AN EVALUATION OF THE SPARK PROGRAMMING MODEL FOR BIG DATA ANALYTICS

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Haripriya Ayyalasomayajula

May 2015

AN EVALUATION OF THE SPARK PROGRAMMING MODEL FOR BIG DATA ANALYTICS

Haripriya Ayyalasomayajula

APPROVED:

Dr. Edgar Gabriel, Chairman
Dept. of Computer Science, University of Houston

Dr. Weidong Shi
Dept. of Computer Science, University of Houston

Dr. Dan Price
Honors College, University of Houston

Dean, College of Natural Sciences and Mathematics

Acknowledgments

I take this opportunity to express my deep regards to all the people who helped me in this journey.

I would like to express my profound gratitude to Dr. Gabriel for being a terrific advisor. You have been a great teacher and a mentor. I would like to thank you for all the coffee sessions, for the patience and kindness you have shown in guiding me through out the research and for helping me endlessly, for the freedom and flexibility at work, which is invaluable. It has been an honour and privilege to be under your tutelage and I thoroughly enjoyed working as your TA.

I would like to thank Dr. Shi and Dr. Price for accepting to be my committee members. I greatly appreciate the time and effort you spent in reviewing my thesis and providing valuable feedback.

I would like to express my special thanks to Dr. Venkat Subramaniam, for being a great teacher and mentor and for his constant encouragement.

For the last nine months, PSTL lab has been a second home for me. I would like to thank my lab mates: Hadi, Shweta, Jyothi and Youcef for the amazing work atmosphere. Thank you for the constant encouragement and sharing joy in all my little accomplishments. I truly cherish working with you.

I have been lucky to have loving friends and family who rested infinite faith in my abilities always. I would like to thank my friends Anudeep and Sejal for their love and unfailing support. Pushkar, Maurya for being source of inspiration and for teaching me that coding is bliss! Anisha, Supriya, Akshay, Yashwant, Navya: my friends in India for constant assurance and encouragement and constant advice to

keep calm and relax. Venkat and my roommates Kavya and Anusha for helping me get on with little chores on tough days, My seniors Vasanthi, Kiran, Tejas, Hamza, Madhuri, Varun, Shravan for guiding me; Karthik, Nikhil, Sachin, Jayanthi, Lakshmi, Nilakshi, Sri Lakshmi, Ankit and Mayur for their love and support.

I am extremely thankful to my cousins Vandana, Santhosh, and Chaitanya who stand as huge pillars of moral support. This thesis would not have been possible, without the unconditional love and support of my Mom, Dad and Sister. For the wake up calls, for teaching me the beauty of hard work and to pursue my passion, this thesis is a humble dedication to my family!

Finally, I thank the Almighty, for his choicest blessings without which I would not have come this far.

AN EVALUATION OF THE SPARK PROGRAMMING MODEL FOR BIG DATA ANALYTICS

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Haripriya Ayyalasomayajula

May 2015

Abstract

The focus of companies like Google, Amazon etc. is to gain competitive business advantage from the insights drawn by processing petabytes of data. Big Data refers to data characterized by large volume, great variety, and ubiquitous nature of its sources. MapReduce is a programming model that provides a highly scalable and efficient solution to analyze massive datasets on large-scale commodity clusters. Though Hadoop, its open source implementation became a de facto for parallel processing of batch workloads, it is inefficient for iterative, incremental algorithms, ad hoc queries, and stream processing.

Apache Spark is a general-purpose cluster-computing framework, which supports in-memory data analytics. It preserves the merits offered by Hadoop and overcomes its limitations. This thesis aims at evaluating the performance offered by the Spark programming model for Big Data Analytics. Code has been developed to perform analyses of historic air quality data using Spark and MapReduce. It involved significant development effort and tuning the configuration parameters. It is observed that Spark offers a performance of up to 20% more than MapReduce.

Applying Machine Learning techniques to Big Data forms the core of data analytics. MLib is a scalable machine-learning library, offered by the Spark eco-system. To extend our analyses, we perform clustering on the air-quality dataset and evaluate the performance, clustering quality and usability of K-Means Clustering algorithm implementation provided by Spark MLib library against that of Apache Mahout. We tried to develop code to evaluate Spark's ability to integrate with HBase as a data source. Though the initial test cases ran successfully with small dataset, due to insufficient documentation available currently, this is reserved for future work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A tremendous digital revolution is seen in the modern world today. Personal computers, mobile phones, and tablets have become an inherent part of our lives. Internet and social media have gained predominant importance. A fundamental paradigm shift and a special focus are seen on e-commerce. Internet companies are growing at a massive scale and there is a huge deluge of data being collected everywhere. The value being extracted from these datasets proves to be crucial for many business decisions and hence the growing research in big data analytics. The all-encompassing term big data refers to data that are characterized by gigantic volume, great variety, and ubiquitous nature of its sources. Conventional databases cannot handle these large datasets. Individual machines do not support efficient processing of these datasets, with their current IO and processing capabilities. To address these challenges, there is a proliferation of large-scale commodity clusters with distributed system architectures that are able to store and use the large data sets effectively.

Though it existed in the early 1970s, it was in 2005 that Roger Magoulas from

OReilly media introduced the term big data [1] to the world of computing. Companies had realized the need for collecting larger datasets to be able to generate more accurate correlations, instead of gathering multiple small datasets and merely investing on sophisticated algorithms. Around the same time, there was a decreasing trend seen in the cost and size of secondary storage devices and the network infrastructure. With the growing popularity of grid and cloud computing, large data centers are being used by enterprises big and small, for collecting data.

With millions of users accessing the Internet every hour, web services evolve to be more and more reliable and highly available. Google with its search engine provides suggestions of what the user is querying for and thereby, highly personalized search results within milliseconds. Cutting edge research is seen not only in the area of sophisticated and efficient search algorithms, but also on the required infrastructure that supports extensive parallel processing of queries with high speed and performance. Big data marks a fusion of multiple domains distributed systems, storages, statistics, business intelligence, data mining, artificial intelligence, sciences, and parallel processing.

The measure of how large the data are varies from company to company, from terabytes to petabytes based on the size of the enterprise, its goals, and its interests. Social networking sites such as Facebook, Twitter, and e-commerce websites like Amazon, eBay etc. are thriving to offer highly personalized experience by analyzing the user logs and understanding user behavior. There exists a perfect synergy in the research in industry and academia, in big data analytics, natural language processing and machine learning. From giving customized predictions, to fraud and

spam detection, to analyses made on sensor networks, GPS traces, complex science simulations and health care - genomics, biological and environmental research, public sector, almost every domain now leverages big data analytics to extract valuable information to make strategic business decisions.

1.1 Brief Overview of MapReduce Programming and Hadoop Eco-system

The existing techniques in traditional database management systems and data warehouse tools do not scale to analyze the petabyte datasets. At an infrastructure level, these large datasets cannot be stored on a single machine. Hence, a cluster of commodity PCs that are interconnected using a network is necessary to store the data. On the software front, it is crucial to be able to achieve high-speed access. While performance remains the prime concern, due to the growing demand and to meet the commercial requirements, availability and reliability are very essential. The system must be able to reconcile the failure of any node, at any point of time without affecting the services, making fault-tolerance a critical challenge. Apart from this, in large scale data intensive computations automatic parallelizing, data distribution, scheduling program execution across multiple systems, load balancing, inter-machine communication management, and being able to handle partial execution failures are some of the major challenges encountered by such applications.

While many complex computational problems exist, at the heart of computer science lies the problem of searching and sorting. Researchers at Google presented

to the world, a new programming model called MapReduce [2] which stands as an answer to all the above mentioned problems of achieving efficient processing of massive datasets on large clusters. Some of the famous problems to which MapReduce provided a highly scalable and efficient solution are: determining frequency of occurrences of words in a large corpus of documents, frequency of URL access, finding matching pattern, inverted index, and sorting of petabytes of data spread across thousands of machines of a distributed cluster.

MapReduce is a functional style-programming paradigm where the problem is decomposed into a set of map and reduce tasks. The user provides input as a key/value pair to the map function that generates an intermediate key/value pair set which are passed as an input to the reduce function. The reduce function then merges all the intermediate values associated with the same intermediate key.

Consider the problem of determining the frequency of occurrence of words in a corpus of documents. Using MapReduce, the problem could be solved easily as follows. The input documents are parsed, and one line or text block is assigned to a map instance. For each word the mapper emits (word, 1) where the key is the word and the value is 1 which denotes the occurrence of a word. All the values corresponding to a key go to one reducer and the logic in the reducer aggregates all the values, which computes the total frequency of occurrence of the word. (Word, count) is the output emitted by the reducer. This is shown in fig. 1.1 [3]. This could be easily applied for a large class of problems that involve computing aggregate values and it is highly scalable for very large sized datasets.

