Institute of Parallel and Distributed Systems
Applications of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Masterarbeit Nr. 3699

Large-scale Data Mining Analytics Based on MapReduce

Sunny Ranjan

Studiengang:	Infotech
Prüfer:	PD Dr. Holger Schwarz
Betreuer:	PD Dr. Holger Schwarz
Begonnen am:	12.05.2014
Beendet am:	16.12.2014
CR-Classifikation:	H.2.8, H.3.4, D.4.3, D.4.7

Table of contents

Abstract **6**

1. Introduction **7**

1.1 Background Of The Problem 7
1.2 Purpose Of Work 9
1.3 Structure Of Work 12

2. Data Mining Concepts **13**

2.1 Motivation Behind Data Mining 13
2.2 Definition Of Data Mining 14
2.3 Data Mining Process 15
2.4 Data Mining Methods And Its Categories 16

3. MapReduce Concepts **21**

3.1 MapReduce Paradigm 21
3.2 Pre-Requisites 21
3.3 Architecture And Programming Model 22

4. Hadoop **24**

4.1 HDFS - Hadoop Distributed File System 24
4.1.1 Design of HDFS 25
4.2 MapReduce Framework 26
4.2.1 Map 27
4.2.2 Reduce 28
4.3 Hadoop MapReduce 1.0 (MRv1) 28
4.4 Hadoop MapReduce 1.0 (MRv1) or YARN 30
4.5 Why YARN? 32
4.6 Conclusion 33

5. Apache Spark **34**

5.1 Basics Of Spark 34
5.2. Overview Of Spark In Cluster Mode 37
5.3 Spark On YARN 38
5.4 Task Scheduling Process In Spark 40
5.5 Apache Spark's Machine Learning Library (MLlib) 41
5.6 Decision Tree Learning Algorithms In Spark's MLlib 41

6. Discussion Of Existing Approaches To Large Scale Data Mining **44**

6.1. Existing MapReduce Based Data Mining Scientific Works 45
6.1.1 MapReduce Naive Bayes Classifier 45
6.1.2 MapReduce K-Means Clustering 47

6.1.3 MapReduce K-Medoids Clustering 48
6.1.4 MapReduce Co-Clustering 48
6.1.5 MapReduce Apriori Association Rule Discovery 49
6.1.6 MapReduce ID3 Decision Tree 50
6.1.7 MapReduce C4.5 Decision Tree 51
6.1.8 MapReduce Decision Tree Ensembles (PLANET) 53
6.2 MapReduce Based Data Mining Tools 55
6.2.1 Apache Mahout 55
6.2.2 WEKA 57
6.3 Conclusion 58

7. Experimental Set-Up 59
7.1 Data Preparation 63

8. Experimental Results and Evaluation 69
8.1 Initial Experiments 70
8.2. Experiment 1 With Alpha Data (Binary Classification) 70
8.3 Experiment 2 With Blog Data (Regression) 74
8.4 Experiment 3 With YearPrediction Data (Regression) 77
8.5 Experiment 4 With Mnist Data (Multi-Class Classification) 82
8.6 Conclusion 86

9. Conclusion Of This Work 89

References 90

List Of Figures 97

List Of Tables 98

John Naisbett:

“We are drowning in data but starving for knowledge.”

Abstract

In this work, we search for possible approaches to large-scale data mining analytics. We perform an exploration about the existing MapReduce and other MapReduce-like frameworks for distributed data processing and the distributed file systems for distributed data storage. We study in detail about Hadoop Distributed File System (HDFS) and Hadoop MapReduce software framework. We analyse the benefits of newer version of Hadoop software framework which provides better scalability solution by segregating the cluster resource management task from MapReduce framework. This version is called YARN and is very flexible in supporting various kinds of distributed data processing other than batch-mode processing of MapReduce.

We also looked into various implementations of data mining algorithms based on MapReduce to derive a comprehensive concept about developing such algorithms. We also looked for various tools that provided MapRedcue based scalable data mining algorithms. We could only find Mahout as a tool specially based on Hadoop MapReduce. But the tool developer team decided to stop using Hadoop MapReduce and to use instead Apache Spark as the underlying execution engine. WEKA also has a very small subset of data mining algorithms implemented using MapReduce which is not properly maintained and supported by the developer team. Subsequently, we found out that Apache Spark, apart from providing an optimised and a faster execution engine for distributed processing also provided an accompanying library for machine learning algorithms. This library is called Machine Learning library (MLlib). Apache Spark claimed that it is much faster than Hadoop MapReduce as it exploits the advantages of in-memory computations which is particularly more beneficial for iterative workloads in case of data mining. Spark is designed to work on variety of clusters: YARN being one of them. It is designed to process the Hadoop data.

We selected to perform a particular data mining task: decision tree learning based classification and regression data mining. We stored properly labelled training data for predictive mining tasks in HDFS. We set up a YARN cluster and run Spark's MLlib applications on this cluster. These applications use the cluster managing capabilities of YARN and the distributed execution framework of Spark core services.

We performed several experiments to measure the performance gains, speed-up and scale-up of implementations of decision tree learning algorithms in Spark's MLlib. We found out much better than expected results for our experiments. We achieved a much higher than ideal speed-up when we increased the number of nodes. The scale-up is also very excellent. There is a significant decrease in run-time for training decision tree models by increasing the number of nodes. This demonstrates that Spark's MLlib decision tree learning algorithms for classification and regression analysis are highly scalable.

1. Introduction

We started this work with the objective of performing large-scale data mining analytics using *MapReduce*. With MapReduce, we mean a distributed data-processing model which has become popular for processing extremely large-scale data. We went on for an extensive search of implementations and tools of data mining algorithms which provided large-scale data mining based on *Hadoop* MapReduce. We narrowed our search for a specific data mining task: decision tree model learning based classification and regression data mining. We found out during our extensive search process that there are rather very few implementations of data mining algorithms based on Hadoop MapReduce. We found only two tools that supported some data mining algorithms based on original MapReduce. But we discovered that both of those tools have discontinued their support for original Hadoop MapReduce. The respective teams responsible for both the tools decided to completely retool their scalable data mining algorithms by using Apache Spark. This is how we came across *Apache Spark*.

Apache Spark is a MapReduce-like distributed data processing model that uses Hadoop data and Hadoop cluster but uses a much more optimised execution engine than the original Hadoop MapReduce [5a]. Their execution engine is optimised for different kinds of workloads as opposed to strict batch-mode processing of original MapReduce. Apache Spark not only provided an efficient execution that exploited the advantage of in-memory computations but also provided in-built low-level data structures and high-level libraries (programming constructs) for data mining algorithms. This library is called *Machine Learning library,* abbreviated as *MLlib,* which contains some high level APIs for performing a wide variety of data mining tasks. We also found out that it contained the implementations for decision tree learning based classification and regression analysis.

We go on to analyse the performance of those algorithms implemented in MLlib. We run programs using Apache Spark' MLlib algorithms on a Hadoop YARN cluster which acts as a cluster manager and a Hadoop distributed files system for data storage. We use Apache Spark's MLlib algorithms to build a decision tree of a fixed depth by using three different kinds of data mining tasks: binary classification, multi-class classification and regression. We perform numerous experiments with real-world data of varying sizes and various number of nodes to measure the performance gain, scale-up and speed-up. We also determine the impact of data characteristics and other factors on the computation time. We also elucidate the limitations and bottlenecks of building distributed scalable decision trees using this execution environment. We finally analyse the effectiveness of this distributed data-processing model and framework for mining large-scale data. We aim to find out if this library successfully serves our purpose for large-scale data mining.

1.1 Background Of The Problem

There is no dearth of data mining algorithms and tools for processing small datasets which can easily fit into a single machine (see a list of them in [1]). In order to mine large datasets, one of the current workarounds is to first sample the huge amounts of data into a smaller

dataset and then feed the resulting smaller dataset into the existing data mining algorithms and tools. This obviously does not alleviate the problem of storing such large-scale data and in addition to that, there are no such sampling algorithms which can guarantee to give always good sampling quality for any kind of data. As a result, this particular approach is not proven to output accurate data models. Therefore, large-scale data mining and analytics is still a much researched topic in Information Technology sector. Large-scale data mining and analytics is a term which usually refers to processing and analysis of “datasets with sizes which is beyond the capability of commonly used software tools to capture, curate, manage and process the data within a tolerable [and acceptable] elapsed time” [2]. The sudden explosion of data has driven the needs for more scalable and faster data mining and analytics methods. The conventional or legacy data mining algorithms, broadly classified into association-rule discovery, classification, regression and clustering suffer from the problems of memory and processing power limitations of the single machine they are running on.

So, how can we do large-scale data mining? MapReduce framework, first developed by Google, is the golden standard for general large-scale data processing and is slowly emerging as an important paradigm in data mining and analytics. MapReduce [3] is a simple programming model for expressing distributed computations on massive amounts of data and also an execution framework for large-scale data processing on clusters built from low-end commodity servers. With MapReduce, queries are split and distributed across parallel nodes and processed in parallel (the map step). The results are then collected from various map outputs and merged (the reduce step). The huge success of MapReduce has led to increasing implementations of data mining algorithms based on this framework. The high throughput and the performance of Google MapReduce subsequently led to the open-source MapReduce implementation by *Apache Software Foundation* which is known as Hadoop [4]. Although MapReduce is very simple because of its high-level abstraction and simplistic programming model, its application to the data mining algorithms remains poorly understood. The conversion of legacy algorithms to MapReduce is, henceforth, a non-trivial process. The process of designing MapReduce data mining algorithms requires identifying the parts of the legacy algorithms which have independence with respect to data and can be parallelized. Moreover, it also obligates the algorithm designers to develop a proper understanding of the MapReduce architecture.

MapReduce has continuously evolved over years in an effort to yield ever-increasing throughput and performance. These optimised MapReduce frameworks are called MapReduce-like frameworks. But in essence, they are MapReduce because they distribute the computations using map function and combine the results in reduce function. Nowadays, there are numerous implementations of MapReduce and that’s why it is a very broad term that encompasses the original MapReduce and the optimised versions. The optimised versions of MapReduce rather than the pure Hadoop MapReduce was found to be interesting for our work because a lot of work has been already done using old MapReduce systems and they clearly point out the drawback of strict batch-processing model for heavily iterative algorithms like data-mining algorithms. Moreover, hadoop MapReduce clusters are hardcoded to perform only MapReduce batch workloads and cannot support other workloads. This was seen as a wastage of valuable computational

resources. As a consequence, it is interesting for us to analyse a computational framework which is basically tailored for our purpose. Apache Spark [5] is a relatively new distributed processing engine for large-scale data which promises a much faster performance as compared to Hadoop MapReduce in general. This is made possible by persisting as much data in memory as possible and performing in-memory MapReduce based distributed computation. This saves the time for multi-stage applications as they do not need to perform the expensive disk read operations in every stages. This concept of data caching is particularly useful for iterative workflows where computations are done on nearly same datasets in every iterations [5a]. Apache spark provides a rich set of tools that take advantage of its optimised core execution engine. They are customised for special workflows like streaming, graph-processing, iterative processing, etc in order to optimise the MapReduce for a particular kind of data processing. One of such libraries is called Machine Learning library (MLlib) which optimises the MapReduce for data-mining algorithms on large-scale data and provides implementations of various data mining algorithms.

1.2 Purpose Of Work

We identified that classification and regression mining algorithms based on decision tree models is particularly interesting for our work due to the current organisational needs of the institute and also due to lack of properly implemented MapReduce data mining algorithms based on decision tree learning. In the section 2.4, we will give a generic description of decision tree learning algorithm which builds a non-distributed decision tree on a single machine. This algorithm suffers from a big limitation that all the training data lying on the disk might not fit into main memory of a single machine because they suffer from the restriction that all the training data should reside in the memory. As a consequence, they suffer from scalability problems with respect to the size of training data. One of the straightforward scalability approach includes sampling the training data into a smaller dataset but this method is not guaranteed to output a sampled dataset with a size which can fit into the main memory. Moreover, there is no proof of guarantee for a good sampling quality for sampling methods. In data mining literature, we can find a few complex scalable decision tree algorithms. RainForest is a scalable fast decision tree algorithm that adapts to the amount of main memory available and uses a special data structure that is more memory efficient [6]. But this algorithm has too much dependence on the size of main memory available and involves time-consuming repeated operations of reading training data from disk for each level of tree. BOAT (Bootstrapped Optimistic Algorithm for Tree construction) is another scalable decision tree algorithm that uses bootstrapping sampling (to sample the training data uniformly with replacement) to create smaller subsets of training data that can easily fit into main memory [7]. Each of these subsets are used to create several trees which are then somehow examined and combined to form a new tree which is very close to the tree which would have been built using all the training data. This tree is incrementally constructed and it requires only two scans of training data lying on the disk. We cannot give a comprehensive list of all the scalable decision tree learning algorithms in data mining literature as they are not the main focus of our work.

The solution strategy to the scalability problem of decision tree learning algorithms or as a matter of fact, for any data mining algorithm, always waters down to a very generic question in data mining community:

"Which is better: Better algorithms or more data?"

In other words, what is more beneficial and less effort consuming: improving the data mining algorithms so that they can output equally accurate patterns from sampled-down large-scale data or use simple algorithms without any optimisations on a large-scale data. "The Unreasonable effectiveness of Data" by NORVIG ET.AL [8] shows that for every increase in order of magnitude of data we have, it surpasses all the improvements in algorithms. Most of the data mining algorithms are paralysed with a restriction that they cannot be solved by some neat and concise mathematical formulas. Instead, the best approach is to embrace the complexity of the problem domain and to address it by harnessing the power of data. RECCHIA and JONES actually performed experiments that demonstrated that simple algorithms with more data actually outperforms and scales more efficiently than applying more efficient and elegant algorithms on smaller datasets [9]. SHOTTON ET AL. at Microsoft Research also support the view that it is worthwhile to look for more data instead of searching for a better classifier that outputs highly accurate prediction model using sampled-down data [10]. The major outcome of their experiment was that the huge amount of data, they were able to generate, was one of the key factors in the success of the classifier algorithm, while using a very simple random decision forest as a decision tree classifier. BRILL [11] and RAJARAMAN [12] also point out that it is better to concentrate more on exploiting the advantages of using more and more data rather than fine tuning the algorithms.

As we can clearly see the advantage of using large-scale data mining using simple models on all data, we actually need to perform data mining on large scale or "web-scale" data in the order of hundreds of gigabytes to terabytes or petabytes. It is quite obvious that a single computer can not store so much data and process them in reasonable amount of time given the limited amount of RAM and CPU processing power. Therefore, we are obligated to use distributed storage and computation.

There are two procedures for scalable data mining: scale-up and scale-out. Scale-up means a small cluster of high-end servers whereas scale-out refers to large cluster of low-end commodity servers. Although scale-out will imply more machine failures and high requirements on fault-tolerance, it is much more economical than scale-up [20a].

Large-scale data has scalability problems in two dimensions: data storage and processing. Therefore, we need a special distributed file system responsible for distributing workloads across the cluster in a controlled manner. Since MapReduce is designed for large clusters built using low-end commodity drives, failure of disks will be a norm not an exception. The underlying layer of distributed file system provides efficient, automatic distribution of large-scale data across clusters of machines. It also handles fault tolerance through replication, load balancing and monitoring in such a way to provision and locate computing at data locations and save the critical resource: network bandwidth. Scheduling of machines in a

grid is also provided by other proprietary systems like *Condor* but it requires a separate SAN to automatically distribute the data and MPI for managing communication system.

There are numerous parallel programming languages such as Orca, Occam, ABCL, SNOW, MPI and PARLOG but they do not clearly illustrate how to parallelize a particular algorithm [13]. There is a vast amount of literature on distributed learning and data mining [14], but very few of them give a common procedure or a framework to write distributed machine learning programs. We have chosen MapReduce for parallel data processing because it is a simple programming model where users only need to specify the computation in terms of a *map* and a *reduce* function and they don't need any specialised knowledge of distributed computing [15]. Underlying runtime system automatically parallelises the computation across large-scale clusters of machines, and also handles machine failures, efficient communications and performance issues. GLEICH explains the ease of use of MapReduce distributed computations in HPC clusters where normally MPI dominates: MPI based distributed programming needs thousands of lines of C code just to perform a single task of loading data and this code needs to distributed to all nodes in the cluster. Whereas, same task in MapReduce needs zero lines of code [16]. Moreover, there has been many successful attempts to adopt various popular data mining algorithms to MapReduce programming model. Various experiments have shown that the major advantage of this data processing model is its flat scalability curve for very large amounts of data [17]. A program written in other distributed programming models will need a large amount of refactoring when scaling from few machines to a cluster of 4000 nodes. In other frameworks, the program needs to be re-written several times as the scale grows while a MapReduce program remains basically same for all scales. TAN ET AL. performed experiments to compare three scalable approaches to Naive Bayes classifier: sampling, ensemble methods and MapReduce techniques. The results clearly illustrate that the MapReduce approach is better due to no limitations and good accuracy [18].

We have given proper arguments for: Why we perform large-scale data mining? And why MapReduce is a suitable distributed programming model for doing large-scale data mining? This work is principally done to answer the following questions:

- How can we perform large-scale data mining based on MapReduce?
- What tools and implementations for MapReduce based data mining algorithms are available?
- Which of these implementations or tools we select?
- What are the reasons for selecting this particular tool or implementation?
- How do we measure the performance of this implementation?
- What is the performance of this tool?
- How does this tool scale with the large-scale data?
- Does this tool give us the benefits that we expected or desired in the beginning?

1.3 Structure Of Work

In this chapter, we have already given an introduction to our work. Chapter 2 explains the general concepts and fundamentals of data mining methods and models. In Chapter 3, an overview of the distributed programming model of MapReduce is presented. Chapter 4 gives a detailed description of an open-source implementation of MapReduce: Hadoop. We will cover the most basic aspects of distributed storage and the distributed execution framework which is indispensable to understand the further chapters. We will also uncover the benefits and drawbacks of this framework. Chapter 5 gives a description of another MapReduce implementation, which is provided by Apache Spark. Here, we focus on the Machine Learning library (MLlib) of Apache Spark. It is rather obligatory to comprehend the details of implementation of this library as we have developed and run data mining programs using this library. In Chapter 6, we will browse through relevant related works that have been done in this field and discuss the available approaches to scalable data mining algorithms. It includes several scientific papers, data mining tools and environments based on MapReduce distributed programming model. In Chapter 7, we describe the experimental set up. In Chapter 8, we present the various experimental results using charts and graphs and evaluate them. In Chapter 9, we put forward the overall conclusions of this work and also we lead to the future work.

2. Data Mining Concepts

In this chapter, we give a general introduction to data mining and its concepts. As data mining is about discovering patterns and knowledge from data, it has two facets: data mining methods (algorithms) and models to represent the discovered patterns and knowledge. Here, we give a brief description of most commonly used data mining methods and models. Since the focus of work is decision tree based classification and regression data mining methods, we give a detailed description of decision tree learning.

2.1 Motivation Behind Data Mining

Automated data collection tools, mature database technologies, inexpensive disks, online storage and Internet has generated tremendous amounts of data which is stored electronically in databases and other information repositories [19]. Actually there is a very famous quote describing this situation:

"We are drowning in data but starving for knowledge."

-by JOHN NAISBETT

Nowadays, information or knowledge is very valuable. We have vast amounts of raw data from various sources, but they are useless unless we can gain useful some information or knowledge from them.

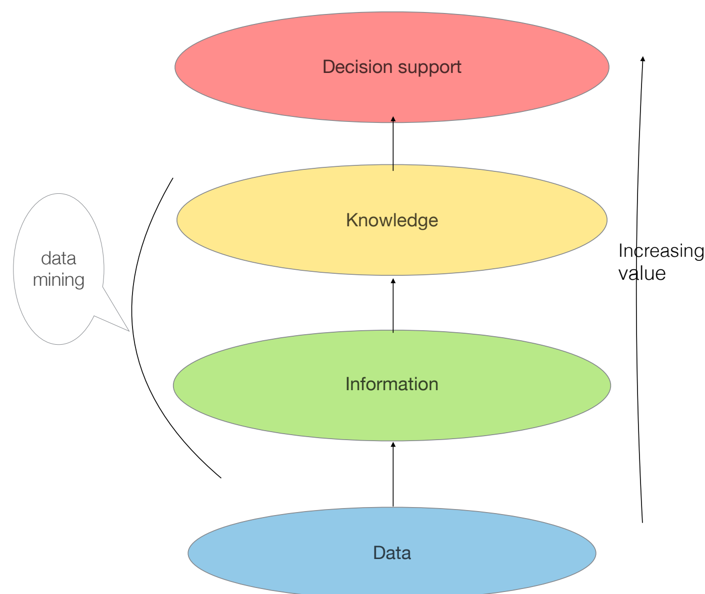


Figure 1. Value chain of Information Gain

In Figure 1, we depict that the mining of data increases the value of collection of vast amounts of data. When we perform data mining on data, we gain some useful information and eventually we are equipped with a better knowledge and understanding of patterns and regularities in the data. This knowledge helps us to make better and more productive

decisions. Therefore, data mining gives us a lot of gain in the value chain which is in often more than the efforts put to perform data mining. The proper knowledge about the patterns in data is even considered very crucial for making decisions in some special scenarios. Such field of applications for data mining is growing day by day, for example, it has now far-reaching applications with the objective of bringing optimisations, increasing the revenues and saving limited resources like medicine, food supply, etc. It saves us a lot of efforts and time in case of complex decision making situations.

2.2 Definition Of Data Mining

Data mining is a process which starts apparently from vast amounts of unstructured datasets in various formats and from various sources to extract useful and interesting information and patterns for making better and timelier decisions [20]. It is an automatic or a semi-automatic process to provide us some useful information which we cannot derive just by looking at the raw data. It is not only limited to business field but also to artificial intelligence, natural language processing, genome sequencing and many other scientific breakthroughs. Data mining has emphasis on utilising data from a particular domain in order to understand some questions in that domain. For gaining useful information from a specified data domain, it applies various data mining algorithms such as prediction, classification, regression, clustering, association-rule discovery, recognition and so on. These data mining algorithms can be broadly classified into *predictive* and *descriptive* techniques. Predictive techniques are considered to be more similar to machine learning where a model learns over given data (training and testing) and then predicts new instances of data using the learned model e.g. decision tree classifiers and regressors, bayesian classifiers, neural nets etc. Descriptive techniques merely describe characteristics or patterns in the data and there is no learning or training phase. Examples include association rules, summarisation, generalisation, inductive algorithms, etc.

The process of data mining in itself is an empirical approach that is driven by data and it is an attempt to extract statistical regularities of data [20a]. There are 3 components to this approach: data, representation of data and some methods to capture the regularities in data. Feature is another term given to the representations of input data. Features can be superficial and easy to extract or can be deep and difficult to extract. These features are essential for extracting useful regularities or patterns using some kind of model or algorithms. There is a growing evidence that of all three components, data is the most important [11]. When we apply simple data mining methods to the superficial features and to a lot of data, we require much less effort and such a process is more scalable. The accuracies of data mining models obtained using this approach can be continuously improved by using more data. When we use complex methods to derive equally accurate models from small dataset, the complexity hardness of such methods are very difficult to increase. But more complex methods are not always guaranteed to give us more accurate models using small datasets because there can be case that they have escaped some important data which was essential for outputting the model. Additionally, we can see that such an approach is hardly scalable. Thus, we can argue that using simple methods on large dataset to output simple model is evidently a better and more scalable approach than applying complex methods on small datasets to output complex models.

Now, we are faced with the problem that the data mining and analytics need to deal with burgeoning amounts of data. Data volume, variety, velocity and complexity all present challenges that must be faced to efficiently address this problem [21]. This is apparently beyond the capability of single machines at a reasonable cost. There are various possible solutions for introducing scalability to the data mining algorithms. Parallel and distributed data-mining is one of them. This kind of data-mining can run on a diversified type of clusters of machines. This strategy has two facets: distribution of the data and an execution framework for performing distributed processing. Data-centres or clusters need to find an appropriate architecture in order to be scalable and cost-effective. Cloud-computing is one of the architectural model for clusters which has emerged to be affordable and flexible. It is predominantly used for cases when we need to handle and store vast amounts of data and to execute a particular execution on multiple number of nodes in distributed fashion. It is beneficial for both the service-provider and the service-consumer because of the element of elasticity in cases of varying loads.

2.3 Data Mining Process

Data mining is an analytic process of discovering hidden, previously unknown and usable information or patterns from a large amount of data that go beyond simple analysis [20]. The data is analysed without any expectation on the result. Data mining is often is described as the analysis step of knowledge discovery process. But for the data mining process to work properly, we must know the domain of the data, understand the data and the mining methods. We need to understand the data in order to ensure meaningful data mining results because data mining algorithms are often sensitive to specific characteristics of the data: outliers (data values that are very different from the typical or expected values), irrelevant columns, columns that vary together, data coding and data that are chosen to be included or excluded.

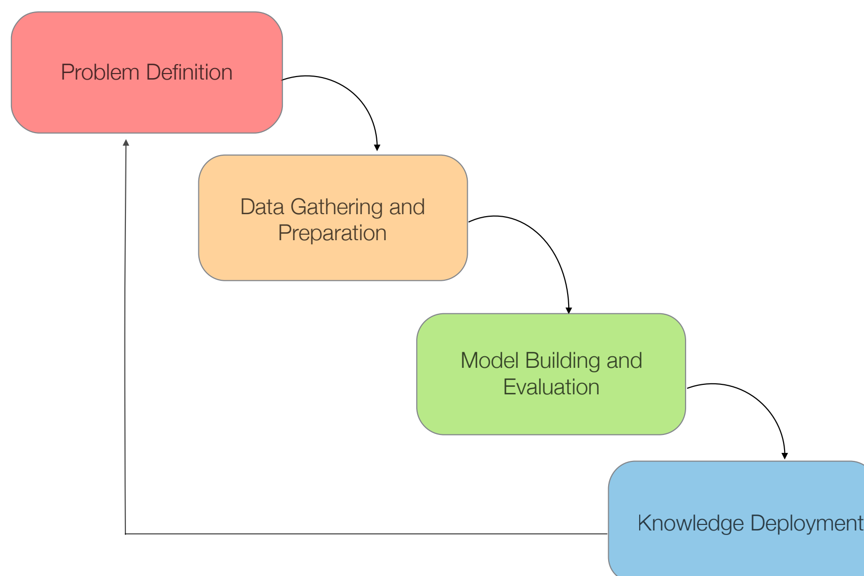


Figure 2. Data mining process [22]

Figure 2 shows the general steps in a data mining process. Such a process is iterative in nature which means that the data mining results trigger new questions where the focus of models is narrowed in the next iteration. The first step is to identify the objectives and the requirements and reformulate them into a data mining problem. The following step is to access the data and identify those datasets which addresses the data mining problem well. The input data should be meaningful for the target application. This step is indispensable for improving the data quality which can subsequently improve the information obtained through data mining. It might also involve tasks like data sampling, cleaning and transformation. In model building and evaluation stage, we apply various modelling techniques and finally evaluate how well the model satisfies our initial problem statement. The last stage of knowledge deployment is to apply the model in the real-world targets to obtain the actionable information that we aimed for initially.

2.4 Data Mining Methods And Its Categories

Like mentioned above, data mining methods can be broadly classified into *descriptive* and *predictive.* The former describes the data in a concise summarised manner and presents the general interesting properties and patterns; whereas the latter constructs one or a set of models using the past values and predicts the behaviour of new datasets. Figure 3 shows the categorisation of data mining methods and the popular methods in the respective categories.

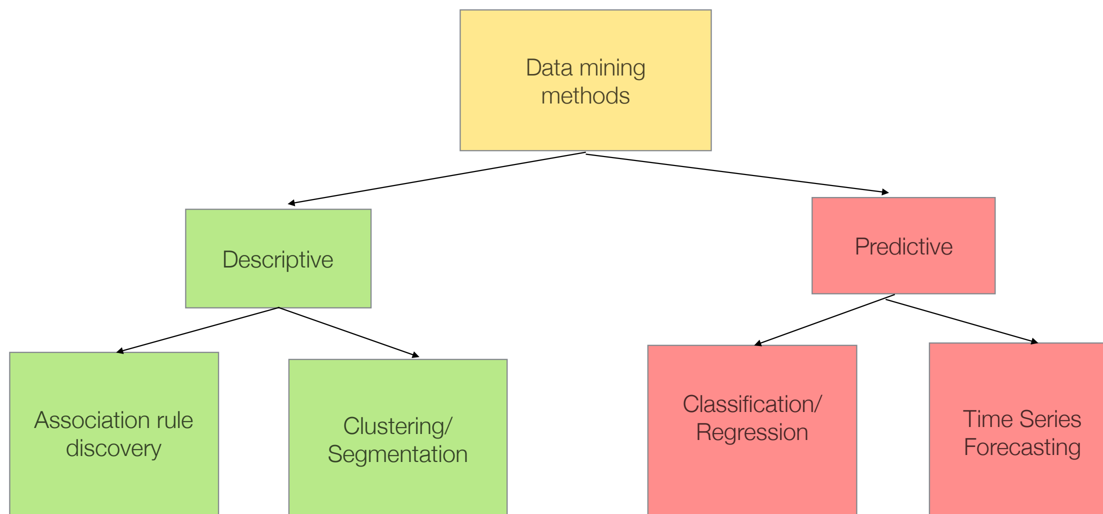


Figure 3. Data mining methods and its categories

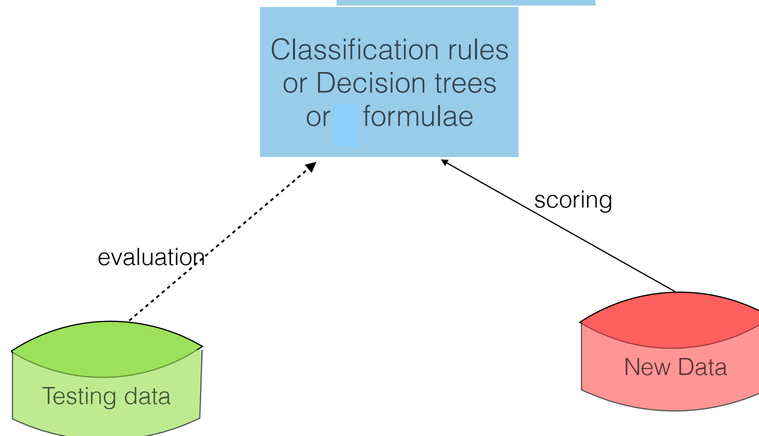
Association-rule discovery or **frequent pattern mining** searches recurring relationships in a given data. It discovers interesting associations and correlations between itemsets in our data [23]. Typically, association rules that do not satisfy the minimum support and the confidence are discarded as uninteresting patterns. They are used to derive interesting statistical correlations between attribute-value pairs. This data mining technique is widely used in recommender systems by large e-commerce and online shopping companies like Amazon, Zalando, eBay, etc which we come across in our daily lives. There is a collection

of algorithms that implement frequent patten mining. *Apriori* is one of the foremost algorithm proposed by R. AGRAWAL and R. SRIKANT [24] for mining frequent itemsets for Boolean association rules [23]. This algorithm implements a level-wise iterative searching where the prior knowledge of frequent itemsets in the previous iteration is utilised to reduce the search space for candidate itemsets in the current iteration. This typically requires several passes to the original database. There are several optimisations to this algorithm but their discussion is beyond the scope of this work. Another novel implementation for this data mining is **frequent pattern growth (FP-growth)** which employs a divide-and-conquer tactic to build a *frequent pattern tree* which contains the itemset association information. The whole set of transactions are then partitioned into smaller sets each corresponding to one frequent itemset in the tree. The tree grows in similar fashion. The advantage of this approach is elimination of candidate itemsets generation step in every iteration.

