:

  $$MemoryOverhead = Max(384\,MB, Memory\_per\_executor \times 7\%)$$
  $$MemoryOverhead = Max(384\,\text{MB}, 5.5\,\text{GB} \times 7\%) = 0.385\,\text{GB}$$

- Finally, the optimal memory for each executor:
  – –executor-memory = $(Memory\_per\_executor - MemoryOverhead)$
  – –executor-memory = $5.5 - 0.385 \simeq 5$ GB.

Resource allocation is a crucial technique while submitting any spark application. Figure 12 shows that the performance has been improved after tuning on some parameters (Table 3). In this analysis, we highlighted the impact of tuning resource allocation, because not only by allocating all the available resources we will get better performance, but it also depends on how to tune the parameters like – –driver-memory, – –num-executors, – –executor-cores and – –executor-memory.

RDDs' persistence in Spark

Persistence is an optimisation technique for interactive and iterative Spark computations. It help saving interim partial results so they can be reused in subsequent stages. These interim results as RDDs are thus kept in memory or in more solid storages.

In general, we need to persist data when a dataset is likely to be re-used such as in iterative algorithms and machine learning algorithms. By default, every RDD operation executes the entire lineage. If an RDD will be used multiple times, it is recommended to persist it in order to avoid re-computation.

There is two types of persistence in Spark:

- Memory persistence: In-memory persistence is a suggestion to Spark, if not enough memory is available, persisted partitions will be cleared from memory (Least recently used partitions cleared first), transformations will be re-executed using the lineage when needed.

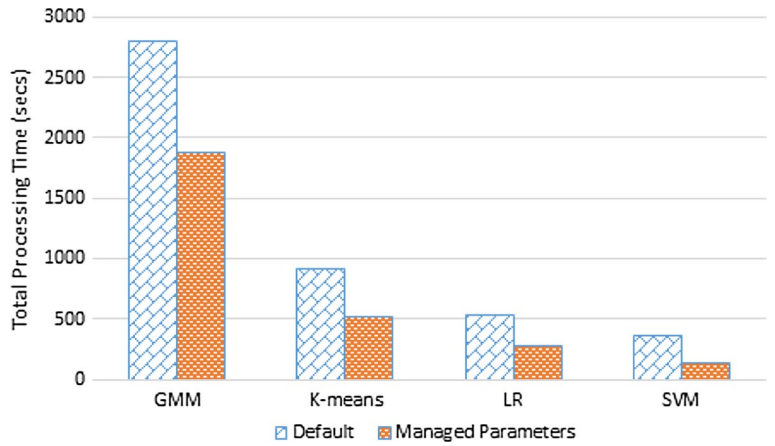

Fig. 12 Processing time with default and Managed parameters. This figure shows the processing time with default and managed parameters, the performance has been improved after making tuning on some parameters

- Distributed persistence: Resiliency is a product of tracking lineage, RDDs can always be recomputed from their base if needed, RDD partitions are distributed across a cluster. By default, partitions are persisted in memory in Executor JVMs.

There is different types of persistence Levels in Spark. By default, the persist method stores data in memory only, the persist method offers other options called storage levels.

Storage levels allow us to control:

- Storage location (memory or disk),
- Format in memory,
- Partition replication.

Table 4 summarizes the characteristics of each storage level.

In this part, we use RDD persistence to optimize data processing, this method is for interactive and iterative computations, it aims to save the results of RDDs evaluations. When we persist data, we can use the results later if it is required, this allows the computation overhead to decrease. Data persistence is done by default using the storage level 'memory_only'.

RDDs persistence is an essential technique for interactive and iterative algorithms. in other words, after persisting an RDD, each node saves the partition in memory and reuse it for future computation. This operation decreases the computation time and makes the whole cluster more efficient and fast.

There are two methods of persistence: **cache()** and **persist()**.