Going further, consider the K-Means clustering algorithm [4, 5]. K-Means is a

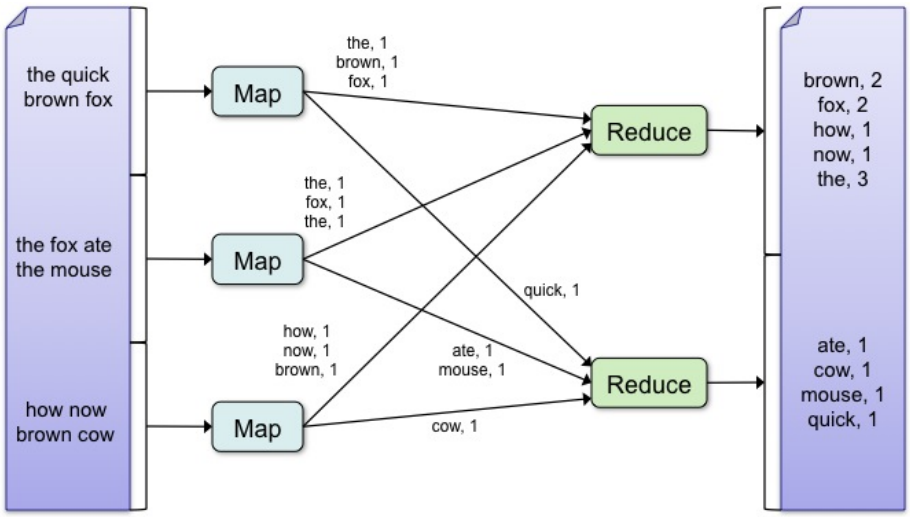

Figure 1.1: Word count example using MapReduce.

very popular unsupervised machine-learning algorithm widely used in many problems. The algorithm iteratively groups a set of data points into clusters. The initial cluster centroids are determined randomly or using a preprocessing algorithm such as Canopy clustering to determine the initial cluster centroids which are provided as input to the K-Means clustering algorithm. In the K-Means implementation each iteration is a MapReduce job. The mapper reads the input, which consists of the data points and the coordinates of data points and calculates the distance of every data point to each centroid. The closest cluster is determined and the mapper outputs (clusterID, coordinatesOfDataPoints) as its output. In the reducer, for each clusterId, the sum of all point coordinates is calculated and emitted as its output and the mean of the cluster is re-computed. A number of MapReduce jobs are run which denote several iterations of K-Means algorithm till the convergence criteria is satisfied.

The master-worker pattern is internally used. Input data are automatically partitioned and distributed across multiple machines. Both the map function and reduce function are distributed. When the user calls a MapReduce function, many copies of the program are started on the cluster of machines. One of the copies is called the master and it controls the rest of the copies - the workers. The master is responsible for distributing the data across the workers and ensuring that all the workers are engaged in successful completion of tasks. In case of any failure, it performs automatic re-scheduling of tasks across the available workers. The intermediate key value pairs generated by the map function is distributed across the multiple workers which run the reduce function. The intermediate values are sorted and then merged by the reduce function which emits them as output.

The input for each task is saved in the local machines to conserve the network bandwidth. The master contains the location information and distribution information of the data and ensures that the map task is scheduled on the machine, which contains a replica of corresponding data. In case of any failure it re-schedules the task to the nearby machine to avoid consuming the bandwidth. Also, to alleviate the problem of the tasks that take unusually long time, which are termed as stragglers, when the MapReduce operation is close to finish, a backup execution of the remaining in-progress tasks is scheduled by the master. When the primary or backup execution is completed, the task is marked complete. Combiner functions, whose functionality is similar to the reduce function can also be used to improve performance. However, the difference is that they are run on the machines after the map task and they write their outputs to the intermediate files.

The ease with which many real-world problems can be expressed as MapReduce computations made this paradigm very popular in academia and industry. Apache Lucene [6, 7], an open source text search library, tried to address the challenges of handling search in big data. Apache Nutch [8], open source web search engine, tried to handle the problems of crawling and indexing websites. Doug Cutting created an open source implementation of MapReduce called Hadoop [7, 9]. Hadoop provides to the programmer with constructs for handling various challenges of distributed computing, which facilitates the programmer to concentrate on the efficient algorithms and program logic instead of spending long hours in taking care of the low-level constructs on a regular basis.

Hadoop provides a shared storage system and an analysis system, which form the core of its stack. A high-level API in JAVA is offered that facilitates MapReduce programming. HDFS [10], the Hadoop distributed file system provides the means of storing data and supports a wide range of formats. Data are replicated across the nodes that help in reading from multiple disks at once. Due to its replication scheme, it is able to provide strong fault tolerance. Further, it addresses a major issue of large-scale data intensive computations moving large data to the disks with executors, which affects bandwidth. Hadoop provides data locality, by moving computation to the disks where the data are present. To address the problems of distributing data among the nodes and handling partial failures, Hadoop provides an execution environment with a YARN. YARN consists of a resource manager that takes care of scheduling jobs and an application master that monitors the progress of tasks.

1.2 Problems and Challenges of MapReduce

Hadoop MapReduce has become a de facto programming paradigm for parallel programming of massive datasets on large-scale commodity clusters. It has gained an unbeatable popularity among different computing communities and enjoys a premier position amongst the existing distributed frameworks. However, like any other architecture, MapReduce has its downside too. There exists some serious limitations to the programming model, and the class of algorithms and real-world problems to which the solutions provided by using MapReduce perform very poorly and offer meager performance.

MapReduce has been designed to offer highly scalable solutions for very large datasets, usually varying in size of terabytes to petabytes. Batch processing lies at the core of the design considerations of MapReduce. Further, it assumes a shared-nothing architecture and the process workflow need to fit within the map-shuffle-sort-reduce sequence. While this works like a charm for many problems, like that of computing the frequency of occurrence of words in a large corpus of documents, there are still many problems to which the nature of the proposed solution does not align with the MapReduce model.

The very design of the paradigm, to express every computation as a sequence of map and reduce tasks, stands also as a critical restriction. Not all problems can be expressed in that format. Further, for very large datasets, the startup costs for map and reduce tasks remains negligible as compared to the size of the data. However, this turns of to be a serious limitation if several map reduce tasks have to be started for multiple small data sets. Some algorithms require more than one

map/reduce tasks. It is not possible to have several map or reduce tasks within one MapReduce job. To be able to solve this problem, the programmer would have to write a chain of map reduce jobs which increase the startup costs, redundant storage caused by multiple disk writes at the end of each MapReduce job, and access time. Map and reduce tasks run independently on several nodes of clusters, and there is an inability to preserve the state among multiple map reduce phases and jobs and provide global synchronization. Moving large intermediate datasets across the nodes of cluster is very expensive and replication proves costly with increase in size of the dataset. Further, it creates heavy network traffic and causes bandwidth bottlenecks. Hadoop MapReduce also fails to support interactive data mining which require running multiple ad hoc queries on same subset of data and handling real-time streaming data which need to maintain and share state across multiple phases. Insufficient support for debugging individual map/reduce tasks stands as another pitfall of MapReduce.

A major class of algorithms where MapReduce failed to offer good performance is iterative and incremental algorithms. By nature, these algorithms require multiple passes over the same dataset. They form the core of many machine-learning problems. Consider the example of a PageRank [11, 12] algorithm, which is one of the most widely used web-scale graph algorithm. It requires the computation of a rank for every vertex of the graph which denotes a website which has to be done iteratively. Another example is an Expectation maximization algorithm [13] where maximum likelihood of parameters in statistical models has to be determined iteratively. The famous K-Means clustering algorithm described earlier, involves running

multiple iterations over the data points to determine the respective clusters. These would require running a MapReduce job for each iteration that makes it cumbersome.

1.3 Goals of this Thesis

Hadoop MapReduce has gained immense popularity in the world of distributed computing for being a highly scalable and efficient paradigm for handling massive datasets. While it is true, that Hadoop MapReduce offers peak performance for batch operations on large datasets, it comes with a heavy downside and some serious limitations too. Several specialized systems exist, such as: Pregel [14], which offers a bulk synchronous parallel model for iterative graph algorithms, Impala [15] that offers support to SQL on Hadoop and Storm [16], which facilitates distributed real-time computations. It is difficult to integrate multiple operations on different specialized systems, ex: if an analytics company is interested in storing the dataset on a SQL style database, apply machine-learning algorithms, and finally perform parallel aggregations in the last phase, it would be difficult to use different systems for each stage and use the output of one analyses as an input to the other. Integrating the operations and workflows on different models requires significant programming effort and also an overhead to switch between multiple frameworks. A general-purpose and advanced computing framework is needed which offers the power of MapReduce, overcomes its drawbacks, and also supports multiple models built on it which offer the functionality of specialized systems. Apache Spark [17] is a general cluster-computing framework that solves these problems.

The goal of this thesis is to evaluate the performance offered by Apache Spark

over Hadoop MapReduce by performing analyses on a large dataset. The evaluation is done in three phases. The Java MapReduce API is evaluated over Scala-Spark API by computing rolling average of pollutant concentration. In the second phase, usability of HBase as the data storage with MapReduce and Spark is evaluated. Spark provides a scalable Machine Learning Library called MLib [18, 19]. In the third phase, the performance offered by this is evaluated against Apache Mahout [20]. A 10GB air-quality dataset containing all measurements of pollutants from various sensors in 2011 all over Texas is used for this purpose.

1.4 Organization of the Document

The rest of the thesis is organized as follows. Chapter 2 outlines the details of Apache Spark, a general-purpose cluster-computing framework. In chapter 3, we present a comparison of the Spark and MapReduce programming models and evaluate their performance. In chapter 4 we evaluate the performance, clustering quality and usability of K-Means Clustering algorithm implementation provided by Spark MLib library against that of Mahout. In chapter 5 we examine the usability of HBase as a data storage from Spark as compared to its ease of integration with MapReduce. In chapter 6 we present the conclusions of the work.