Clustering or **Segmentation** is also a non-predictive technique that aims to group a set of objects into k clusters such that objects in the same cluster are more similar and they are very dissimilar from objects of other clusters. Clustering is called as **unsupervised learning** because it organises the observed data into meaningful taxonomies, groups or clusters that are initially unknown or not pre-defined [25]. It implies that the training data do not need to have the class labels. That's why it is sometimes also referred as **automatic classification**. Clustering itself is not an algorithm but refers to a general data mining task which can be implemented by using various algorithms [26]. These algorithms can use various schemes for clustering like distance-based, partitioned-based, hierarchical, density based and model-based clustering. Discussion of such algorithms in detail is beyond the scope of this work.

Classification and **regression** are two main predictive data mining techniques or methods that we will discuss in more detail as they are the main focus of our work. **Classification** is a data mining method where it learns a predictive model or a function by analysing a set of properly labeled training examples. The derived model is then used to predict the class labels for new data for which the class label is previously unknown. The term **classification** is used when it predicts categorical (discrete, unordered) class labels. It is rather called **regression** when the target class labels have continuous values [23]. Both the predictive techniques are a two-step procedure, comprising of a *learning step* (to learn a predictive model or a function) and the *application step* (where the derived model is used to predict class labels of given data). The second step is also popularly called as *scoring* in data mining literature. This is depicted in Figure 4 where testing or evaluation of output model is rather optional (shown by dotted line).These techniques are also known as supervised learning techniques because the training phase involves training data whose target or predicted labels are previously known. Therefore, the training data should have special characteristics when being applied to these data mining methods. One part of the training example is called explanatory (independent) variable which is grouped together to form a feature vector. The other part is the dependent variable or the predicted class label. In data mining terminology, the feature vector is said to be composed of attributes or features.

(a)



(b)

Figure 4. Two-step classification and regression data mining [23].
(a) Learning step (b) Testing and application step

Typically, the first step of classificatio[illegible] or regression is to learn the mapping y —> $f(X)$ or the function $y = f(X)$ that can predict the [illegible]iated class label y of a given training record X. In other words, we aim to find c[illegible]es which establish some kind of relationship between the feature vector ar[illegible] class label. However, this mapping can be represented using a variety of [illegible] output representations like classification rules, decision trees or just ma[illegible]lae. In the second step, we can also measure the accuracy of the predictor mode[illegible]is purpose, a set of testing examples are formulated and then the accuracy of the m[illegible]el is derived by the percentage of test set examples that are correctly classified by the model.

For our work, we have chosen classification and regression data mining methods which uses decision tree model as a predictive model. Decision tree models, where the target class labels can only have discrete and finite set of values (categorical values), are called **classification trees** [26a]. Decision trees, where the target labels can take continuous values (typically real numbers), are called **regression trees** [26a]. We will use the term **decision tree classifiers** and **decision tree regressors** for the corresponding classification and regression methods to build the decision tree models. We will use an umbrella term **decision tree learning** for both methods.

Decision tree is a flowchart-like tree structure where each leaf or terminal node represents the class label or a probability distribution over the set of class labels while each non-leaf or internal node represents a test on an attribute and the branch depicts the result of the test. Therefore, each non-terminal node is labeled with an input feature and the branches from this node is labeled with the possible values of the feature. We h[illegible] an example to illustrate the decision [illegible] in Figure 5. This decision tree learning models the classification rules for the data mining problem: “Whether a customer buys a smartphone?” We have used various features or attributes like age, salary and pension as a test parameter to divide the customer set into two class labels: yes or no.

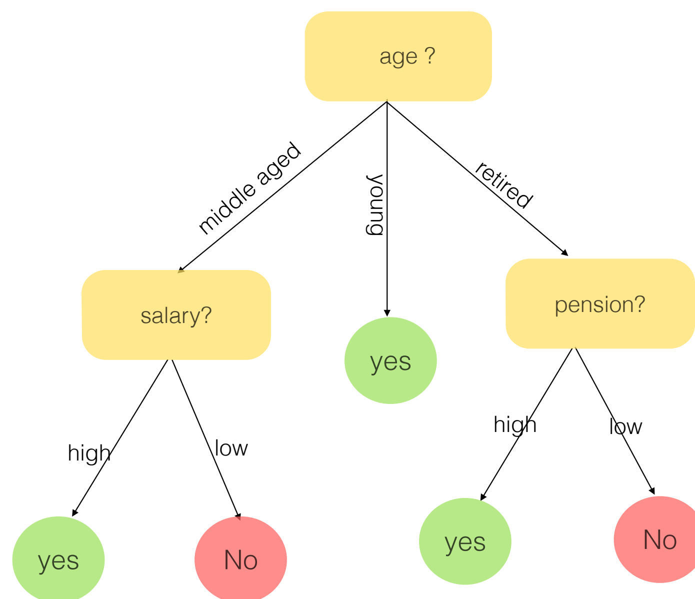


Figure 5. Decision tree for the date mining goal: “buy a smartphone”

Now, one can easily argue that why we have chosen decision tree learning as a specific data mining algorithm to be parallelized using the MapReduce. This question can be reformulated as: “Why decision tree classifiers and regressors are so popular?” First of all, the construction of decision trees does not require any prior knowledge of the data domain. This means that they are able to handle all kinds of data: both numerical and categorical. They do not feature scaling, data normalisation or dealing with the missing values. They are also able to capture non-linearities and feature interactions. Decision models are rather simple and easy to understand and interpret.

There are several specific algorithms that implement decision tree learning. Iterative Dichotomiser (ID3) is the seminal algorithm invented by J. Ross Quinlan, to generate decision trees [27]. ID3 is a precursor for a better algorithm called as C4.5, which was again developed by Quinlan [28]. Around the same time, group of statisticians (L. Breiman, J. Friedman, R. Olshen, and C. Stone) developed a new decision tree algorithm called as *Classification and Regression Trees (CART)* [29]. Albeit being developed independently, all these algorithms yet follow a similar approach to learn decision trees.

All three algorithms employ a greedy (also known as non-backtracking) strategy where decision trees are built in a top-down induction and recursive partitioning of data in a divide-and-conquer way [23]. The algorithm proceeds by the recursive partitioning of the training set by selecting a best split candidate or an attribute that maximises the homogeneity of the target labels of training subsets at each child node. In other words, it maximises the value addition to the prediction.The measure of homogeneity or purity of target labels for a set of training tuples at child nodes can be expressed by various statistical properties knows as gini index, entropy and variance. They are the heuristics for splitting criteria. Gini index and Entropy are strictly used for classification trees whereas variance is used for regression trees. When we use gini index, the tree is strictly binary but entropy is able to output multiway trees. Information gain refers to the difference between the purity of child nodes and the parent node. So an attribute is considered as a best split candidate when it maximises the information gain. The node is then labeled with this best split attribute and branches are grown out of this node. If the splitting attribute is a discrete value, a branch is created for each known unique value of this attribute. The training data is partitioned by putting tuples having the same value of the attribute to the corresponding branch. If the splitting attribute is continuous-valued, there are two possible outcomes each corresponding to the attribute values below and above the splitting point. This splitting point is implicitly calculated when calculating best split candidate. In practice, the splitting point is often taken as the midpoint of two known adjacent values of the continuous attribute and therefore may not actually be a preexisting value of the attribute from the training data [23]. The recursive partitioning terminates when all the training tuples in the node have same class labels (completely homogeneous) or there is no significant information gain or there are no remaining attributes on which data can be partitioned. In the end, a decision tree is returned.

Classification and regression trees work in a similar way but they differ in the use of impurity measures: classification use gini and entropy while regression use variance. Moreover, they also differ in the representation of the leaf nodes: classification tree uses the most commonly occurring class label as a leaf node whereas regression uses the average value of all the target values in the leaf node. However, decision tree learning algorithms might suffer from overfitting problem and create over-complex trees that do not generalise properly due to some anomalies in the training data such as outliers or noise. This problem of overfitting is rectified by a pruning approach where pruned trees tend to be smaller and less complex. There are two types of pruning: pre-pruning and post-pruning. Pre-pruning halts the construction of tree at earlier stages. This can be done by specifying the maximum depth of the tree or by specifying a minimum threshold for information gain. If the information gain is below this threshold, the leaf nodes are constructed. In this case, leaf nodes contain the most commonly occurring class labels or the average value of target variables. Post-pruning removes sub-trees from a fully-grown tree by replacing them with a leaf node which is labeled with the most frequently occurring class among the subtree being replaced. The criteria for selecting sub-trees to be pruned is the error rates of prediction and the improvement in this error rate after pruning the tree.

3. MapReduce Concepts

Google MapReduce [15] was the first successful attempt to implement distributed computing on a cluster of low-end cheap commodity servers while hiding the system level details from the programmers. Programmers need to focus only on what computations are to be performed and they don't need to bother about how computations are actually distributed. The basic idea behind MapReduce is to move the processing closer to the data and to spread the data across the various nodes in the cluster. This is in contrast to the conventional data processing models where data is moved to the processing nodes. The complex task of managing the data is done automatically by the underlying distributed file system called GFS (Google File System). Nowadays, the term MapReduce can refer to three different but related concepts. First, it can refer to the distributed programming model, second it can refer to the runtime environment that facilitates the distributed computations and lastly it can refer to the software implementations like Google, open-source Hadoop, MapR, Apache Spark etc. The first concept is relevant for our work. The term MapReduce refers to distributed data processing model rather than to a specific implementation for our work.

3.1 MapReduce Paradigm

MapReduce programming model derives its inspiration from functional programming models like LISP and ML. The key concept of functional programming is that the higher-order functions can accept other functions as arguments. Just as *map* and *reduce* functions in the functional programming languages like LISP, we have two stage processing structure in MapReduce. First stage is a embarrassingly parallel *map* step which is basically a *transformation* stage where it reads and processes discrete chunks of data called as 'splits' in parallel. The second stage is called *reduce* stage where the sorted output from the map step is aggregated and stored back to the permanent storage. So the process of converting complex algorithms into equivalent MapReduce algorithms is to express all the computations (those that can be parallelised) in the original algorithm using map and reduce functions only.

3.2 Pre-Requisites

The first and the foremost task is to determine if a particular algorithm can be parallelised and implemented using MapReduce paradigm. CHU ET AL. have given a generalised approach to test whether a particular algorithm can be adopted to MapReduce. They have proved that any algorithm that satisfies the *"Statistical Query Model"* can be expressed in a summation form. Moreover, MapReduce is one of the easiest way to express this summation form and write the corresponding program. Algorithms that compute enough statistics or gradients satisfy this model. They have shown that if the summations are batched, it can be easily implemented using two stage process: first stage to compute the statistics and second stage to sum all of them [13]. Now, we will enumerate some of the general restrictions with respect to the input data to the MapReduce processing. Data should have "Write Once Read Many" (WORM) characteristics because such data fulfil the

necessary condition for being processed in parallel fashion without the need for mutexes. The tuples in data should not have any dependencies on other tuples so that out-of-order processing is possible. In order to fully exploit the potential of MapReduce, data-set should be very large because MapReduce is an execution framework that has been designed for processing large-scale data. MapReduce is a partial parallel algorithm because the input of the map and reduce should always be of the form `<key, value>`. This key-value structure is imposed on the input when the data tuples are read from the underlying special Distributed file systems such as HDFS, GFS by the MapReduce run-time.

3.3 Architecture And Programming Model

The architecture of MapReduce programming model, as proposed in [15], is of master-slave paradigm. One of the nodes is a special node: master node. It has the responsibility of assigning map and reduce tasks to the slave nodes, monitoring task progress on each of the slaves nodes, periodically detecting slave node failures and restarting failed map and reduce tasks preferably on alive slave nodes which have the replicated data. Slave nodes only run map or reduce tasks and periodically inform the progress of the task completion back to the master node. Figure 6 shows the original architecture of MapReduce and its execution flow. The client program forks a master process on the master node and slave processes on other slave nodes. From this point, master node assumes the responsibility of scheduling the map and reduce tasks on slave nodes. Slave nodes execute the map and the reduce tasks which are assigned to them by the master node. Before the reduce tasks can be started, all the map tasks should be finished. This can be viewed as a synchronisation step between map and reduce tasks. Map tasks send the output to the reduce tasks by using a distributed shuffle and sort operation. Combiner and partitioner steps are optional and are basically optimisations. Combiner reduces the size of map outputs by performing the local reduce. Combiner reduces the size of shuffle data between map and reduce tasks. Partitioner divides the map output into number of reducers. Various kind of partitioning schemes can be employed like hash partitioning. In reduce step, all the values with same key are combined and the merged results are stored back to the DFS storage. The reduce results are first sorted by their keys before they are stored to DFS.

The key aspect of the MapReduce programming model is that if every Map and Reduce is independent of all other ongoing Maps and Reduces, then the operation can be run in parallel on different keys and values [30]. One of the most important aspect of MapReduce processing model is that it runs the map tasks on the nodes where the data lives. This ensures that there is no or very little movement of data between nodes and preserves the valuable network bandwidth. This property is called as *data locality optimisation*. All the independent computations need to be specified in 2 kinds of functions: **map** and **reduce**. We will now give a general definition of map and reduce transformations:

Map. A map transform is provided to transform an input data row of key and value to an output list containing zero or more (key,value) pairs: **map(key1,value) -> list<key2,value2>**
Reduce. A reduce transform is provided to take all values for a specific key, and generate a new list of the reduced output: **reduce(key2, list<value2>) -> list<value3>**

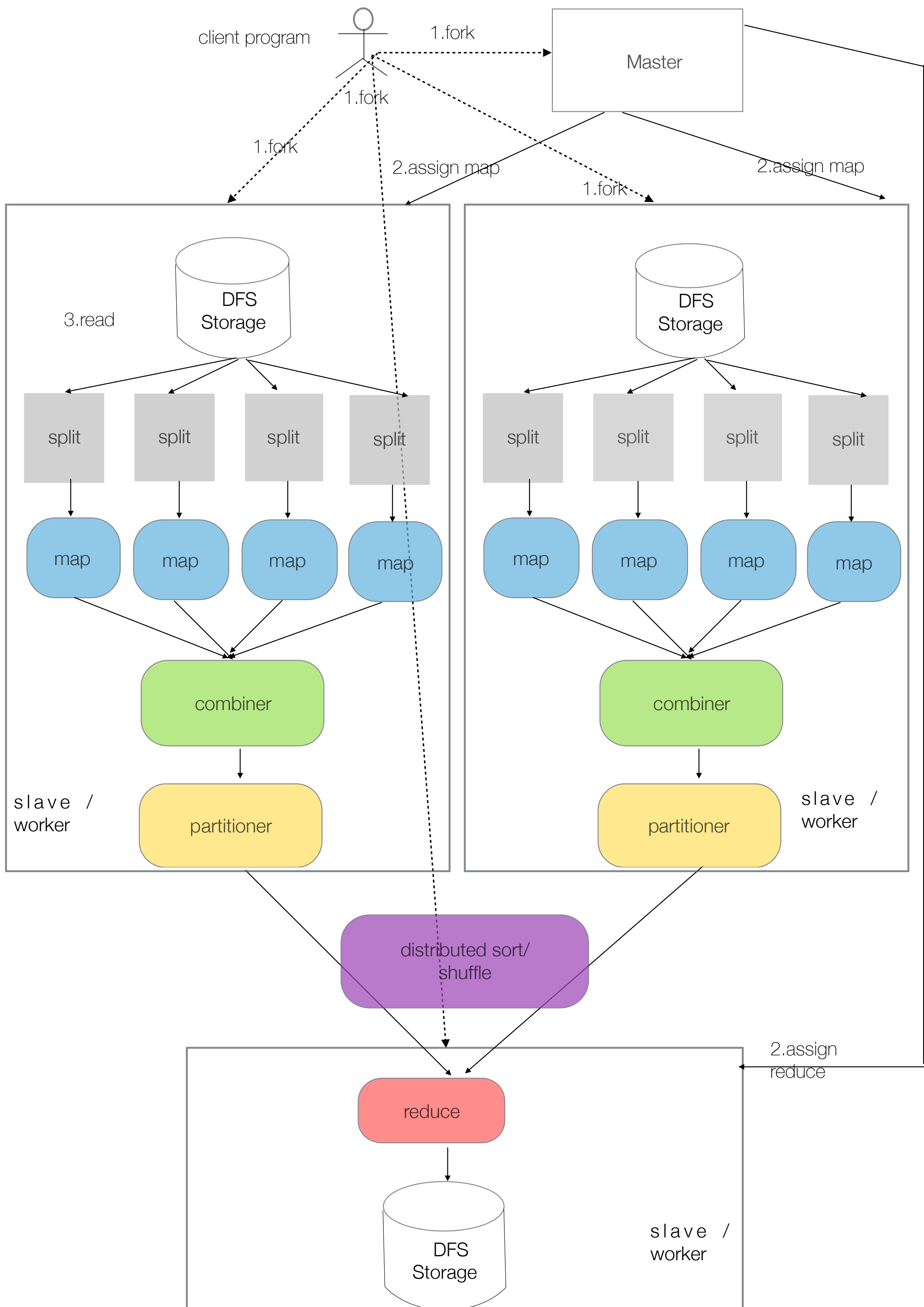


Figure 6. Overall MapReduce programming model architecture [15]

4. Hadoop

This section gives an overview to Hadoop, an open-source project under Apache Software Foundation and distributed under Apache License 2.0 [4]. It is necessary to understand the basics of Hadoop because we have used Hadoop based cluster to perform our experiments later. There are numerous reasons to use this framework to build the cluster. While MapReduce software framework from Google is proprietary, Hadoop is an open source, the most accessible and the most mature implementation of MapReduce. In Hadoop, data storage and computation both live on the same machines within the cluster. By leveraging this proximity of data, hadoop is capable of efficiently processing large volumes of data. Therefore, we can leverage the data mining analytics easily and cost effectively using Hadoop. Since it is written in Java, it can run on any platform running JVM. It implies that there are no special requirements on machines that will comprise the cluster. Building up distributed systems requires coordination of large number of machines, large-scale data storage and analysis and Hadoop provides very simple and easy-to-use tools for that purpose. All the modules in Hadoop framework are designed to automatically handle hardware failures. This fault-tolerance is made possible by the automatic and efficient data replication in Hadoop. There are 2 basic modules in Hadoop which are essential to understand for this work: Hadoop Distributed File System (HDFS) module for distributed data storage and MapReduce module for distributed data processing. We have used HDFS for storing large-scale data in our experiments. Although, we will see later that we have not used the MapReduce processing framework of Hadoop, we need to understand this original and the most mature MapReduce framework because it is the reference framework for other advanced and optimised MapReduce frameworks. It is of immense help to understand the fundamentals of Hadoop's MapReduce processing, so that we can later compare this with the optimised MapReduce framework (which is Apache Spark) that we have used for our work. We need to have a basic idea about the original MapReduce so that we can comprehend the optimisations that are implemented in other MapReduce-like frameworks.

Hadoop has namely two major versions: Hadoop 1.x and 2.x. The 2.x version is scalable up to 10,000 nodes whereas 1.x version is scalable only up-to 4000 nodes because the newer version has better cluster management capabilities and provide fine-grained memory management capabilities. 1.x uses slot-based mechanism for memory allocation and it leads to under-utilisation of overall cluster memory. 2.x uses container-based memory allocation which is more flexible. Nevertheless, both versions have improved tremendously in terms of scalability, reliability, manageability and performance over incremental releases.

4.1 HDFS - Hadoop Distributed File System

There are several distributed file systems that has been in use for several decades, then *why do we need a special DFS like HDFS?* Let us take an example of the most ubiquitous file system: NFS (Network File System) [17]. It has been so popular because of its major advantage being the location transparency of files. All the file commands for local filesystem also works on files hosted on NFS as if they are locally hosted. But there are several limitations of NFS. The first problem is that all files belonging to one NFS volume should

reside on a single machine. Thus the size of single machine is the upper limit of the size of a NFS volume. If there are large number of clients requesting files from this single NFS volume, the machine will be probably overloaded. Moreover, it does not provide any out-of-the-box reliability guarantees because there is no replication supported. The primary requirement is that distributed File system should tolerate node failures without any data loss. Now we will see how the unique design of HDFS that solves all the shortcomings of NFS.

4.1.1 Design of HDFS

The architecture of HDFS is described in the original paper from Yahoo! Inc. [31], which is based on the design of GFS [32]. HDFS can store large amounts of data (in order of terabytes or petabytes) and the size of files is not limited by the size of single DFS machine because it stores files by breaking them into blocks of fixed sizes (128 MB by default) which might typically reside on different machines across the cluster. It also provides automatic fault-tolerance and handles the node failures by replicating the blocks on various machines in an efficient manner. The size of the blocks is usually very large as compared to traditional filesystem blocks so that seek times are much smaller as compared to time for reading data from the disks. Therefore, it gives very high throughput and performance in case of applications that require long sequential reads from the disks. It is also designed to use commodity disk drives because it provides automatic fault tolerance in case of disk failures. Hadoop has the fundamental flexibility to handle unstructured data regardless of the data source or format. Additionally, disparate types of data stored on unrelated systems can all be deposited in the Hadoop cluster without the requirement to predetermine how the data will be analysed. But the design of HDFS which makes it highly scalable and gives very high performance also makes it limited to a particular class of applications. It is specially suitable for data which has Write Once and Read Many (WORM) and strong independent properties. Applications which require very instant seek accesses are not suitable for HDFS storage. But the *data locality optimization* of HDFS makes it very suitable for any kind of MapReduce applications. Data locality optimisation implies that the blocks are stored in such a manner that there are minimal movements of blocks between machines belonging to the same rack and even lesser block movements between machines in different racks. HDFS follows a rack-aware architecture.

In Figure 7, we illustrate the architecture of HDFS using an example where there is one master node and 3 slave nodes.The architecture of HDFS is based on master-slave design [33]. The master node is called *namenode* whereas the slave nodes are called *datanodes* [34]. *Namenode* holds the filesystem tree and its metadata persistently on a local disk but it stores the metadata about block locations in memory. Therefore, namenode needs typically more memory than data nodes. Large size of HDFS blocks also means that there will be relatively low amount of metadata per file as the memory needed by namenode is typically determined by total number of blocks in the HDFS. Datanodes manages, store and retrieve blocks in a HDFS directory as directed by the client or namenode. In Figure 7, we show that a read or a write client can do the respective operations on any slave or master node but it first needs to consult the namenode. In case of read operation, the read client needs to fetch the metadata information about the physical location of the blocks and their replicated

versions on data nodes from namenode. In case of a write operation, the write client informs the namenode about its intention to write a block of a file. Namenode replies back with the address of the datanode where the requested block can be stored. Write client writes the block to the specified datanode. Datanodes are then responsible for replicating the particular block by a replication factor value specified for the HDFS system. In the given example, the replication factor is 2. This cycle repeats for next blocks until all the blocks of file are written to the HDFS ensuring that the required replication factor is always fulfilled.

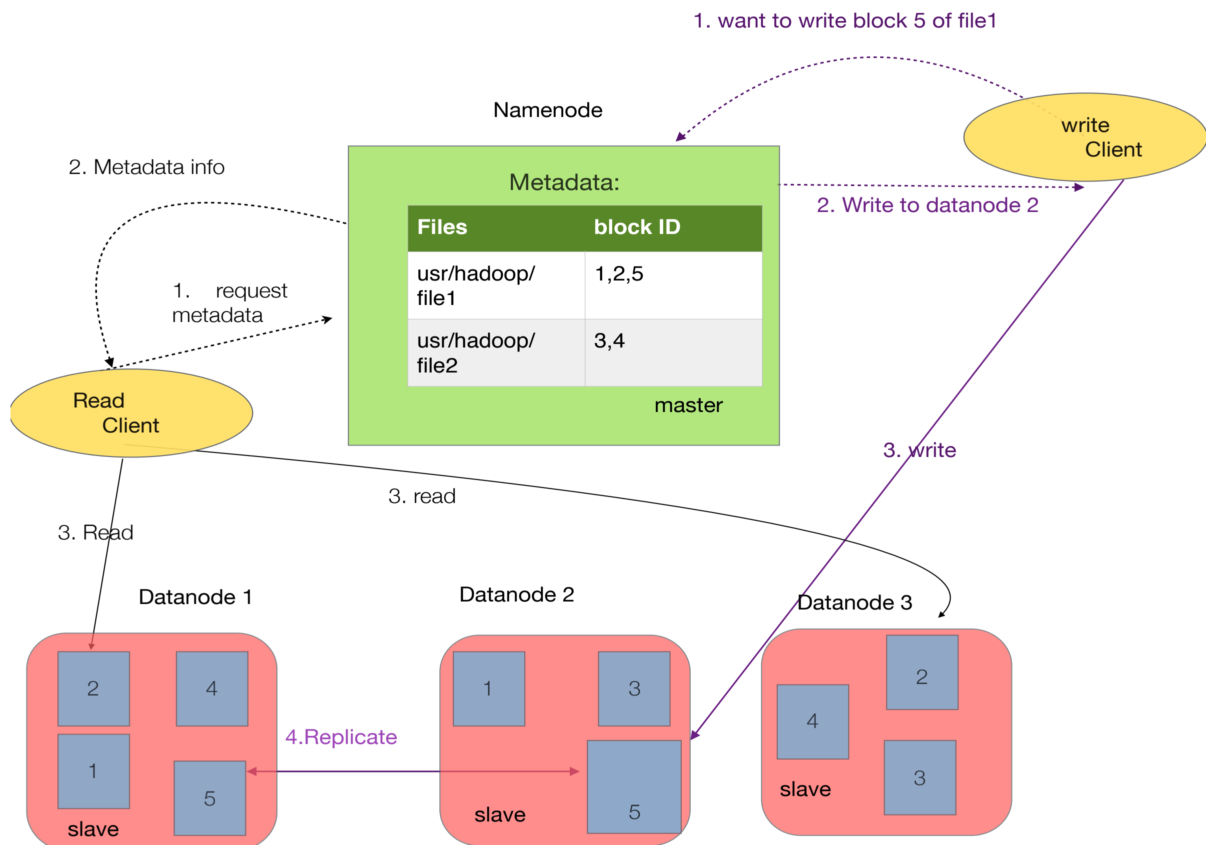


Figure 7. HDFS architecture and the flow of execution for read and write operations [34]

4.2 MapReduce Framework

In this sub-section, we develop a comprehensive understanding of fundamentals of MapReduce processing framework [35]. This specifically refers to the original implementation by Hadoop. We will describe only the important characteristics of this framework which is important for building a concept about MapReduce distributed programming model. In Hadoop, a *MapReduce job* is the highest unit of abstraction of work in Hadoop and it is submitted by a *job client*. The *job* consists of three minimal components: specification of input and output paths, definition of *map* and *reduce* tasks and various other configuration parameters. For this purpose, **JobConf** is the primary

interface for a user to describe and configure a MapReduce job. MapReduce execution framework also works in the master-slave fashion. The master-slave architecture is completely different in two versions of Hadoop. So, we will be using the more generic term master and slave for this section and we will see their specific implementations and their respective names in different versions in the sub-section 4.3 and 4.4. Hadoop run-time is responsible to divide a job into tasks which can be only of 2 types: *map* and *reduce*. For this, users are required to define a **Mapper** class that implements the map function and a **Reducer** class that implements the reduce function. Here, we will give a foundational concept of map and reduce which is valid for both versions of Hadoop. It gives us a general idea about how map and reduce actually distribute the computations and what characteristics yield performance and throughput benefits.

4.2.1 Map

Hadoop divides the input data into input splits and creates one map task per input split and processes the list of `<key1, value1>` entries in the split one by one to produce an intermediate list of `<key2, list of value2>`. In map stage, user need to define some classes: **InputFormat, InputSplits, RecordReader, Mapper, Combiner** and **Partitioner.**

InputFormat is a class that defines the source of input files, defines **InputSplits** to break input and provides a factory of **RecordReader** objects to convert binary data from **InputSplits** into meaningful `<key, value>` tuples per record.

InputSplits defines a single unit of work for a single map task. Therefore, the total number of map tasks is equal to the number of input splits. This is not user configurable because it depends only on the split size. Smaller splits means less time to process the splits concurrently as compared to larger ones. Smaller splits also enable better load-balancing because faster processor can process more splits than slower machines within the same time frame. Split size should not be too small, otherwise overhead of managing the splits and time to create map tasks is much higher than the map task execution time. In most scenarios, a fair split size is one HDFS block because if input split contains several HDFS blocks, it is unlikely that all HDFS blocks is present on the same node. Some of them will need to be transferred to the running map task and therefore will consume the cluster bandwidth. This goes against data locality optimization efforts of Hadoop to conserve the network bandwidth. This shows that one DFS block size is the right size for input split. If data-local map task is not possible, Hadoop MapReduce tries to execute map task on a node in the same rack (rack local). Off-rack map tasks are spawned when not possible in the same rack by transferring the split over the network.

RecordReader recursively creates `<key, value>` pairs from InputSplits until an entire split has been consumed. Map tasks are called for every invocation of RecordReader and all the intermediate values associated with a given output intermediate key are subsequently sorted and grouped by the framework.

Combiner (which is a optional stage) attempts to reduce the size of data shuffled between map and reduce tasks. It runs the *reduce* function on the map output locally on the node. Since, it is just an optimization, Hadoop does not guarantee the number of times combiner function might be called if at all.

Partitioner partitions the intermediate key space per Reducer. The total number of partitions is the same as the number of reduce tasks for the job.

The output from the local buffer is finally written to a local file not to the HDFS because it is just an intermediate output which is trashed when read by the reducer.

4.2.2 Reduce

Reducer reduces a set of intermediate values which share a common key to a smaller set of values. The number of reduce tasks is a user controllable parameter unlike the map tasks. When number of reduce tasks is high, the shuffle is more complicated and will consumes more bandwidth. While lesser number of long running reduce tasks will negatively impact the overall execution time due to lesser degree of parallelism. Reducer has 3 primary phases: *shuffle, sort* and *reduce*. *Shuffle* is the process of moving relevant partitions of map outputs to the reduce tasks using HTTP. The set of intermediate keys on a single node is automatically *sorted* by Hadoop before they are presented to the Reducer. A Reducer instance is created for each *reduce* task. It merges the values which have the same key and the output is written back to the HDFS as defined by **OutputFormat**.

Most of the applications especially for machine learning and data mining require several parameters to be passed to the MapReduce tasks. If the parameters are read-only large files, they must be distributed by **DistributedCache**. Small parameters are passed by using the setter and getter functions in **JobConf** class.

In sub-section 4.3 and 4.4, we will see the differences between the two versions of Hadoop. There is very minor difference between HDFS in two versions and it does not need any further elaboration. The architecture and job execution flow in MapReduce has undergone a complete overhaul from first to second version. Although, we will be using MRv2 because of its numerous advantages, it is worth to see both versions for a comparison.

4.3 Hadoop MapReduce 1.x (MRv1)

Hadoop MapReduce Software component has three major facets:

- The end-user MapReduce API for programmers and users.
- MapReduce run-time which is the actual implementation of various phases of MapReduce applications.
- MapReduce framework which is the back-end infrastructure to run the MapReduce job, manage cluster resources, schedule queues of numbers of concurrent jobs, provide fault tolerance etc.

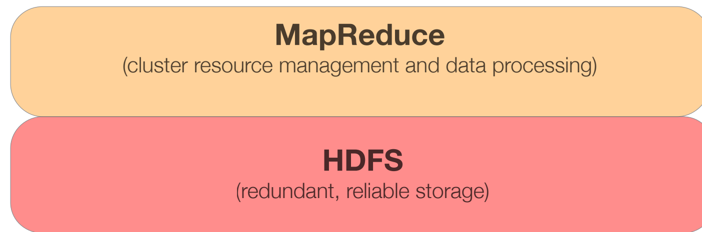


Figure 8. Hadoop MapReduce version 1.x (MRv1)

In Figure 8, we can see that in MRv1, there are only two abstract layers: HDFS and MapReduce. The lower layer facilitates a reliable and fault-tolerant distributed data storage whereas the upper layer is responsible for both cluster resource management and distributed data processing in MapReduce fashion.