The first method cache() is for storing all the RDD in-memory, it has only one storage level 'memory_only'. While persist() has different storage levels, including: 'memory_only', 'memory_and_disk', 'disk_only', 'memory_only_ser', and 'memory_and_disk_ser'.

In this context, we tried to carry out experimental simulations to assess the RDD persistence by choosing several storage levels, using environment 3 (described above) and using HIGGS dataset.

Table 4 Storage levels of RDDs persistence

Storage level	Characteristics
memory_only	Storing data in memory if it is possible. when the RDD size is higher than memory size, It will not cache the partitions which have not enough space, In consequence, it will not recompute these partitions whenever required. This level provides very high space for storage and reduces the CPU computation time
memory_and_disk	For this level, it is possible to store the partitions in disk if there is no enough space in memory. consequently, it will retrieve these partitions whenever required. This level provides very high space for storage and reduces the CPU computation time
disk_only	Storing all the partitions only on the disk, it provides more space efficient. For this level, the storage space becomes small and the computation time becomes high
memory_only_ser	This level stores the RDD as serialized Java object and only in memory. It provides more space efficient compared to deserialized levels. However, it raises the CPU overhead. For this level, the storage space becomes small and the computation time becomes high
memory_and_disk_ser	This level stores the RDD as serialized Java object in memory and on disk. It provides more space efficient compared to deserialized level. However, it raises the CPU overhead. For this level, the storage space becomes small and the computation time becomes high

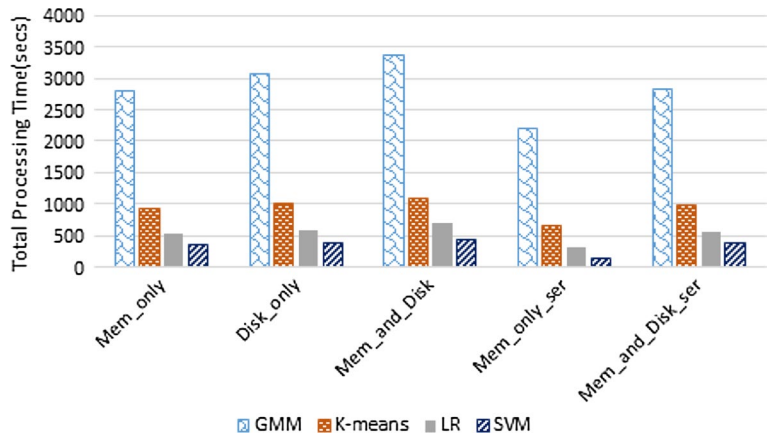

Fig. 13 Processing time using different storage levels of RDDs persistence. This figure shows the results of experimental simulations to assess the RDD persistence by choosing several storage levels. 'memory_only_ser' level has less execution time compared to the other levels, because it stores serialized data in memory, it provides more space efficient, and it raises the CPU overhead

Figure 13 shows the results of the simulations. We persist a dataset when it will be reused, that is to say, when an RDD will be used several times, we will persist it to avoid the re-computation and save the processing time. As shown in Fig. 13, 'memory_only_ser' level has less execution time because it stores serialized data in memory, it provides more space efficiency, and it raises the CPU overhead. We persist in order to use the RDD efficiently over parallel operations and it saves space by saving the serialized objects in memory if required. For Disk level, we can use it when the re-computation is more expensive than disk read for example with expensive functions or filtering large datasets.

Conclusion and future works

Resource management in large-scale is a very challenging problem. In this paper, we used the framework Apache Spark to manage the resources (CPU, memory and Disk).

First of all, we performed a scalability analysis using Spark. We analyzed the speedup and the processing time. We conducted several simulations using different number of nodes and using different data size. We deduced from these simulations that from a certain number of nodes in the cluster, it is no longer necessary to add additional nodes to improve the speedup and the processing time. So, we must investigate other techniques in order to improve the performance of Spark applications.