Chapter 2

Background

Performing analyses on large datasets is common in many companies today. The focus is on improving the infrastructure on par with the growing needs of advance frameworks to decipher high business value from the data available. Hadoop and its components have proved to be invaluable for large-scale data intensive analysis on commodity clusters. They offer high scalability and fault tolerance. It has gained a de-facto status for being the framework that best facilitates single-pass batch processing of large-scale data, stored on-disk. However, its design does not support interactive data exploration and processing of multi-pass algorithms efficiently. Researchers realized the need for an advance framework that preserves the merits offered by Hadoop and also overcomes its limitations.

2.1 Overview of Spark

Apache Spark [17, 21] is a general-purpose cluster-computing framework, which comes with a power to run applications 100x faster than Hadoop in memory or 10x faster on disk [21]. It can be considered as an extension to MapReduce. It supports a wide range of distributed computations and facilitates re-use of working data sets across multiple parallel operations. The problem of not being able to share data across multiple map and reduce steps posed by Hadoop MapReduce is resolved by Sparks potential to keep its data in memory. It offers sub-second latency and strongly supports interactive ad hoc querying on large datasets. Its design helps to solve multi-pass iterative algorithms efficiently. Not only for batch processing, Spark also offers strong support to a variety of workloads, graph processing, and handling real-time streaming data within the same runtime. It facilitates applications to scale up and down elastically on clusters with ease and responsive resource sharing; work placement is done automatically based on data locality.

Spark provides to its users, a great ease of programming with its language integrated programming interface and rich Application Programming Interface (API) in Scala, Java, and Python. Its integration with the Scala/Python interactive shell facilitates easier development and debugging. Spark is originally implemented in Scala. Scala is hybrid programming language on the Java Virtual Machine(JVM). It is a statically typed programming language and supports type inference. The scala compiler performs type checking at compile time. It verifies and enforces constraints of types and the programmer need not explicitly mention the type of the variable, it is automatically inferred. Since it is built on the JVM, it interoperates and can

be easily integrated with other languages on the JVM. It provides its users with high expressiveness offered by its functional style and an ease of solving real world problems by decomposing into Object Oriented Programming model. For every line of code typed by the user into the interactive shell, the interpreter compiles a class. It is then loaded into the JVM and a function is invoked on it.

Spark provides unified programming abstractions called Resilient Distributed Datasets(RDD) [22]. They extend the data flow programming model introduced by MapReduce, Dryad [23, 24]. High-level operations are used to perform computations without worrying about internals such as work distribution and fault tolerance. RDDs are the primary storage primitives that facilitate the user to store the data in-memory and across multiple compute nodes. They give control of data sharing to the user and make efficient data sharing possible across multiple stages of parallel computations on a distributed dataset. RDDs are immutable parallel data structures. This simplifies consistency and supports straggler mitigation possible by running back up tasks. Stragglers are the slow nodes that are common in large clusters. They affect the performance and need to be handled carefully. Read-only copies of RDDs are shipped across worker nodes and will be acted upon by the operations in parallel. Most of the parallel operations performed in real-time are naturally coarse grained. Each operation is applied to multiple data elements.

Datasets are initially stored in storage system such as HDFS. On these data, operations are carried out which generate RDDs. These operations are referred to as transformations. Spark supports a wide range of transformations such as map, filter, etc. The output of these transformations is a new RDD, which will be used

by subsequent transformations. Spark also provides operations called actions, which returns the result to the program instead of an RDD. RDDs are ephermal and are computed every time they are used in an action.

Fault tolerance is crucial to any large-scale compute intensive, distributed framework. Spark provides automatic recovery from sudden failures through its RDDs, which are its distributed memory abstractions. It maintains a track of the series of transformations applied in the form of lineage graph that it builds as the application progresses. In case of any failures, it applies these transformations on the base data to reconstruct any lost partitions. A program cannot reference an RDD that it cannot reconstruct after a failure. This property proves to be critical for fault tolerance. The major difference in the Spark model is that the lineage graph is used instead of the actual data itself to efficiently achieve fault tolerance. Memory remains a prime concern with the constantly increasing volumes of data. Not having to replicate the data across disks saves significant memory and storage spaces. Network and storage I/O account for a major fraction of the execution time. RDDs offer great control of these, hence attributing to better performance.

2.2 Operations on RDDs

There are two types of operations on RDDs: Transformations and Actions. Transformations return pointers to new RDDs while Actions return values or results to the driver program. Multiple transformations and actions can be chained together to perform complex analyses on datasets. RDDs can be created from Hadoop InputFormats such as HDFS files or by transforming other RDDs. There are two other

essential aspects of RDDs the programmer can have a control of, Persistence and Partitioning. The RDDs which are likely to be re-used can be indicated using the `persist()` operation. Further, the programmer has options on chosing a storage strategy for them such as preserving them `MEMORY_ONLY, MEMORY_AND_DISK` etc. The programmer can also mention if the RDDs can be partitioned across machines based on a key in each record. RDDs are computed lazily from the lineage graph when they are first used in an action.

Programmers write a driver program, which connects to a cluster of workers. One or more RDDs are defined in the driver and actions are invoked on them. Spark code on the driver tracks the RDD lineage. Workers are the long-lived processes that can store RDD partitions in memory across operations.

2.3 An Application using Spark for Mining Flight Data

Consider an example of mining flight data. A flight dataset with all the flights that occurred in 2008 in the US is used for the purpose of illustrating simple Spark operations. The input file is a comma-separated list of data. For the task in hand, the following fields are required: Month represented by Integers whose value ranges from 1-12, DepDelay which is the duration by which the flight is delayed, given in minutes and Origin which is the IATA airport code.

```
/** Create a Spark configuration object with
    corresponding properties set */
```

```
val conf = new SparkConf().setAppName("Flight Data Analyser")
/** Create a Spark Context Object */
val sc = new SparkContext(conf)
```

The SparkContext needs to be initialized as part of the program and the SparkConf object that contains information about the application needs to be passed as argument to the SparkContext constructor.

```
val inputFile = "/Priya/Asst1Input/Inputfile.csv"
```

The above line stores the input file path which is a path of HDFS on the Shark cluster, whose details are described in section 3.3.1

```
val inputRDD = sc.textFile(inputFile)
```

The input file path is passed to the `textFile()` method which is invoked using the Spark Context object. This creates an RDD of the input file. RDDs are statically typed java objects parameterized by an element type. Various parallel operations are invoked on these objects to return new RDDs or actions on them. Spark uses the direct Scala compiler. Scala's type inference makes it easier for the programmer, to concentrate on the other programming tasks rather than worrying about the RDD types.

```
val flightDetails = inputRDD.flatMap(line => line.split(" "))
  .collect
```

On the RDD of the input file, a `flatMap()` method is invoked which maps each entry of the input RDD to 0 or more output items creating a new RDD. Arguments

to this are scala function literals (closures) and can use any language features or scala/java library features. After this transformation, a `collect()` action is invoked which returns an array of the elements at the driver program .Until the collect is invoked, all the transformations remain lazy, and no work has been performed on the cluster.

```
val flightData = sc.parallelize(flightDetails)
```

A parallelized RDD is created using the SparkContext's `parallelize()` method on the RDD computed previously. The elements of the collection are copied to form a distributed dataset that can be operated in parallel.

```
val flightParameters = flightData.map( line => line.split(","))

val delayPerFlightPerMonth = flightParameters.map( retorigin =>
           ((retorigin(16),retorigin(1)),
 if ((retorigin(15).equalsIgnoreCase("NA"))
 || (retorigin(15).toInt < 0 ))  0.toInt
  else retorigin(15).toInt))
```

In the above line, `map()` operation is performed on the `mapInputLines` RDD . The delay value is retrieved from the input file line by line and the key is stored as a tuple of origin airport and the month and the delay value is the value for the corresponding key in the new RDD.

```
val averageDelay = delayPerFlightPerMonth.combineByKey(
          (v) => (if(v > 0) 1 else 0, 1),
```

```
    (acc: (Int, Int), v) => ( acc._1 + 1, acc._2 + 1),
    (acc1: (Int, Int), acc2: (Int, Int))
  => (acc1._1 + acc2._1, acc1._2 + acc2._2)
    )
```

Using the `combineByKey()` operation, each value corresponding to a key is aggregated.

```
val percentageDelayPerFlightMonth = averageDelay.map{
 case (key, value) =>
 (key, ((value._1/value._2.toFloat)*100))}
```

Finally, the percentage of delay is computed for each key using the above `map()` operation.