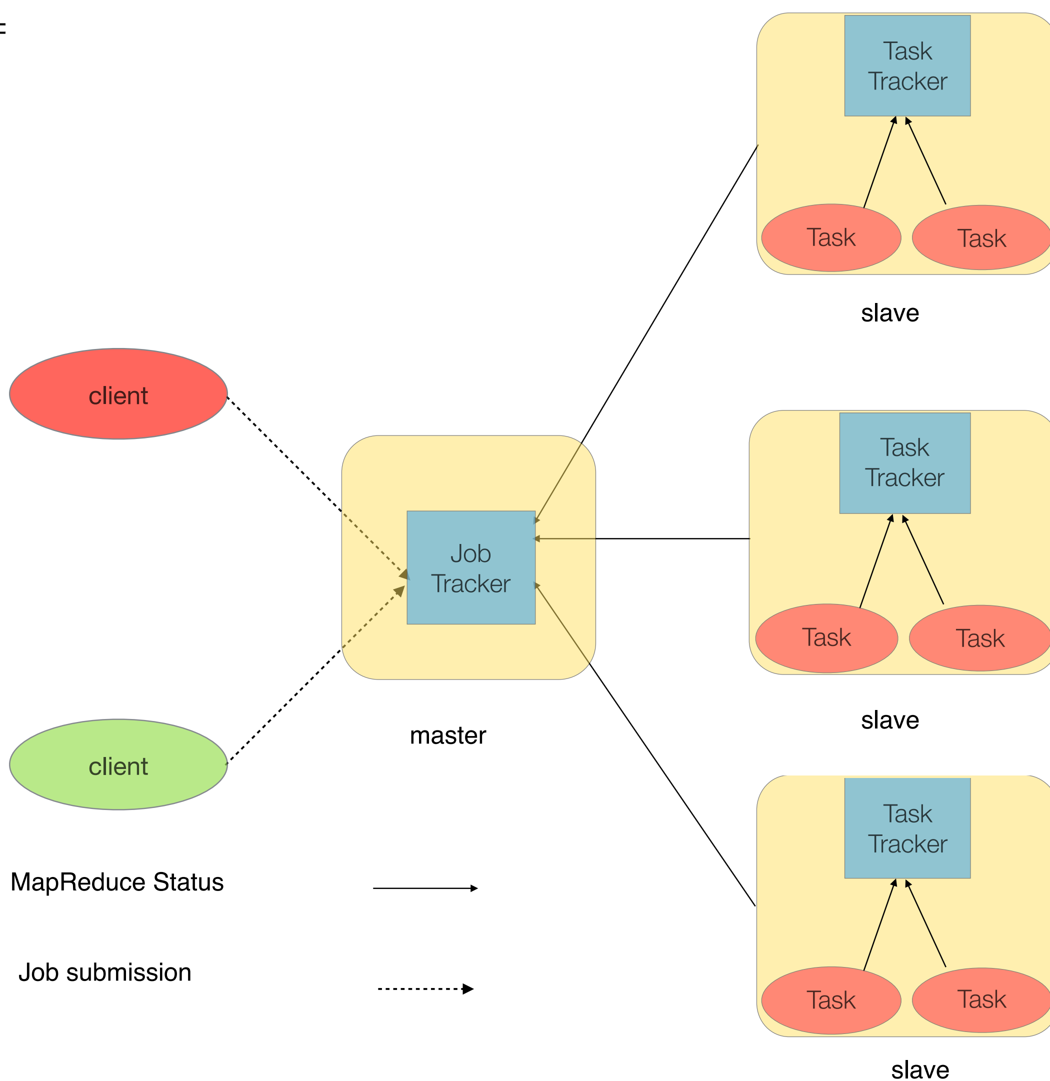


Figure 9. Architecture and job execution flow in Hadoop MapReduce version 1.x (MRv1)

In Figure 9, we show the architecture of MapReduce framework specific to this version. We also illustrate the specific daemons running on master and slave nodes which form the fabric of MapReduce distributed processing for this version. The master node is controlled by a ***JobTracker*** daemon, that farms out and schedules the MapReduce tasks to specific nodes in the cluster, ideally the nodes that have the data, or at least are in the same rack [35]. Slave nodes are controlled by a ***TaskTracker*** daemon that manages the map and the reduce tasks on the individual node [35]. The ***TaskTrackers*** communicate with and are controlled by the ***JobTracker***. The ***JobTracker*** is a single point of failure for the Hadoop MapReduce service.

The responsibilities of ***JobTracker*** are resource management (which implies monitoring the task trackers on worker nodes), tracking resource availability and consumption and job life-cycle management (for example, scheduling individual tasks of the job, tracking process, providing fault tolerance, etc). Whereas ***TaskTrackers*** have very simple responsibilities to launch the map and reduce tasks ordered by the ***JobTracker*** and inform the task progress status information periodically back to the ***JobTracker***.

4.4 Hadoop MapReduce 2.x (MRv2) or YARN

Hadoop YARN is an attempt to enable other data-processing models beyond MapReduce on the Hadoop clusters. Job scheduling and running the MapReduce applications still remains in the MapReduce component but the cluster resource management is moved to a new component called YARN [36].

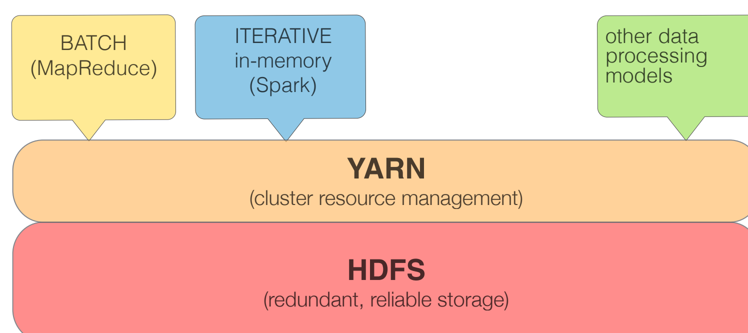


Figure 10. Hadoop MapReduce version 2.x (MRv2) or YARN

In Figure 10, we can see that there are 3 abstraction layers in Hadoop MRv2: HDFS, YARN and data processing layer. Cluster resource management has been separated from data processing so that YARN only acts a cluster manager and can support a variety of data processing models like batch, iterative, interactive, streaming and so on.The major redesign of MRv2 is to split up the two facets of the ***JobTracker*** daemon: global resource management and application-specific job scheduling, into separate daemons: global ***ResourceManager*** and per-application ***ApplicationMaster***. The ***ResourceManager*** is the central arbitrator of all cluster resources (cpu and memory) among the competing applications. Container is an abstract notion for representing a collection of physical

computational resources like CPU cores, disks and memory. Yarn uses container-based allocation of resources. Per-node ***NodeManager*** daemons run on slave machines. They are responsible for allocating containers as scheduled and requested by the ***Resource Manager,*** managing and monitoring the resource usage of all the containers on the single machine and reporting back the monitored values to the ***Resource Manager***. Per-application ***ApplicationMaster*** is a special container which acts as application master for a specific application launched. The ***ApplicationMaster*** is a data-processing framework-specific container that negotiates resources from the ***ResourceManager*** and works with the NodeManager(s) to execute and monitor the component tasks. This means that the ***ApplicationMaster*** container for Spark will be different from Hadoop MapReduce.

The ***ResourceManager*** has two main components: *Scheduler* and *ApplicationsManager*. The *ApplicationsManager* is responsible for accepting job-submissions, negotiating the first container for executing the application-specific ***ApplicationMaster*** and provides the service for restarting the ***ApplicationMaster*** container on failure. The *Scheduler* is a pure scheduler which is responsible for allocating cluster resources to the various running applications subject to familiar constraints of capacities, queues etc. The *Scheduler* performs its scheduling function based the resource requirements of the applications and schedules the resources in the form of containers.

YARN – Walkthrough

In Figure 11, we give an illustration about the general execution sequence in YARN which is explained in the following steps:

1. A client program submits an application jar (where configurations and compiled code are packaged together) to ResourceManager.
2. The ResourceManager requests a container to start an ApplicationMaster from any NodeManager.
3. The ApplicationMaster registers itself with the ResourceManager and provides the necessary information.
4. After reading the configurations from jar, the ApplicationMaster negotiates appropriate resource containers from ResourceManager via the resource-request protocol.
5. The ResourceManager requests the required number of containers from NodeManagers.
6. NodeManagers launch specified containers as requested by the ResourceManager.
7. NodeManagers return the container IDs back to the Resource Manager.
8. Resource Manager sends the list of Container IDs to the ApplicationMaster.
9. Application master communicates with launched containers and the application code executing within the containers via an application-specific protocol.
10. During the application execution, the client that submitted the program communicates directly with the ApplicationMaster to get status, progress updates etc. via an application-specific protocol.
11. Once the application is complete, and all the necessary work has been finished, the ApplicationMaster deregisters with the ResourceManager and shuts down, allowing its own container to be repurposed.

Figure 11. Architecture and job execution flow of Hadoop YARN

4.5 Why YARN?

In this sub-section, we give proper arguments for selecting Hadoop YARN as the underlying cluster for our experiments. Hadoop YARN provides numerous benefits and addresses various drawbacks of the first version of Hadoop. First of all, Hadoop YARN clusters are more scalable than clusters built from previous version of Hadoop. How YARN is more scalable? Resource Manager which is the central manager for all cluster resources does not have to deal with other specifities of MapReduce like providing application-specific fault tolerance. The previous version of Hadoop MapReduce JobTracker spends a very

significant portion of time and effort managing the life cycle of applications. Thus, it becomes overloaded as the size of cluster grows and becomes a major cause for software mishaps and breakdowns. YARN supports data-processing models beyond batch mode MapReduce which are optimised for iterative, interactive or graph jobs. Specifically for our work, YARN is suitable because it can handle iterative kind of workloads for data mining. Additionally, Apache Spark (introduced later), which we have used for MapReduce data mining, is capable of running very efficiently on YARN. Another interesting aspect is to analyse the resource allocation models in both versions. In previous version of Hadoop, each node was statically assigned a fixed number of slots for map and reduce tasks. These slots could not be dynamically shared between map and reduce tasks. But MapReduce job execution flow shows that there is a more demand for map slots in the beginning and reduce slots at the completion of the job. The static allocation of slots in the previous version led obviously to the under utilisation of the cluster resources. YARN addresses this wastage of cluster resources in the old version of Hadoop by providing a more flexible resource allocation model. Resources are requested in the form of containers. They have non static attributes because they have maximum and minimum defined memory limits. ApplicationMaster can obtain only containers in the multiple of the minimum until the maximum is reached.

4.6 Conclusion

The strict batch job scheduling of map and reduce tasks in Hadoop MapReduce is very restrictive and not suitable for other data-processing workflows. Map and reduce tasks must be written as acyclic data flow programs, i.e., a stateless mapper followed by a stateless reducer, that are scheduled in a batch fashion [37]. As a consequence, it introduces high-latency to the programs and falls short of real-time data analysis. MapReduce paradigm also hinders the repeated querying of datasets and makes them painfully slow. Such limitations become more obvious in fields such as data mining, where it is very typical that iterative algorithms will revisit a single working set multiple times. In Hadoop, reduce tasks can not take place until all maps tasks have completed (or have failed and been skipped). As a result, we do not get any data back until the entire mapping has finished.

5. Apache Spark

Although the original MapReduce provided scalability, fault tolerance and throughput, it suffered from the problem of latency due to the strict batch processing architecture. Original MapReduce executed jobs in a simple but rigid structure: a processing or transform step (“map”), a synchronization step (“shuffle”), and a step to combine results from all the nodes in a cluster (“reduce”). Most of the data mining algorithms are highly iterative in nature and to port these algorithms to MapReduce framework, the only way was to string together a series of MapReduce jobs and execute them in a sequence. Each of those jobs was of high-latency because they had to read the data from the local disks, and none of them could start until the previous job had finished completely. Therefore, such complex, multi-stage applications are distressingly slow because of the non-trivial MapReduce run-time overhead in every iteration.

Apache Spark tries to relax the rigid batch architecture of the original MapReduce and tries to eliminate costly snychronisation blockages. It allows programmers to construct complex, multi-step Directed Acyclic Graphs (DAGs) of work that must be done, and to execute those DAGs all at once, not step by step. Moreover, it also let the jobs to keep the data in memory for faster query and processing. Spark was purposely designed to support in-memory processing. The net benefit of keeping everything in memory is the ability to perform iterative computations at blazing fast speeds — something MapReduce is not designed to do. But it won’t be far to compare Spark with MapReduce because MapReduce is just a programming model for distributed computation whereas Spark is a general purpose cluster computing system which uses MapReduce-like parallel operations. Apache Spark is a MapReduce-like distributed data processing model which supports a rich set of distributed transformations in addition to simple “map” and “reduce”. It has following components:

- High level APIs (Application Programming Interfaces) in Java, Scala and Python.
- An optimised execution engine that supports fast distributed processing.
- A set of higher-level tools including Shark (Hive on Spark), Spark SQL for structured data, MLlib for machine learning, GraphX for graph processing and Spark Streaming.

Machine Learning Library MLlib is of special interest for our work because there is a growing number of machine learning algorithms being implemented and ready to be used.

5.1 Basics Of Spark

We know that Hadoop is the most widely used open source implementation of MapReduce which scales out computation and storage across cheap commodity servers and allows other applications to run on top of both of these — Spark is one of these applications. Although Hadoop is effective for storing vast amounts of data cheaply, the computations it enables with MapReduce are highly limited for the reasons we have seen above.

According to Spark documentation, Spark application comprises of a *driver* program that executes the user's main program and runs several kinds of parallel operations on a cluster [38]. **Resilient distributed datasets (RDDs)** forms the fundamental programming abstraction of Apache Spark. They are immutable (which means read-only) collections of records partitioned across cluster that can be operated in parallel [37a]. So how are RDDs actually created? Before, we can answer this question, we need to have the knowledge that RDDs support two types of operations: *transformations* (to create new RDDs from existing dataset) and *actions* (to compute and to output results to the main driver program after performing computations on the transformed RDDs). In Table 1, we have given some examples of transformations and actions which are currently provided by Apache Spark. RDDs are created by starting with a source file stored in HDFS (or any other Hadoop-supported file system), or an existing Scala collection (which we do not use for our work) in the driver program, and applying any kind of transformations (map, filter, group-by…) on it. Spark supports text files, SequenceFiles and any other Hadoop InputFormat.

When the source file is HDFS, number of partitions of RDDs is equal to the number of HDFS blocks in a file. However, transformations in Spark are lazy. As such, RDDs are materialised on demand when an action is required to be employed to the RDDs and result is to be returned to the driver program. Users can explicitly cache or persist RDDs in memory across machines and reuse it in multiple MapReduce-like parallel operations. Therefore, we can control the storage level (memory, disk, etc) for RDDs. RDDs are currently cached in memory by default, but when the memory is insufficient, RDDs will spill to disk. In case of spill to disk, there will be a degradation in the performance of Apache Spark but not worse than current MapReduce. In our experiments, we will also analyse this correlation between the size of data and the performance output.

RDD Transformations (define a new RDD)	RDD actions (compute and return the result to driver program)
map filter sample groupByKey reduceByKey sortByKey flatMap union join cogroup cross mapValues	reduce collect save lookUpKey count

Table 1. RDD operations

Resilient distributed dataset uses a clever way of guaranteeing fault tolerance that minimises network I/O. The Spark academic paper [37] says, RDD achieves fault tolerance through a notion of lineage. If a partition of an RDD is lost, the RDD always tracks the graph of transformations that is used to build it in order to be able to rebuild just that partition. This implies that lost partitions can be reconstructed without needing replication to achieve fault tolerance. Hadoop, on the other hand, uses replication to achieve fault tolerance.

The second abstraction of Spark is **shared variables** which is passed to the functions that invoke parallel operations. Spark supports two kinds of shared variables: *broadcast variables*, which is used to copy a large read-only parameter to all the nodes just once, and *accumulators*, which are variables that are added to associative operators like counters and sums which are specifically used by the driver program.

In order to understand the abstract programming model of Spark, we will use an example as shown in Figure 12. This gives an in-depth explanation of RDD and its parallel operations. This pseudo code of Spark application performs log mining to count error messages containing specific errors. The first line of the pseudo code refers to reading a text file from HDFS. This is actually the base RDD where each HDFS block forms a partition of RDD. Second and third lines intend to perform transformations to get new RDD. But these transformations are rather lazy and are not performed until actions are invoked. Fourth line again stores an intention to cache the calculated RDD after invoking action. Until now, there has been no reading of data and any kind of computations in the program. As soon as the first count action is invoked, the partitions of base RDD, each corresponding to one HDFS block, are read in parallel manner. Then filter and map transformations are applied to get the final transformed RDD on which count action is ready to be invoked. Since fourth line gives the intention to cache the RDD, they are persisted in the memory and the second count action is applied on the cached RDD. If any partition of RDD is lost, it tracks the lineage information: read data from HDFS, filter, map and cache to rebuild the lost partition.

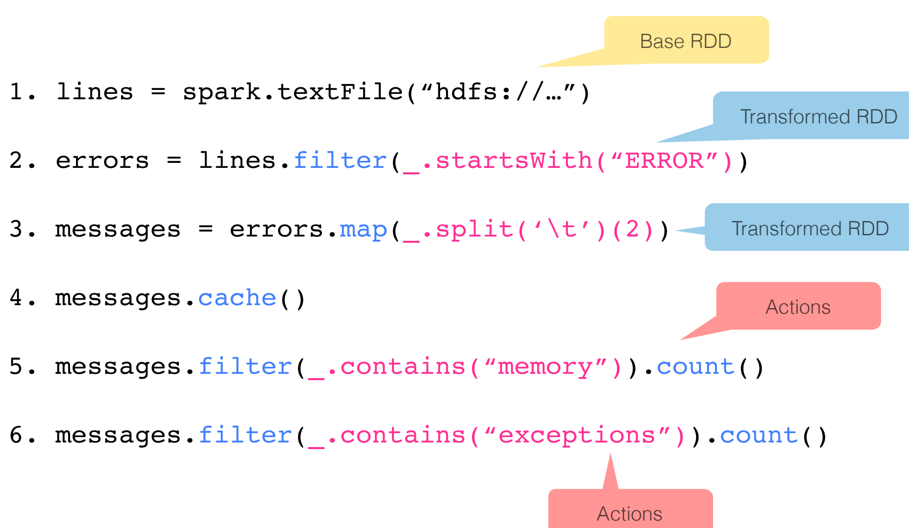


Figure 12. An example for Spark program for log mining

Additionally, we also need to know the internal representation and the building blocks of RDD. RDD contains following components: set of partitions ("splits"), list of *dependencies* on parent RDD, function to *compute* a partition given parents, optional preferred locations and optional partitioning info (Partitioner). Since we are using Hadoop data for our experiments, we need to see the internal representation of Hadoop RDD. In case of Hadoop RDD, there is one *partition* per HDFS block, there are no dependencies, there is a function to compute the partition which is reading a corresponding block, there is a preferred location which is the HDFS location and there is no partitioner which is required to be defined.

5.2 Overview Of Spark In Cluster Mode

We will give a short overview of how Spark runs on clusters, to make it easier to understand the components involved. In Figure 13, we show the mechanism of working of Spark in cluster mode. Each Spark application creates one SparkContext object which is the main entry point to the Spark functionality. It is used to specify all the application-specific configurations. SparkContext is also responsible to connect to a central cluster manager and request compute resources from the cluster. The cluster manager can be of following types in the current version: Standalone cluster from Spark, Hadoop YARN, Apache Mesos and Amazon EC2. Next SparkContext acquires executors on worker nodes. Executors are processes to run computations and store relevant data. Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads. Next, it sends the application code (defined by JAR or Python files passed to SparkContext) to the executors. Finally, the executors run the tasks submitted to them.

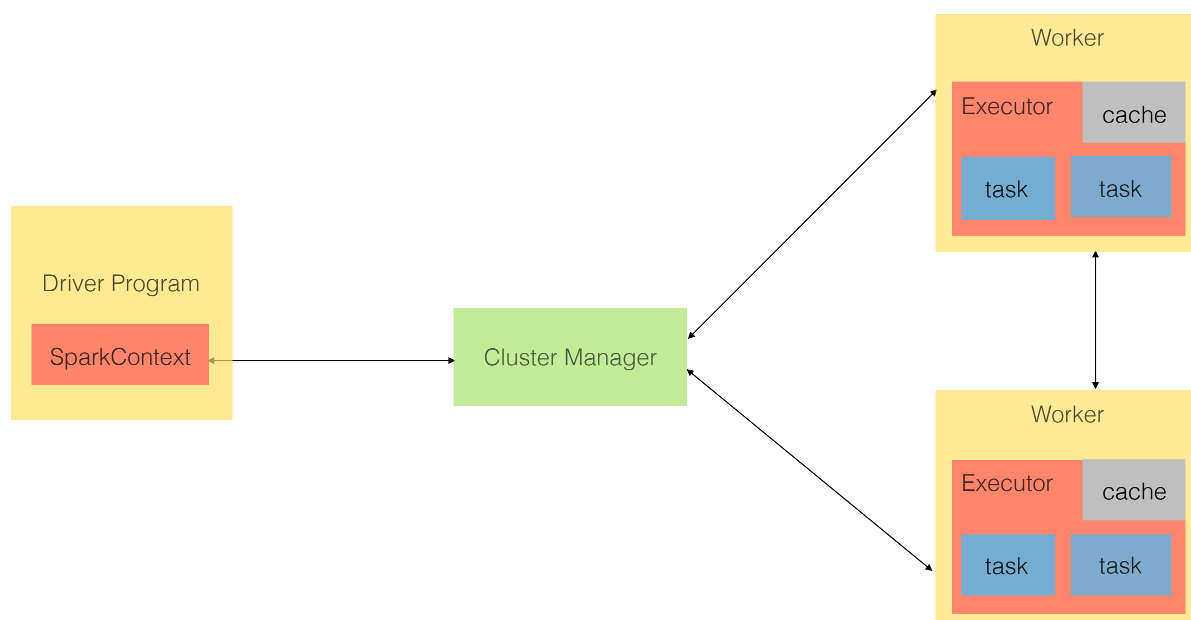


Figure 13. Architecture and job execution flow for Spark in cluster mode [39]

In order to write applications for Spark, we need to understand the underlying architecture. It comprises of following components:

Application: This is the job that user intends to perform. It may be a single job, a sequence of jobs, a long-running service issuing new commands or an interactive exploration session.
Spark Driver: The Spark driver is the process running the spark context (which represents the application session). This driver is responsible for converting the application to a directed graph of individual steps to execute on the cluster. There is one driver per application.
Spark Application Master: The Spark Application Master is responsible for negotiating resource requests made by the driver with ClusterManager and finding a suitable set of hosts/containers in which to run the Spark applications. There is one Application Master per application.
Spark Executor: A single JVM instance on a node that serves a single Spark application. An executor runs multiple tasks over its lifetime, and multiple tasks concurrently. A node may have several Spark executors and there are many nodes running Spark Executors for each client application.
Spark Task: A Spark Task represents a unit of work on a partition of RDD of a distributed dataset.

Apache Spark applications can run both in local and cluster mode. For our work, we have specifically used “Apache Spark on YARN” cluster. It is rather difficult to comprehend how Spark works on YARN. That’s why we present a simplified explanation of interaction between Spark and YARN in the following sub-section.

5.3 Spark On YARN

When running Spark on YARN, each Spark executor runs as a YARN container. But the Spark container is more efficient than the MapReduce container. In MapReduce, Hadoop schedules a container and fires up a JVM for each task. Spark hosts multiple tasks within the same container. This approach enables several orders of magnitude faster task startup time because of elimination of overhead of starting each container for a new task. The same container is reused for multiple tasks in Spark for the whole life-time of an application.

Spark supports two modes for running on YARN: “yarn-cluster” mode and “yarn-client” mode. Broadly, yarn-cluster mode makes sense for production jobs, while yarn-client mode makes sense for interactive and debugging uses where the users want to see the application’s output immediately.

In yarn-cluster mode, the driver runs in the Application Master. This means that the same process is responsible for both driving the application and requesting resources from YARN, and this process runs inside a YARN container. The client that starts the app doesn’t need to stick around for its entire lifetime.

In yarn-client mode, the driver runs inside the client program. The Application Master is merely present to request executor containers from YARN. The client communicates with

those containers to schedule work after they start. This mode allows driver to access client side resources, including inputs from users. Since the user program is the driver of the Spark job, if the client dies in this mode, the entire application exits.

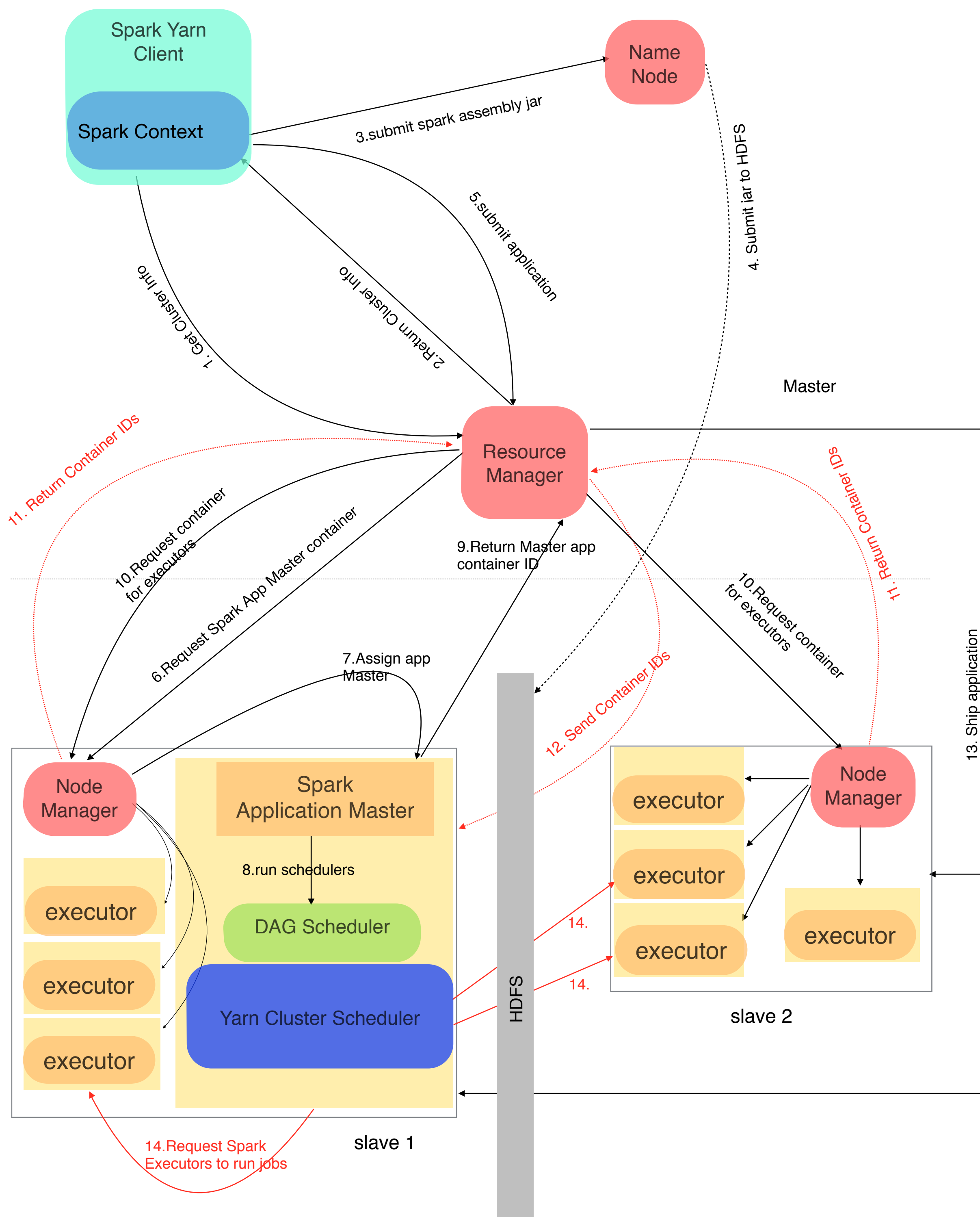


Figure 14. Spark on Yarn Execution Cycle scenario for 2 slave nodes

The study of general flow of execution in "Spark on YARN" environment is an indispensable step to understand how exactly our experiments run in this execution environment. In Figure 14, we explain exactly how Spark applications run on YARN clusters by taking an example where there is one master node and two slave nodes: The region above the grey dotted line runs locally on the master node while two boxes below the grey dotted line represent two slave nodes. The distributed nature of HDFS is depicted in an abstract view. It actually represents the local disks of two slave nodes as a single storage location (location transparency). SparkContext object is first created inside the user's main program which serves as an entrance to Spark API. Using this object, we specify various configurations for a Spark job. For Spark application to run on YARN in a client mode, the configuration parameter "master" should be specified as "yarn-client". SparkContext object uses an environment variable HADOOP_CONF_DIR of the underlying YARN cluster to contact Resource Manager. It requests various cluster information from the Resource Manager such as number of Node Managers, Name Node, data replication, etc. After fetching this information, client program submits the Spark assembly jar (packages all dependencies into a single jar file) to Name Node. Name Node assumes the responsibility to submit the submitted jar to the underlying HDFS ensuring that the required replication factor is fulfilled. The client submits a separate application jar to the ResourceManager. The ResourceManager now assumes the responsibility to negotiate and arbitrate cluster resources for executing the application code. First step is to request for a special container known as *Spark Application Master* which acts as a Spark framework specific Application Master. The Node Manager on one of the slave nodes launches this requested container with the required configuration parameters and sends back the corresponding ID of the Application master container. Now, Resource Manager requests the allocation of containers for Spark executors from Node Managers. Executors are the containers requested by Spark application where the actual parallel distributed computation will run. Node Managers return the container IDs of the launched executors to the Resource Manager. The Resource Manager sends this list of container IDs to the Spark Application Master so that it can directly communicate with the launched containers and schedule the tasks to the executors. Finally, the Resource Manager ships the application jar to all the node managers where containers have been launched and will execute the application code.

5.4 Task Scheduling Process In Spark

As we can see in the last Figure 14, Spark Application Master container runs a Directed Acyclic Graph (DAG) scheduler which is a general scheduler for tasks. Task scheduler pipelines the operations wherever possible. It is data-locality, cache aware and also partitioning-aware, the last one to avoid shuffle of data. Figure 15 shows task scheduling process in detail. The first step is to create a Directed Acyclic Graph of operations that will happen in a Spark Application. In the given figure, DAG scheduler builds a graph corresponding to join, groupBy and filter operations. For our experiments, where we run programs to implement multi-stage decision tree learning algorithms using Machine Learning library of Spark, the number of stages in DAGs specifically depends on the maximum depth of tree. Before each stage is submitted as ready, DAG scheduler needs to split each stage corresponding to one operation into a number of tasks. The number of tasks in every stage depends on the number of input partitions to the particular stage. In

our experiments, the number of tasks in a stage depends on the total number of HDFS blocks in the input file stored on HDFS. The tasks are then submitted to the worker nodes for execution via Cluster Manager. Basically, there is one Block Manager per executor and it manages the storage for most of the data in Spark, to name a few: block that represent a cached RDD partition, intermediate shuffle data, broadcast data etc. Cluster Manager also takes care of re-submitting the failed stages to the worker nodes.