In this paper, we were also interested in tuning the resource allocation in Spark. We showed that it is not only by allocating all the available resources we get better performance but it depends on how to tune the resource allocation. We presented some recommendations for tuning and for a specific configuration, we proposed new managed parameters and we showed that these managed parameters give a better total processing time than the default parameters.

Finally, we studied the RDDs' Persistence in the context of machine learning algorithms use. We discussed the types of persistence in spark and we explained the different storage levels of RDDs' persistence. We conducted several simulations using these different storage levels. The results show that the storage level 'memory only ser' gives the best execution time among all tested storage levels. We have studied the RDDs' persistence by choosing two storage levels that let us control storage location (memory or disk) and the format in memory (serialized or not). An interesting direction for future work is to study a new storage level which is Disk Persistence with Replication. It is important to control Partition Replication because we can save data on multiple locations. So, in case a node goes down, replicated data on disk will be used to recreate the partition if possible without recomputation. Using Disk Persistence with Replication is especially interesting when recomputation is more expensive than storing in memory.

In this paper, we have also studied the tuning of resource allocation. Another direction for future work would be to study the other aspects in tuning the performance and scalability of Apache Spark. We will study the tuning of the number of partitions of a dataset, the tuning of Spark Shuffle Operations (for example: (i) How to choose the right arrangement of actions and transformations in order to minimize shuffles ; (ii) When to Add a shuffle transformation because sometimes an extra shuffle can be advantageous when it increases parallelism). We will also study how to reduce the size of data structures (using Kryo serialization) and how to choose efficiently the data formats (using extensible binary format like Avro, Parquet, Thrift, or Protobuf and storing in a sequence file).

Abbreviations

ADMM: alternating direction method of multipliers; API: application programming interface; CPU: central processing unit; DAG: directed acyclic graph; GCP: Google Cloud Platform; GMM: Gaussian mixture model; GPU: graphics processing unit; HBase: Hadoop database; HDFS: Hadoop distributed file system; KNN: K-nearest neighbor; KVM: Kernel-based virtual machine; LR: logistic regression; ML: machine learning; MLlib: Machine Learning library; MPI: message passing interface; OS: operating system; RAM: random access memory; RDD: Resilient Distributed Dataset; RMI: remote method invocation; SSD: solid state drive; SVM: support vector machines; YARN: Yet Another Resource Negotiator.

Acknowledgements

Not applicable.

Authors' contributions
KA, DZ, and MB contributed equally to this work. All authors read and approved the final manuscript.

Authors' information
Khadija Aziz received the Computer Science engineering degree from the National School of Applied Sciences, Cadi Ayyad University, Morocco in 2015. She is currently Ph.D. student at the National Institute of Posts and Telecommunications since 2016. Her research interests include Data Analytics, Machine Learning, and Big Data.

Dounia Zaidouni is an Associate Professor at the National Institute of Posts and Telecommunications, Morocco. Her main research interests are in the area of scheduling techniques and parallel algorithms for distributed systems and also in the area of Big Data Analytics. Recently, her research emphasizes the optimization of various aspects of Big Data platforms. In particular, resource management, fault tolerance and real-time processing in Big Data platforms. She obtained her Master's degree from the "Institut National des Sciences Appliquées de Lyon", France in 2011, and her PhD degree from the "Ecole Normale Supérieure de Lyon", Fance in 2014. Prior to joining the INPT of Rabat, she held position as Research and Teaching Assistant at the University Pierre and Marie CURIE (Paris 6), France in 2015.

Mostafa Bellafkih is a Professor at the National Institute of Posts and Telecommunications, Morocco. He received his Ph.D. degree from Pierre and Marie Curie University (Paris 6), France in 1994. He has also received the Ph.D. degree from the Mohammadia School of Engineers, Mohamed V University, Morocco in 2001. His research interest include Big Data, Resource Management, Machine Learning, Data mining, Database, Software Engineering, Artificial Intelligence, and Information Systems.