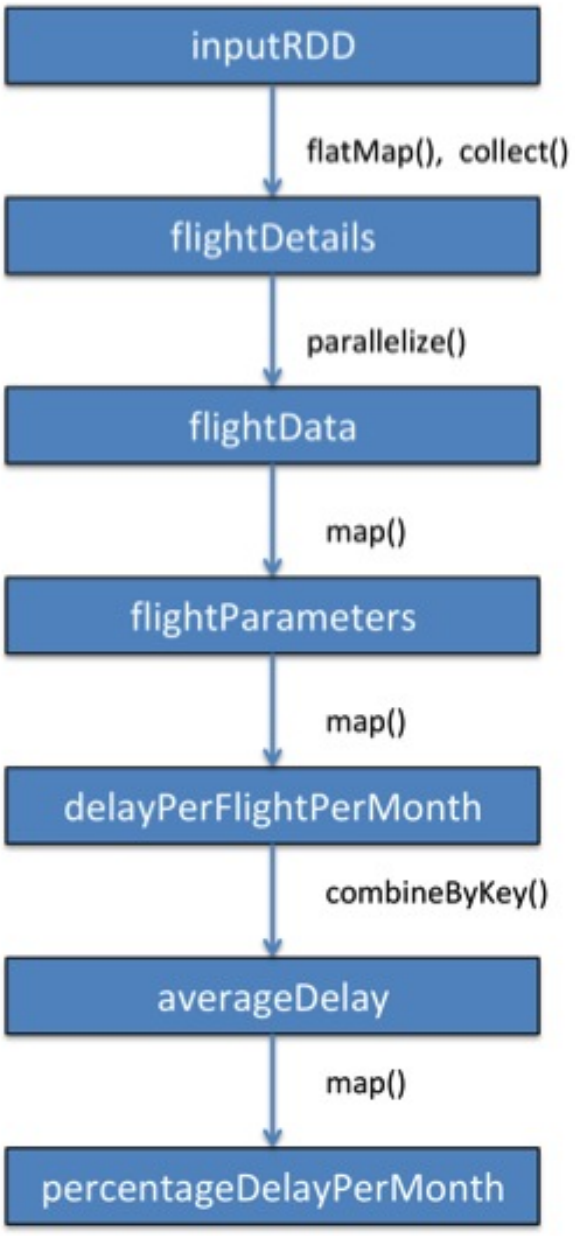

Figure 2.1: Lineage graph for the flight data mining program. RDDs are represented by boxes and transformations are represented by arrows.

Chapter 3

A Comparison of Spark and MapReduce

In the earlier chapters, an overview of Hadoop MapReduce and Spark programming models has been presented. To evaluate the performance of these two programming models, multiple analyses have been performed. A dataset containing all measurements of pollutants from various sensors in 2011 all over Texas is used for our data analysis. The input file is a comma-separated list of data. Each line consists of the following fields: year, month, day, hour, min, region, parameter id, parameter name, site, cams, value, and flag. The size of the input file is 10 GB and it is stored in HDFS.

3.1 Eight-hour Rolling Average

The first problem we tried to solve is to compute the eight-hour rolling average of O3 concentration. For every site in Houston, we calculate the hourly average first and combine them next to calculate eight-hour averages.

3.1.1 Using Hadoop MapReduce

Using Hadoop MapReduce the eight-hour rolling average has been computed using two MapReduce jobs. The first MapReduce job computes the pollutant concentration average for every hour. The mapper receives a text file as an input. It emits `(key,value)` pairs which are used by the reducer to perform the required aggregation. In this case, the key is composed as a combination of siteId, year, day of the year, and the hour. Only those data points whose site Ids correspond to Houston, parameter name is listed as O3, whose flag is not set and the pollutant value is not null are considered as valid data points for our measurement. Their corresponding pollutant concentration value is emitted by the Mapper. The following code sample indicates the core functionality of map() function of Mapper:

```
//check if the site id of the input data point belongs to houston
if(HoustonSiteIds.contains(psite))
{
    // check if the input data point belongs to O3
    if( pparam_id.equals(O3_pid) && pparam_name.equals(O3_pname))
    {
        /*check if the flag of the input data point is empty
```

```
            and the pollutant concentration value is not null*/
        if(!pflag.matches("[A-Za-z]+") && !pvalue.equals("NULL"))
        {
            mapOutputValue.set(Double.parseDouble(pvalue));
            context.write(mapInputKey, mapOutputValue);
        }
    }
}
```

Every instance of the Reducer gets all the values associated with a key. For each key, the sum of values and the frequency of values are computed. If the frequency count for a given hour is above a certain threshold (ex: greater than five in our case), the corresponding hourly average is computed. If the frequency count is less, a placeholder value (ex: `"-1"` in our case) is emitted by the reducer for the corresponding hour that indicates that it is insignificant. The following code sample indicates the core functionality of reduce() function of Reducer :

```
for (DoubleWritable val : values)
{
    String p_value  = val.toString();
    double p_value_D  = Double.parseDouble(p_value);
    total += p_value_D;
    count++;
}
/* compute the hourly average if count is greater than five
```

```
    if the count is less than five, emit "-1" indicating that
    the hourly average is insignificant */
if(count > 5){
    average = total/count;
    hourlyAvg.set(String.valueOf(average));
    context.write(key,hourlyAvg);
}
else {
    String invalid = "-1";
    hourlyAvg.set(invalid);
    context.write(key,hourlyAvg);
}
```

The next MapReduce job computes the eight-hour rolling average of the O3 concentration. The mapper receives as its input, the hourly averages computed in the earlier MapReduce job. The mapper emits eight keys that indicate the eight consecutive hours starting from the hour indicated in the input key and the input average value corresponding to the base hour. Important care had to be taken to ensure the hours emitted by the mapper match the clock (reflecting roll over after 24).

Every instance of reducer, receives all the average values corresponding to an hour. For every hour, the sum of the values and a frequency count is computed as in case of earlier MapReduce job. If the number of valid entries determined by the frequency count for a given hour is above a certain threshold (ex: greater than six), the corresponding eight-hour rolling average is computed. If the frequency count is

less, a placeholder value (ex: `"NA"`) is emitted by the reducer for the corresponding hour that indicates that it is insignificant.

The input and output files are provided as command line arguments to the driver method of the MapReduce program. A configuration Object is used to initialize a job instance. To the Class which comprises of the Jar, the Mapper, and Reducer classes, the input output formats of the input output files, the classes of Output Key and Value are explicitly passed through the job instance. The method `waitForCompletion()` is called on every job instance, to ensure that the first job completes its execution before the second job begins to execute. This is very important, as we are pipelining the output of first job as input to the second, and the second job will throw an error if it is unable to receive the complete output of the first job.

3.1.2 Using Apache Spark

Using Apache Spark, the eight-hour rolling average is computed in a single program. The programmer is not limited to decompose the problem strictly into a sequence of map and reduce steps. We present two implementations to computing the eight-hour rolling average: using `combineByKey()` and `reduceByKey()`. There is no significant performance difference seen in our context, since both the transformations are designed to group in order to perform aggregation over each key. We illustrate the choice and flexibility offered by the Spark to the programmer to chosoe either of them.

The first implementation uses `combineByKey()` transformation to perform aggregation. In the Spark Driver program, we start by creating a Spark configuration object to which we pass the required application parameters. An essential configuration parameter which is set in the current context is `spark.akka.frameSize()`. It indicates the maximum message size to allow in control plane communication in MB. The default size is ten. To be able to handle the large data size, we set it to twenty. A SparkContext object is created which is used for the subsequent parallel computations. The input and output files are stored on the HDFS. The air-quality file is provided as an input. A significant difference is that the hourly average values can be directly passed to the subsequent computations without having to write to the disk.

The input file is stored as an RDD using the Spark's `textFile()` method. The data points with no flags set, whose site Ids correspond to Houston, parameter name is listed as O3 and the pollutant value is not null are relevant for our analyses. A sequence of `map()` and `filter()` transformations are applied on the input RDD to extract the valid data points for our measurement into a new RDD. This new RDD is cached using the `persist()` method. We apply the `map()` operation on this RDD, which results in an RDD of `(key,value)` tuples. The key is composed as a combination of siteId, year, day of the year and the hour and value is the pollutant concentration value for the corresponding hour. Using the `combineByKey()` operation, the sum of the pollutant concentration values and their frequency for each key is computed. The hourly average is then computed by applying a `map()` transformation on the RDD obtained as a result of the earlier aggregation. As in case of

Hadoop MapReduce implementation described in section 3.1.1, before computing the average, the frequency count is ensured to meet the minimum threshold requirements. In the code sample given below, we illustrate the map() transformation and aggregation using combineByKey() transformation.

```
/**  Apply map on the data points */
val mapOutputFinal = mapInput.map(param =>
        ((param(8), param(0),
        getKeyPart1(param(0),param(1),param(2)),
        param(3)), param(10).toDouble))

/** Perform reduce on the data  points */
val reduceOutput = mapOutputFinal.combineByKey(
        (v) => (v , 1),
        (acc: (Double, Int), v) =>
            ( acc._1 + v, acc._2 + 1),
        (acc1: (Double, Int), acc2: (Double, Int)) =>
            (acc1._1 + acc2._1, acc1._2 + acc2._2)
)

/** If there are more than 5 values, emit the corresponding value
 else emit -1 */
val job1Output= reduceOutput.map{ case (key, value) =>
        if (value._2 > 5)
            (key,BigDecimal(value._1/value._2)
```

```
        .setScale(3, BigDecimal.RoundingMode.HALF_UP)
        .toDouble)
    else (key,(-1).toDouble)}
```

In the next phase, we compute the eight-hour rolling average. On the RDD consisting of the hourly averages, we apply a map() transformation which generates an RDD consisting of `(key,value)` tuples. The `map()` operation emits eight keys which indicate the eight consecutive hours starting from the hour indicated in the input key and the input average value corresponding to the base hour. The method `inputKeyIncrementer()` (called within the `map()` to generates the eight `(key,value)` tuples) ensures that the hours match the clock (reflecting roll over after 24). The sum and the frequency of hourly average values is computed using the combineByKey() transformation and later using a `map()` transformation, the eight-hour rolling average is computed.