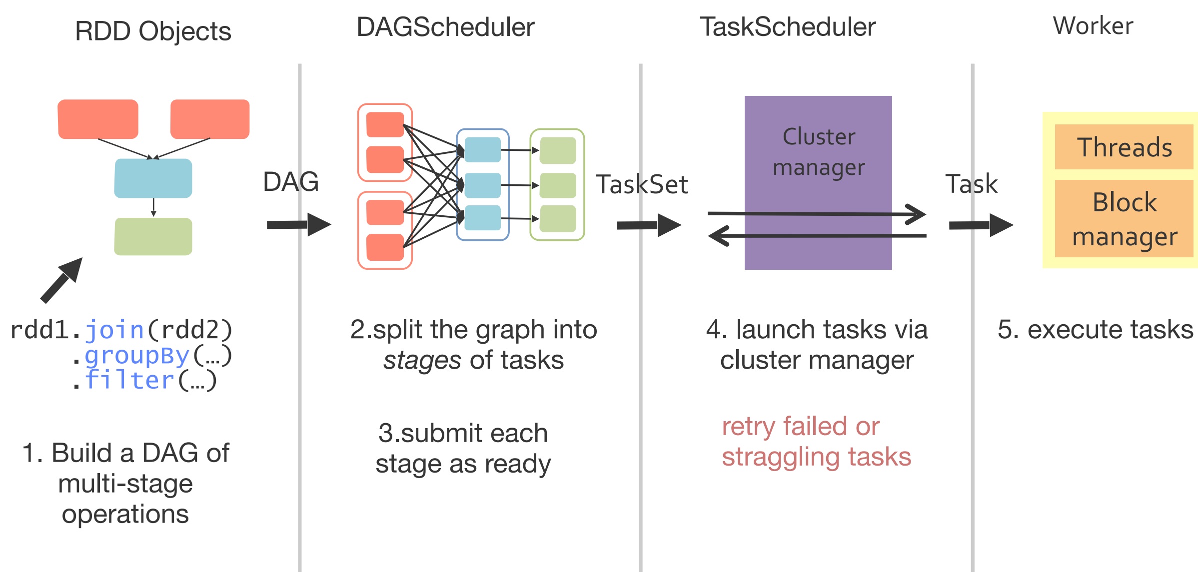


Figure 15. Task Scheduling Process in Apache Spark.

5.5 Apache Spark's Machine Learning Library (MLlib)

MLlib is a standard component of Spark which provides machine learning primitives on top of Spark's core execution engine. It is a low-level library for distributed Machine Learning on Spark and also introduces high-level machine learning programming abstractions. MLlib enables data analysers to develop machine learning programs using rich and user-friendly APIs in Java, Scala and Python. However, we have used Java APIs to write our programs that train and evaluate decision tree models based classification and regression data mining. The current version of MLlib consists of many data mining algorithms, such as, classification, regression, clustering, collaborative filtering and dimensionality reduction [64].

5.6 Decision Tree Learning Algorithms In Spark's MLlib

In data mining literature, there are two popular predictive data mining methods: classification and regression that can output the prediction models in various formats. But, in MLlib, there are rather three kinds: *binary classification*, *multi-classification* and *regression*. Spark's MLlib documentation points out that there are classification and regression data mining algorithms can be performed using naive bayes, decision trees and

linear models (SVMs, logistic regression, linear regression). As such, there are three kinds of decision tree learning based classification and regression algorithms. These decision tree learning algorithms output decision tree model which are represented as hierarchical if-else statements that the test feature values in order to predict a label. Binary classification refers to data mining method where objects can be classified only into one of two classes by using some kind of classification rule [40]. Multi-class classification is a similar procedure to classify objects into one of more than two classes [41]. Both binary and multi-class classification can predict only discrete values for the target labels. Regression refers to the classification problem where the target or the predicted values are continuous. Decision tree model can be built by using two static methods: trainClassifier(for both binary and multi-class classification trees but works internally differently for them) and trainRegressor (for regressor trees) [42],[43].

It is mandatory to understand the details of the implementation of decision tree classifiers and regressors in Spark's MLlib for our work. All the three decision tree learning algorithms work in a similar fashion and share a similar implementation. Therefore, we can give a general description for the implementation that is valid for all three algorithms. First and foremost, the parallel scaling of those algorithms is achieved by partitioning the input data horizontally by rows and processing them in a parallel distributed fashion. This implies that it performs same kind of computation on all nodes but using different subsets of datasets that have been partitioned across several nodes. The algorithms build a decision tree using level-by-level approach. The algorithms proceed by performing a recursive partitioning of the feature space [43]. This partitioning is done by choosing the best split candidate which leads to maximum information gain. The algorithms select the splits for all nodes at the same level of the tree simultaneously. This level-wise optimization reduces the number of passes over the dataset tremendously because it make one pass for each level, rather than one pass for each node in the tree. It also leads to significant savings in I/O, computation and communication overheads [44]. As we know that information gain used for selecting the best splits, is the difference between the parent node impurity and the weighted sum of the child node impurities, the current version of MLlib supports three kinds of impurities: gini and entropy (for classification) and variance (for regression). As we will see later, we have performed experiments using all three impurity measures. The recursion of algorithm terminates when the tree has reached its maximum depth or there are no split candidate that can give an information gain at the node.

We need to also understand how the MLlib decision tree algorithms calculate the split candidates for large-scale distributed data, in case where the domain of all features is continuous. We have assumed later in our experiments that all features are continuous. For small-scale data on single machine, non-distributed decision tree implementations typically use sorted unique feature values for continuous features as split candidates for the best split calculation [43]. This implies that, for continuous features, number of split decisions that legacy algorithms need to consider for a feature will be equal to the total number of instances in the whole dataset. However, finding sorted unique values is an expensive operation over a distributed dataset. Moreover, the number of split decision considerations for a feature will be extraordinarily high for large-scale data which might contain millions of instances. MLlib decision tree solves this problem by first discretising the continuous

features. The first step of MLlib decision tree algorithms is to store metadata information of decision tree such as dimension of feature, whether features are continuous or categorical, etc. Then it uses a sampled fraction of data (obtained by sampling with replacement) to calculate the quantile (distribution of values for feature in the sampled data) for each feature. This quantile calculation is also done in a parallel distributed manner on a very large number of sampled data on multiple nodes. This quantile statistics from a large number of sampled data is then collected and merged together to calculate an overall quantile statistics of data. This aggregated statistical information about the distribution of values in a feature is used to determine the boundary of bins. The maximum number of bins is a user-defined parameter (maxBins parameter). Then the binned representation of each feature is precomputed using: the `maxBins` user-defined parameter and the aggregated quantile information. The binned representation of each feature is subsequently saved on disk and memory, to be used in later stages for computing sufficient statistics for splitting. This saves computation in each iteration. As a result, in later stages, where the algorithms need to calculate the best split candidates, they know the split locations for each feature beforehand. After binning, each continuous feature becomes an ordered discretised feature with at most `maxBins` possible values.Therefore, the number of split locations considerations for each feature is drastically reduced from a unusually high value to a user-defined controllable value. This is a standard tradeoff for improving decision tree performance without significant loss of accuracy. This parameter has also impact on communications and computation time because it controls the graininess of split decisions. Higher value results into considering more split candidates. Now, we give a brief description about the process of decision tree construction. The steps are as follows:

- Master node selects a node of tree. It first starts from the root node and then goes into the deeper levels of the tree. Specifically, it selects all the nodes belonging to one level of the tree at a time.
- For the nodes selected by the master node, each worker makes one pass over its subset of instances to collect the statistics about splitting.
- For each node in construction, the statistics for that node from all workers are aggregated to a particular worker by a reduce function. The designated worker for reduce operation chooses the best (feature, split) pair, or chooses to stop splitting if the stopping criteria are met.
- The master collects the all decisions about splitting nodes and updates the model.
- The updated model is passed to the workers in the next iteration.
- This process continues until the maximum depth of tree has reached or there are no features left that could be used for splitting.

Most of the methods in this implementation support the statistics aggregation, which is the heaviest part of the computation. In general, this implementation is bound by either the cost of statistics computation on workers or by communicating the sufficient statistics. We will study about also this characteristic in our experiments later.

6. Discussion Of Existing Approaches To Large Scale Data Mining

As stated in the introductory chapter, we look for tools and implementations for data mining based on MapReduce for performing large-scale data mining analytics. We identified that the first step for this task was to do a comprehensive literature research of existing MapReduce scientific papers, libraries and toolkits available. This will help us to develop a comprehensive concept for developing such algorithms. We will need to carefully analyse the results of existing experimental set-ups with real-world data in order to find out if the performance benefits are good enough to channelize our efforts to perform MapReduce based data mining. Another advantage of this work is that we also learn about the possible bottlenecks of frameworks used and their suggested optimisations during the algorithm design. Moreover, it guides us to identify parts of MapReduce framework which are essential to understand in order to design an effective MapReduce data mining algorithms for large-scale data. We use this knowledge to find the appropriate MapReduce implementation of data mining algorithms that can process large-scale data for our work.

In this section, we will study various scientific works and available tools to implement data mining algorithms based on MapReduce. The basic objective of this chapter is to develop a general concept about formulating MapReduce based data mining algorithms. This chapter will give us a deeper insight about how legacy data mining algorithms can be transformed into MapReduce based data mining algorithms. What are the generic steps thereof? Do we need to properly understand the legacy data mining algorithms in order to develop MapReduce based data mining algorithms? Which parts of traditional non-distributed algorithms can be implemented using MapReduce paradigm? How do we identify those parts? How can we string together all those MapReduce parts and non MapReduce parts to output algorithms that behaviour like single-machine implementations? What trade-offs are necessary for processing large-scale data? Such design questions and decisions pertain to a general MapReduce programming concept and not to a particular MapReduce implementation. This knowledge will help us to grasp the basic concepts and fundamentals of data mining algorithms which intends to harness the potentials of generic MapReduce distributed data-processing model. Some of those concepts are rather more general for all data mining algorithms and very few of them rather specific to a particular data mining algorithm. In this section, we will attempt to capture both non-specific and specific concepts about parallelising data mining algorithms using MapReduce. Therefore, we will discuss the MapReduce implementations of decision tree learning based classification and regression algorithms and as well as other data mining algorithms like naive bayes classifier, frequent pattern mining, clustering etc. This concept building will be important to understand the data mining algorithm implementations in Apache Spark MLlib, which we have used for conducting our experiments. This is because they also use various map-like and reduce-like operations on key-value pairs of input data to parallelise the algorithms. We will also see the extent of support for large-scale data mining analysis using MapReduce distributed computation in popular data mining suites.

6.1 Existing MapReduce Based Data Mining Scientific Works

6.1.1 MapReduce Naive Bayes Classifier

TAN ET AL. [18] performed a comparison of various existing approaches to introduce scalability to the data-mining algorithms using real-world large scale datasets. They have also used the results of the comparison to suggest to new framework to combine the different approaches to provide fast, scalable and accurate data mining. We will evaluate how far their suggested methodology applies to the general-purpose data mining algorithms which are scalable without sacrificing considerable accuracy. So far, they found out that there are three major ways to deal with vast amounts of data:

- *Sampling:* The data mining model is constructed using a sampled data of massive dataset. Sampling is done to reduce the size of the training data so that it can easily fit into the main memory of the machine. This method is fast and efficient and they found out that it can be as accurate as model built from entire dataset. But there are problems to determine the right size of the samples and to increase the quality of samples. TAN ET AL. did not mention about any algorithms that could deliver good quality samples. This approach also suffer from the problem of memory limitations on local classifiers.
- *Ensemble Method:* This method has been applied on the classification algorithms where data is randomly partitioned and mined in parallel on local classifiers and finally merged to build a global classifier. Normally, training data is fed into the local classifiers and a voting mechanism is used to find the category of each file. This is illustrated in Figure 16. Again, the problem could be that size of the partitioned training data is larger than the memory available on local machines.

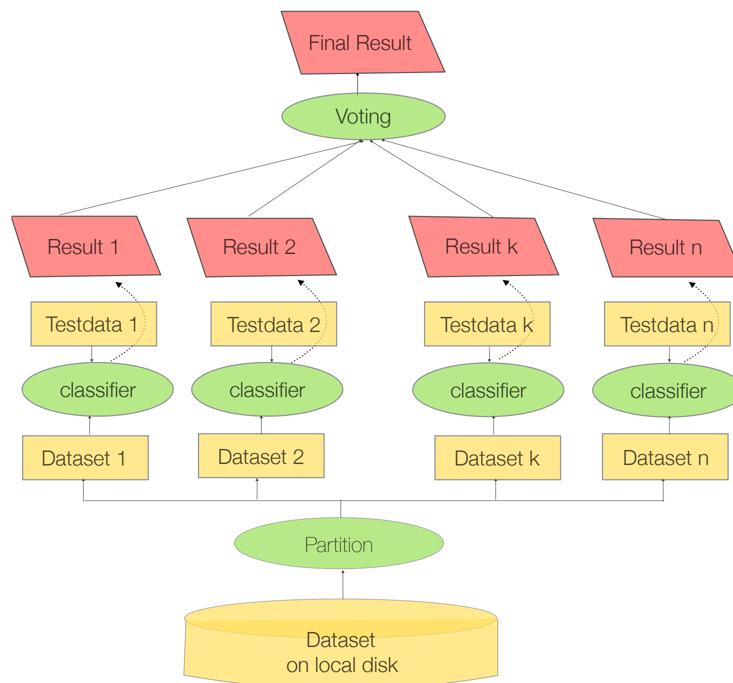


Figure 16. Ensemble Approach for Naive Bayes Classifier [18]

- *MapReduce based Approaches:* TAN ET AL. used Hadoop MapReduce for distributed computing on large-scale data. Hadoop MapReduce provides a very simple interface to parallel scale the data mining algorithms on large scale data alleviating the memory limitation problems in other two approaches. Obviously in MapReduce, the whole dataset was used as a training dataset. As we know that in classification data mining algorithms, there are two distinct phases: Training phase and evaluation phase, both phases are parallelized using MapReduce framework as shown figure 17. They have used the Mahout Naïve Bayes classifier for this purpose.

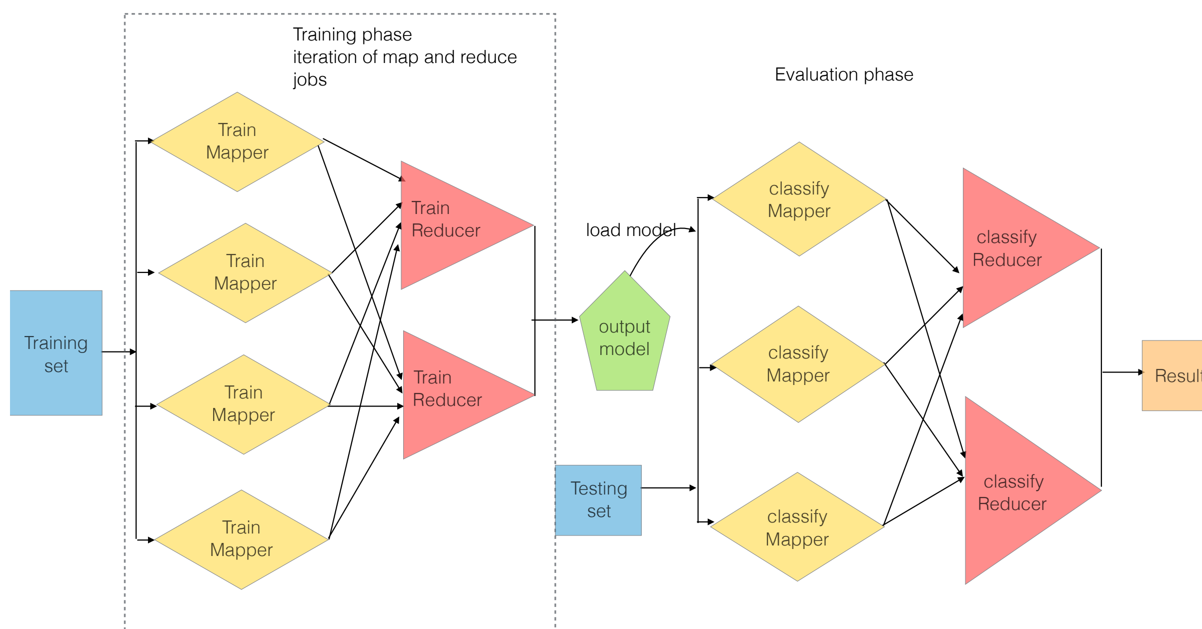


Figure 17. MapReduce Classifier Model [18]

A Naive Bayes classifier is a simple probabilistic classifier based on applying Bayes' theorem (from Bayesian statistics). Naive Bayes Classifier was selected because it can be used on all the aforementioned three approaches and have strong independence assumptions. In simple terms, a naive Bayes classifier assumes that the presence (or absence) of a particular feature of a class is unrelated to the presence (or absence) of any other feature.

The experimental results demonstrated that the accuracy of all the three models improved by using larger training dataset. Ensemble model gives the highest accuracy although the accuracies of all three models are very close. Ensemble model is paralysed by memory limitations of a single machine and is not scalable. MapReduce inherently gives following benefits: scalability, reliability and efficiency. Sampling can give good results only if sampled size is big enough and sampling quality is good enough. For improving implementation time and to increase the quality of sampling model, they suggested to use MapReduce only in the sampling phase. The sample data is extracted from entire dataset using the

MapReduce and it is then fed into data mining tools like WEKA to build data model. This is an iterative procedure where there is a feedback loop to the next iterations and where the accuracy of data mining model is improved by adjusting the sampling data and the size of the sampling data over the iterations. This hybrid is again not guaranteed to give data samples that will ensure that important patterns are not missed. Sampling might remove datasets that are important for data mining model.

6.1.2 MapReduce k-Means Clustering

ZHAO ET AL. give a very methodical way to design parallel k-means clustering algorithm based on MapReduce [45]. In order to design the MapReduce algorithm, the foremost part is to identify the serial and the parallel (or independent) parts of the traditional algorithm. The original k-means algorithm partitions a set of n objects into k clusters. It randomly selects k objects as cluster centers and calculates the distances of remaining n-k objects and assigns them to the nearest centers. The new mean, respectively the center of the cluster is recalculated and the above process is iterated using the updated centers. The process is terminated when there are no significant changes in the position of the centre of clusters. They identified that the calculation of distances is the most intensive calculation in every iteration. Of course, the distance calculations of different points to the corresponding centre are independent and can be largely parallelized. Map function assigns every point to the nearest centre and reduce function calculates the new mean and respectively the new centre of the cluster. They also use a combiner function which is used to calculate the partial sum of all points belonging to the same centre to reduce communication overheads during the shuffling period. MapReduce k-means is an iterative process with a termination condition which is as follows:

- Repeat {
  - Map {
    - Input to map function is: a data point and k centers broadcasted using an array.
    - Map outputs a key which is the closest centre to the given input point and value is the input point}
  - Reduce {
    - Input is one of the k cluster as key and all data points having this centre as the closest centre as value.
    - Reduce calculates the new centre of these points}

  }// end of loop
- Until all of new centers do not change or the change is insignificant. // termination condition

Their experimental setup gave very encouraging results with respect to speed-up, scale-up and size-up. Speed-up measures the increase in performance by increasing the number of nodes while keeping the other parameters constant. The overall speed-up, despite being not close to linear is good. It improves and becomes more acceptable for larger datasets. The gain in speed-up however reduces when the number of nodes are increased because of the greater communication overheads between more nodes. Scale-up measures how

well the larger system handles the equivalently larger data in same time consumed by the original system. The algorithm was found to have an acceptable scale-up. Size-up measures the ability of the system to handle larger data. The MapReduce algorithm sizes up according to their expectations. The overall experiment validates the effectiveness of the MapReduce algorithms for k-means clustering algorithm.

6.1.3 MapReduce k-Medoids Clustering

K-medoids clustering is performed when the particular application requires more robustness against outliers and noise. K-medoids algorithm uses medoid-shift as opposed to mean-shift in k-means to find the clusters in an iterative way. ZHOU ET AL. have attempted to use the Hadoop MapReduce to parallelize this algorithm [46]. First of all, they propose an algorithm for the optimal search of medoids which aims to reduce the number of distance calculations to find the medoids. They use this optimisation in the MapReduce based algorithm which is as follows:

- Randomly select k centers of the clusters.
- Map function calculates distances of the data elements to each of the centers (medoids) and assigns them to the closest medoids.
- Reduce function uses medoids centers as the key and calculates the new medoids using the optimal search of medoids.
- If the medoids have changes, then new medoids are broadcasted to the next iteration of MapReduce jobs.
- The iteration is terminated when there is no significant change in the medoids.

In their experiments, they have only measured the speed up using their experimental set-up which was found to be excellent and almost linear versus number of nodes.

6.1.4 MapReduce Co-Clustering

Co-Clustering or biclustering generates simultaneously two clusters by performing a simultaneous clustering of rows and columns of a data matrix. The pre-requisite of this algorithm is that the data should be represented as a matrix: data tuples as rows and features as columns. For row-wise clustering, we need to identify a correlation or similarity across some of the columns that will guide the clustering of rows. The process is repeated likewise for column-wise clustering. PAPADIMITRIOU and SUN have made a successful attempt to formulate and implement a MapReduce algorithm of checkerboard style partitioning of a matrix into sub-matrices where each sub-matrix represents a cluster [47]. The goal of the algorithm proposed here is to divide an $(m \times n)$ matrix into $(k \times l)$ sub-matrices.

They have used the MapReduce and HDFS for all the four stages of data mining: data gathering, pre-processing, mining and post-processing. Here, our main interest is to find out the results of their experiments and collect their experiences in using MapReduce for

this particular data mining algorithm. For simplicity and brevity, we have removed the discussion of details of their implementation.
The aim of the experiment was to design an effective distributed co-clustering algorithm based on MapReduce and to collect various experiences to implement it using the Hadoop MapReduce framework. For implementation, they needed to tune various parameters to get the best results out of distributed computations. The first parameter was the thread pool size which needed to be configured per node. They found out that when it was set to 1.5 times the number of cores in the node, it gave very reasonable node utilisation. The number of map tasks was implicitly controlled by the minimum input split size and 256 MB was found to be an optimal size. The number of reduce tasks was set to 1 for actual data mining step. But for pre-processing steps, one reduce task was found to be a bottleneck and therefore this number was increased to the number of machines in the cluster. First, we see their experimental findings of the data pre-processing step. The aggregate throughput scaled almost linearly (rather below linear) against the number of nodes by using the initial configurations: input split size as 256 MB, number of reduce tasks as 5 and max number of concurrent map tasks as 6. Next, they found out that the utilisation of each node increased by increasing the number of concurrent map tasks until an optimum value is reached. This optimum value was found to be a value which was slightly more than the number of cores on the machine. The number of reducers for this task that gives peak throughput was found to be 5. Finally, they found out that too small or too large input splits degraded the performance. A small size caused large overheads since there was a large number of map tasks, and therefore a larger number of HTTP request to each reducer, to transfer intermediate results. However, as they increased the split size to several multiples of HDFS block size, it became much more difficult to place map tasks on local copies of the data, so performance degraded due to unnecessarily high network traffic for transferring HDFS blocks to map tasks. For co-clustering step (actual data mining step), they found out that optimal number of nodes to be 25 and after that there were no gains in the scalability. This was possibly because the hadoop run-time overhead began to dominate the job run time as the actual time of job execution decreased with increasing number of nodes. The behaviour of co-clustering process with respect to other three parameters, as in pre-processing step, was found to be same as in the pre-processing step. Lastly they found out that the pre-processing step took significant time portion of the whole process. Therefore they suggest that it is better to employ MapReduce on pre-processing.

6.1.5 MapReduce Apriori Association Rule Discovery

Apriori Algorithm for Association rule discovery is one of the most prevalent data mining methods to find frequent itemsets from given transactional data collected by the companies. It is a very effective algorithm which uses the Apriori knowledge that non-empty subsets of frequent item sets must also be frequent. This Apriori property is used to effectively reduce the search space for itemsets in each iteration. It needs k iterations to find k cardinality frequent itemsets and finally outputs strong rules from it which satisfy both the minimum user defined support and confidence. In every iteration, the algorithm needs the scan the entire database to find frequent k-itemset whose support is greater than minimum support defined. LI ET AL. [48] proposed a parallel Apriori Algorithm based on MapReduce. They found out that scanning the entire transactions database to find the

occurrences of k-itemset candidates and respectively their support is the most time-consuming step in every iteration. Furthermore, the counting of occurrences of itemsets across the transactions is independent of each other. So this step can be parallelized using the MapReduce distributed computation. Map function basically counts the occurrences of the given itemset and reduce function sums up the counts outputted by all the mappers. Then the frequent item sets are generated whose support is more than minimum support.

The experimental set up gave not so good results for smaller datasets. But the results with respect to scale up, size up and speed up subsequently improved by using larger datasets. Consequently, their experiments demonstrated the fact that MapReduce data mining algorithms can give performance enhancements only when the data size is really big.

6.1.6 MapReduce ID3 Decision Tree

The supervised methods of data mining tasks like classification are often represented and stored in the form of decision trees. ID3 (Iterative Dichotomiser 3) is one of the basic algorithms to build decision tree models. We will see MapReduce version of this algorithm which has been designed by Purdila and Pentiuc [49]. Unlike original ID3, their algorithm can handle both continuous and discrete attributes. The algorithm has two distinct phases: decision tree building using training data and post-pruning using the test data. Both phases share the same architecture for MapReduce based distributed processing. In this architecture, there is a central controller which runs only on the master node and performs no data-intensive computations. The controller runs the process to build the decision tree and this process is controlled by a recursive ID3 function running on the controller. The task of calculating the class entropy, information gain for each attribute, the most common class for current dataset, the number of records processed, the splitting value (threshold) for each numeric attribute is done by MapReduce jobs running in parallel on worker nodes. These values are sent back to the recursive function ID3 function running on the controller. They provide enough information to the ID3 function running on master node to choose the best split and its value for a particular node of the tree in construction. ID3 function uses four parameters to control the recursive construction of decision tree and they are as follows:

- *Attributes:* It is a list of attributes which can be used for splitting a node.
- *Input Path*.:The HDFS location to read the files.
- *Filters:* list of <best split, corresponding value> tuples that have been calculated in the previous iterations. This is used to split the node using each value of the best attribute.
- *MostCommonClasses:* Most common class labels found in the previous iteration.

In the beginning, the attribute list is set of all possible attributes in the training set. If there are elements in the attribute list, the controller dispatches a MapReduce job to find out the best split. Map and reduce tasks compute the class entropy, information gain for every attribute and the most common class labels. These values are communicated back to the controller. Then the controller propagates the construction of tree using these values in the following fashion:

- If the class entropy is 0 (all patterns from current dataset belong to the same class) or no patterns were processed (all were excluded by the filters) the algorithm returns a new leaf node that is labeled with the most common class for current dataset.
- Else, it chooses the attribute with maximum information gain and creates a new node.
- The filters and attribute list are accordingly updated and broadcasted to all mappers in the next iteration.
- Then, the ID3 function is called recursively for each value of best split attribute in case the best split attribute is categorical. In case the best split attribute is continuous, the ID3 function is called only twice, once for values ≤ threshold value and once for values > threshold value. This threshold values was implicitly calculated earlier in best split computation step. Filters contain the best split attribute and its corresponding value which is used to split the datasets into the partitions in order to reduce the search space for calculating the nodes at lower depths of the tree.
- The terminating condition for the whole procedure is when the list of attributes is empty.

Their experimental set-up achieved a near linear scale-up. They attributed the success of their experiments to clever MapReduce decisions, for example, selecting the right split size of input data for MapReduce run-time and choosing the appropriate number of map and reduce slots. This ensured the proper utilisation of nodes in the cluster.

6.1.7 MapReduce C4.5 Decision Tree

Now we will see another MapReduce implementation of a very popular decision tree learning data mining algorithm: C4.5 is basically an extension of the ID3 algorithm, both of them invented by J. ROSS QUINLAN. The operation of both algorithms is same but C4.5 can handle continuous attributes and missing values of attributes better and uses entropy gain ratio instead of pure entropy gain as in ID3 to eliminate the bias of selecting the attributes with many values or large range of continuous values. Without going into the further details of C4.5, we will analyse the MapReduce version of this algorithm designed and implemented by DAI and JI [50]. For porting the traditional single machine C4.5 algorithm to MapReduce, they have designed appropriate data structures. These structures store the metadata information of the training dataset which helps in building the final decision tree and also provide a representation for the final tree constructed. The first data structure is an *attribute table* which stores the basic information about an attribute (represented as key of the table), such as row identifier for training tuples (row_id), domain of the attributes and the class labels of tuples. C*ount table* stores the number of training tuples with a specific class label if split by a particular attribute and its value. *Hash table* which is the last data structure stores the link information between node_id (each node in a tree is represented by a node_id) and row_id and link between parent node (node_id) and it's sub-node (subnode_id). Their approach was to divide the M attributes equally on N nodes. This is how the data is vertically (logically only) partitioned and it ensures the maximum localisation of data processing. In other words, they are actually parallelising the job of finding the best split. They have used four kinds of MapReduce jobs corresponding to each of the four stages in this algorithm: *data preparation, selection, update and tree growing.* The explanation of each phase is as following:

- *Data Preparation*: This stage converts the raw data stored in HDFS into the three data structures mentioned above for further MapReduce processing. In this stage, map function transforms the training data tuples into the *attribute table* by using each attribute as key, row_id and class labels as values. To perform this aforementioned conversion, a map function is used to calculate the invert indexes of training data (which is viewed as a matrix) to calculate the *attribute table*. The input to the map function is a `<key, value>` pair where key is the row_id of each training row and value is the training row itself. The output of map is a `<key,value>` pair which is keyed by an attribute which gives information for every attribute, such as, the training tuples (row_id) that have those attributes and the class labels. Reduce function receives this output from the map function. It can simply apply count operation on map output to count the number of training tuples with a specific class label if they are split by a particular attribute. This forms the count table. So the output of map function is an attribute table and reduce function is a count table. However, the hash table is set to NULL initially.
- *Selection*: This stage selects the attribute that is the best split and its corresponding value. First step of this stage is a reduce function whose input is the output of the reduce function of the previous stage. It uses summation to compute the total number of training rows for a particular attribute. Next function is a map function which computes the information gain and split information for every attribute. The next function is a reduce function that computes the information gain ratio for each attribute. The attribute that yields the maximum information gain ratio is selected as the best attribute for splitting and the corresponding split information is selected as the split location or split value.
- *Update*: This stage updates the count table and the hash table. Map function reads the training rows from the attribute table whose keys are the best attribute calculated in previous step and calculates the counts of the class labels for the best attribute. Next map function computes the hash value of the best attribute so that the training rows with same class labels split into the same partition in the next step. This hash value of best attribute is assigned to the node_id. Next, it assigns the node_id to every row_id. This stage can be viewed as node creation step.
- *Tree growing*: The next iteration is repeated by following the same steps in selection and update stages as in previous iteration. This stage grows the tree by creating a sub-node of the parent node created in the previous iteration. It builds tree by creating linkage between the node created in the previous step and the node created in the current iteration. If the node_id for a particular row_id calculated in the current iteration is similar to the node_id of the previous iteration, it is outputted as a leaf node. Otherwise, a new subnode_id is created and assigned to the row_id.
- The whole process is a pipeline of jobs as described in four stages. Only data preparation MapReduce job is one time, the rest of the jobs are iterative in nature which terminates when all the node_id become leaf nodes.

Their experimental set up comprised of 4 machines and they used 6 attributes. Therefore, 1 attribute on is assigned to each of first 3 machines and 3 attributes are assigned on the last one. First, they compared their algorithm with the original C4.5 algorithm. Interestingly, their algorithm outperformed the original C4.5 algorithm even on single node. This was attributed to the dual core machines they have used which provides some parallelism using

MapReduce algorithm. Then, they measured the characteristics of their algorithm using single node. The results showed that the speedup of MapReduce C4.5 algorithm in single node environment was considerably increased when the data size was increased. In the last, they evaluated the behaviour of their algorithm in a truly parallel distributed environment consisting of 4 nodes. In parallel distributed environment, the execution times decreased as the number of nodes were added. The scale-up and speed-up of the system improved when the size of the dataset was increased. For smaller datasets, the performance results were rather poor. Their experiment clearly shows that a properly designed MapReduce decision tree learning algorithm can be effective for large-scale data only. In other words, MapReduce is advantageous when there are long-running jobs.

6.1.8 MapReduce Decision Tree Ensembles (PLANET)

Classification and regression tree learning is one of the most important data mining operations done at Google Inc. PLANET, developed by PANDA, HERBACH, BASU, and BAYARDO, is an attempt at Google to properly understand the application of MapReduce distributed framework to one of the most popular data mining algorithm and also explore the underlying challenges and bottlenecks and find their respective workarounds [51]. PLANET stands for "Parallel Learner for Assembling Numerous Ensemble Trees".