Funding
I certify that no funding has been received for the conduct of this study and/or preparation of this manuscript.

Availability of data and materials
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Received: 3 May 2019 Accepted: 8 August 2019
Published online: 23 August 2019

References

1. Bakshi K. Considerations for big data: architecture and approach. In: 2012 IEEE aerospace conference, 2012. Piscataway: IEEE; 2012. p. 1–7.
2. Aziz K, Zaidouni D, Bellafkih M. Big data optimisation among RDDs persistence in apache spark. In: International conference on Big Data, cloud and applications, 2018. 2018; Berlin: Springer; p. 29–40.
3. Aziz K, Zaidouni D, Bellafkih M. Big data processing using machine learning algorithms: Mllib and mahout use case. In: Proceedings of the 12th international conference on intelligent systems: theories and applications, 2018. 2018; New York: ACM; p. 25.
4. Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ, et al. Apache spark: a unified engine for big data processing. Commun ACM. 2016;59(11):56–65.
5. Apache Spark. http://spark.apache.org. Accessed 20 Feb 2019.
6. Databricks. https://databricks.com/spark/about. Accessed 15 Mar 2019.
7. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. Commun ACM. 2008;51(1):107–13.
8. Shanahan JG, Dai L. Large scale distributed data science using apache spark. In: Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining, 2015. 2015; New York: ACM; p. 2323–2324.
9. Salloum S, Dautov R, Chen X, Peng PX, Huang JZ. Big data analytics on Apache Spark. Int J Data Sci Anal. 2016;1(3–4):145–64.
10. Shahrivari S. Beyond batch processing: towards real-time and streaming big data. Computers. 2014;3(4):117–29.
11. Shi J, Qiu Y, Minhas UF, Jiao L, Wang C, Reinwald B, Özcan F. Clash of the titans: mapreduce vs. spark for large scale data analytics. Proc VLDB Endow. 2015;8(13):2110–21.
12. Liu X, Wang X, Matwin S, Japkowicz N. Meta-mapreduce for scalable data mining. J Big Data. 2015;2(1):14.
13. Singh D, Reddy CK. A survey on platforms for big data analytics. J big data. 2015;2(1):8.
14. Herodotou H, Lim H, Luo G, Borisov N, Dong L, Cetin FB, Babu S. Starfish: a self-tuning system for big data analytics. Cidr. 2011;11:261–72.
15. Wang G, Xu J, He B. A novel method for tuning configuration parameters of spark based on machine learning. In: 2016 IEEE 18th international conference on high performance computing and communications; IEEE 14th International conference on smart city; IEEE 2nd international conference on data science and systems (HPCC/SmartCity/DSS), 2016. 2016; Piscataway: IEEE; p. 586–593
16. Renner T, Thamsen L, Kao O. Adaptive resource management for distributed data analytics based on container-level cluster monitoring. In: DATA. 2017. p. 38–47.
17. Apache Hadoop. http://hadoop.apache.org. Accessed 15 Feb 2019.
18. Doulkeridis C, NØrvåg K. A survey of large-scale analytical query processing in mapreduce. VLDB J. 2014;23(3):355–80.
19. Gu L, Li H. Memory or time: Performance evaluation for iterative operation on Hadoop and spark. In: IEEE 10th international conference on high performance computing and communications & 2013 IEEE international conference on embedded and ubiquitous computing (HPCC EUC), 2013, 2013. Piscataway: IEEE; p. 721–727.

20. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, Mccauley M, Franklin M, Shenker S, Stoica I. Fast and interactive analytics over Hadoop data with spark. Usenix Login. 2012;37(4):45–51.
21. Lin C-Y, Tsai C-H, Lee C-P, Lin C-J. Large-scale logistic regression and linear support vector machines using spark. In: 2014 IEEE international conference on Big Data (Big Data), 2014. 2014; Piscataway: IEEE; p. 519–528.
22. Li P, Luo Y, Zhang N, Cao Y. Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms. In: IEEE international conference on networking, architecture and storage (NAS), 2015 , 2015. Piscataway: IEEE; p. 347–348.
23. Maillo J, Ramírez S, Triguero I, Herrera F. knn-is: an iterative spark-based design of the k-nearest neighbors classifier for Big Data. Knowl-Based Syst. 2017;117:3–15.
24. Siegal D, Guo J, Agrawal G. Smart-mllib: a high-performance machine-learning library. In: IEEE international conference on cluster computing (CLUSTER), 2016, 2016. Piscataway: IEEE; p. 336–345.
25. Assefi M, Behravesh E, Liu G, Tafti AP. Big data machine learning using apache spark mllib. In: IEEE international conference on Big Data (Big Data), 2017 , 2017. Piscataway: IEEE; p. 3492–3498.
26. Dhar S, Yi C, Ramakrishnan N, Shah M. Admm based scalable machine learning on spark. In: IEEE International conference on Big Data (Big Data), 2015, 2015. Piscataway: IEEE; p. 1174–1182.
27. Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, et al. Apache hadoop yarn: Yet Another Resource Negotiator. In: Proceedings of the 4th annual symposium on cloud computing, 2013. 2013. New York: ACM; p. 5.
28. Hindman B, Konwinski A, Zaharia M, Ghodsi A, Joseph AD, Katz RH, Shenker S, Stoica I. Mesos: a platform for fine-grained resource sharing in the data center. NSDI. 2011;11:22.
29. Karau H, Warren R. High performance Spark: best practices for scaling and optimizing Apache Spark. Sebastopol: O'Reilly Media, Inc.; 2017.
30. Penchikala S. Big Data processing with Apache Spark. Lulu. com, 2018.
31. Karau H, Konwinski A, Wendell P, Zaharia M. Learning Spark: Lightning-fast Big Data analysis. Sebastopol: O'Reilly Media, Inc.; 2015.
32. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. HotCloud. 2010;10(10–10):95.
33. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. Resilient Distributed Datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on networked systems design and implementation, 2012. 2012; Berkeley: USENIX Association; p. 2.
34. Spark Architecture. https://spark.apache.org/docs/latest/cluster-overview.html. Accessed 13 Feb 2019.
35. Meng X, Bradley J, Yavuz B, Sparks E, Venkataraman S, Liu D, Freeman J, Tsai D, Amde M, Owen S, et al. Mllib: machine learning in apache spark. J Mach Learn Res. 2016;17(1):1235–41.
36. Armbrust M, Das T, Davidson A, Ghodsi A, Or A, Rosen J, Stoica I, Wendell P, Xin R, Zaharia M. Scaling spark in the real world: performance and usability. Proc VLDB Endow. 2015;8(12):1840–3.
37. Venkataraman S, Yang Z, Franklin MJ, Recht B, Stoica I. Ernest: Efficient performance prediction for large-scale advanced analytics. In: NSDI, 2016. 2016; p. 363–378.
38. ML Guide for Apache Spark. https://spark.apache.org/docs/latest/ml-guide.html. Accessed 16 Apr 2019.
39. Schölkopf B, Burges CJ, Smola AJ, et al. Advances in Kernel methods: support vector learning. Cambridge: MIT Press; 1999.
40. Dua D, Graff C. UCI Machine Learning Repository 2017. http://archive.ics.uci.edu/ml. Accessed 11 Jan 2019.
41. Baldi P, Sadowski P, Whiteson D. Searching for exotic particles in high-energy physics with deep learning. Nat Commun. 2014;5:4308.
42. Cloud Dataproc Service. https://cloud.google.com/dataproc/. Accessed 1 Jan 2019.
43. Spark Tuning. https://www.cloudera.com/documentation/. Accessed 20 Apr 2019.

Publisher's Note