Another implementation to compute the eight-hour rolling average uses the `mapValues()` transformation provided by the Spark API which maps every entry of the key to `(value,1)` tuple. Here 1 indicates the occurence of the value for the corresponding key. Using a `reduceByKey()` transformation, the sum of the values and the frequency count is computed. This RDD is used to compute the hourly average in the next `map()` transformation.

3.2 Eight-hour Rolling Average for Multiple Pollutant Concentrations

We now extend our analyses to SO2 and NO2 which are two other significant pollutants.

3.2.1 Using Hadoop MapReduce

In the first task, the eight-hour rolling average is calculated for O3 concentration. To be able to perform these analyses, a chain of six MapReduce jobs is used. Two MapReduce jobs are used for each pollutant concentration to compute the eight-hour rolling average. Each mapper uses the data points whose parameter id and the parameter name correspond to the specific pollutant concentrations for subsequent aggregations. The sequence of computations used is the same as described in section 3.1.1

3.2.2 Using Apache Spark

Using Spark-Scala API offers the programmer, high expressiveness, and conciseness. The task of computing eight-hour rolling average of multiple pollutant concentrations requires performing the same set of parallel operations on different subset of the data. On the existing dataset, the filter being imposed varies from one pollutant to the other. We can use the functional style of Scala to design the problem into making multiple calls to the core function that computes the eight-hour rolling average.

We use a single Spark program for multiple pollutant concentrations. The air-quality data set stored in the HDFS and a base output path are passed as inputs to the driver program. A function `generateOutputFilePaths()` is used which generates dynamically, an output file path that stores the eight-hour rolling average computed for each pollutant concentration. For example: if the base HDFS path provided is Priya/MultipollutantAvg/ , three HDFS paths are created Priya/MultipollutantAvg-44201-O3/, Priya/MultipollutantAvg-42602-NO2/, Priya/MultipollutantAvg-42401-SO2/.

The list of pollutant concentrations is stored as map with tuples of the parameter name and parameter id. For each pollutant concentration stored in this map, the method `rollingAvgCalculator()` is invoked. This method consists of all the parallel computations indicated in the earlier subsection 3.1.2 which are a series of transformations on the dataset performed to compute the eight-hour rolling average. The arguments passed to the function are the parameter name of the pollutant concentration and the corresponding output path. By calling the same function multiple times on different pollutant parameters, we are able to compute the eight-hour rolling average for each of the pollutant concentrations in a single program.

3.3 Performance Evaluation

In the following section we evaluate the performance offered by Spark over MapReduce on the basis of the results obtained by the analyses performed. First, a description of the resources used for executing the tasks has been provided, followed by a discussion on the results.

3.3.1 Technical Resources

The Shark cluster located at the University of Houston is used to perform analyses for the research work. It has 17 SUN X2100 nodes (shark01 - shark20). Each node has a 2.2 GHz dual core AMD Opteron processor. The main memory for these nodes is between 2-4 GB. There are three SUN X2200 nodes (shark25 - shark28) with two 2.2 GHz quad core AMD Opteron processor (8 cores total) and 16 GB main memory. It is connected by a 48 port Linksys GE switch. Further, the secondary storage comprises of 20 TB Sun StorageTek 6140 array and 4 TB distributed HDFS storage.

The cluster uses openSUSE operating system with version 13.1. The Hadoop version we are using is 2.6.0. The Java version: 1.7.0_51, Spark version: 1.2.0, Hbase version: 0.98.9.

3.3.2 Parameters Evaluated

To evaluate performance of Apache Spark over Hadoop MapReduce, we have considered two major parameters.

- Programmability
- Performance and Scalability

Programmability The ease of programming offered by any new technology stands as a crucial parameter that determines how gracefully it can be adapted and used. MapReduce and all other components on Hadoop use a Java API. Java being a very popular programming language makes it easy and simple for programmers to quickly

learn and use the API provided by MapReduce for their big data solutions. However, with increase in the complexity of the problem and growing need to expand and add new modules, programmers find it difficult to manage to get good solutions with the limitations posed by the model.

Spark is written in Scala. Scala, being a functional programming language, provides conciseness and expressiveness. The number of lines of code can be considered as a good measure to evaluate the programmability of the model.

It can be observed from figure 3.1 that the number of lines of Code in Spark is very less as compared to that of Hadoop MapReduce. This is mainly due to the conciseness offered by Scala [25] API. Particularly, in case of computing the eight-hour rolling average of the Multiple Pollutant Concentrations using Spark there is no overhead of replicating code blocks, no need for passing very long command line arguments. The program can be designed to dynamically adapt to the increase in the computational complexity.

For O3 pollutant concentration, the number of lines of code for computing eight-hour rolling average using Scala-Spark API is two and half times lesser than the number of lines of code to simulate the same parallel operations using MapReduce. For computing the eight-hour rolling average for multiple polluntants, this difference is large. The number of lines of code using Scala-Spark API is 3.16 times lesser than the number of lines of code used by MapReduce. The size of the program is important as the complexity of the problem grows, where code maintainability needs to be addressed with care. Spark version of the code supports efficient code reusability handle increasing number of pollutant concentrations.

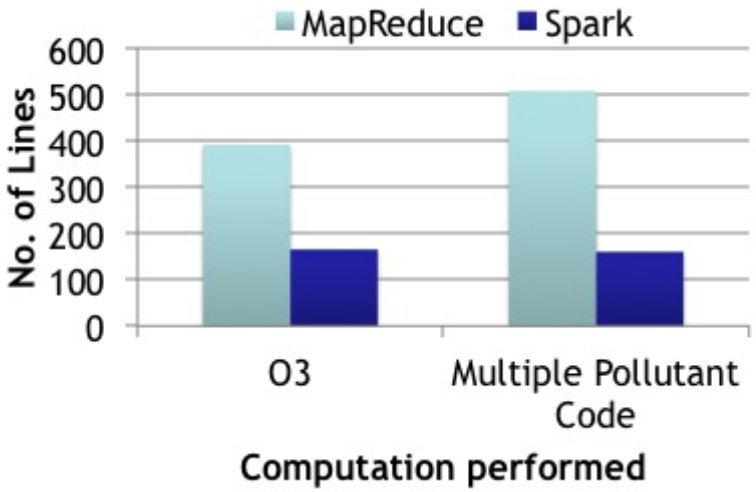

Figure 3.1: Number of lines of code

Spark also offers APIs both in Java and Python. The choice of using Scala for our research is simple and obvious, in terms that it is the language in which the Spark model has been natively written. It requires a significant effort and time to learn and adapt to Scala syntax initially. However, when one gets a good hold of the language, Scala being both functional and object orientated, one enjoys the benefits offered by its dual nature.

Also, code development is easier with Spark as it provides the interactive shell which facilitates testing the code as we develop it. With the Spark-Scala shell, we can see the output of each line of code that we write. This is a huge drawback in MapReduce, as there is no proper debugging facility available. It is difficult to debug a complex big data application running on thousands of clusters. It would be a tedious task to analyse the logs to understand which part of the program is causing the error. Spark with its interactive shell helps us conquer the error handling and

debuging at the initial stages of code development.

Performance and Scalability The execution time taken for the analyses performed is critical in big data applications. We measure the execution time to precisely evaluate the performance. Lesser execution times indicate that the program runs fast and there by giving good performance. It is also well noted that proper resource utilization is also crucial with large datasets. A good application should ideally offer high performance with minimal resource utilization.

To achieve good performance in the applications using Hadoop MapReduce, we vary the number of reducers based on the size of the dataset and the ratio of the number of values for each key. The number of reducers should be slightly less than the number of reduce slots on the cluster. By increasing the number of reducers, we allow all the values corresponding to a key finish in one wave, fully utilizing the cluster using the reduce phase.

It can be observed from figure 3.2 that MR jobs run faster with increase in the number of reducers from 1 to 5. However, there is no significant performance improvement seen by increasing the number of reducers from 5 to 10. This reason could be that more number of values are aggregated for some keys than the others and there cannot be any further distribution among the reduce instances resulting in approximately same computational time.

Every Spark application (i.e. for every SparkContext instance) has its own independent set of executor processes. Executor refers to a process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. The cluster manager that Spark runs on provides scheduling

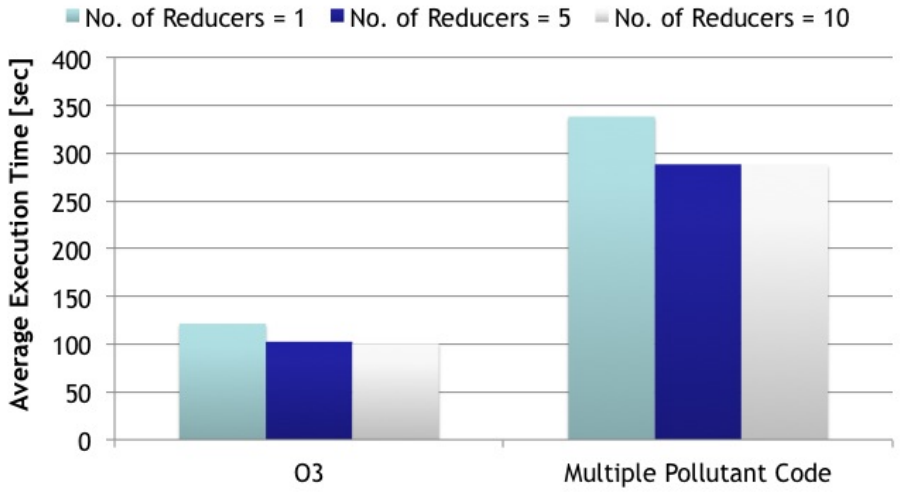

Figure 3.2: Average execution time for calculating eight-hour rolling average with HDFS as data source using Hadoop MapReduce

facilities across applications. Using the `--executor-cores` option, we can control the resources per executor. This is the major advantage for achieving the desired performance using Spark in the YARN cluster mode.