We will give a brief description about architecture and working of PLANET. It is an oversimplified MapReduce decision tree learning algorithm for small decision trees that must fit into memory of single machine. But the advantage of this implementation is that it is designed to handle large-scale training dataset. Therefore, size of training dataset is not a limitation for this implementation. However, it is limited to regression trees with binary splits only and it can use only variance as a impurity measure. They argue that the main objective of their work is to derive and develop general architectural principles of MapReduce based decision tree learning algorithms so that they can be later utilised to extend this algorithm to other options like classification trees, n-ary splits and other impurity measures such as gini and entropy.

PLANET achieves parallelism by breaking the large-scale datasets across many processing units. PLANET builds a breadth-first tree which grows one level in every iteration. The central component of PLANET is a *Controller* which runs on the master node. It controls the entire process for constructing a decision tree model. For coordinating this process, it uses a special data structure called as *ModelFile*. This data structure stores the current state of the tree which has been constructed so far. By inspecting the *ModelFile*, controller determines which nodes and split predicates to consider. Suppose, the ModelFile contains the information that the tree has been built until level k, then the master considers all the nodes in the level k+1 and the set of remaining attributes and its values that can be considered for splitting those nodes at k+1 level of the tree. The process of decision tree building is described in the following steps:

- *Initialisation phase.* The first stage of this implementation is a initialisation phase which is implemented using map and reduce jobs. This phase has only one kind of MapReduce job called as Initialisation MapReduce job. This job pre-identifies all the values for each

attribute which need to be considered later for splits. In other words, it calculates the attribute metadata such attribute domain of each attribute. For continuous attributes, it computes an approximate equi-depth histogram and uses the boundary points of the histogram as split locations. For categorical attributes, it identifies the whole domain of each attribute by identifying each distinct categorical value that were found in all data instances. This task of pre-computing the metadata of each attribute is executed in a parallel distributed fashion on a number of nodes using the MapReduce job mentioned earlier in this paragraph. This kind of metadata information about attributes is stored in an *attribute file* which is later loaded in the memory for MapReduce tasks of the succeeding stage. The initialisation stage runs only one and is followed by a stage which builds the decision tree. This stage is iterative in nature and uses the attribute metadata information given by the initialisation phase.

- *Decision tree construction phase.* This phase uses MapReduce jobs to calculate the best splits. After the master has decided for which nodes split predicates to compute, it sends the attribute file and the model file to all mappers on worker nodes. The mapper function first scans the whole partition of dataset that is residing on the same node and uses the ModelFile to determine if a particular training tuple is a part of input dataset. After determining the input dataset to a particular node, mapper needs to evaluate the possible splits for the node and select the best split for the partition of data residing on the node. Each Mapper runs on a subset of data and computes partial statistics for all splits to be considered. Mappers output the result where key is the best split that the mapper computes for the particular subset of data residing on the same node and values are the statistical information when that best split is used on the same subset of data. Reducers combine map outputs and compute the best split that it has seen for the particular node. Each reducer informs the controller about following information: (1) the quality of the best split and its value, (2) the average value predictions or class labels in the left and right branches, and (3) the number of training records in the left and right branches. The *Controller* takes the splits produced by all the reducers and finds the best split for the current node, then updates the *ModelFile* with this information. This process is repeated until there are no splits to consider. They have also introduced one very important optimisation that the algorithm switches to the original algorithm for small-scale data where training tuples are kept in memory over the iterations. In PLANET, as tree induction progresses, the size of the input dataset for many nodes becomes small enough to fit in memory. In that case, subtrees below those nodes are built by keeping the training tuples in the memory as opposed to scanning the database to find the input dataset.

Here we have seen one of the implementations of simple decision tree induction on MapReduce framework. But they also found out that their algorithm often suffers from the problem of overfitting. Therefore, they build an ensemble of trees in parallel. Each of these trees are built exactly in the same manner explained above. Boosting and bagging is applied to this ensemble of trees which speeds up the decision tree building process. Additionally, they also remove the problem of overfitting eliminating the need of any post-pruning. Since building tree is a iterative process, they ran into various issues which significantly affected the performance and speed up negatively. They observed that every set-up for MapReduce task took significant time. So they used forward scheduling to

handle this problem. Forward scheduling means *Controller* performs the set-up for next MapReduce task when a MapReduce task is currently running rather than waiting for the current MapReduce task to finish. Another significant latency in map jobs was introduced by traversing the tree model to determine if the records are candidate for the test on the node being expanded. They used fingerprinting to solve this issue. They also introduced some reliability measures to the controller because it was a single point of failure. Controller checkpoints the growth of the tree on a permanent storage. In case the controller fails, it can be restarted from the last tree model that checkpoint by the failed controller. The experimental results shows that the training time increases with the increase in the training data size but significantly decreases by adding more machines. They also concluded that optimal number of workers for their experimental set up is under 400 machines.

In this sub-section, we have summarised the effectiveness of MapReduce for data mining algorithms. Most of these algorithms achieve close to linear speed up (always lower than linear exactly) against number of nodes and good scale up and size up against various data sizes. But there are several bottlenecks and limitations of this software framework that compelled the programmers developing distributed data mining algorithms to come up with many work-arounds and optimisations tailored for their particular implementation.

6.2 MapReduce Based Data Mining Tools

6.2.1 Apache Mahout

Apache Mahout is a project of Apache Software Foundation to develop open source implementations of highly scalable data-mining algorithms [52]. It is a scalable machine learning and data mining open-source software based mainly in Hadoop MapReduce. For years, this was the only tool that provided that provided ready to use data mining tool based on Hadoop. But the development team of Mahout has stopped support for Hadoop MapReduce based data mining tools in favour of developing tools based on Apache Spark. The current version is still based on Hadoop which has implementations of a wide range of machine learning and data mining algorithms: clustering, classification, collaborative filtering and frequent pattern mining on top of distributed and scalable Apache Hadoop platforms. The scalability of Mahout has 3 dimensions:

- Scalable to large data-sets because it uses scalable and distributed systems.
- Scalable for individual business case as it uses friendly Apache license.
- Scalable developer and contributor community which is responsive and interactive.

In order to use Mahout with Hadoop, Mahout needs to be installed only on the master node of Hadoop cluster. Mahout works by submitting the data mining job as a Hadoop jar to the Master node in Hadoop environment. Mahout can also be used on a single machine without MapReduce. In data mining field, Mahout is the only most complete suite that contained most of the best-in-class data mining algorithms based on Hadoop MapReduce. Table 2 gives an exhaustive list of all algorithms implemented in Apache Mahout. Detailed discussion of implementations of each algorithm in Mahout is not required for our work.

Data mining categories	Data mining algorithms
Collaborative filtering/frequent pattern mining	1. user-based recommender
	2. item-based recommender
	3. recommendations based on Alternating Least Squares
Classification	1. Naive Bayes Classifier
	2. Complementary Naive Bayes classifier
	3. Random Forest Classifier
Clustering	1. K-means clustering
	2. Canopy clustering
	3. Fuzzy k-means clustering
	4. Streaming k-means clustering
	5. Spectral clustering

Table 2. List of Hadoop MapReduce based data mining algorithms in Apache Mahout

After going through the full documentation of Mahout, we have come into several interesting observations about this project. We found that this machine learning software tool is relatively strong in unsupervised learning, offering a number of clustering algorithms and others. But Mahout's supervised learning algorithms, however, are weak. There is only one decision tree algorithm supported and it have several issues regarding the memory limitations of tree. We found two major criticisms of Mahout project, which we will explain: the project itself is a mess and Mahout's integration into MapReduce is suitable only for high latency analytics. First, Mahout certainly does seem eclectic in the way the different data mining algorithms are implemented. Some of the algorithms are distributed, others are single-threaded, in turn others are simply imported from other projects. Many algorithms are underdeveloped, unsupported or both. The project is constantly in a cleanup phase as it upgrades its version: a number of underused and unsupported algorithms are continuously deprecated and removed. Secondly, the "Slow analytics" solution provided by Mahout is not acceptable in the real world. The issue here is that machine learning performance suffers from MapReduce's need to persist intermediate results after each pass through the data; for competitive performance, iterative algorithms require an in-memory approach. For this reason, they are going to develop algorithms based on Apache Spark. So, currently the project is in experimental status. MICHAEL SEVILLA shares his experiences using the Mahout library for data mining purposes [53]. The results are not very exciting. He found out that Mahout is not so intuitive and does not provide easy-to-use and out-of-box solutions for parallelising data mining on hadoop clusters. The abysmal scaling performance from his experimental set up was because of the poor understanding of the underlying data mining algorithms. There were several bugs which users of Mahout need to always track in their bugs and issues webpages. He found out that Mahout did not suffer from the problem of

poor scalability but rather improper tuning of parameters could give poor performance. So in order to get maximum performance out of Mahout, data scientists must have the proper understanding of both input data and the underlying algorithms implemented in Mahout.

In April 2014, Apache Mahout project announced that they will stop developing data mining algorithms based on Hadoop MapReduce. After releasing Mahout version 0.9, the team decided to begin an aggressive retool using Apache Spark. So, we can see that even the most complete suite of data mining algorithms based on Hadoop MapReduce are moving to a more modern and more efficient data-processing model. One could argue that the Mahout community had to embrace Spark, at least, if it wanted to remain relevant. This was one of the strongest arguments for us to select Apache Spark for our data mining tasks. This is a clear indication that even big open-source projects that aim to develop distributed and scalable data mining algorithms are abandoning Hadoop MapReduce for future development.

6.2.2 WEKA

WEKA (Waikato Environment for Knowledge Analysis) is a popular suite of machine learning software written in Java, developed at the University of Waikato, New Zealand. WEKA is free software available under the GNU General Public License [54]. It contains a collection of visualisation tools and algorithms for data analysis and mining together with graphical user interfaces. WEKA is mainly a tool for use on a single machine. The development team has attempted to develop some scalable data mining packages for large-scale data. There are some development efforts to provide some data mining functionality using Hadoop MapReduce. Without delving into the depths of this software, we will analyse this MapReduce based data mining functionality provided by WEKA in form of extension packages which is collectively called as *distributed weka* [55]. The version 3.7.2 of WEKA contains two packages for this purpose: *distributedWekaBase* which provides base "map" and "reduce" tasks that are not tied to any specific distributed platform and *distributedWekaHadoop* which provides Hadoop-specific wrappers and functions for the aforementioned map and reduce base tasks. *Distributed Weka* package for WEKA defines various Mappers and Reducers that can be directly used in user’s programs for executing data mining tasks inside of Hadoop MapReduce. But there are no documentations and API specifications about Hadoop MapReduce data mining algorithms implemented in WEKA. On the top of that, the number of data mining algorithms that have been implemented using Hadoop MapReduce is abysmally low. The current version of WEKA has only implementation for classification or regression type of data mining algorithms based on MapReduce. There are only logistic regression and naive-bayes based classification algorithms. There are no decision tree based classification and regression data mining algorithms. The implementation of clustering and association rule discovery data mining practices are still missing or are under development. But the project team has given clear indications that they will invest their development efforts in developing distributed scalable data mining algorithms using Apache Spark.

6.3 Conclusion

The outcome of this chapter is that although MapReduce is a mature technology and a lot of large-scale data processing is still done using this technology, there is a rather disappointing number of MapReduce data mining implementations in scientific literature. This scenario is even worse when we look for MapReduce based data mining tools. Moreover, the development teams that involved in scalable data mining projects for large-scale data are more interested in investing and focussing their development resources to implement distributed data mining algorithms based on Apache Spark because its in-memory computations is clearly advantageous for iterative date mining algorithms. This extensive literature review helped us analyse to various alternatives for scalable and distributed data mining algorithms that are capable of processing large-scale data.

When we summarise the results of all experiments that has been done in section 6.1, we can conclude that MapReduce makes sense when the data size is massive. All the experiments pinpointed the same trend that increasing the data size gave better speed-up and scale-up. But for all data sizes, the scale-up was always much less than 1 and speed up approached 1 when the data size was increased by tremendous amount. Increasing the data size ensured that the length of the running job was much larger as compared to the MapReduce overhead. The insignificant MapReduce overhead in case of longer-running jobs improved the speed-up. The performance benefits from Hadoop MapReduce are clearly not enough for data mining tasks when we desire output in reasonable amount of time. Therefore, the current trend in data mining field is to explore better distributed execution platforms that are more suited for data mining and analytics. But Hadoop MapReduce is a mature technology and its use case is justified when stability is desired for our applications.

Now, we have gathered a lot of strong arguments that favours using Apache Spark's distributed processing engine for scaling out data mining algorithms. After doing extensive research about experiments that have been done using Apache Spark, we could not find any experiments that demonstrate the behaviour and performance of Spark for decision tree based classification and regression analysis of real-world datasets. This particular kind of data mining algorithm has been implemented in Spark's MLlib. We will evaluate the performance and behaviour of this implementation in forthcoming chapters.

7 Experimental Set-Up

In this section, we give a detailed description of the experimental set-up, the various configuration parameters tuning and the data retrieval process for performing decision tree based classification and regression analysis using the Apache Spark's MLlib. Our experiments are conducted using Apache Spark's distributed computation model which uses the Resource Manager of Hadoop YARN for negotiating the cluster resources and the underlying HDFS for storing and accessing data in distributed manner. Therefore, all Spark applications have dependencies on Hadoop YARN client, HDFS and on Spark core execution services. All the Spark applications that use MLlib APIs must also specify dependency on Spark MLlib module. Although spark applications can be developed using Java, Scala or Python, we have chosen to use Java as a programming language for our applications development. In order to submit the standalone java programs (not using Spark's interactive distributed Scala shell) to Spark execution framework, we need to create an assembly jar which packages the compiled Java application with all the dependencies. We have used Apache Maven to create jar files. Maven is a very popular Java project building tool. All applications are run on "Spark on YARN" in "client mode". We have used Java version 1.8.0_25, Spark version 1.1.0 and YARN version 2.4.0 for all our experiments. This section is unusually long because we need to discuss the experimental set-up specifics of two technologies: Spark and YARN. We also discuss the data preparation in this section as it is a very essential and time-consuming step for general data mining tasks which becomes even more complicated in case of large-scale data preparation.

First of all, we need to set up the underlying cluster. As such, it is very essential to have an overview of the YARN cluster and the underlying HDFS. This is represented in Figure 18 below. Our infrastructure consists of a cluster of six nodes: dmcl0, dmcl1, dmcl2, dmcl3, dmcl4 and dmcl5. Each node has 4 cores, 8 GB RAM and 50 GB of disk storage. All of these virtual machines are running CentOS 6.5. Nodes dmcl0-5 are locally connected by a software-switch running on the VM host. Hence, the bandwidth will depend on the overall load of the host. In Figure 18, we see also that dmcl1 acts as master node, dmcl2 - 5 act as slave nodes and dmcl0 acts as secondary master node.

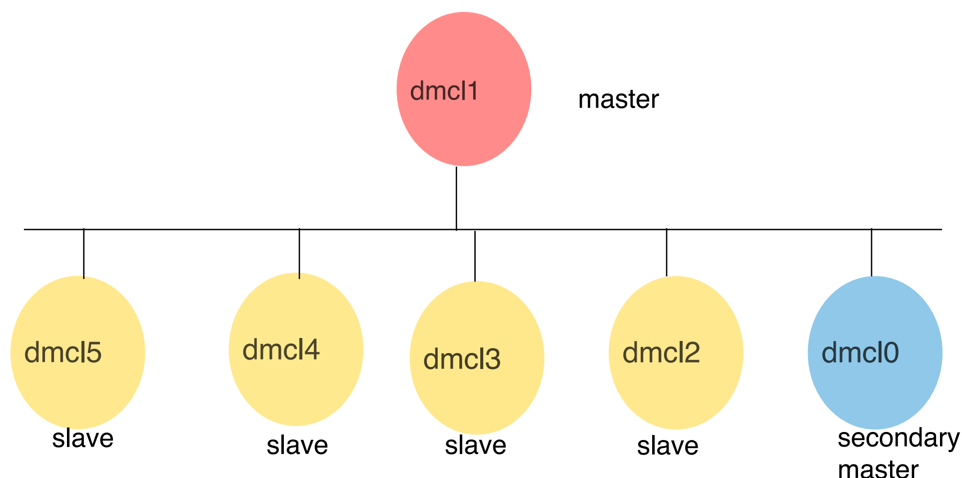


Figure 18. YARN Cluster showing number of nodes, their respective roles and the network topology

The next step is to calculate and set the values of various configuration parameters for YARN and HDFS. After determining the necessary configuration values, they are set with the help of a group of configurations files provided by Hadoop: conf-site.xml, hdfs-site.xml, yarn-site.xml and mapred-site.xml. There is a bewildering number of configuration parameters when one wants to deploy a YARN cluster. But only few of them are worth mentioning. Considering that we have a rather small cluster with limited number of cores, disk space and memory, we need to carefully choose some important configurations and cleverly calculate their values. In other words, we have a cluster of low-end machines. This is a very important characteristic of our cluster which we need to keep in mind and which we evaluate to find out Spark's concrete hardware requirements. It is a popular fact that Hadoop was designed for low-end commodity servers but there is no general documentation about such requirements for Spark.

Due to the small nature of the cluster, we have reduced the default size of HDFS blocks from 128 MB to 64 MB. There are two reasons for this decision: First, we have to accept the fact that we cannot process very large amounts of data within a reasonable amount of time on such a small cluster. This is specially true for decision tree classification and regression algorithms as they are multi-stage applications. Secondly, reducing the block size ensures maximum parallelization even for smaller amounts of data because spark calculates the number of tasks from the total number of blocks in any HDFS data. Typically, as the Hadoop YARN and Spark documentation says, the total number of tasks should be more than the maximum number of cores (which is 16 in our cluster) for optimal cluster utilisation. For our experiments, as we will see later, the minimum size of input data is 2 GB which is when divided by 16 gives us a value of 64 MB. This parameter was not set to a lower value because for larger data sizes, it would create very large number of blocks. Henceforth, data seek times for blocks would become very high, disk read speeds would deteriorate and it would affect the performance of algorithms for larger data sizes negatively. The default replication factor for blocks was left at the default level of 3.

There are other configuration parameters specific to spark environment which needs to be taken care of in order to avoid Java heap space memory problems while ensuring maximum parallelization of tasks and in-memory computations at the same time. Spark configuration parameters need to be closely controlled due to the limited amount of RAM we have (only 8 GB). For calculating the values of those parameters, we should carefully inspect and understand how Spark works on top of a YARN cluster in a client mode. This makes it necessary to have an overview of all the processes and daemons running on each node and their respective memory usage. This is graphically depicted in Figure 19. There are 3 kinds of nodes: master, secondary master and slaves. They have different sets of daemons and processes running on them. There are four yarn daemons running on the master node: *Resource Manager, Name Node, Hadoop Job-History Server* and *Hadoop Proxy Server*. The secondary master has only one YARN daemon: *Secondary Name Node*. Slaves have two YARN daemons: *Node Manager* and *Data Node*. Each YARN daemon on master, secondary master and slaves requires 1GB of memory. Therefore, maximum memory that can be allocated to YARN containers on each slave node is left to 6 GB. YARN container scheduler is configured to allocate containers in multiple of 512 MB up-to a maximum of 6 GB. Spark executors are just containers which has been requested by a Spark application.

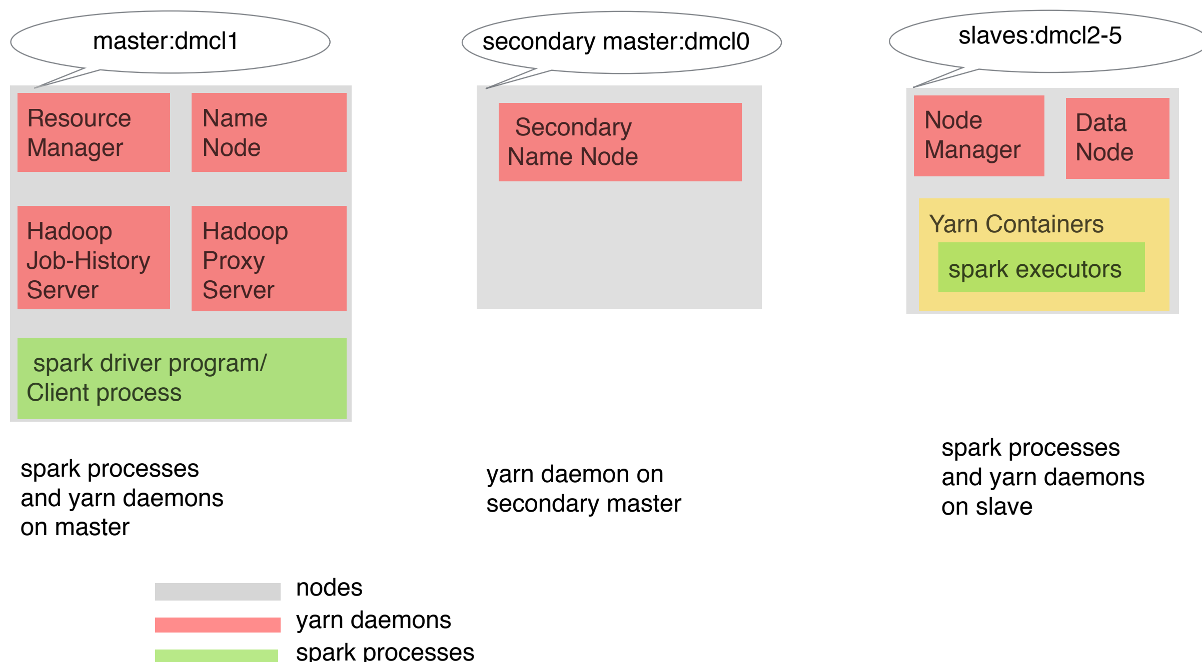


Figure 19. An overview of YARN daemons and Spark processes running on different nodes.

.

One of the most tedious job in the process of configuring a Spark application is to configure the exact amount of memory for Spark executors and driver. Figure 20 shows a very systematic way to calculate the value for a maximum amount of memory that can be allocated to each executor in our cluster. Since each slave node has four cores, four containers may run on each node at the maximum in a truly parallel manner. At first, it seems that we can specify a maximum of 1.5 GB to each executor. In that case, we found out that the number of executors is less than four and one of the cores will go idle on each node. This is due to the fact that when we specify spark executor or driver memory, this amount of memory refers only to the amount of memory that will be requested by Spark application to store objects. Here, objects refer to Java RDDs which are constructed by transforming the training tuples when a Spark application reads the training dataset stored on HDFS. Such RDDs will typically reside in Java heap space and will also be subject to Garbage collection of JVM. In addition to the memory requested for each executor, there is a “Spark on YARN” memory overhead per executor which refers to off-heap memory to store serialised objects (which are stored outside the Java heap space and are also not subject to Garbage Collection). This memory overhead is used to store or cache stuff like JVM overheads, interned strings, other native overheads, etc. Spark run-time adds this “Spark on YARN” memory overhead value to the executor memory value to determine the size of the containers that it will request from the YARN Resource Manager. The Spark on YARN memory overhead can have a minimum value of 384 MB. We have increased this value to 512 MB to avoid out-of-memory problems. Therefore, each spark executor memory is configured to 1 GB. In Figure 20, we can also see that the amount of executor memory space is further divided into 2 regions: 60% of this memory is used to cache and keep old RDDs and the rest 40% is used for storing new RDDs which are created during the task execution (40%) of every iteration. The Garbage Collection occurs first on the part

of executor memory where new objects are created and stored (the 40% part). Why it is designed like this? If the application revisits the same dataset in every iteration there is no need for Garbage collection and to create new RDDs in each iteration. But this is not the typical case in decision tree learning algorithms as the input dataset to compute each node in each iteration is different. Therefore, we need a separate cache to store new RDDs where Garbage collection occurs to evict the old RDDs to make room for new RDDs.

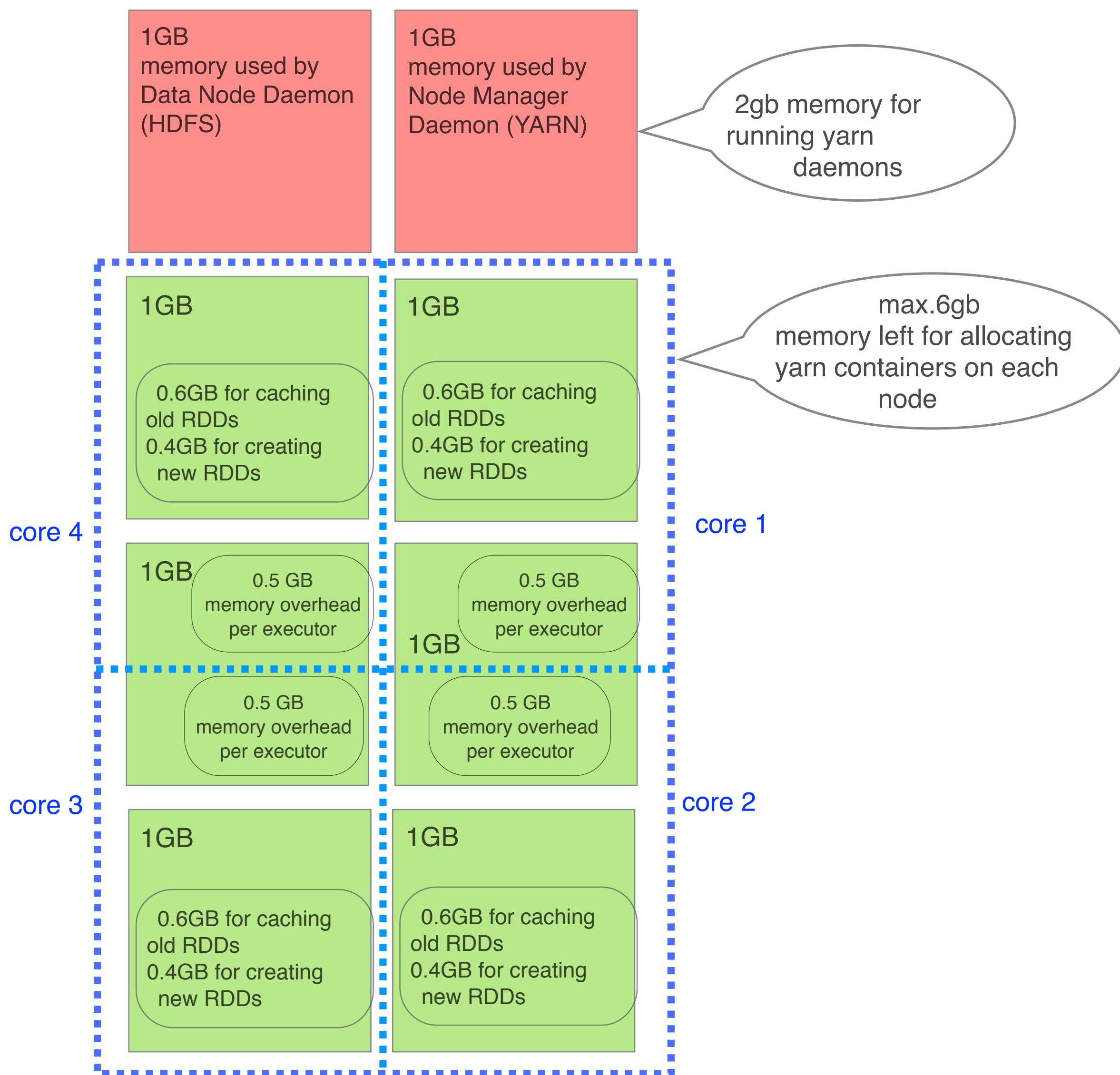


Figure 20. Memory is allocation to each executor on a node.

7.1 Data Preparation

The current version Spark's MLlib supports three kinds of data structures to represent the RDDs of the input training and testing data: Matrix, Labeled point or Vector form [56]. For supervised learning algorithms like decision tree classification, the only supported data structure is a Labeled Point. We know that the training data for classification and regression algorithms should be a labelled training data which contains one column as a label (response) and the rest of the columns represent the feature vector used to predict the corresponding label. The Labeled Point refers actually to the RDDs which is obtained by reading such labelled training data from a source file. In Labeled Point RDDs, a double is used to store a label so that it can be used for both classification and regression algorithms. Figure 21 represents the Labeled point example for a multi-class classification.

label/response	index 0	index 1	index k	index n-1
1	33,5	45,0	……………………..	90,0
0	34,0	50,9	……………………	99,7
3	22,9	51,2	…………………..	100,8
2	34,9	89,0	…………………	110,9

label/response for multi-class classification

Feature vector of dimension n

Figure 21. Labeled points for multi-class classification (an example).

For data preparation, we need to discuss the specific formats of source file that are supported by Apache Spark's MLlib. It actually supports both CSV and LIBSVM format for input data file. If the input data is in CSV format, MLlib reads training tuples row by row and transform them into a Labeled point RDD which has two parts: a label and a feature. We need to specify which column is a label, which columns form feature vector and what delimiters is used (in this case always a comma). However, the internal representation of a Labeled point RDD is exactly the same as LIBSVM format. Therefore, MLlib supports reading data directly from files in LIBSVM format where users do not need to specify the procedure to convert the input data into Labeled Point. LIBSVM format is optimised for both sparse and dense data while CSV format is not optimised for sparse data. The conversion of data in CSV format to a Labeled Point is currently supported only in Scala API but not in Java API. Therefore, Spark's MLlib Java applications for decision tree learning expects the input data only in a LIBSVM format and not in a CSV format. Thus, we have used input data files in LIBSVM format rather than CSV format. LIBSVM uses the so called "sparse" format where zero values do not need to be stored. An example for LIBSVM format training input for the corresponding CSV training input is shown in Figure 22. The first value in LIBSVM format data is defined to be a label. Therefore, MLlib classification and

regression algorithms automatically know the label and the feature vector and they don't need to calculate them while loading LIBSVM data into memory. This saves valuable computation time as well. The only drawback of this format is that it currently supports only numerical data.

CSV format:	**comma separated zero and non-zero values**
CSV input data:	**9,1,0,8,97,0,0,0,0,12**
LIBSVM format:	**label <index 1>:value1 <index2>:value2.....**
LIBSVM input data:	**9 1: 1 3: 8 4: 97 9: 12**

Figure 22. An example for LIBSVM format for corresponding CSV format

Very fortunately, several tools are available that convert CSV format files to LIBSVM format files, for example, MATLAB [57] (proprietary) and Phraug [58] (open-source python scripts). We have used the Phraug tool to pre-process large files line by line. In addition to converting files from CSV to LIBSVM, we have used other functions of this tool, for example, to count lines in a file and to split a file into a number of smaller files to create data with sufficiently varying characteristics.

Now we will see the procedure to retrieve, transform and augment the training data which serve as input for programs that perform decision trees learning. For regression, we selected two kinds of datasets. We call them "Blog" data and "YearPrediction" data. The dimension or the number of features in YearPrediction data is 90 and in Blog data it is 281. For our experiments, we have used all the 90 features of YearPrediction data and all the 281 features of Blog data. YearPrediction data is obtained from an online repository [59] which hosts some properly labelled training data-sets in the LIBSVM format for classification and regression mining tasks. The original size of YearPrediction data is 567 MB. Since the size of data-set is not large enough, we have use the tools from Phraug to break the file into large number of small files. We create files of required sizes by randomly concatenating those smaller files. This ensured that training tuples are randomly distributed and their duplication is also randomised in the output files of desired sizes. Since the computation time for training the decision tree models depends on number of training rows and not on the size of the file, we have used the Phraug tool to count the number of training instances in different sizes of file. Table 3 shows the exact number of training instances in varying sizes of the YearPrediction data file (exact sizes in GB).

YearPrediction data

Data Size (GB)	No of training instances
2	1704860
4	3409720
8	6819440

Table 3. Number of training instances in 2,4 and 8 GB of YearPrediction data

Blog data is obtained from another online repository [60] but it stores data in CSV format. So we used phraug tool to convert the format of data from CSV to LIBSVM and then we apply same procedures as in YearPrediction data to create files of desirable sizes. The original size of Blog data is 72 MB.