As mentioned above, to be able to achive desired performance, the user should explicitly tune the resources while executing the Spark application. This is critical as it gives us a control over the memory to be used, the number of processes to be started. We can try to get high performance with minimal resource utilisation. It can be observed from figure 3.3 and table 3.1 that when the `--num-executors` and `--executor-cores` are not set for the Spark application, it can be noticed that Spark application takes longer time to execute. However, in the latter runs, when we set the number of executors to 2 and increase the number of cores, we achive the desired performance which is higher than that obtained by using MapReduce.

The same flexibility to control the resources shows a significant performance difference in computing the eight-hour rolling average for the Multiple Pollutant concentrations. It can be observed from figure 3.4 that the analyses performed using Spark run faster than that of MapReduce.

In terms of scalability, Hadoop is used in real-time big data companies as it is known to scale well to clusters of thousand nodes. Spark is used actively in clusters of upto hundred nodes. However, active research is going on to making necessary improvements to the existing Spark model to match the scalability of Hadoop MapReduce.

Figure 3.3: Average execution time for calculating eight-hour rolling average for O3 concentration with HDFS as data source using Spark

To summarize the analysis, Spark outperforms MapReduce when the Spark application is set to utilize the resources correctly. Spark's ability to do this stands as a great advantage for performing analyses on massive datasets.

Table 3.1: Average execution time for calculating eight-hour rolling average for Multi Pollutant concentration with HDFS as data source using Spark

Number of Executors	**Number of Cores**			
	Not Set	2	4	8
4	6m54.97s	6m47.90s	5m23.82s	3m56.75s

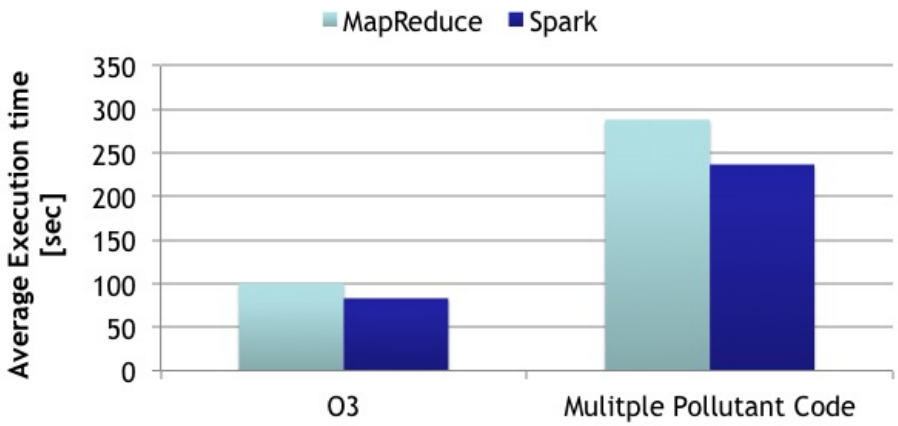

Figure 3.4: Comparision of average execution time for analyzing air-quality dataset with HDFS as data source using Spark and Hadoop MapReduce

Chapter 4

Machine Learning with Spark

4.1 Introduction

Machine Learning (ML) [26] helps to program systems to automatically learn and improve with experience. Applying Machine Learning techniques to big data forms the core of data analytics. Insights drawn by applying various ML algorithms play a vital role in the improving business. Some of the examples where ML algorithms see their importance in the recent time are: personalized treatment by analyzing patient's medical records, improved navigation systems which operate based on experience from sensors, recommender systems that drive sales of e-commerce industry, classifiers to keep spam out of email accounts, recommender systems to provide enriching user experience, friend suggestions in social network, etc. With the rise in the number of ML-driven applications, there is a growing demand for scalable solutions and large-scale distributed computing systems that support them.

4.1.1 Existing Tools

ML researchers prefer to code in statistical computing languages like Matlab [27] or R [28]. Weka [29] and Google Predict [30] offer ML tools with an intuitive interface. These languages and tools are sufficient for small-scale data exploration but the prototypes built using these technologies do not scale to larger datasets. Implementing these on massive datasets, to fit to the industry needs significant development effort, and becomes tougher as the problem complexity expands with time.

4.2 Comparision of Spark MLib to Mahout

Apache Mahout [20] is a machine learning library built on Hadoop. Though the recommendations, clustering and classification implementation provided by Mahout have gained popularity, developing new ML models on Mahout is not easy as it relies on HDFS to store and communicate intermediate state. This calls for the need of an advance framework, which supports wide range of Machine Learning algorithms on a large cluster of commodity machines.

MLib [18] is a scalable machine learning library built on Spark. It makes a perfect fit for the iterative ML algorihtms and with its capacity to store data in-memory. It offers high performance and ease of use, which favors both the ML researchers as well as end users. It can be used with the other components of Spark's eco-system without having to worry about portability issues. It presents to the programmers APIs in Scala, Java, and Python. It can be used with the Hadoop workflows. MLib facilitates API for computing basic statistics such as mean, median,

correlation, sampling and random data generation that are used as part of many ML algorithms. Some of the common machine learning algorithms supported by MLib are classification, regression, clustering, collaborative filtering, dimensionality reduction, feature extraction, and transformation. Active research and development is being carried out by the Spark developer community to implement more algorithms in the MLib library.

K-Means algorithm(described in 1.1) implementation provided by Mahout and MLib is evaluated. An air quality dataset that is stored in the HDFS is used for this purpose.

4.2.1 K-Means Clustering using Apache Mahout

An input file with site id's and feature vector, which contains the latitude and longitude and the daily maximum of O3 value for that site is used. The length of the feature vector is 367. The dataset has 106 data points. The expected input format by Mahout is a Sequence file. SequenceFile is a binary file with key/value pairs used vastly in MapReduce as input/output format. We evaluate the clustering quality obtained by K-Means algorithm by using compactness of the clusters as a measure. Running the K-Means algorithm using Mahout is achieved in two steps:

```
mahout kmeans -i /Priya/Clustering/dailymax.seq
              -o /Priya/Clustering/kmeans-out
              -c /Priya/Clustering/centers/centers.seq
              -x 10 -k 5 cl
```

The above command is used to run K-Means algorithm for 5 clusters with a maximum of 10 iterations on the sequence file. The clusterdump tool is used that makes it easier to read the output of any clustering algorithm in output. The command used is as follows:

```
mahout clusterdump
        -i /Priya/Clustering/kmeans-out/clusters-9-final
        --pointsDir /Priya/Clustering/kmeans-out/clusteredPoints
        -o /home/priya/outputk.txt
```

The output file consists of 5 clusters, centroids, and points corresponding to each cluster. This is passed as an input to a java program that computes the compactness of algorithm using the formula in 4.1.

To measure how well the algorithm performed, we compute the 'clustering quality'. This is obtained by analyzing the output of clustering. After clustering, we obtain the clusters, their centroids, and the points corresponding to each cluster. Using these details, we compute the clustering quality. For each cluster, we take the ratio of sum of square of distance between the data point that belong to the cluster and the cluster centroid and the sum of square of distance between the data points that do not belong to the cluster and the cluster centroid. Clustering quality is the sum of the ratio obtained for each cluster.

$$c = \sum_{C=0}^{NumClusters} \frac{\sum_{i \in C} (x_i - \mu_c)^2}{\sum_{j \ni C} (x_j - \mu_c)^2} \tag{4.1}$$

4.2.2 K-Means Clustering using Apache Spark MLib

To perform K-Means Clustering using Apache Spark, a text file containing site id's and feature vector which contains the latitude and longitude and the daily maximum of O3 value for that site is passed as an input to the Spark driver program. MLib uses a parallelized variant of the `k-means++` method called `kmeans||`. It accepts the following parameters:

- Number of desired clusters.
- Maximum number of iterations to run.
- Number of times to run the k-means algorithm.

Optionally we can provide:

- `initializationMode` : specifies either random initialization or initialization via `k-means||` The default is `k-means||`.
- `initializationSteps` : determines the number of steps in the `k-means||` algorithm.
- `epsilon` : Distance threshold within which we consider k-means to have converged.

The input file is loaded into an RDD using the Spark's `textFile()` method.

```
// Load and parse the data
val data = sc.textFile(pollutantInputFile)
```

The input is parsed to a new RDD containing `Vector` using a map() transformation on the RDD containing the data points. This RDD `[Vector]` containing the training data is passed to the `train()` method of the `KMeans` object.

```
/** Perform clustering*/
val parsedData = data.map(s =>
        Vectors.dense(s.split(' ').map(_.toDouble))).cache()

// Cluster the data into five clusters using KMeans
val numClusters = 5
val numIterations = 10
val clusters = KMeans.train(parsedData,
                numClusters, numIterations)
```

Once the clustering is performed, to retrieve the cluster centers, we use the `makeRDD()` method, which gives the list of cluster centers. Each cluster center is then mapped to the cluster index using the `predict()` method. The RDD `clusterCentroidPoints` is saved as a text file on HDFS that consists of the cluster ids and cluster centers

```
/** Generate (clusterId, clusterCenter) */
val clusterCentersGenerated =
            sc.makeRDD(clusters.clusterCenters, numClusters)
val reqClusterCenters = clusterCentersGenerated.
```

```
                map{ case(x) => (clusters.predict(x),x)}
val clusterCentroidPoints = reqClusterCenters.
                map{ case(x) => (x._1.toInt, x._2)}
```

Further, to compute the compactness, it is important to know to which cluster every point in the data belongs. To achieve this, we perform a similar computation as mentioned in the previous step. Each data point is mapped to the cluster index using the `predict()`method.The RDD `pointAndClusterId` is saved as a text file on HDFS which consists of the cluster ids and cluster centers

```
/** Generate (clusterId, dataPoint) */
val points = parsedData.map{case(x) =>
                    (clusters.predict(x),x)}
val pointAndClusterId = points.map{ (x) =>
                    (x._1.toInt, x._2)}
```

The output files are passed as input to a java program which computes the compactness of the algorithm using the formula mentioned in 4.1.

4.3 Performance Evaluation

The resources described in 3.3.1 is used to perform analyses for the research work.

From table 4.1 that the compactness value obtained for Spark MLib is higher than that of Mahout

It can be observed from table 4.2 that MLib runs faster than Mahout. Mahout requires the input to be converted to a Sequence File. This conversion is not trivial

Table 4.1: Comparision of Clustering Quality obtained by K-Means algorithm implementation by Mahout and MLib

	Mahout	MLib
Trial1	0.8051	1.3201
Trial2	0.7331	1.4504
Trial3	0.8131	1.3388

Table 4.2: Average execution time for performing K-Means clustering using Mahout and MLib

	Mahout	MLib
AverageExecutionTime(secs)	205.384	46.83

as it requires a lot of preprocessing and stands as a serious limitation. On the other hand, MLib accepts the input in a simple text format, and uses it own Vector class to convert the input to an RDD `[Vector]`. This offers programmers with great flexibility. MLib is also available through the Spark's interactive shell which makes development and debugging easier.

Further, the conciseness obtained by using the Scala API can be seen by very few lines of code written to use the K-Means clustering provided by Spark MLib.

Table 4.3: Number of lines of code to use the K-Means implementation provided by MLib

	MLib
#LinesOfCode	67

Chapter 5

Integration With HBase

In the age of massive scale analytical processing, there is an increasing demand for efficient storage systems that can facilitate large queries and scans on wide range of records and in some cases very big tables. Traditionally, Relational Database Management Systems (RDBMS') [31] have been used by large-scale enterprises since the 1970s. However, there are multiple problems with scaling up traditional database technologies. For example: it is difficult to scale up vertically by adding more database servers and to handle the problem of waits and deadlocks that increase with the number of transactions and high concurrency. De-normalizing the databases does not help either, as it results in costly join operations at later stages. Sharding is a technique used to spread data across multiple storage files and servers. This too is impractical, as it is very costly to re-shard the data and it incurs huge I/O overload due to its massive copy operations. Due to these limitations, RDBMS' make a poor fit to the big data needs.

To address these limitations, a new class of databases emerged called NoSQL [32].

Multiple variants of NoSQL databases are seen in practice. They are designed to offer horizontal scalability without a need to repartition as data grows and also supporting high fault tolerance and data availability. Around the same time, Bigtable [33], a distributed storage system for managing structured data presented by Google, addresses the limitations of RDBMS mentioned before. A big table is a sparse, distributed, persistent, multidimensional-sorted map. It is indexed by row key, column key, and a timestamp. Each value in the map is an uninterpreted array of bytes. The data model offers dynamic control over data layout and format to the clients. With its wide applicability and high availability, it proved be a scalable solution with its implementation seen in various Google projects such as Google Finance, Google Earth, etc. In 2007, HBase [34] project started as an open source extension of Bigtable. In 2010, it became an Apache top-level project and gradually gained a significant place in the NoSQL community.

HBase is a distributed, persistent, strictly consistent storage system. It scales efficiently to large-scale commodity clusters. It adopts a column-oriented architecture with huge, wide sparse tables. It supports compression. It is designed to offer good read performance and near optimal writes. The basic unit of HBase is a column. One or more columns merge to form rows. Row keys uniquely address rows. Row keys are arbitrary array of bytes. The read and write operations are atomic for each row key. This is to prevent problems caused during concurrent updates to the same row. Multiple rows form a table. Further, each column has multiple versions. Distinct values of each version are stored in a separate cell. Their row key sorts rows lexicographically. Columns are grouped into 'Column Families' that are the basic

unit of access control. All columns of a column family are stored together in same low level storage called `HFile`. Column families are defined while creating HBase table. Each column is referred as `family:qualifier`, where qualifier is any arbitrary array of bytes. Every column value or cell is timestamped. Multiple versions are sorted in decreasing timestamp order. Hence, whenever there is a query the most recent version is read first. To facilitate efficient searches, contiguous ranges of rows are stored as Regions. Regions serve as unit of load balancing and scalability. Initially, there is one region per table, which is dynamically split as the system gets large. By this, HBase supports Auto Sharding. A region server processes each region. In case of failure, regions can be quickly moved between region servers.

HBase tightly integrates with MapReduce. It provides wrappers that convert tables into input sources and output targets for MapReduce jobs. HBase also provides an extensible jruby-based (JIRB) shell as a feature to execute some commands (each command represents one functionality) [35]. It facilitates interactive development. We examine its integration with MapReduce in section 5.1. As mentioned in the earlier chapters, there is a slow shift seen in the big data industry from MapReduce to Spark.

Spark supports accepting input from data sources supported by Hadoop, local file system, HDFS, Cassandra, HBase, and Amazon S3. It supports data formats such as text files, Sequence Files, and any other Hadoop Input Format. Because HBase is a vital component of Hadoop eco-system, we examine Spark's ability to actively integrate and interoperate with the HBase in section 5.2.

5.1 Using HBase with Hadoop MapReduce

To examine MapReduce integration with HBase we calculate the eight-hour rolling average as described in section 3.1.1. However, the main difference is that, we use HBase as the data source. To achieve this, we write two MapReduce jobs that are chained together. The first MapReduce job computes the hourly average. It accepts as an input, a HBase table with the data stored in it.

The air-quality data set containing all measurements of pollutants from various sensors in 2011 all over Texas is used. A MapReduce job is used to perform bulk import of the data into a HBase table using completebulkimport tool. A table `2011-full` is created in the HBase. The schema has been defined manually using the hbase shell commands. The fields of the dataset are grouped into column families as follows:

- `date: year, month, day, hour, min`
- `location: region and site`
- `data: cams, param_id, param_name, value, flag`

```
//Create a scan instance
Scan scan = new Scan();

//choose families and columns for scanning
scan.addFamily(Bytes.toBytes("date"));
scan.addFamily(Bytes.toBytes("location"));
```

```
scan.addColumn(Bytes.toBytes("data"),
Bytes.toBytes("param_id"));
scan.addColumn(Bytes.toBytes("data"),
Bytes.toBytes("value"));
scan.addColumn(Bytes.toBytes("data"),
Bytes.toBytes("flag"));
```

The code sample given above indicates the scan operation performed on HBase table. Only those data points whose site Ids correspond to Houston, parameter name is listed as O3, whose flag is not set and the pollutant value is not null are considered as valid data points for our measurement. We check for the valid data points by using scan and filter operations provided by the Client API of HBase. Using Scan class we narrow down the dataset being passed to the Mapper. We create a scan instance, which will scan through the entire table. Once Scan is constructed we further narrow down the search by mentioning the family and corresponding column to be scanned and set a filter. Since we need more than one filter, we create a list of filters and set the filter to the list. The following code sample indicates the filter operation.

```
//Filter list contains all the sites in Houston and O3 for each site
FilterList list = new FilterList(FilterList.Operator.MUST_PASS_ALL);

SingleColumnValueFilter O3ParamIdFilter = new SingleColumnValueFilter(
                            Bytes.toBytes("data"),
                            Bytes.toBytes("param_id"),
```

```
                                CompareOp.EQUAL,
                                Bytes.toBytes("44201"));
list.addFilter(O3ParamIdFilter);

SingleColumnValueFilter flagFilter = new SingleColumnValueFilter(
                                Bytes.toBytes("data"),
                                Bytes.toBytes("flag"),
                                CompareOp.NOT_EQUAL,
                                Bytes.toBytes("\"\""));
list.addFilter(flagFilter);

SingleColumnValueFilter valueFilter = new SingleColumnValueFilter(
                                Bytes.toBytes("data"),
                                Bytes.toBytes("value"),
                                CompareOp.NOT_EQUAL,
                                Bytes.toBytes("NULL"));
list.addFilter(valueFilter);

scan.setFilter(list);
```

Hence, only the valid data points are sent as an input to the Mapper of the first MapReduce job. The Mapper receives as an input the data from the table as arrays of bytes. These are then converted to strings to perform the subsequent operations as described in section 3.1.1 to initially compute the hourly average and compute the eight-hour rolling Average in the next MapReduce job.