Blog Data

Data Size (GB)	number of training instances
2	5179877
4	10357049
8	20714098

Table 4. Number of training instances in 2,4 and 8 GB of Blog data

For Binary classification, we have used another interesting data source: “Pascal Large Scale Learning Challenge” [61][62]. This repository provides various large-scale data files which are named as: alpha, beta, delta, dna, epsilon, fd, gamma, zeta ocr and webspam. We selected the alpha data file and therefore, we call this data as “Alpha”. This is the only true large-scale properly labelled data (original size is 3,7 GB) that we found for classification mining tasks. The dimension of features in this data 500. We have used all features of this dataset. The original data comes in CSV format, but the online repository also provides an accompanying python tool which can be used to specify exact number of training tuples that can be extracted in LIBSVM format. The original file contains 500.000 training instances and we apply this conversion tool on this file. Then, we use the Phraug tool to break the converted large file into a large number of smaller files which are again merged to form files of desired sizes. This was clearly the best and the easiest way to create large-scale which did not require the duplication of training instances. But training data for multi-class classification or regression is unfortunately not available.

Alpha Data

Data Size (GB)	number of training instances
2	292278
4	583221
8	1166442

Table 5. Number of training instances in 2,4 and 8 GB of Alpha data

For multi-class classification, we have used a rather high dimension data obtained from Mnist database for handwritten digits [63]. We have directly used the LIBSVM format of the same data stored at [59]. We call this data as “Mnist" data. The original size of this data file is 69,4 MB. It has a maximum dimension of 780. But we have used 100, 150 and 200 dimensions for our experiments in order to study the impact of number of features on the computation time. We could not use higher dimensions due to memory restrictions. We have applied similar procedure as in YearPrediction data to consolidate the size of the files to desired values.

Mnist Data

Data Size (GB)	number of training instances
2	1800000
4	3600000
8	7200000

Table 6. Number of training instances in 2,4 and 8 GB of Mnist data

The last step is to discuss some important decisions pertaining to the data mining algorithms implementations in Apache Spark’s MLlib which we have used for our experiments. MLlib decision tree learning algorithms support both discrete and continuous values for features. But in our experiments, we have assumed that all features have continuous values for the sake of simplicity. To extract categorical features information, we will need to do a lot of pre-processing for the input data and this is particularly very difficult and time consuming for large-scale data. Additionally, it does not gives any significant advantage for our work because MLlib decision tree learning algorithms internally discretises the continuous features into discrete features by binning the features as described in section 5.4. We viewed extracting categorical features information in a pre-processing step also as a deviation from the main focus of our work.

It is also important to identify different stages in all three MLlib's decision tree learning algorithms. Later, in our experimental results, we will see that we have assigned certain names for those stages: *Initial Spark setup, Read data, Read data & Count labels, Train Model and Evaluate Model*. We need to understand what those terms stand for. Figure 23 (a), (b) and (c) respectively depicts the various stages in decision tree learning based regression, binary classification and multi-class classification algorithms. For all three algorithms, the first stage is called *Initial Spark set up*. In this stage, Spark environment-specific configurations for a spark application is set up and the required number of executors are requested from the Resource Manager. The second stage is called *Read data* whose responsibility is to load as much data as possible into memory. This is done by calling a spark action which loads the LIBSVM data file. But, as we can see in Figure 23 (c), the loading of LIBSVM data becomes a lazy transformation rather than an action. Spark's MLlib API specifies that when we specify the number of features that we want to use for data mining, then the loading of LIBSVM data becomes a lazy transformation and is delayed until first action is performed. In Figure 23 (c), we can also see that the first action for multi-class classification is to count the number of target labels. This implies that the loading LIBSVM data is delayed until counting of labels is required. Therefore, the stage to count the labels includes the time for caching the LIBSVM data and the time for counting the labels. That's why, the second stage in mutli-class classification is called *Read data & Count labels.* As we can see in Figure 23 (b), that there is no operation for counting the labels because we know beforehand that there are only two target labels for binary classification. Third stage is called *Train model*. This stage actually builds the decision tree model for classification and regression mining. This stage requires a lot of iterations to construct a decision tree. The number of iterations depends on maximum depth of tree. Time taken by this stage is most interesting for our work. As we can see in figure 23 (a), that this stage also performs the counting of labels before it performs the decision tree induction. The last stage is to evaluate the output model from train phase by calculating the mean square error of trained model against a testing data. Here, we use the training data as testing data.

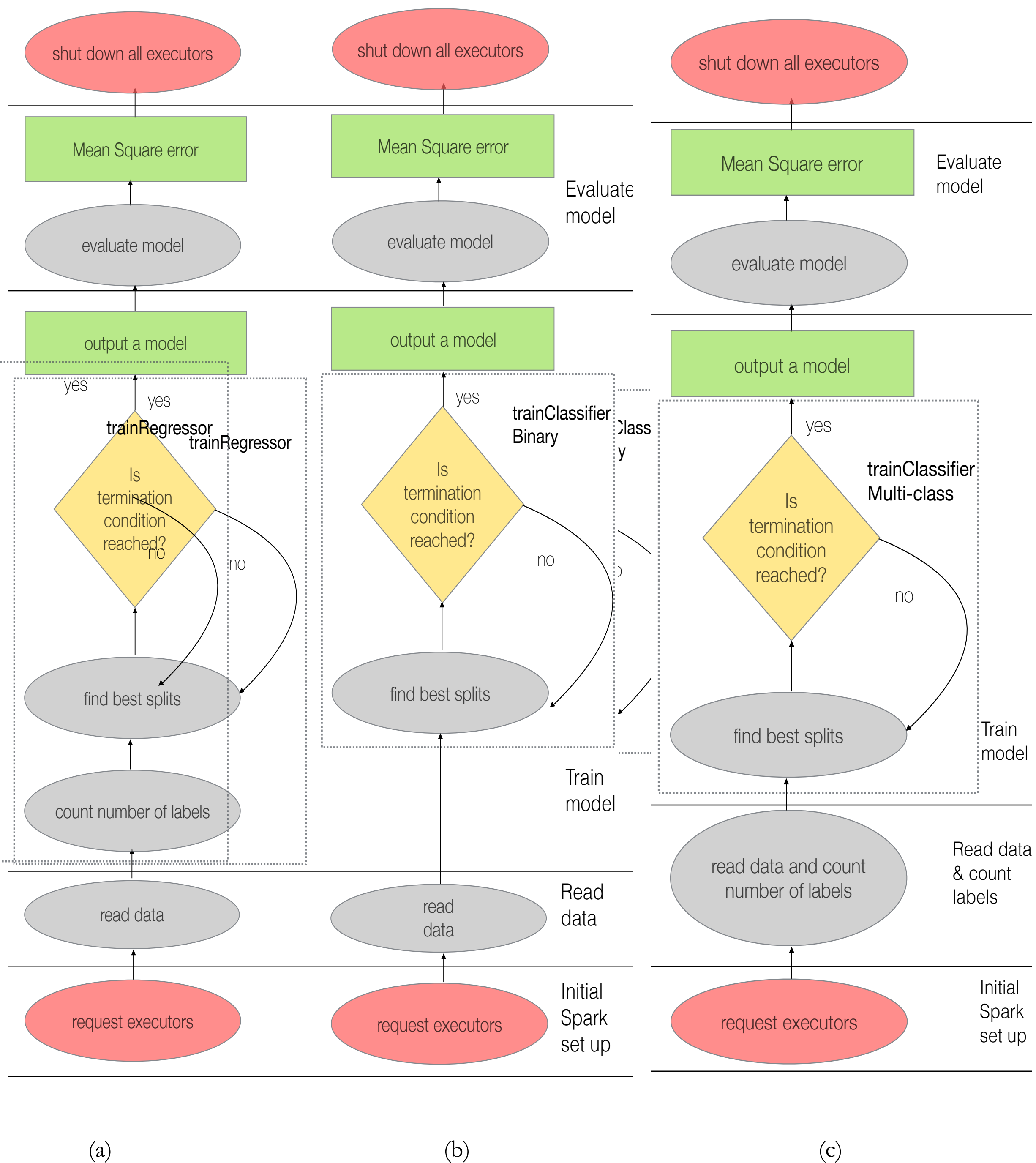


Figure 23. Stages in decision tree learning for regression and binary classification in MLlib

8 Experiment Results And Evaluation

In this section, we evaluate the performance of decision tree classification and regression algorithms implemented in Spark's MLlib for large-scale data. We also analyse their run-time behaviour and its characteristics. We compare the time to train decision model by varying the number of nodes and by varying the data size. Number of nodes actually refer to number of worker nodes in our experimental set-up. The performance evaluation is also done by calculating speed-up and scale-up. We perform experiments by using 1 master node and 1, 2 and 4 slave nodes. In order to properly measure the speed-up and scale-up, we have taken 2, 4 and 8 GB data which are the exact multiples of the node numbers.

Speed-up factor demonstrates how much faster an algorithm is when it is run on more than one node. To measure the speed-up, we keep the dataset constant and increase the number of nodes in the system. A perfect parallel algorithm demonstrates a linear or an ideal speed-up. It is obtained when the system with p-times the number of nodes yields a speed-up of p. However, the linear speed-up is difficult to achieve because the communication and other overhead increases as the size of the cluster grows.

Scale-up evaluates the ability of the algorithm to grow both the number of nodes and the dataset size. It is defined as the ability of a p-times larger system to process a p-times larger dataset in the same run-time as the original system. For measuring the scale-up, since there are 1, 2 and 4 nodes, we need 2, 4 and 8 GB data. Scale-up is measured by increasing the data size and the number of nodes by same factor at the same time.

Before discussing the results of all three algorithm implementations, we need to understand some factors that will influence the run-time. The most obvious factor is network and communication overhead which increases when we increase the number of nodes. More number of node implies that there is more shuffling of intermediate data between map and reduce tasks. There are some factors that are more specific for decision tree learning based classification and regression algorithms. One of those factors that we need to consider is the dimension of features. We perform some experiments to show the effect of number of features on the run-time of training models. There are some factors that are even more specific to the Apache Spark's MLlib implementation of decision tree learning algorithms. As we know from section 5.4 that MLlib implementations of decision tree learners actually discretises the continuous features by binning them, the user specified maxBins parameter which specifies the maximum graininess that is used for split decisions also affect the computation times and the communication overhead. We also attempt to demonstrate the effect of maxBins parameter on run-time by performing appropriate experiments.

For all experiments, we repeat the same experiment five times and then we use minimum, average and maximum values for fine tuning the presentation of the results. We also specify the maximum depth of tree to be 6 because higher values of tree depths were giving out-of-memory errors. Such errors finally led to the complete halt of the programs without finishing the execution. In this section, we will present the results of our experiments by using several graphs and pie charts that will demonstrate the behaviour and performance of all

decision tree learning algorithms of MLlib that we have used for our experiments. We have organised this section by assigning one sub-section to each dataset.

8.1 Initial experiments

After performing rigorous initial experiments with various datasets having different dimensions and maximum number of bins, we found out that we can build a tree of maximum depth of 6. Since, spark uses in-memory computations, building tree of more depth threw out-of-memory errors in most cases.
In the beginning, when we started performing our experiments, we found out that Spark run-time was throwing very often out-of-memory errors. We tried all kinds of optimisations suggested by Spark online guide but of no avail. Therefore, we performed some initial experiments to find out the cause of this problem. This problem was mainly due to the Garbage Collection mechanisms during the task execution. As we know that Spark performs in-memory computations, it reserves 60% of the executor memory for storing old RDDs and 40% for storing new RDDs created during particular stage execution. It uses Garbage Collection to make space for the new RDDs in the same fashion as JVM organises its own heap [65]. Normally, when decision tree construction reaches lower levels of the maximum specified depth, a lot of new RDDs are produced which causes the Garbage Collection mechanisms to kick in too often. As we know that garbage collections introduce intermediate pauses, this can interfere with task working memory. The pause time for each Garbage Collection process is proportional to the number of Java objects. The garbage collector will throw an out-of-memory error if too much time is being spent in garbage collection: if more than 98% of the total time is spent in garbage collection and less than 2% of the heap is recovered, an out-of-memory error will be thrown. We found out that the programs were throwing out-of-memory errors at lower depths of tree which implies that garbage collection becomes more prominent at lower depths of tree. Again after performing lot of initial experiments, we found out that the rate of churning in of new java RDDs increases when the feature dimension and the maximum number of bins increase. Again, we found out this fact by increasing feature dimension and keeping the maximum number of bins constant. This led to the out-of-memory errors for the higher dimensions of feature. The same procedure was applied to the maxBins parameter and yielded out-of-memory errors for higher values of maxBins.

8.2 Experiment 1 With Alpha Data (Binary Classification)

In Experiment 1, we perform decision tree based binary classification using Alpha data. Since, we already know that the maximum depth of tree is 6 and the dimension of training data is 500, we only need to determine the value of one more parameter: maxBins. Since the dimension of the pascal data is very high, we fixed the value of maxBins to 25. When we increased the value of maxBins to 30,40 and 50, we found out that spark was throwing out-of-memory error at the lower depths of 6-depth tree for all data sizes. This led to lost executors problems and considerable time loss to recover the lost stages. The cause behind out-of-memory errors can be attributed to the fact that higher feature dimension and higher bin value led to higher memory requirements for histogram computations at lower level. So, we could not perform experiments with bin value 50 and this option was

eventually dropped. Of all the datasets that we have used for our experiments, this dataset has the lowest number of training instances for same volumes of data in GB. The MLlib documentation [64] says that the training time depends on the number of instances. For conciseness, we are referring to data sizes in GB rather than the number of instances. However, we can refer to tables in section 7 for information about the exact number of training instances in each dataset.

As mentioned before, the objective of this experiment is to measure the run-time for training decision tree model by varying the data size and the number of nodes. This is depicted by a number of graphs in Figure 24. We also find out the proportion of total time taken by different phases. This is depicted by various pie-charts in Figure 25. This gives a general idea about how the total computation time is distributed among various phases.

Figure 24 (a), (b) and (c) shows respectively the minimum, average and maximum time taken for training the decision tree models versus number of nodes for a specific data size. For 2 GB data set, there was a considerable decrease in run-time when the number of nodes was increased from 1 to 2. This result was according to our expectations. This result can be explained by the arguments that increasing the number of nodes from 1 to 2 facilitated more main memory for caching the RDDs which enabled more in-memory computations. Moreover, there are more number of cores that can run computations in parallel. When the number of nodes was further increased from 2 to 4, we see some improvement in run-time as well but his improvement is rather small. This shows that 2 nodes were sufficient to cache 2 GB data in memory. This data set has the lowest number of training instances. The small improvement that was obtained was primarily due to parallel processing of data rather than increase in amount of cached RDDs. Moreover, when number of nodes is increased to 4, the performance is dampened because there is more communication overhead for shuffle and sort of intermediate results of map and reduce which begins to dominate the small execution time. We conclude from the results that 2 nodes are rather optimal for 2 GB of Alpha data. For 4 GB data, we see better improvements in run-time when we increase the number of nodes from 1 to 2 and 2 to 4. By improvements, we mean an absolute amount of decrease in run time rather than the factor of decrease. For 8 GB data, the improvement in run-time is even more pronounced. Therefore, the amount of decrease in computation times definitely depends on the training time on 1 node for all data sets.

Figure 24 (d), (e) and (f) shows respectively the minimum, average and maximum time taken for training the decision tree models versus different sizes of data while keeping the number of nodes constant. All the graphs in the second row of Figure 24 demonstrate that there is a considerable increase in training time when the data size is doubled or quadrupled. There is also a significant decrease in training time when the number of nodes are increased from 1 to 2 to 4.

Figure 24 (g) demonstrates the speed-up of MLlib's binary decision tree classifier. In sub-section 6.1, we have seen that the decision tree learning algorithms based on Hadoop MapReduce yielded near-linear speed-up. Figure 24 (g) clearly shows that we have much better than close to linear speed-up for Apache Spark's MLlib decision tree classifier for all

data sizes. As expected, we achieve super good speed-up for 2 (6,3 for 2 nodes and 7,6 for 4 nodes) and 4 GB data (2,9 for 2 nodes and 8,1 for 4 nodes) because most of the data can fit in memory for in-memory processing. We can also conclude from the results that 4 nodes is an overkill for a small data set (2 GB). There is no significant gain in speed up when we increase number of nodes from 2 to 4 because 2 GB data is not enough data to fully utilise the processing power of 4 nodes.The best speed-up was observed for 4 GB data because this amount of data is neither too less nor too much to utilise the full processing power and caching benefit of 4 nodes. So, we can conclude that when the data size is less than the capacity of cache memory, we can get much better than ideal speed-up. For 8 GB data, we have speed-up of almost 2 for 2 nodes and 5,5 for 4 nodes which is still better than an ideal speed-up. Therefore, for all dataset sizes of Alpha, we achieve much better results than we expected.

Figure 24 (h) reveals the scale-up for this system. As we can see that system scales very well when data size is increased from 2 to 4 GB (even over 1) but decreases slightly when it is increased to 8 GB (0,89). This scale-up (both data sizes) is considerably better than Hadoop MapReduce based algorithms in section 6.1. Moreover, it shows that 4GB data is the optimum data size for this system when we aim that all training tuples are processed using fast in-memory computations. When data size is increased to 8 GB, the scale-up decreases due to various factors: more frequent Garbage Collection occurs, more percentage of total data is lying on the disk and less percentage of total data is lying in memory; more number of nodes means there is more shuffling of intermediate data between map and reduce because we increase number of nodes to 4 for 8 GB. Our experimental set-up exhibit far better than expected results for decision binary classifiers implemented in Apache Spark' MLlib.

Figure 25 shows the percentage of total run-time taken by each phase. Now analyse the behaviour and the general trend of each phase. First, we take the training phase. We can see that training phase takes the most percentage of total time for all cases (except when data size is 2 GB). We can also see that the proportion of total time spent in training phase increases when the data size increases and it decreases when the number of nodes increases. As the size of training data increases, most of the total time is spent training the model. For evaluation phase, the proportional time also decreases with more number of nodes. Proportion of time spent for initial spark set up increases as we increase the number of nodes. This shows that the overhead for requesting executors increases when the number of nodes requested are higher. For reading phase, the share of time for reading phase decreases when the data size increases because training and evaluation phase take much longer time and dominate the run-time.

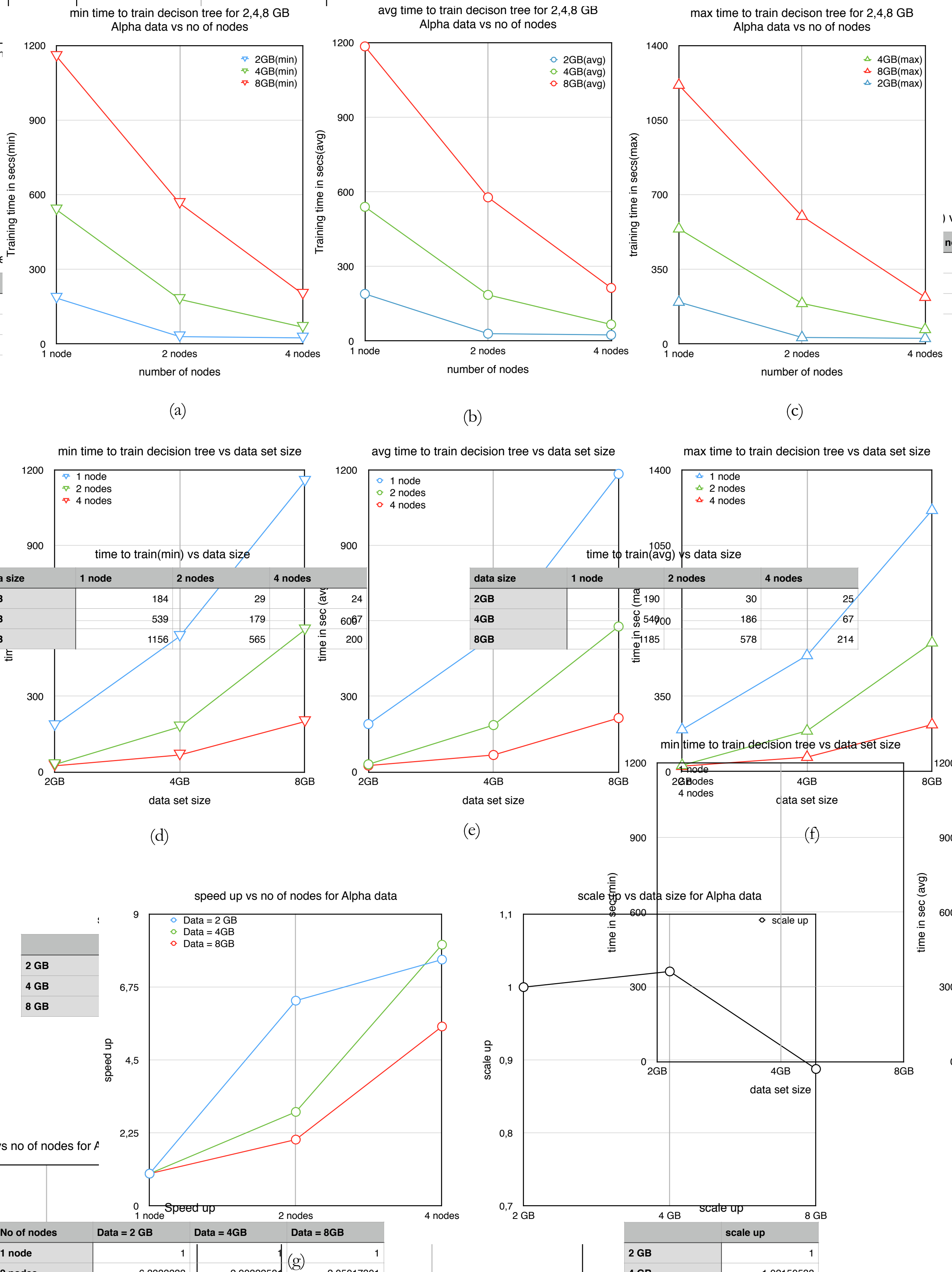


Figure 24. Experimental results for Alpha data using training time

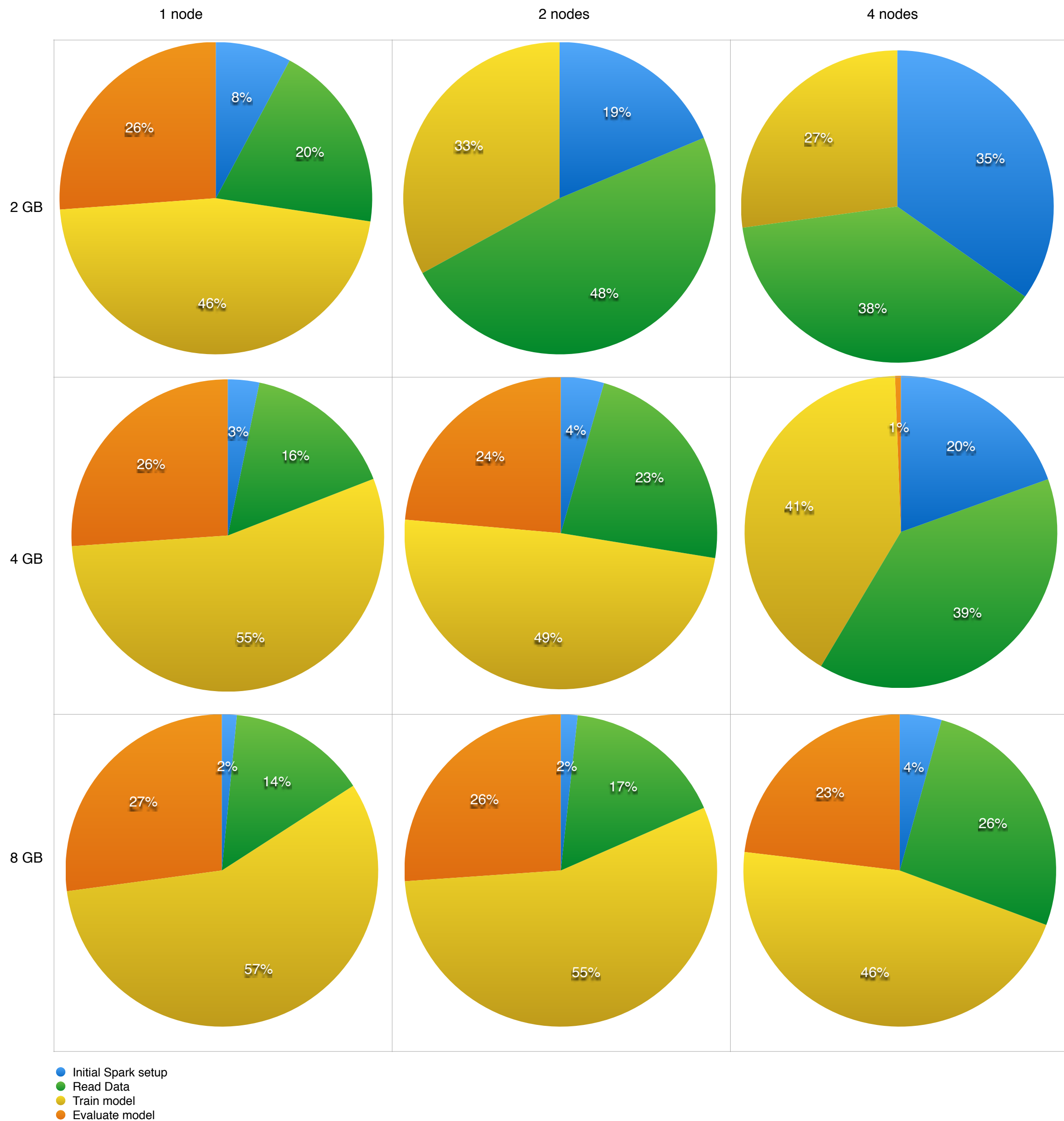


Figure 25. Share of total time for different phases for Alpha data

8.3 Experiment 2 With Blog Data (Regression)

For regression analysis using decision trees, we have used both high dimension and low dimension input training data. Blog data is the high dimension data with 281 number of features. We have set the maximum number of bins to be 25 as higher value threw out-of-memory errors. We build a tree of depth 6. Here we see the behaviour of run time of MLlib's decision tree regressors using high dimensional data. Although the sizes of data sets in Alpha [Table 5] and Blog [Table 4] data is same in terms of disk space, the number of training instances in Blog Data is 17,7 times the number of training instances in Alpha data. Therefore, in terms of number of training instances, Blog data is considerably larger data than Alpha data; but it has lower dimension. The number of instances and their feature

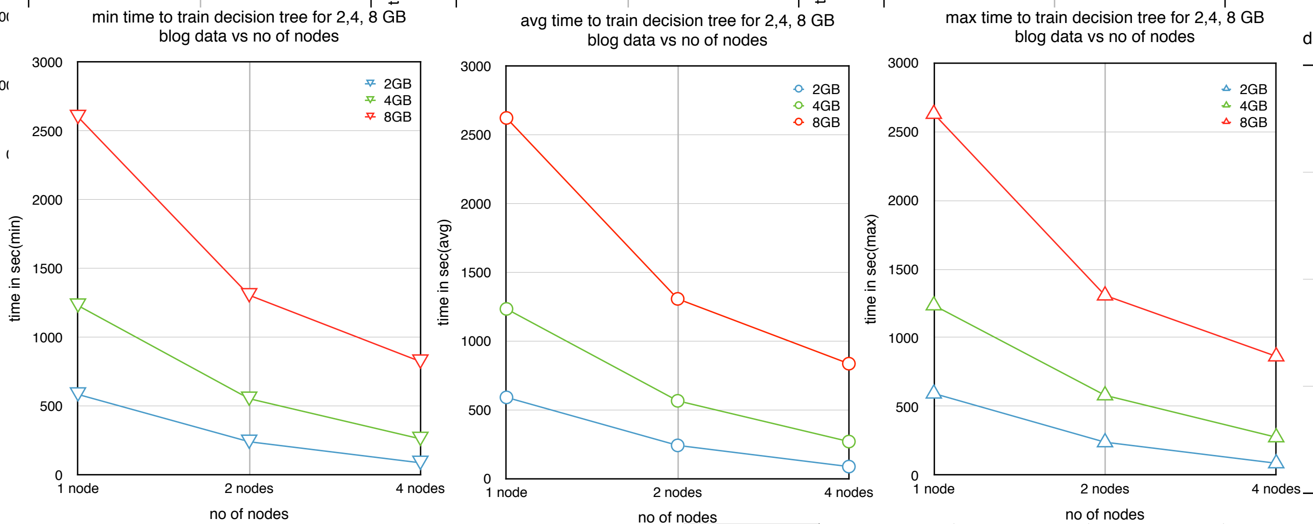
dimension finally determine the number of RDDs that algorithms in MLlib will need to process. Thus, number of instances is the actual size of input data for MLlib.

The first row of Figure 26 shows the behaviour of training time for regression decision trees versus number of nodes for a particular data set size. Figure 26 (a), (b) and (c) shows correspondingly the minimum, average and maximum observed values for training times. As expected, there is considerable decrease in training times with more number of nodes for all data sizes. However, we do not observe any anomaly in the case of 2GB data which we observed in the Alpha Data because there are significantly more number of training instances in Blog Data even for 2 GB. Again, the amount of decrease in computation time is observed highest for 8 GB data and it decreases as the size of data decreases.

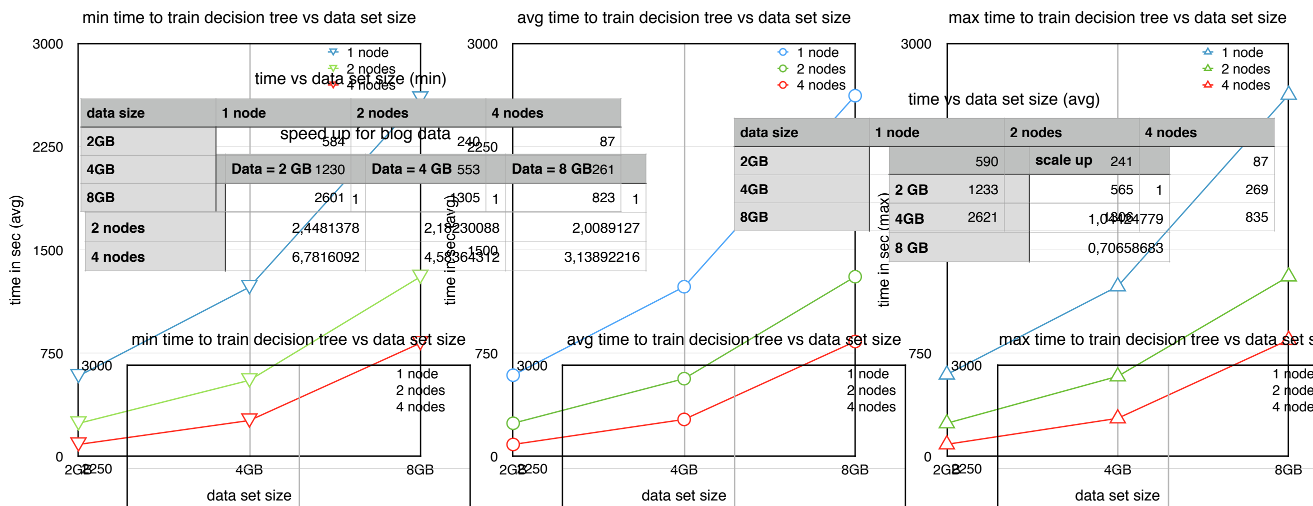
In Figure 26 (d), (e) and (f), we can see how the training time changes against data sizes while keeping the number of nodes constant. When the data size is increased from 4 to 8 GB, the increase in training time is far much more than the increase in training time when data size is increased from 2 to 4.