5.1.1 Performance Evaluation

The performance offered by using HBase as a source by computing eight-hour rolling average has been evaluated by measuring the execution time. Figure 5.1 captures the average execution time taken for the computation. We use the UNIX time command to measure the execution time. Each job has been run 3 times.

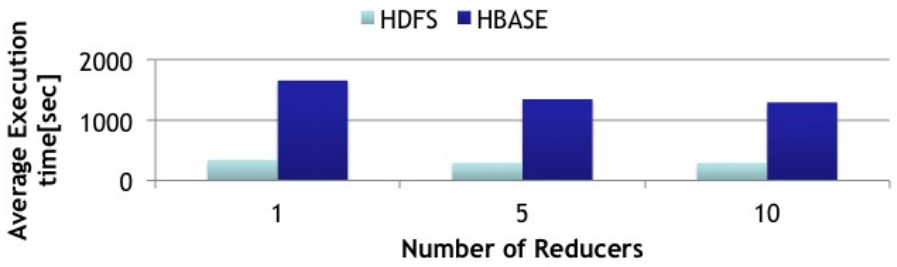

Figure 5.1: Execution time for calculating eight-hour rolling average with HBase as data source using Hadoop MapReduce

It can be observed that, though from the usability perspective using HBase as a data source is very good, it takes long time for computations as compared to that of using HDFS as a data source. With increase in number of reducers, there is an improvement in the performance. As mentioned earlier, HBase internally uses HFiles to store data on the HDFS. Slow performance offered by using HBase as compared to HDFS can be accounted to the extra overhead of the hbase operations, which inherently access HDFS too. However, HBase can be used as a NoSQL data store that offers high scalability for batch-oriented data processing with excellent read performance.

5.2 Using HBase with Apache Spark

Spark offers high performance and great ease of use when performing analyses on large datasets. While it is extremely flexible to use data stored in HDFS as input source to a Spark driver program, integrating with HBase has been a herculean task. Some of the challenges encountered in the process of using HBase from Spark are described below.

HBase runs on top of Hadoop in a fully distributed mode. When Spark runs on a cluster, the SparkContext needs to connect to cluster managers to allocate resources across applications. It was difficult to set the configuration parameters in the Spark driver program to be able to establish the connection and access table existing in HBase. Very few sources available on the Internet discuss how to access HBase from Spark, and very little description is available about the API and how to adapt it to a generic cluster set up. It took significant time and effort to get a working test case of Spark.

In the sample test case, we had to explicitly pass all the configuration details through the driver program to the Spark configuration object. This is not the right approach, as it gets difficult for the end user to mention the configuration details in every application. Further, it reduces portability of the application across different clusters. The Spark environment should be able to read the configuration automatically from the files. Though the current Spark version already supports this, there is no documentation available for common users that give instructions about how to achieve it.

The code snippet given bellow illustrates a sample test case that scans a HBase

table and prints them.

```
def scanTable(table: HTableInterface): Unit = {
    println("Scanning table")
    val scan = new Scan();
    val tableScanner = table.getScanner(scan);
    tableScanner.foreach { result =>
            result.raw.foreach { kv =>
                println("Row name \t "
                        +Bytes.toString(kv.getRow))
                println("Column family \t"
                        +Bytes.toString(kv.getFamily))
                println("Timestamp family \t"
                        +kv.getTimestamp)
                println("Value family : \t"
                        +Bytes.toString(kv.getValue))
                        }
                }
    println("Done")
}
```

A test case has been developed. A small table is created in HBase and a Spark application is written that establishes connection to the HBase table and retrieves the data from HBase and prints it. While the test case ran successfully, work is

still in progress to implement other HBase operations using Spark. To perform any computations, Spark requires the data to be transformed to RDD, which is its primary abstraction. To be able to perform the analyses on the air quality data set, the scan() and filter() functions of the HBase API have to be used from a Spark driver program, to transform the data read from the HBase table to a Spark RDD and then perform the required transformations. The documentation available does not provide sufficient details about how this can be done and how it can be extended to handle large dataset stored on a cluster of machines. It is reserved for the future work to understand and get a working implementation of a Spark application integrated with HBase.

Chapter 6

Conclusions and Outlook

Hadoop MapReduce is a distributed programming model, which supports parallel processing of large-scale data efficiently. It supports batch workloads and offers scalable solutions to a wide range of real world problems.

Though it has gained popularity, MapReduce has some serious limitations too. Some classes of real-world problems cannot be easily decomposed into map and reduce computations. Increase in start up costs of map and reduce tasks in case of multiple chained MapReduce jobs and need to move large dataset across the cluster proves expensive with increase in dataset size. The design of MapReduce does not inherently offer good support to iterative and incremental algorithms. It does not support interactive and random workloads. There is no support to share the intermediate state between map and reduce stages without writing to the secondary storage. With growing research, there is a need seen for an advance framework that preserves all the merits of MapReduce and also overcomes its limitations.

Apache Spark is a general-purpose computational engine which facilitates in-memory analytics on large-scale data stored in cluster of commodity machines. It supports wide range of workloads ranging from ad hoc queries to batch and iterative processing. It provides to its user, a great ease of use with its APIs in Scala, Java, and Python.

We have evaluated the performance offered by Apache Spark over Hadoop MapReduce by performing analyses on an air-quality dataset of 10 GB. In the first phase, the Java MapReduce API is evaluated over Scala-Spark API by computing rolling average of pollutant concentration. From the analyses performed, it is observed that Spark offers a performance of up to 20% more than Hadoop MapReduce. It offers to its programmers' conciseness and expressiveness mainly due to the Scala syntax.

Machine Learning is an emerging discipline, which helps in making predictions by learning from existing data. ML research is rapidly growing in the present day data analytics. Machine learning algorithms are actively used in e-commerce, social networking cites, fraud detection, etc. There is a growing demand seen for scalable solutions and large-scale computing systems which support ML. Spark provides a scalable Machine Learning Library called MLib [18, 19]. In the third phase, the performance offered by this is evaluated against Apache Mahout.

Spark is designed to actively interoperate with Hadoop eco-system. In the next phase, we evaluate the usability of HBase as the data source with Spark as compared to that of HBase integration with MapReduce. However, since it is in its inital stages and inadequate learning resources available, it has been difficult to integrate it with HBase. We wish to continue working on this in the future.

In the future, performing analyses on a larger dataset is of great interest. It would be interesting to explore the performance and cluster tuning for a data set of size up to 50 GB as an initial step to understand the real-time scenario. We will continue to work on integration with HBase. Further, we would love to explore the other components of the Spark eco-system such as Spark SQL and also the other algorithms provided by MLib.

Bibliography

[1] Elena Geanina ULARU, Florina Camelia PUICAN, Anca Apostu, and Manole VELICANU. Perspectives on big data and big data analytics. *Database Systems Journal*, 2012.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[3] Matei Zaharia. Introduction to MapReduce and Hadoop. http://www.cs.berkeley.edu/ demmel/cs267_Spr09/Lectures/Cloud_MapReduce_Zaharia.ppt.

[4] Jimmy Lin and Chris Dyer. *Data-intensive Text Processing with MapReduce* . Morgan and Claypool, 2010.

[5] Edgar Gabriel. COSC 6339 Big Data Analytics. http://www2.cs.uh.edu/g̃abriel/courses/cosc6339_S15.

[6] Lucene . https://lucene.apache.org/core/.

[7] Tom White. *Hadoop: The Definitive Guide* . O'ReillyMedia, Inc., second edition, October 2010.

[8] Nutch . http://nutch.apache.org/.

[9] Hadoop . http://hadoop.apache.org.

[10] HDFS Architecture Guide . https://hadoop.apache.org/docs/r1.2.1/hdfs_design.pdf.

[11] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, April 1998.

[12] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International Conference on World Wide Web 7*, WWW7, pages 107–117, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V.

[13] Expectation-maximization Algorithm . http://en.wikipedia.org/wiki/Expectation

[14] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[15] Cloudera Impala . http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html.

[16] APACHE STORM . https://storm.apache.org/.

[17] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[18] Spark MLib. https://spark.apache.org/mllib/.

[19] Nick Pentreath. *Machine Learning with Spark* . Packt Publishing, February 2015.

[20] Mahout. http://mahout.apache.org/.

[21] Spark Lightning-fast cluster computing. http://spark.apache.org.

[22] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[23] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[24] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, March 2007.

[25] Bill Venners Martin Odersky, Lex Spoon. *Programming in Scala* . Artima, Inc., second edition, 2010.

[26] Machine Learning . http://en.wikipedia.org/wiki/Machine_learning.

[27] MATLAB . http://www.mathworks.com/products/matlab/.

[28] The R Project for Statistical Computing . http://www.r-project.org/.

[29] WEKA . http://www.cs.waikato.ac.nz/ml/weka/.

[30] Google Cloud Platform: Prediction API . https://cloud.google.com/prediction/docs.

[31] RDBMS. http://en.wikipedia.org/wiki/Relational_database_management_system.

[32] NoSQL. http://en.wikipedia.org/wiki/NoSQL.

[33] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[34] Lars George. *HBase: The Definitive Guide* . O'Reilly Media, Inc., first edition, September 2011.

[35] Learn HBase . https://learnhbase.wordpress.com/2013/03/02/hbase-shell-commands/.