However, the speed-up results, as shown in Figure 26 (g), are rather worse than for Alpha data because this dataset is considerably larger than Alpha dataset. Here, we get better than linear speed-up for 2 GB data, linear speed-up for 4 GB data and close to linear speed up for 8 GB. Thus, for 8 GB data, the performance of MLlib regression algorithms approaches the performance of Hadoop MapReduce based regression algorithms because all the Hadoop MapReduce experiments in section 6.1 always yielded close to linear speed-up (which is actually lower than the linear speed-up). We found out that there is not much variation in the speed-up for the number of nodes equal to 2. This is because all the data sizes are much larger than the total size of memory of 2 nodes and for all data sizes, most of the data is fetched from the disk. But there is a much larger variation for speed-up when the number of nodes is 4. We begin to see the caching effect for 2 GB but this effect again diminishes for 4 GB and 8 GB data. Caching effect means most of the data can be cached into memory and speeds up the execution. We see that for 8 GB data and 4 nodes, the speed-up falls slightly lower than linear because there is a higher communication overhead for shuffle and write of intermediate map and reduce outputs, most of the data is lying on the disk and the garbage collection is occurring more frequently. Figure 26 (h) demonstrates the scale-up of this system. The system scales again superbly up-to 4 GB (again more than 1). But, for 8 GB data, the scale-up decreases because the system becomes larger and there is higher communication overhead which overshadows the little benefit we are getting from in-memory computation because most of the data is processed from the disk.

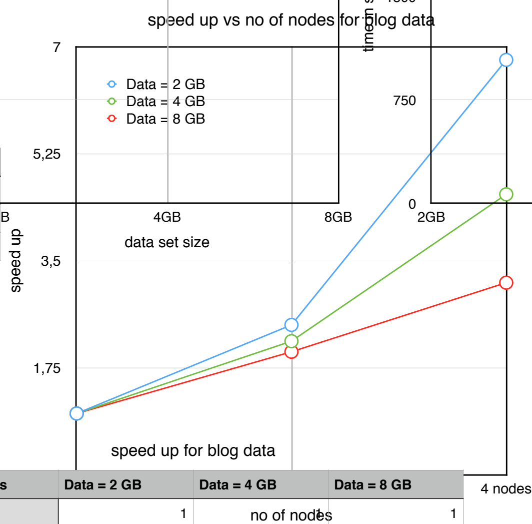
In Figure 27, we show the percentage of total computation that is spent on a particular stage. As we can see, the training time percentage remains almost constant around 65 % except for the case of 2 GB and 4 nodes because the training is sped up due to more caching effect. Evaluation time percentage decreases with increase in number of nodes.

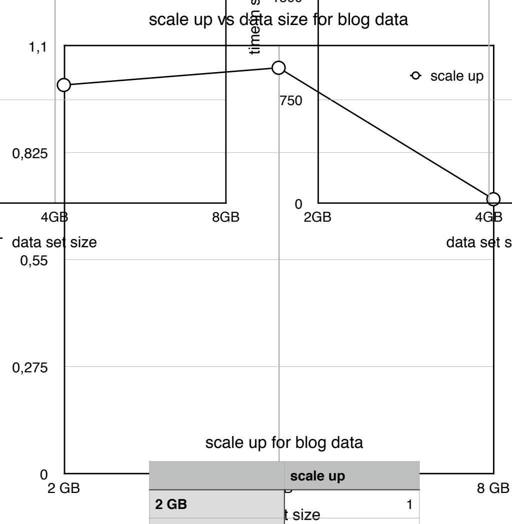


(a) (b) (c)



(d) (e) (f)





(g) (h)

Figure 26. Experimental results for Blog data using training time

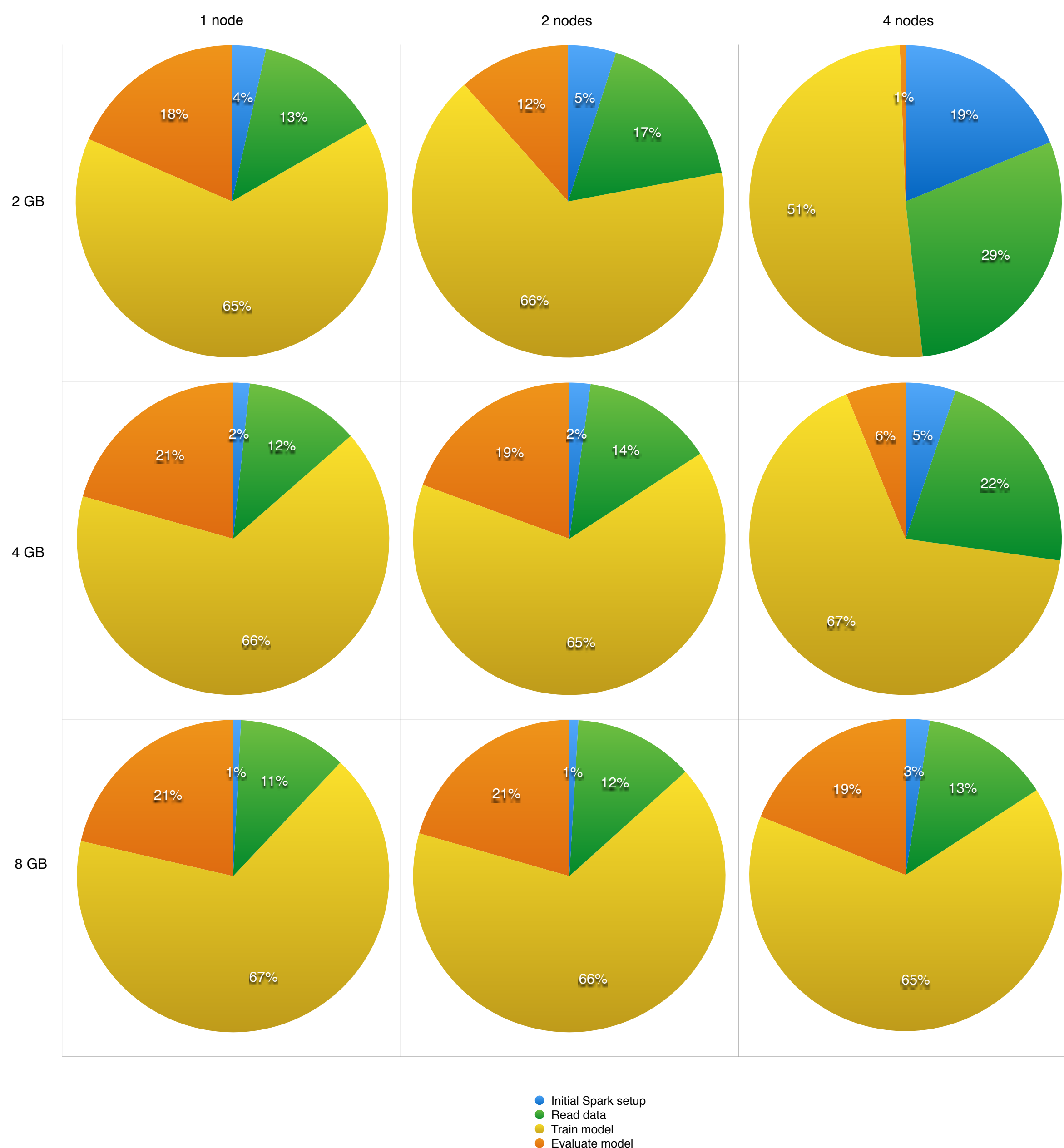


Figure 27. Share of total time for different phases for Blog data

8.4 Experiment 3 With YearPrediction Data (Regression)

We perform another decision tree based regression analysis using a lower dimension data. We have used YearPrediction data for this purpose whose features have dimension of 90. If we compare the data sets of Blog data and YearPrediction data of same sizes, the number of training instances in YearPrediction is one third of the number of training instances in Blog data. Although it is a smaller data set as compared to Blog data, it is still much larger than Alpha data when we talk in terms of number of training instances.

Figure 28 (a), (b) and (c) depicts the minimum, average and maximum training time for regression decision tree versus number of nodes for a specific data size. We can see that the amount of decrease in training time by increasing the number of nodes is prominent and more evident in case of 8 GB. For 4 GB data, we also see a considerable reduction in training times but it is less than 8 GB data. For 2 GB data, there is not much reduction in the training time because this data is small and the training time on even 1 node is rather small. We can also see for 2 GB data that there is a significant reduction in training time when the nodes are increased from 1 to 2 but rather insignificant reduction in training time when the number of nodes are increased from 2 to 4. This shows that most of the RDDs created from 2 GB data can fit in the memory of 2 nodes. Therefore, 2 nodes are just sufficient for this amount of data. Increasing the number of nodes to 4 for this size of YearPrediction data is not justified because of the poor gains. This shows that there is much reduction in training time when the training time on 1 node is very long but we cannot gain much reduction in training time when it is initially very small on 1 node.

Figure 28 (d), (e) and (f) shows the behaviour of training time versus data size when the number of nodes are kept fixed. We can see that there is a considerable increase in training times when the number of nodes are increased in two steps: 1 to 2 and 2 to 4. In the latter step, the increase in computation time is even more pronounced.

Figure 28 (g) demonstrates the speed-up of this system. The speed-up for the 2 GB data is excellent and far better than just linear speed up (7,7 in case of 2 nodes and 12,5 in case of 4 nodes). Better than the linear speed-up shows the effectiveness of memory caching. The memory of 1 node is not sufficient to cache all the RDDs of the 2 GB YearPrediction data due to which the on-disk processing dominates the in-memory processing. 2 nodes cached most of the data into memory and 4 nodes could cache even more. So, there was a significant shift from on-disk to in-memory processing. Therefore, more caching in the case of 4 nodes gives the benefits of in-memory computation and speeds up the algorithm by a large factor. In the case of 4 GB data, the speed-up is lesser than 2 GB data but it is still better than ideal speed-up (2,7 in case of 2 nodes and 7,9 in case of 4 nodes). This shows that there is also a continuous increase in percentage of total data being cached into the memory by increasing the number of nodes. The proportion of the RDDs in the memory to the RDDs lying on the disk is large enough for the in-memory computations to show the caching effect. In case of 8 GB data, this proportion becomes very low even in the case of 4 nodes and therefore speed-up is close to linear speed-up (2,1 in case of 2 nodes and 3,3 in case of 4 nodes).

Figure 28 (h) shows how well the system scales with the respect to data. The system scales very well for 4 GB data but this scalability drops when the data size is increased to 8 GB. To measure the scale-up, we use 4 times larger system to process 8 GB data. So, there is a high communication overhead and there is a shuffle of larger amounts of intermediate outputs of map and reduce when the number of nodes increases and so far we are not benefitting much from in-memory data processing as most of the data is lying on the disk.

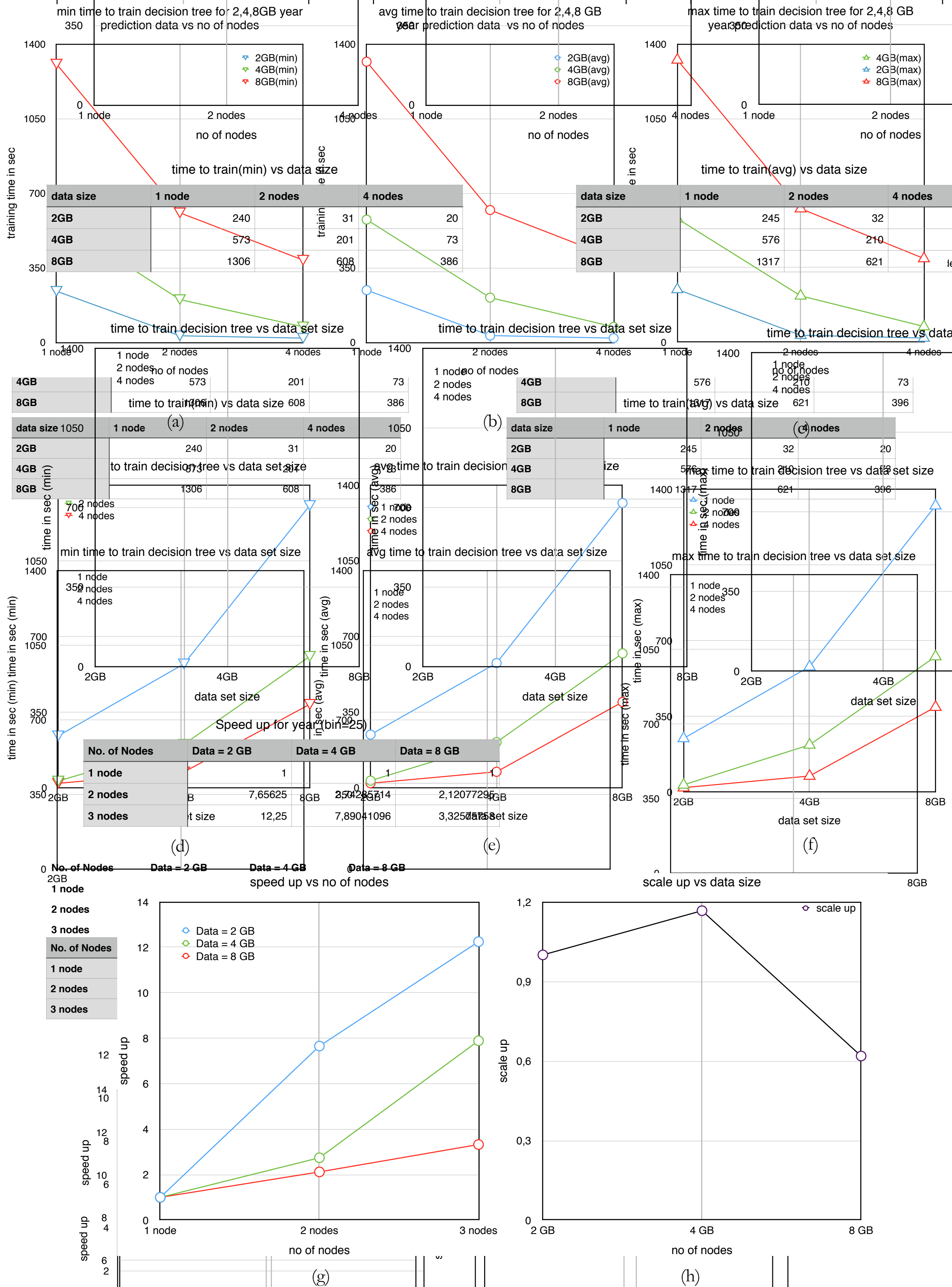


Figure 28. Experimental results for Year Prediction data using training time

Figure 29 shows the proportion of total time that is shared by different phases of decision trees learning based regression analysis. The percentage of time taken by the training and the evaluation phase decreases with more number of nodes for the 2 GB and the 4 GB data but remains more or less constant for the 8 GB data. For the 2 and the 4 GB data, the training time is substantially improved due to the effective caching of greater portion of the input data into the main memory of more nodes.

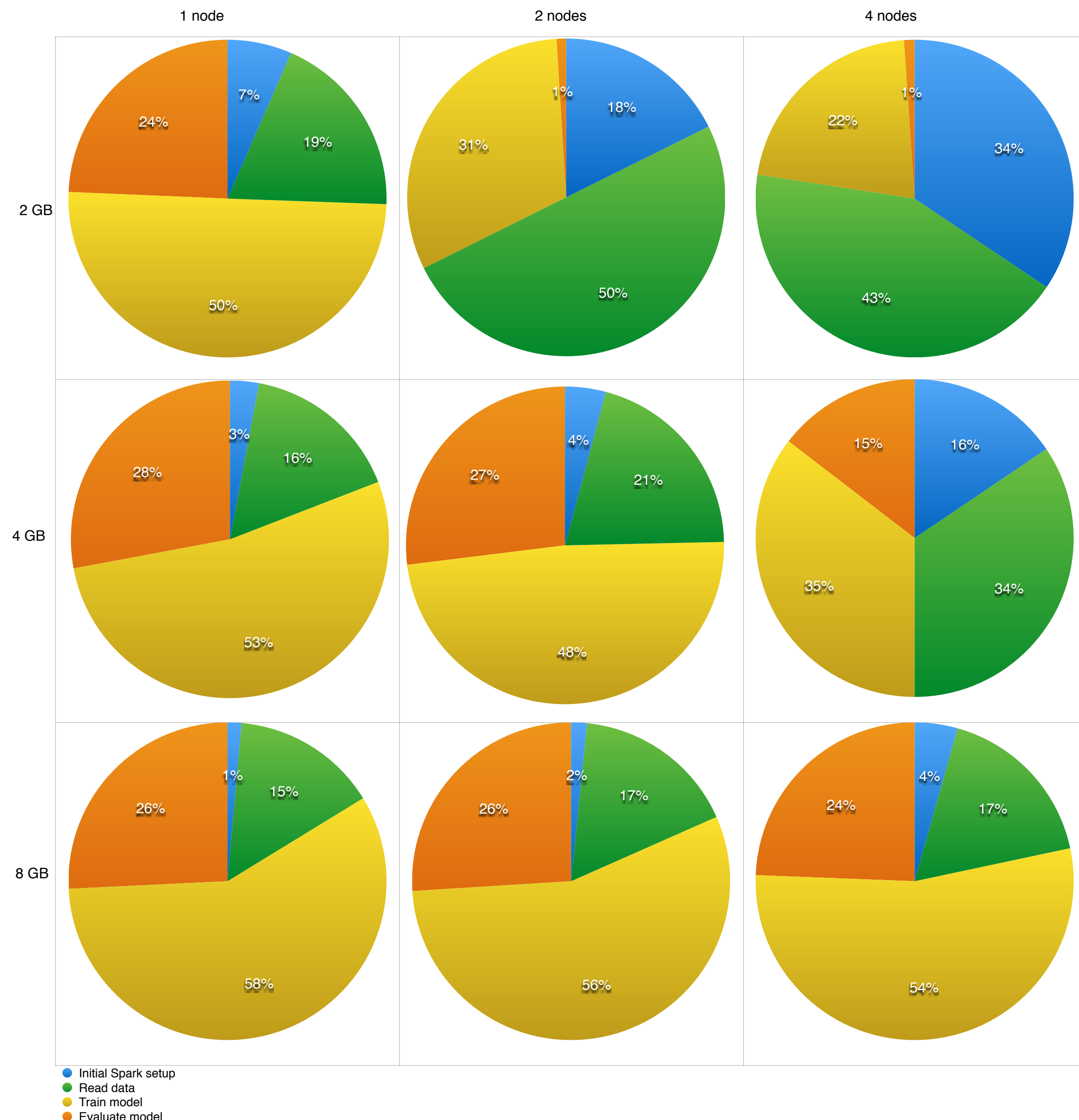


Figure 29. Share of total time for different phases for YearPrediction data

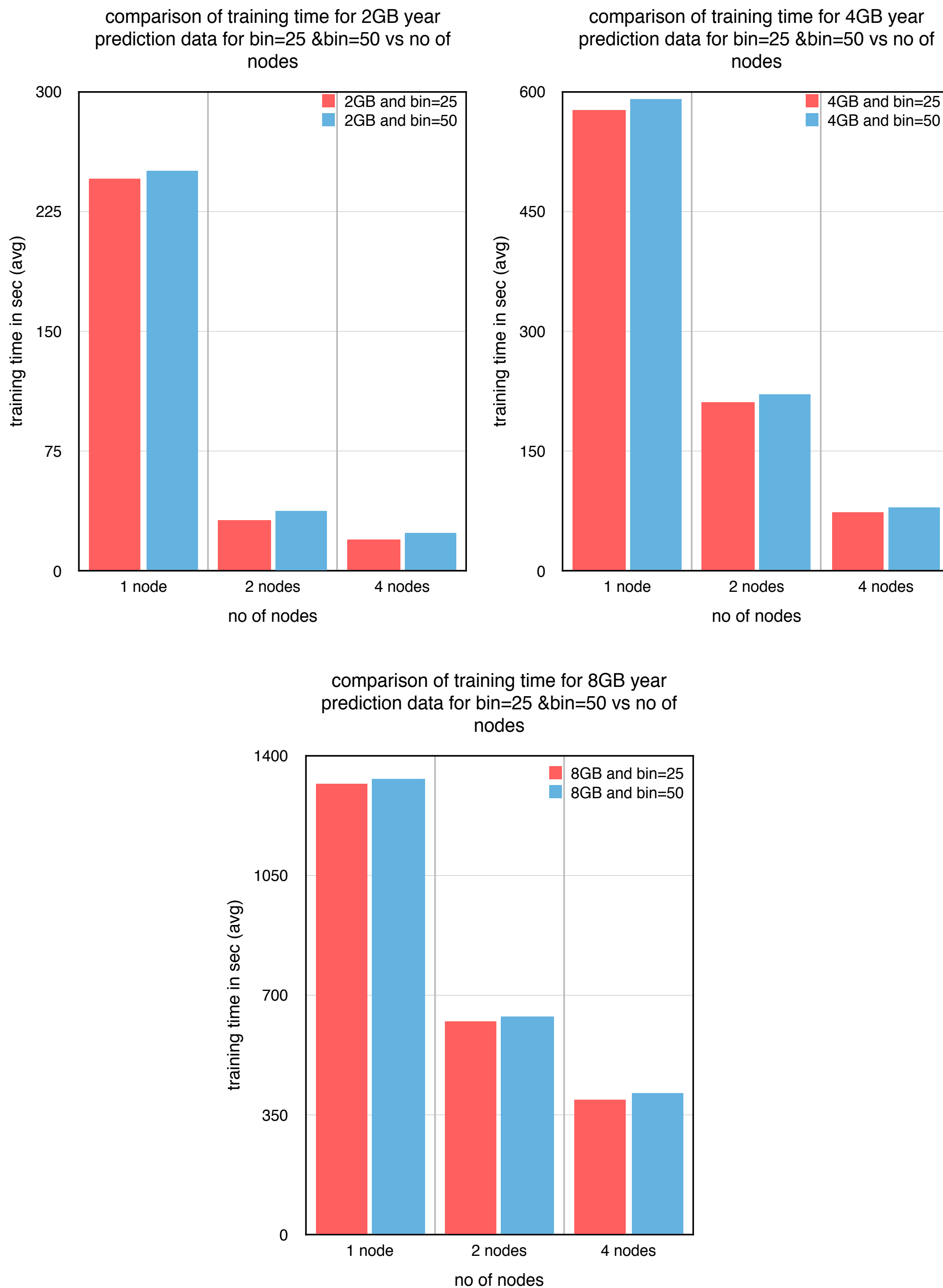


Figure 30. Comparison of training time for Year Prediction data using different bin values

In this experiment, we have also tried to measure one more criterion that may affect the training time of decision tree models. This criterion is the maximum number of bins. We conducted the experiments by using two values of maximum number of bins: 25 and 50. In this data, we were successfully able to increase the maximum number of bins because of the low dimensionality of the features in this data. Figure 30 compares the training time for

the two values of maximum number of bins. As expected, the training time increased for the larger bin value for all data sizes. We could not further increase the bin value due to the memory limitations of this set-up.

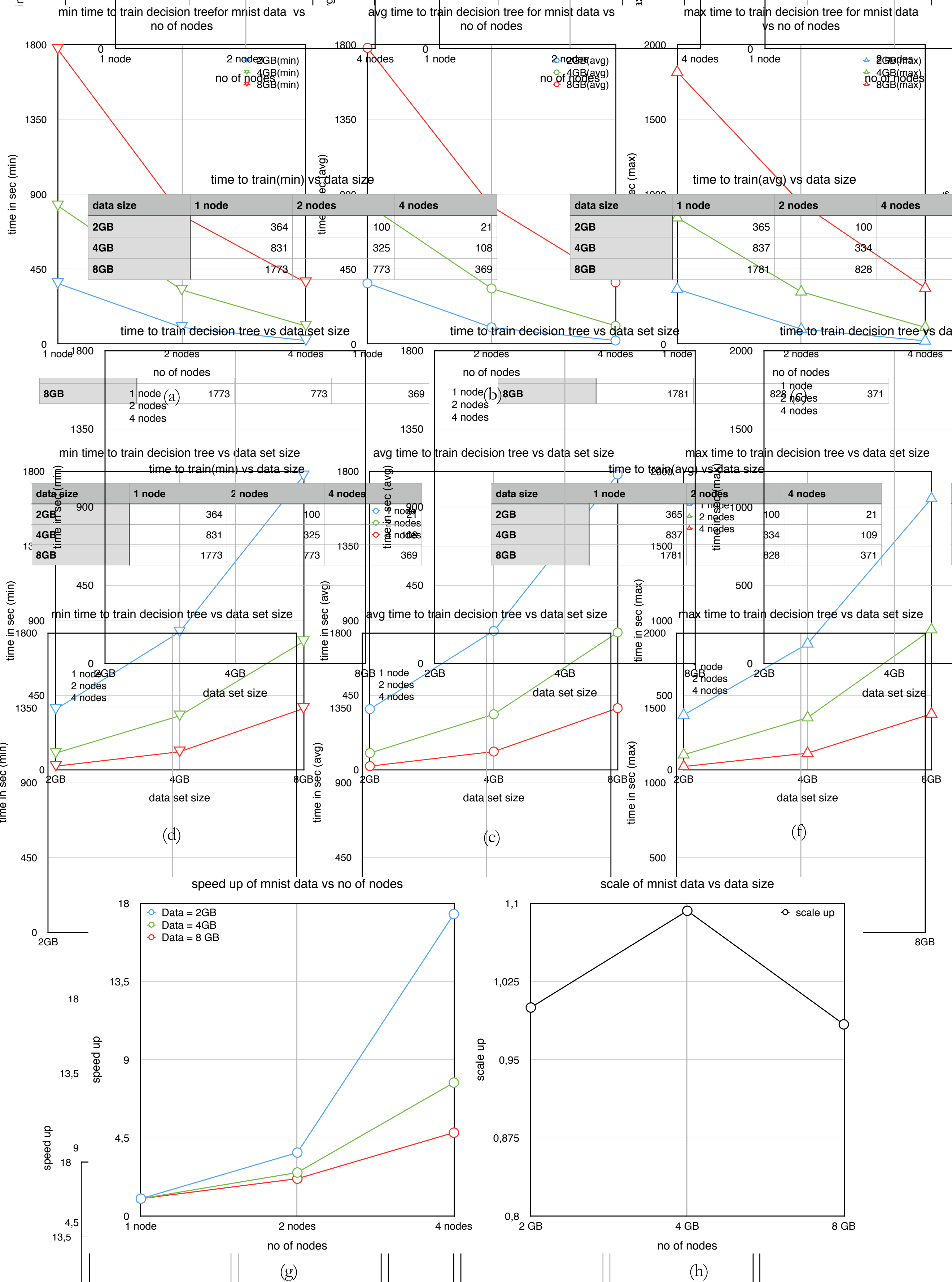
8.5 Experiment 4 With Mnist Data (Multi-Class Classification)

We perform decision tree learning based multi-class classification using Mnist data. The Mnist data has slightly more number of training instances as compared to the YearPrediction data for same data sizes. Mnist data is a very high dimensional data (it has 780 number of features). After running various experiments, we found out that we could not use all the features of this data without running into out-of-memory problems. As such, we decided to use only a sub-set of the features of this data. MLlib allows this functionality which enables us to specify the first n features that we want to use for the decision tree construction. Therefore, we have used the first 100 features to build the decision tree of depth 6 and maximum bin value of 25. Figure 31 and 32 show the experimental results using these parameters.

Figure 31 (a), (b) and (c) shows the minimum, average and maximum values of the training time for decision tree building versus the number of nodes given a specified data size. Here, we see considerably good performance gains for all data sizes and for all node values. We can see that there is a considerable reduction in run-time for training the model for the 8 GB data by increasing the number of nodes from 1 to 4. This shows that more and more data are being fetched into the memory and being processed in parallel which significantly reduces the amount of run-time. However, this absolute reduction in running time decreases as the data sizes decreases.

Figure 31 (d), (e) and (f) shows how the minimum, average and maximum values of training time changes with respect to the data sizes when the number of nodes is kept fixed. We see that the increase in training time when the data size is increased from 4 to 8 GB is much greater than the increase in training time when the data size is increased from 2 to 4 GB.

In Figure 31 (g), we see the speed-up of this experiment. When the number of nodes is increased to 2, we see that the speed-up for all data sizes are close to each other. This points to the fact that the total memory of 2 nodes is too small for all data sizes. Therefore, for all the data sizes, most of the data processing is done by fetching the data from the disk. Thus, there is a negligible benefit from in-memory computation because only a very small proportion of data is fetched into the memory. But when the number of nodes is increased to 4, we see a considerable jump in the speed-up for the 2 GB data (from 3,65 for 2 nodes to 17,3 for 4 nodes). We can conclude that there is a considerable increase in the proportion of data that is fetched into the memory when the number of nodes is increased from 2 to 4. This means that mostly on-disk processing of 2 nodes moves to mostly in-memory processing of 4 nodes. For the 4 GB data, we also see a considerable jump in the speed-up which is much better than a linear speed-up (2,5 for 2 nodes to 7,7 for 4 nodes). However , 8 GB data is still too large for the total memory of 4 nodes. Therefore, we



time to train(min) vs data size

data size	1 node	2 nodes	4 nodes
2GB	364	100	21
4GB	831	325	108
8GB	1773	773	369

time to train(avg) vs data size

data size	1 node	2 nodes	4 nodes
2GB	365	100	21
4GB	837	334	109
8GB	1781	828	371

Figure 31. Experimental results for Mnist data using training time

Table 4

	scale up
2 GB	1
4 GB	1,09282437
8 GB	0,983832749

observe only a slightly better speed-up than a linear speed-up in this case (2,2 for 2 nodes and 4,8 for 4 nodes).

Figure 31 (h) shows the scalability of this experimental set up with respect to increasing data size. This system scales well both for both data sizes. Both scale-up values are very close to 1.

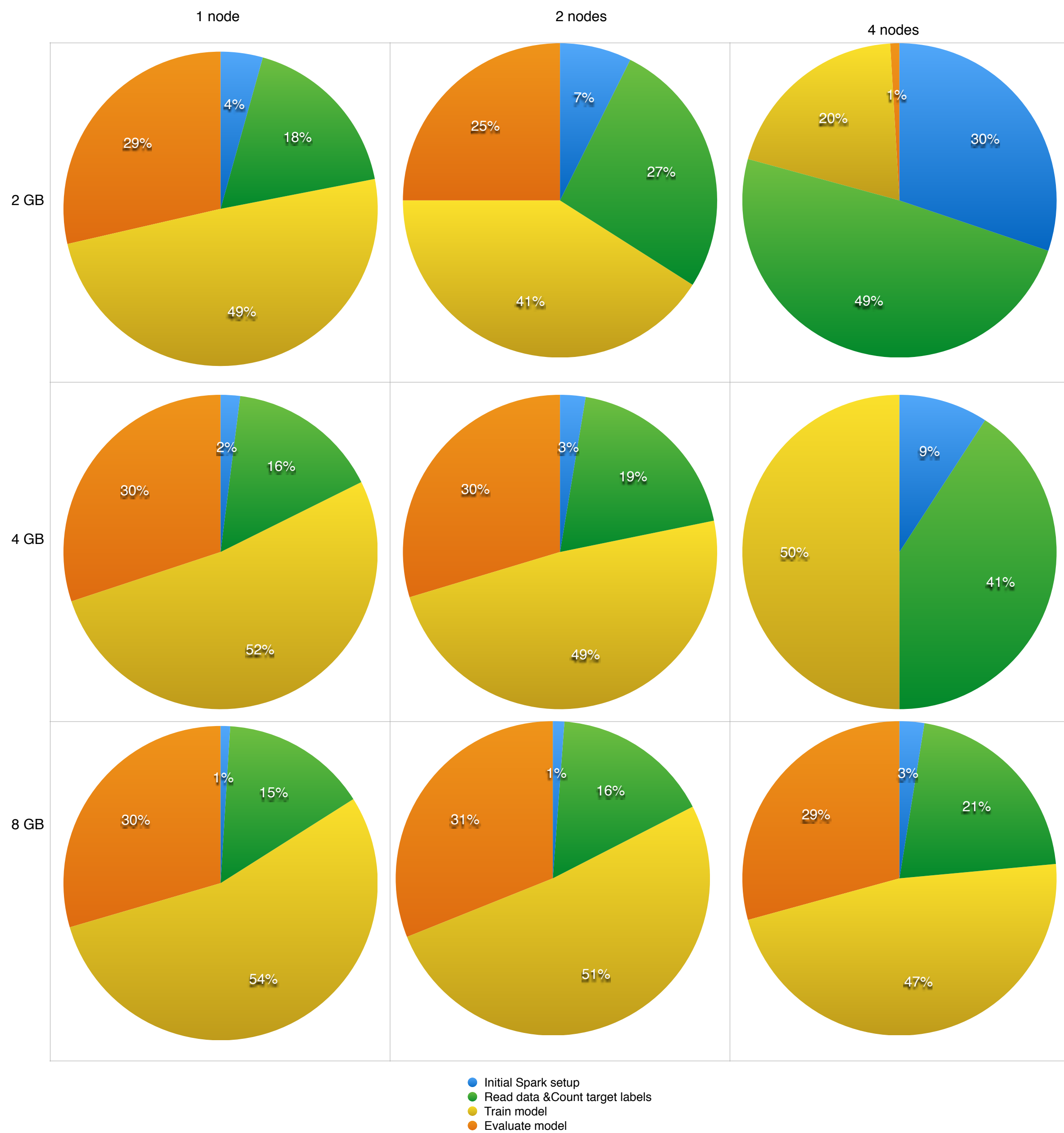


Figure 32. Share of total time for different phases for Mnist data

Training Time for mnist100

No.of nodes	2GB(min)	2GB(avg)	2GB(max)	4GB(min)	4GB(avg)	4GB(max)	8GB(min)	8GB(avg)	8GB(max)
1 node	364	365	367	831	837	844	1773	1781	1802
2 nodes	100	100	100	325	334	349	773	828	939
4 nodes	21	21	21	108	109	111	369	371	374

In Figure 32, we can see that the proportionate share of the total time for the training phase rem[illegible]ost constant around 50 % but decreas[illegible]lerably for 2 GB data and 4 nod[illegible]se training phase is immensely fastene[illegible]he small size of data and the more memory availability.

Training Time for mnist data with dim=150 & bin=25

No.of nodes	2GB(avg)	4GB(avg)	8GB(avg)
1 node	[illegible]	854	1837
2 nodes	108	[illegible]	[illegible]
4 nodes	26	122	381

Training Time for mnist data with dim=200 & bin = 25 (sec)

No.of nodes	2GB(avg)	4GB(avg)	8GB(avg)
1 node	404	904	1943
2 nodes	128	357	937
4 nodes	31	143	415

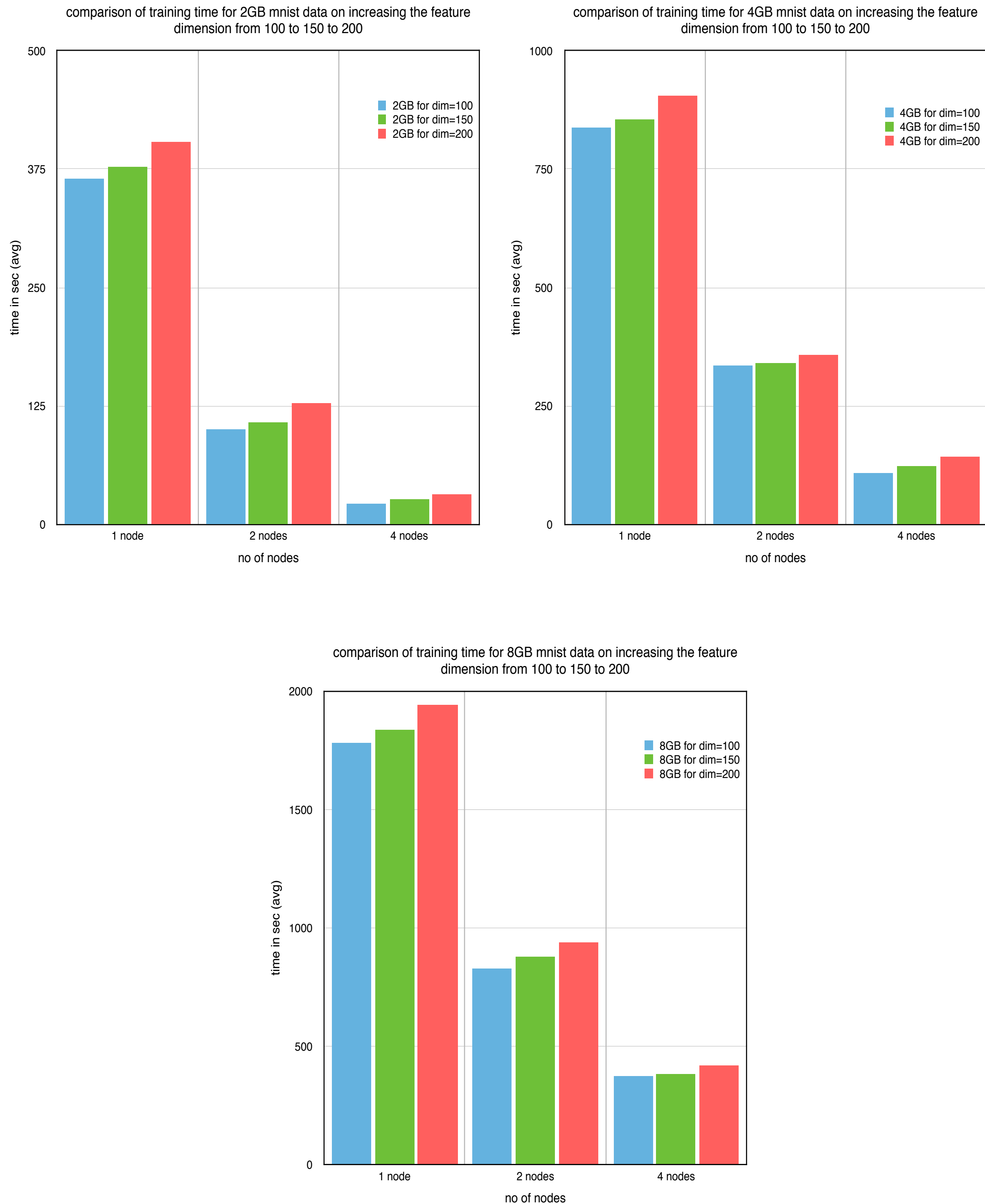


Figure 33. Comparison of training times for different sizes of feature dimension of Mnist data

Using the Mnist data, we also aim to conduct more experiments to analyse the impact of the dimension of features on the training time. We perform the experiments by increasing the feature dimension from 100 to 150 and then to 200. Using these values for feature dimensions gave similar graphs for speed-up and scale-up and therefore they are omitted for space. We compare the training time for training the model by using different sizes of feature dimensions. This comparison is shown in Figure 33. Here, we can clearly see that the training time increases when we increase the size of feature dimension.

8.6 Conclusion

Here, we summarise the general conclusions from our experiments. In a typical parallel distributed experiment, our aim is to achieve a linear speed-up and a linear scale-up. But such ideal speed-up and scale-up is hard to obtain because there is always a non-trivial overhead of distributed systems to the normal program execution time. Therefore, we expected to achieve an almost linear speed-up and scale-up with this distributed system using Spark, YARN and HDFS. We analyse the performance of decision tree based classification and regression algorithms implemented in Spark's MLlib which runs on this system.

We found out that when the memory is large enough to cache all the input data and the system could perform most of the data processing in the memory, the system consequently yielded excellent results. We achieved much better than expected performance gains, speed-up and scale-up for decision tree learning algorithms for classification and regression data mining in MLlib. The in-memory caching effect of Apache Spark enabled much better performance benefits than the simple Hadoop MapReduce. Speed-up was sometimes much higher than a linear speed-up. It happened in cases when increasing the number of nodes and thereby increasing the total amount of main memory to cache the training tuples moved the previously mostly on-disk processing to mostly in-memory computations. This clearly shows the advantage of in-memory computations for highly iterative decision tree learning algorithms that are implemented in MLlib. In our experiments we have achieved the best speed-up for 2 GB Mnist data where the speed-up is almost 18 times when we increase the number of nodes by just 4 times. This is particularly due to the reason that most of the RDDs were lying on the disk in case of 1 node but most of these RDDs move to the main memory cache when the number of nodes was increased to 4. The scale-up was always maximum in case of 4 GB data. The scale-up for 4 GB data and 2 nodes was even more than 1 in all the experiments with different datasets. However, this scale-up dropped slightly for 8 GB data and 4 nodes. For regression algorithm, this drop was more than classification algorithms. Since we measure scale-up for 8 GB data by increasing the node to 4, there are two factors that brings down the scale-up: Firstly, there was increased communication overhead in case of 4 nodes due to shuffle of large amounts of intermediate data between map and reduce; Secondly, increasing the data to 8 GB implied lesser proportion of the total data was processed in memory.

Apache Spark's MLlib was not so much advantageous in the case of very large data sizes containing very high number of training instances. In that case, most of the RDDs were lying on the disk and few proportion of them could be fetched into main memory. This means

disk based computations overly dominated the in-memory computations. In this case, garbage collectors also ran more frequently to make space for the new RDDs of task execution in every iteration. This also interfered in the run-time of program execution. Despite of the garbage collection mechanisms impacting the run-times, the speed-up and scale-up were found to better or at-least as good as the original Hadoop MapReduce based decision tree learning implementations. From this, we can conclude that for larger sizes, the proportion of the total data cached in memory became very low and the benefit from in-memory computations started to diminish. As a result, the performance gains from Spark based decision tree learning algorithms started to approach the performance of original MapReduce based decision tree learning algorithms that we saw in section 6.1.6, 6.1.7 and 6.1.8. So, we can derive a conclusion that the Apache Spark's distributed processing model behaved more like the original MapReduce from Hadoop when most of the data processing is done on-disk rather than in-memory.

There is no general statement about the hardware requirements or characteristics of cluster of machines that should be met for running applications for Apache Spark. We found out that the basic hardware requirement for Spark is a cluster of high-end machines as Spark applications have typically higher memory requirements. This is in contrast to the hardware requirements for Hadoop MapReduce based applications. It is a known fact that Hadoop was built around large clusters of low-end commodity servers. Apache Spark applications have a higher demand for memory but not on CPU cores and disks. However, the main memory is becoming increasingly cheap and we can see nowadays that most of the systems are equipped with much higher RAMs. This requirement for high amount of RAM is specifically true for decision tree based learning algorithms because because there is a higher memory requirements for histogram computations in the later iterations of this algorithm. This requires garbage collection methods of JVM to evict the old RDDs to make space for new RDDs. When we have a cluster of machines with low amounts of memory, we should not be too optimistic about processing very large volumes of data with very high feature dimension.

During our work, we found out that the Spark's run-time was throwing a lot of out-of-memory errors when we attempted to increase the maximum depth of the tree, the feature dimension or the maximum number of bins. We were being overly optimistic about the capabilities of our small cluster with low-end virtual machines. The conclusion is that when we have small number of machines with low amounts RAM, we should keep the depth of tree to a low value. In our case, this value should be higher than 6. We should try to use a lower dimension of data and use a lower value of maximum number of bins to discretise the continuous features, to avoid memory problems.

The requirements of MLlib' s decision tree algorithms in the terms of input data is rather very strict. As we have seen in the data preparation stage earlier in section 7.1, when we use Java API to develop programs that does decision tree learning of MLlib, data should be in specific format: LIBSVM format. For both CSV (which we can convert to LIBSVM when using Scala API) and LIBSVM formats of input files, there is a very strict requirement in case of target labels. For binary classification, the labels should be 0 and 1 and for multi-class classification, the labels should start from: 0,1,2…. to n-1. If the labels are not specified in this manner, MLlib run-time will throw run-time exceptions (ArrayIndexOutOfBounds

Exceptions). The current version of MLlib can handle only numerical data for both the feature vector and the labels. There should no extra or trailing white-spaces in the data because MLlib cannot handle them out-of-box. We found out that YearPrediction data had double whitespaces between labels and the feature vector which cause the MLlib to interpret the feature vector as NULL and throw run-time exceptions.

By comparing our experiments to Hadoop based MapReduce implementations in section 6.1.6, 6.1.7 and 6.1.8, we can see the ease of application of Spark MLlib for large-scale data mining. In case of Hadoop, the speed-up and scale-up was generally very poor for smaller datasets which demonstrates the outcome that Hadoop MapReduce has a high overhead which dominates in the case of small dataset. This speed-up and scale-up improved for larger dataset because the high MapReduce overhead is smaller in comparison to very long running times for processing very large datasets. However, we found a completely opposite outcome in our experiments. The speed-up and scale-up was excellent for smaller dataset but it dropped for larger datasets because of diminishing effect of in-memory computations.

9. Conclusion Of This Work

In tis chapter, we summarise the overall conclusion of our work. Nowadays, there are multiple alternative approaches for large-scale distributed data mining analytics. We are not only limited to the strict batch-mode data processing of old Hadoop MapReduce. Although Hadoop MapReduce is still being predominantly used for distributed processing of data, there is a significant shift in the solutions for data mining and analytic tools. These tools are already in the process of a complete overhaul in favour of Apache Spark. Therefore, it was interesting for us to evaluate the data mining capabilities of machine learning libraries provided by Apache Spark. The Machine learning library from Apache Spark demonstrated the superb performance benefits of distributed large-scale data mining analytics. The performance improvements, speed-ups and scale-ups were in most of the cases better than expected. Even in the case of very larger datasets, the performance benefits were substantial and useful. Our experiments clearly show the effectiveness of distributed data processing (with MapReduce-like data processing model of Apache Spark) for large-scale data mining analytics. This effectiveness is specifically proven for decision tree learning based classification and regression analysis of large-scale data.

Apache Spark is definitely not without problems. We felt that it requires some optimisations, fixes, and a deeper integration with Hadoop YARN. It doesn’t yet provide the depth of information about its library functions that traditional data mining tools do. But as a data mining platform, it is already sufficiently interesting for data scientists to look at it seriously.

Future work could include implementing decision tree learning based data mining algorithms using old Hadoop MapReduce which are equivalent to the implementation of algorithms that we used from Spark’s MLlib and compare them. This would require investigating the whole source code of Apache Spark which will give us an idea about the general design of this algorithm. This will give us an opportunity to compare the distributed processing engine of Spark and Hadoop MapReduce. We could also provision higher end machines to analyse the behaviour of Spark’s distributed processing model for building trees of more depths.

References

[1] List of available Data Mining Tools and Software (exemplar)
unknown author, found in:
http://en.wikipedia.org/wiki/Data_mining,
2014, last accessed 03.06.2014,

[2] C. Snijders, U. Matzat & U.-D. Reips:
Big Data. Big gaps of knowledge in the field of Internet,
Eindhoven University of Technology, The Netherlands; University of Deusto, Spain;
IKERBASQUE, Basque Foundation for Science, Spain, in:
International Journal of Internet Science 2012, 7 (1), 1–5,
Eindhoven, 2012

[3] Jeffrey Dean & Sanjay Ghemawat:
MapReduce. Simplified data processing on large clusters, in:
Proceedings of the 6th Symposium on Operating System,
Design and Implementation (OSDI 2004),
San Francisco CA USA, 2004, pages 137–150,

[4] The Apache Software Foundation, found in:
http://hadoop.apache.org;
last accessed on 11.10.2014

[5] The Apache Software Foundation, found in:
http://spark.apache.org
last accessed 12.12.2014

[5a] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,
Murphy McCauley, Michael J. Franklin, Scott Shenker & Ion Stoica:
Fast and Interactive Analytics over Hadoop Data with Spark,
University of California, Berkeley,
Berkeley CA USA, *unkown*,

[6] Johannes Gehrke, Raghu Ramakrishnan, Venkatesh Ganti:
RainForest. A Framework for Fast Decision Tree Construction of Large Datasets,
Department of Computer Sciences, University of Wisconsin-Madison, in:
Proceedings of the 24th VLDB Conference,
New York USA, 1998

[7] Johannes Gehrke, Venkatesh Ganti, Raghu Ramakrishnany & Wei-Yin Lohz:
BOAT - Optimistic Decision Tree Construction,
Department of Computer Sciences and Department of Statistics,
University of Wisconsin-Madison,
Madison USA, unknown

[8] P. Norvig, A. Halevy & Fernando Pereira:
Unreasonable Effectiveness of Data,
Google Inc., in:
Expert Opinion, by IEEE Computer Society,
unknown, 2009

[9] Gabriel Recchia & Michael N. Jones:
More data trumps smarter algorithms.
Comparing pointwise mutual information with latent semantic analysis,
Indiana University, Bloomington, Indiana, in:
Behavior Research Methods 2009, by The Psychonomic Society, Inc.,
unknown, 2009

[10] Jamie Shotton, Andrew Fitzgibbon, Mat Cook, Toby Sharp, Richard Moore, Alex Kipman, Andrew Blake & Mark Finocchio:
Real-Time Human Pose Recognition in Parts from Single Depth Images,
Microsoft Research, Cambridge & Xbox Incubation,
unknown, unknown

[11] Eric Brill:
Processing Natural Language without Natural Language Processing,
Microsoft Research, Redmont WA USA, in:
A. Gelbukh (Ed.): CICLing 2003, LNCS 2588,
Berlin & Heidelberg, 2003, pp. 360–369

[12] Anand Rajaraman:
More data usually beats better algorithms; found in:
http://anand.typepad.com/datawocky/2008/03/more-data-usual.html
2008, last accessed 15.06.2014

[13] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng & Kunle Olukotun:
Map-Reduce for Machine Learning on Multicore,
CS. Department Stanford University & Rexee Inc.,
Standford CA USA, unknown

[14] K. Bhaduri, K. Das, K. Liu, H. Kargupta and J. Ryan:
Distributed Data Mining Bibliography,
Computer Science and Electrical Engineering Department,
University of Maryland Baltimore County,Baltimore, found in:
http://www.csee.umbc.edu/~hillol/DDMBIB/ddmbib.pdf
Baltimore MD USA, 2008

[15] J. Dean & S. Ghemawat:
MapReduce. Simplified data processing on large clusters,
Google Inc., in:
OSDI '04 . 6th Symposium on Operating Systems Design and Implementation,
by USENIX Association,
unknown, 2004, p. 137-149

[16] David Gleich:
Why MapReduce is successful. It's the IO!; found in:
http://dgleich.wordpress.com/2012/10/05/why-mapreduce-is-successful-its-the-io
2012, last accessed 30.06.2014

[17] Yahoo! Developer Network:
Yahoo! Hadoop Tutorial,
Yahoo! Inc., found in:
https://developer.yahoo.com/hadoop/tutorial/
last accessed 04.10.2014

[18] Amy Xuyang Tan, Valerie Li Liu, Murat Kantarcioglu & Bhavani Thuraisingham:
A Comparison of Approaches for Large-Scale Data Mining.
Utilizing MapReduce in Large-Scale Data Mining,
Department of Computer Science, The University of Dallas, in:
Technical Report UTDCS-24-10,
Dallas TX USA, 2010

[19] R. Ladner & F. E. Petry (Eds.):
Net-Centric Approaches to Intelligence and National Security,
by Springer Science+Business Media, Inc.,
New York USA, 2010

[20] Data Management and Interchange Secretariat: United States of America (ANSI):
ISO/IEC JTC1/SC32 WG4 SQL/MM Part 6 WD,
Administered by Pacific Northwest National Laboratory on behalf of ANSI,
Washington USA, 2000

[20a] Jimmy Lin & Chris Dyer:
Data-Intensive Text Processing with MapReduce,
Draft of January 27, 2013,
by Morgan & Claypool Publishers,
unknown, 2013

[21] G. Aloisioa, S. Fiorea, I. Foster & D. William:
Scientific big data analytics challenges at large scale,
Euro-Mediterranean Center on Climate Change, Italy, University of Salento, Italy,
Computation Institute, University of Chicago and Argonne National Laboratory, Chicago,
Lawrence Livermore National Laboratory, Livermore, CA, USA,
unknown, unknown

[22] Oracle Online Database Documentation:
What is Data Mining? Oracle Data Mining Concepts,
Oracle Corp., found in:
https://docs.oracle.com/cd/B28359_01/datamine.111/b28129/process.htm#CHDEFGIE
2014, last accessed 05.12.2014

[23] Jiawei Han, Michael Kamber & Jian Pei:
Data Mining Concepts and Techniques,
University of Illinois at Urbana–Champaign & Simon Fraser University,
by Morgan Kaufmann Publishers,
Waltham MA USA, Third Edition 2012

[24] R. Agrawal, T. Imieliński & A. Swami:
Mining association rules between sets of items in large databases,
Proceedings of the 1993 ACM SIGMOD International Conference on Management of data,
unknown, 1993, p. 207.

[25] Mohammed J. Zaki & Limsoon Wong:
Data Mining Techniques,
Department of Computer Science, Rensselaer Polytechnic Institute Troy, New York, USA,
Institute for Infocomm Research, Singapore,
Troy NY USA, Singapore, 2003

[26] V. Estivill-Castro:
Why so many clustering algorithms — A Position Paper,
University of Newcastle, Callaghan, NSW, Australia
by ACM SIGKDD Explorations Newsletter 4 (1): doi:10.1145/568574.568575,
Callaghan Australia, 2002, p. 65–75.

[26a] L. Rokach & O. Maimon:
Data Mining with Decision Trees: Theory and Applications,
Ben Gurion University & Tel-Aviv University, Israel,
by World Scientific Pub Co Inc.,
Singapore, 2008

[27] J. Ross Quinlan:
Induction of Decision Trees
by Kluwer Academic Publishers,
Boston, 1986, p. 81-106

[28] J. Ross Quinlan:
C4.5. Programs for Machine Learning,
by Morgan Kaufmann Publishers,
Amsterdam etc., 1993

[29] Leo Breiman, J.H. Friedman, R.A. Olshen, C.J. Stone:
Classification And Regression Trees, in:
The Wadsworth Statistics / Prohability Series,
by Chapman and Hull / CRS
Boca Raton etc., 1984

[30] Apache Software Foundation:
Hadoop Wiki. MapReduce, found in:
http://wiki.apache.org/hadoop/MapReduce
last edited 2011, last accessed 12.10.2014

[31] Konstantin Schvachko, Hairong Kuang, Sanjay Radia & Robert Chansler:
The Hadoop Distributed File System,
Yahoo! Inc., in:
Proceedings of MSST2010, by IEEE,
unknown, 2010

[32] Sanjay Ghemawat, Howard Gobioff & Shun-Tak Leung:
The Google File System,
Google Inc., in:
SOSP'03, October 19–22, 2003,
Bolton Landing NY USA, 2003

[33] Tom White:
Hadoop. The Definitive Guide,
by O'Reilly Media Inc
Cambridge etc., Third Edition 2011,

[34] The Apache Software Foundation:
Hadoop HDFS Architecture Guide, found in:
http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
last edited 2013, last accessed 07.12.2014

[35] The Apache Software Foundation:
Hadoop MapReduce Tutorial, found in:
http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html
last edited 2013, last accessend 07.12.2014

[36] The Apache Software Foundation:
Apache Hadoop NextGen MapReduce (YARN), found in:
http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
last edited 2014, last accessed 13.12.2014

[37] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker & Ion Stoica:
Spark: Cluster Computing with Working Sets,
University of California, Berkeley,
Berkeley CA USA, 2010,

[37a] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,
Murphy McCauley, Michael J. Franklin, Scott Shenker & Ion Stoica:
Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing,
University of California, Berkeley,
Berkeley CA USA, *unkown*,

[38] Apache Software Foundation:
Spark Programming Guide, found in:
http://spark.apache.org/docs/latest/programming-guide.html
last accessed 08.12.2014

[39] Apache Software Foundation:
Cluster Mode Overview of Spark, found in:
http://spark.apache.org/docs/latest/cluster-overview.html
last accessed 06.12.2014

[40] http://en.wikipedia.org/wiki/Binary_classification

[41] Christopher M. Bishof:
Pattern Recognition and Machine Learning,
by Springer,
Cambridge MA USA, 2006,

[42] Apache Spark Online Documentation,
package found in:
http://spark.apache.org/docs/1.1.0/api/java/
last accessed 10.12.2014

[43] Apache Spark Online Guide
MLlib - Decision tree, found in:
https://spark.apache.org/docs/1.1.0/mllib-decision-tree.html
last accessed 09.12.2014

[44] M. Amde & J. Bradley:
Scalable Decision Trees in Mllib, found in:
http://databricks.com/blog/2014/09/29/scalable-decision-trees-in-mllib.html
last edited 2014, last accessed 10.12.2014

[45] Weizhong Zhao, Huifang Ma & Qing He:
Parallel K-means Clustering Based on MapReduce.
The Key Laboratory of Intelligent Information Processing at Institute of Computing
Technology, Chinese Academy of Sciences,
Graduate University of Chinese Academy of Sciences, in:
Cloud Com 2009, edit. by M.G. Jaatun, G. Zhao & C. Rong,
Berlin & Heidelberg, 2009, p. 674-679

[46] Ying-ting Zhu, Fu-zhang Wang, Xing-hua Shan & Xiao-yan Lv:
K-medoids Clustering Based on MapReduce and Optimal Search of Medoids, in:
The 9th International Conference on Computer Science & Education (ICCSE 2014),
Vancouver Canada, 2014

[47] Spiros Papadimitriou & Jimeng Sun:
DisCo: Distributed Co-clustering with Map-Reduce.
A Case Study Towards Petabyte-Scale End-to-End Mining,
BM T.J. Watson Research Center Hawthorne, NY, USA, in:
2008 Eighth IEEE International Conference on Data Mining,
Hawthorne NY USA, 2008

[48] Ning Li, Li Zeng, Qing He & Zhongzhi Shi:
Parallel Implementation of Apriori Algorithm Based on MapReduce, in:
2012 13th ACIS International Conference on Software Engineering, Artificial
Intelligence, Networking and Parallel/Distributed Computing,
Beijing & Boading PR China, 2008

[49] Vasile Purdila & Ştefan Gheorghe Pentiuc:
MR-Tree. A Scalable MapReduce Algorithm for Building Decision Trees,
Stefan cel Mare University of Suceava, Romania, in:
Journal of Applied Computer Science & Mathematics, no. 16 (8),
Suceava Romania, 2014

[50] Wei Dai & Wei Ji:
A MapReduce Implementation of C4.5 Decision Tree Algorithm,
School of Economics and Management, Hubei Polytechnic University, Huangshi PR China,
International Journal of Database Theory and Application Vol.7, No.1
unknown, 2014, pp.49-60

[51] B. Panda, J. S. Herbach, S. Basu, & R. J. Bayardo:
PLANET: Massively parallel learning of tree ensembles with MapReduce,
Google Inc., in:
VLDB 09, August 24-28,
Lyon France, 2009

[52] Apache Mahout Introduction:
What is Apache Mahout?, found in:
http://mahout.apache.org/
last accessed 12.11.2014

[53] Michael Sevilla:
Applicability of MAHOUT for Large Data Sets. Experiences and Lessons Learned,
University of California, Santa Cruz
Santa Cruz CA USA, unkown

[54] http://en.wikipedia.org/wiki/Weka_(machine_learning)

[55] Mark Hall (Core developer of the Weka data mining software):
Data Mining & Weka, found in:
http://markahall.blogspot.co.nz/2013/10/weka-and-hadoop-part-1.html,
2013, last accessed 20.08.2014

[56] Apache Spark Online programming Guide for MLlib library:
MLlib - Data Types, found in:
https://spark.apache.org/docs/latest/mllib-data-types.html
last accessed 13.12.2014

[57] The MathWorks, Inc.:
MATLAB Proprietary Software, found in:
http://de.mathworks.com/products/matlab/
last accessed 01.11.2014

[58] FastML. Machine learning made easy:
Introducing phraug, found in:
http://fastml.com/introducing-phraug/
2013, last accessed 02.12.2014

[59] Rong-En Fan:
LIBSVM Data: Classification, Regression, and Multi-label, found in:
http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/
2011, last accessed 10.10.2014

[60] K. Bache & M. Lichman:
UCI Machine Learning Repository,
University of California, School of Information and Computer Science, Irvine, CA, found in:
http://archive.ics.uci.edu/ml
Irvine CA USA, 2013

[61] S. Sonnenburg & V. Franc:
Index of Large Scale Data,
Machine Learning Group, Technische Universität Berlin, found in:
ftp://largescale.ml.tu-berlin.de/largescale
Berlin, 2010, last accessed 28.10.2014

[62] S. Sonnenburg & V. Franc:
Pascal Large Scale Learning Challenge,
Machine Learning Group, Technische Universität Berlin, found in:
http://largescale.ml.tu-berlin.de/instructions/
Berlin, 2014, last accessed 28.10.2014

[63] Y. LeCun, C. Cortes & C.J.C. Burges:
The Mnist Database of handwritten digits,
Courant Institute, NY University; Google Labs, New York; Microsoft Research, Redmond
found in: http://yann.lecun.com/exdb/mnist/
New York & Richmond USA, unknown, last accessed 28.10.2014

[64] Apache Software Foundation,
Spark's Machine Learning library Guide,
http://spark.apache.org/docs/latest/mllib-guide.html
unkown, last accessed 11.12.2014

[65] Oracle Inc
Java SE 6 HotSpot Virtual Machine Garbage Collection Tuning
http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html
unknown, last accessed: 11.12.2014

List Of Figures

Figure 1 : Value chain of Information Gain
Figure 2: Data mining process
Figure 3. Data mining methods and its categories
Figure 4. Two-step classification and regression data mining
Figure 5. Decision tree for the date mining goal: “buy a smartphone”
Figure 6. Overall MapReduce programming model architecture
Figure 7. HDFS architecture and the flow of execution for read and write operations
Figure 8. Hadoop MapReduce version 1.x (MRv1)
Figure 9. Architecture and job execution flow in Hadoop MapReduce version 1.x (MRv1)
Figure 11. Architecture and job execution flow of Hadoop YARN
Figure 12. An example for Spark program for log mining
Figure 13. Architecture and job execution flow for Spark in cluster mode
Figure 14. Spark on Yarn Execution Cycle scenario for 2 slave nodes
Figure 15. Task Scheduling Process in Apache Spark.
Figure 16. Ensemble approach for Naive Bayes Classifier
Figure 17. MapReduce Classifier Model
Figure 18. YARN Cluster showing number of nodes, their respective roles and the network topology
Figure 19. An overview of YARN daemons and Spark processes running on different nodes
Figure 20. Memory is allocation to each executor on a node
Figure 21. Labeled points for multi-class classification (an example)
Figure 22. An example for LIBSVM format for corresponding CSV format
Figure 23. Stages in decision tree learning for regression and binary classification in MLlib
Figure 24. Experimental results for Alpha data using training time
Figure 25. Share of total time for different phases for Alpha data
Figure 26. Experimental results for Blog data using training time
Figure 27. Share of total time for different phases for Blog data
Figure 28. Experimental results for Year Prediction data using training time
Figure 29. Share of total time for different phases for YearPrediction data
Figure 30. Comparison of training time for Year Prediction data using different bin values
Figure 31. Experimental results for Mnist data using training time
Figure 32. Share of total time for different phases for Mnist data
Figure 33. Comparison of training times for different sizes of feature dimension of Mnist data

List Of Tables

Table 1. RDD operations

Table 2. List of Hadoop MapReduce based data mining algorithms in Apache Mahout

Table 3. Number of training instances in 2,4 and 8 GB of YearPrediction data

Table 4. Number of training instances in 2,4 and 8 GB of Blog data

Table 5. Number of training instances in 2,4 and 8 GB of Alpha data

Table 6. Number of training instances in 2,4 and 8 GB of Mnist data

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.
Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.
Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.
Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Unterschrift:

Declaration

I hereby declare that the work presented in this thesis is entirely my own.
I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations.
Neither this work nor significant parts of it were part of another examination procedure.
I have not published this work in whole or in part before.
The electronic copy is consistent with all submitted copies.

Signature: