Politecnico di MILANO

Faculty of Computer Science & Engineering

Master Program in Computer Sciences & Engineering

DATA ANALYSIS AND RHADOOP: CASE STUDIES

Supervisor:

Prof. Marco Gribaudo
Professor
Dipartment of Computer Sciences & Engineering
Politecnico di Milano

Author:

Kh.Ehsanur Rahman
Matricola: 833342
Academic Session 2014-15

Acknowledgement

I would first like to thank my project advisor Professor Marco Gribaudo of the Department of Computer Science & Engineering of Politecnico di Milano. The door to Prof. Gribaudo's office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently steered me in the right direction whenever he thought I needed it.

I would also like to acknowledge all my Professors of the department of Computer Science & Engineering of Politecnico di Milano who have always encouraged me to work on the topic related to Data Engineering.

Finally, I must express my very profound gratitude to my parents, for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Thank you.

Kh.Ehsanur Rahman

Abstract

The era of "Big Data" is here. Rapid growth in big data and application of analytical algorithms has created massive opportunities for data scientists. From Facebook to small business organizations, everyone is relying on big data for their business forecast. With data firmly in hand and with the ability given by Big Data Technologies to effectively store and analyze this data, we can predict and work to optimize every aspect of our behavior. Amazon can know every book we have ever bought or viewed by analyzing big data gathered over the years. With the advent of many digital modalities all this data has grown to BIG data and is still on the rise. Ultimately Big Data technologies can exist to improve decision making and to provide greater insights faster when needed but with the downside of loss of data privacy.

This project has two phases. In initial phase, the main goal is to understand how RHADOOP works for data analysis. Hadoop was working fine with its own java based environment, but we needed more flexibility and data analysis capability. Arising from this constraint, the requirement of something new has emerged. Data analysts are using R for data analysis, and the use of R is increasing rapidly. Our intention here is clear, to utilize the power of Hadoop and R for Data analysis and combining the power of both technologies, the solution is RHadoop. It uses special packages like rmr2, rhdfs, plyrmr and rhbase for accessing HDFS files and mapreducing jobs. The second phase is the investigation of mapreduce job over RHADOOP. Mapreduce job contains mapper and reducer functions. In some case studies, we have used both functions to understand how they are compatible with each other. At the beginning, system assigns an id for overall mapreduce job, and then handles mapper and reducer respectively. Mapreduce job processing approach is using in most of the big data analysis, so it is highly important to deal with it.

Hadoop and R is a natural match and are quite complementary in terms of visualization and analysis of big data. This work mainly focuses on RHADOOP and its operational features for data analysis.

Table of Contents

Chapter 1: Introduction

The data explosion is upon us, with increasing amounts produced each day. The trend shows no sign of stopping, or even slowing down. In 2009, research firm IDC noted a 62 percent increase in worldwide data over the previous year, and predicted that digital information, which in 2009 had reached 0.8 zettabytes , could reach as much as 44 zettabytes by 2020. And, as if that wasn't enough, the Berkeley School of Management forecasts that more data will be created in the next three years than in the previous 40,000. In the last few years, digital technology, social networks, and forums have been found propagating much more than in the past decades. Moreover, their growth is expected to be much quicker paced in the coming days. As regards the most of private companies, their data volume is much smaller, but which is also experiencing incredible growth nowadays. These are just a few examples of the data deluge affecting almost everything.

The hot IT buzzword of 2012, big data has become viable as cost-effective approaches have emerged to tame the volume, velocity and variability of massive data. The *Harvard Business Review* (HBR) in "Customer Intelligence Tames the Big Data Challenge" explains,

"Big Data examines what people say about what they have done or will do. That's in addition to tracking what people are actually doing about everything from crime to weather to shopping to brands. It is only Big Data's capacity for dealing with vast quantities of real-time unstructured data that makes this possible."

In all areas of enterprise, teams and departments are looking for actionable data. There are endless benefits to managing Big Data instead of either ignoring it or allowing it to outpace any organization. A Big Data and analytics strategy benefits the organization in several ways:

Creating Smarter, Leaner Organizations:

A well thought out and executed Big Data and analytics strategy ultimately makes organizations smarter and more efficient. Today, Big Data is being leveraged in many industries from criminal justice to health care to real estate with powerful outcomes. The same common sense approach to Big Data should be employed by organizations desiring similar results.

For example, *HBR* reports that the New York City Police Department uses Big Data technology "to geolocate and analyze 'historical arrest patterns' while cross-tabbing them with sporting events, paydays, rainfall, traffic flows, and federal holidays." Essentially, the NYPD is utilizing data patterns, scientific analysis, and technological tools to do their job and to do it to the best of their ability. Using a Big Data and analytics strategy, the NYPD was able to identify crime "hot spots." From there, they deployed officers to locations where crimes were likely to occur before the crimes were actually committed.

Gone are the days of "guessing," even in the police force, arguably one of the most instinct- and experience-driven vocations. This does not mean that instinct, human emotion, and reason are gone. It does mean that data is creating leads and context in which the NYPD can hopefully operate at an optimal level. Big Data and analytics are helping the NYPD and other large police departments to anticipate and identify criminal activity before it occurs.

There are plenty of examples like these, in every industry, as leading organizations continue to practice what GE's CMO Beth Comstock recently called "machine whispering":

"The same logic is being applied to economic forecasting. For example, the number of Google queries about housing and real estate from one quarter to the next turns out to predict more accurately what's going to happen in the housing market than any team of expert real estate forecasters."

The question before us today is, how can Big Data and analytics be similarly leveraged by an organization to provide powerful results?

Equipping the Organization to Have Cross-Channel Conversations:

As most organizations will agree, it's simply not possible to carry out the conversations with the customers like earlier times. There's too much feedback and dialogue coming in from various sources, which require help to accomplish the desired means. It's simply not possible to manage the delivery of dynamic, targeted, consistent content, offers, and products, across digitally enabled customer touch points when marketing tasks are semi automated with a series of unintegrated software tools.

Best industry practices today suggest staying close to the customer "by investing in customer insight." Today's digital ecosystem demands strong market intelligence: Innovative teams will integrate emerging digital, social and mobile tools into more traditional 'voice of the customer' processes, and effectively build feedback loops into key business functions such as product development and sales.

Preparing Your Organization for the Inevitable Future

The inevitable future is the digitization of all customer-facing organizational systems from customer service to sales to marketing.

What is interesting and noteworthy is that, structural changes within organizations (related to Big Data) are necessary now as reversals are likely to come. Two interesting reversals can be taken into consideration here. The first reversal was in the newspaper industry that moved from booming to near obsolete with the advent of online publishing. This happened within a decade. The second reversal was in the recording/music industry that moved from booming CD sales to obsolete (CD sales) with the advent of digital music. This also happened within a decade. Both reversals were gradual until they were sudden.

These are both great examples of the gradual takeover that Big Data management tools are having within the marketing teams and departments of every organization today. From the smallest mom and pop shop to the largest, international organizations, organizations that resist the scientific and systematic approach to data analysis, online advertising, and more will become obsolete. Fortunately, we are still in the era of gradual shift.

The Big Data Reversal Is Coming: Are the Organizations Ready yet?

It's just a matter of time before the sudden reversal comes. All the signs point to its arrival. The question is, will the organizations have the proper Big Data and analytics strategy in place to survive the reversal?

Through the techniques of data analysis, it is possible to extract useful information from the customers. The target is the solution of a wide set of problems, such as interaction with customers, strategic decisions, and processes optimization. For instance, many chain shops track their customers' purchases using data extracted from their cards. In addition, the integration between data coming from companies and from the web can produce further information. In this way, some companies try to understand the opinions about their products by analyzing forum discussions. These examples are only a small part of the multitude of results that can be extracted from the huge mine of data.

Regarding current approaches, such as the use of databases, the starting point is often the selection of small datasets from the whole, including only information that is highly relevant to the problem, on the base of analysts' opinions. Since most of the tools do not allow a cheap treatment of wider sets of data, this is the most common way to proceed. However, this approach has some disadvantages since it is not always possible to determine with enough precision which information is useful and which can be excluded. Maybe it was not a considerable problem when the overall data were small, but the data spread has worsened the situation. Moreover, the introduction of less relevant data in analysis can lead to more precise results and to a bird's eye understanding of the problem, so the ideal tool exploits all the available information.

There are some tools that can be integrated in the current devices, in order to handle wider datasets, but these solutions are often afterthoughts and they imply some efficiency problems. In fact, the cost often becomes much higher. Furthermore, the maximum amount of data analyzable is not growing as fast as data.

The target is to extend the analysis to all available information, so it is necessary to use new techniques.

Big Data, Big Challenges:

When we think of Big Data, the three Vs come to mind – volume, velocity and variety. According to some companies, such as IBM, the new challenges are the Three V that stands for Volume, Velocity, and Variety. Big Data phrase refers to all data analysis that present any of these new necessities.

Just as the amount of data is increasing, the speed at which it transits enterprises and entire industries is faster than ever. The type of data we're talking about includes hundreds of millions of pages, emails and unstructured data, such as Word documents and PDFs, as well as a nearly infinite number of events and information from every type of enterprise data center— such as financial institutions, utility companies, telecom organizations, manufacturing facilities and more. Content can be generated by everything from common customer transactions, such as phone calls and credit card usage, to manufacturing facility transactions, like machine maintenance and operational status updates. All of this information needs to be analyzed, acted upon (even if that action is deletion), and possibly stored.

1.1 Volume

According to some IBM estimations, the amount of data produced every day is about 2.5 quintillion of bytes and the most of it (about 90%) belongs to the last two years. Furthermore, according to the current trend, the growth is speeding up, so it is just the beginning. This fact is just an index of how recent the volume problem is. In regards to the total size of data, in 2009 it was about 0.8 zetta bytes, and in 2020 it will be 44 times more. The main data source is the Internet since the web diusion aects billions of people and each one of them produces a lot of data. Indeed, almost every forum, social network, and chat gathers the thoughts of thousands or even millions of people. Just to give some numbers, Facebook generates every day about 10 tera bytes of data and Twitter about 7 tera bytes. These data can be used to extract information about many situations. For instance, it is possible to evaluate the opinions about some products, assuming that they will have a significant impact on the sales.

Of course not all data will be used, but, combining stored data with web information and devices tracks, a single problem can be related with terabytes of even peta bytes of useful data. Clearly, even if the analysis does not consider the web, the volume can be very big. Unfortunately, current technologies imply an elevated cost for the treatment of big volumes. For this reason, there is a need for a new kind of data storage and processing.

1.2 Velocity

Nowadays, the world is a lot quick change, so it is often necessary to take fast decisions. The amount of data that are generated every day is very big and it is necessary to gather, store, and analyze them in a fast way. Sometimes it is even necessary to have a real time track of facts that comes from the collection and analysis of streams of data coming from different sources. Some examples of problems that need this kind of information are nancial investments planning, weather forecast, and track management.

In order to speed up an analysis, the easiest way is to store data in a fast and efficient memory, i.e. the RAM of the computer. This approach, called in-memory processing, is now used a lot. For example, the statistical software R is mostly based on that. Unfortunately, fast and efficient memory devices are a very expensive, so their size is limited. For that reason, the RAM of computers can contain only a few gigabytes. Thats why we use the software alternatives to improve this problems.

1.3 Variety

A few years ago, analysis used to deal only with well-structured sets of data. However, most of data growth is due to websites as they contain texts, images, videos, and log files. Each of these kinds of data is structured in its own way, so it requires a proper handle. Furthermore, also data generated by sensors are usually unstructured. Of course it is not ways effective to ignore all these useful information, so new tools must be able to treat all kinds of data, or at least the most of them.

A possibility is to use an approach that is partly equal to the current one. It consists in the structuring data with new tools, before conducting an analysis through classical tools. For example, data scraping is a technique that translates human-readable information, such as texts, in computer language, i.e. in structured data.

A better way is to develop new techniques that directly analyze all data. Since there are different kinds of unstructured data, each one of them needs a proper tool. Therefore, this approach leads to some complications and to the use of a wider number of new techniques.

Protecting and preserving information:

Another important aspect of Big Data involves protecting information and keeping it moving, even during disruptive events. Things like inclement weather, a sudden load on an energy grid (such as people plugging in their electric vehicles every evening) or mechanical failure can cause brown outs and black outs that will have utility companies scrambling to get their service trucks out the door before the flood of service calls begins. For example, last summer in Dublin, Ireland, a transistor failure caused a power outage at major cloud computing data hubs for Amazon and Microsoft – what followed was a series of failures that resulted in partial corruption of the data base and the deletion of important data.

Chapter 2: Present Trends in Data Analysis

Along with social, mobile and cloud, analytics and associated data technologies have earned a place as one of the core disruptors of the digital age. In last few years we saw big data initiatives moving from test to production and a strong push to leverage new data technologies to power business intelligence. Based on previous analysis we can perfectly assume that the next years will be for analysis of Data and business organizations will try to convert more data to meaningful information.

2.1 Trends for Big Data:

Fortunately, the following trends promise to provide tools and technologies that can help industries and enterprises involved with handling, storing and transmitting data:

- Faster data capture and analysis. New tools allow this to happen as quickly as the data is generated. One example: real-world models of events.
- More intelligent, automated decision-making. Developers are creating software and languages designed to handle intricate "if/then" scenarios, empowering administrators to customize responses to fit any possible scenario.
- Distributed storage techniques and cloud computing. These include the conversion from tape to disk, de-duplication, flash storage and the rapid adoption of 100 Gigabit Ethernet, replacing the fibre channel. All of this allows for more storage capacity and new challenges of retrieval of data and on the fly computing, without necessarily storing everything.

There is a market need to simplify big data technologies, and opportunities for this exist at all levels: technical, consumption and so on. In the next years, there will be significant progress toward simplification of Data analysis, especially Big Data analysis. It doesn't matter who the person is: cluster operator, security administrator or data analyst, everyone wants Hadoop and related big data technologies to be straightforward. Things like a single integrated developer experience or a reduced number of settings or profiles will start to appear across the board. We can predict that Hadoop will be used to deliver more mission critical workloads, beyond the "web scale" companies. While companies like Yahoo, Spotify and TrueCar all have built businesses which significantly leverage Hadoop, we will see Hadoop used by more traditional enterprises to extract valuable insights from the vast quantity of data under management and deliver net new mission critical analytic applications which simply weren't possible without Hadoop.

Big data is growing up: Hadoop adds to enterprise standards. The enterprise capabilities of Hadoop will mature in next year's. A lot of research is going on to standardize the Hadoop Environment. As further evidence to the growing trend of Hadoop becoming a core part of the enterprise IT landscape, we'll see investment grow in the components surrounding enterprise systems such as security and extendibility. Apache Sentry project provides a system for enforcing fine-grained, role-based authorization to data and metadata stored on a Hadoop cluster. These are the types of capabilities that customers expect from their enterprise-grade RDBMS platforms and are now coming to the forefront of the emerging big data technologies, thus eliminating one more barrier to enterprise adoption.

Big data gets fast: Options expand to add speed to Hadoop. Hadoop will gain the sort of performance that has traditionally been associated with data warehouses. With Hadoop gaining more traction in the enterprise, we see a growing demand from end users for the same fast data exploration capabilities they've come to expect from traditional data warehouses. Generally to meet that end user demand, we see growing adoption of technologies such as Cloudera Impala, AtScale, Actian Vector and Jethro Data that enable the business user's old friend, the OLAP cube, for Hadoop further blurring the lines behind the "traditional" BI concepts and the world of 'big data'.

The number of options for "preparing" end users to discover all forms of data grows. Self-service data preparation tools are exploding in popularity. This is in part due to the shift toward business-user-generated data discovery tools such as Tableau that reduce time to analyze data. Business users now want to also be able to reduce the time and complexity of preparing data for analysis, something that is especially important in the world of big data when dealing with a variety of data types and formats. We've seen a host of innovation in this space from companies focused on end user data preparation for big data such as Alteryx, Trifacta, Paxata and Lavastorm while even seeing long established ETL leaders such as Informatica with their Rev product make heavy investments here.

MPP Data Warehouse growth is heating up…in the cloud! The "death" of the data warehouse has been overhyped for some time now, but it's no secret that growth in this segment of the market has been slowing. But we now see a major shift in the application of this technology to the cloud where Amazon led the way with an on-demand cloud data warehouse in Redshift. "Redshift was AWS's fastest-growing service, but it now has competition from Google with BigQuery, offerings from long-time data warehouse power players such as Microsoft (with Azure SQL Data Warehouse) and Teradata, along with new start-ups such as Snowflake, winner of Strata + Hadoop World 2015 Startup Showcase, also gaining adoption in this space. Analysts cite 90 percent of companies who have adopted Hadoop will also keep their data warehouses, and with these new cloud offerings, those customers can dynamically scale up or down the amount of storage and compute resources in the data warehouse relative to the larger amounts of information stored in their Hadoop data lake.

2.2 R and Big Data Analysis:

R is becoming the most popular language for data science. That's not to say that it's the only language, or that it's the best tool for every job. It is, however, the most widely used and it is rising in popularity. In recent survey, we found that R is the most popular programming language. R is not only a programming language but also a stable software environment for statistical computing, Data modeling, Machine learning and graphics. The power of R relies on some basic properties.

- R is **data analysis software**: Data scientists, statisticians, and analysts, anyone who needs to make sense of data, really can use R for statistical analysis, data visualization, and predictive modeling.
- R is a **programming language**: An object-oriented language created by statisticians, R provides objects, operators, and functions that allow users to explore, model, and visualize data.
- R is an **environment for statistical analysis**: Standard statistical methods are easy to implement in R, and since much of the cutting-edge research in statistics and predictive modeling is done in R, newly developed techniques are often available in R first.
- R is an **open-source software project**: R is free and, thanks to years of scrutiny and tinkering by users and developers, has a high standard of quality and numerical accuracy. R's open interfaces allow it to integrate with other applications and systems.
- R is a **community**: The R project leadership has grown to include more than 20 leading statisticians and computer scientists from around the world, and thousands of contributors have created add-on packages. With two million users, R boasts a vibrant online community.

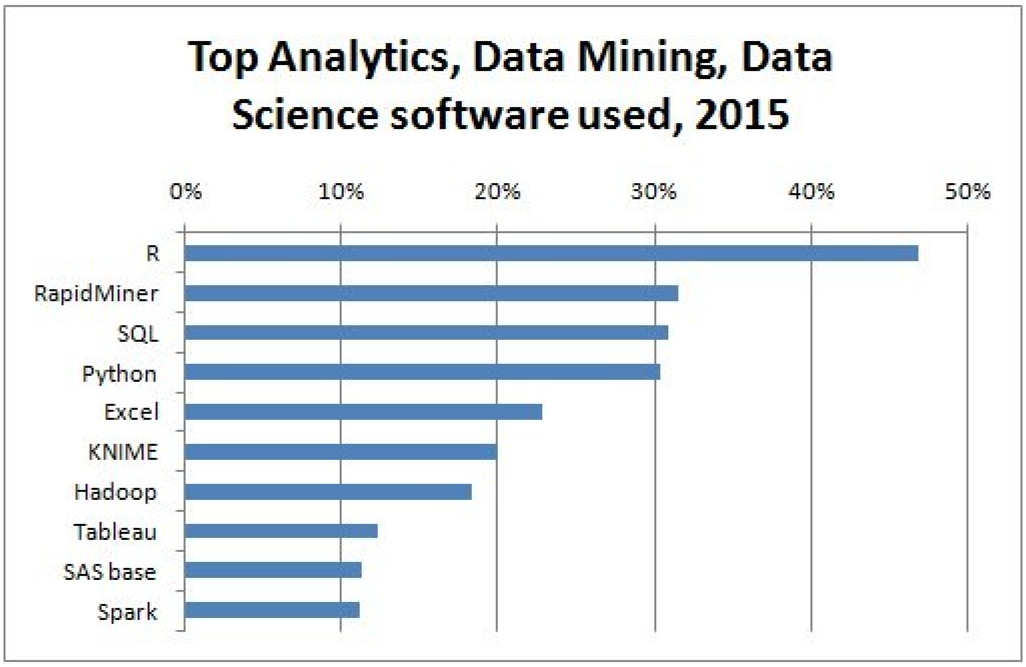


Table: Programming Languages for Data Analysis

Companies using R for Data Analysis: R is in heavy use at several of the best companies who are using data for their business. As Revolution Analytics, R is also the tool of choice for data scientists at big technological organizations, who apply machine learning to data from Bing, Azure, Office, and the Sales, Marketing and Finance departments. In below section I am giving the detail of the companies who are using **R**:

- **Bank of America**: While banking analysts have traditionally pored over Excel files, R is increasingly being used for financial modeling, Bank of America Vice President Niall O'Connor told Fast Company, particularly as a visualization tool.
- **Chicago**: Thanks to an automated system that uses real time textual analysis implemented with R, the Windy City has a powerful tool for identifying the sources of food poisoning from tweets.
- **Facebook**: Data scientists at Facebook use open-source R packages from Hadley Wickham (e.g., ggplot2, dplyr, plyr, and reshape) to explore new data through custom visualizations.
- **New York Times**: The Gray Lady uses R for interactive features like Election Forecast, data journalism, whether from Mariano Rivera's baseball career or the Facebook IPO.
- **Twitter**: Twitter created the Breakout Detection package for R to monitor user experience on its network.

Chapter 3: Hadoop Eco System

Big data is a huge pool of various kinds of data in various formats, which contains hidden potentials for better business decisions. Hadoop, on the other hand, is an open source program designed to handle those big data cheaply and efficiently. It is not only capable of storing and processing many large files at once but also enables moving them over networks quickly, which normally exceeds the capacity of a single server. Hadoop has been well known for its high computing power, strong fault tolerance and scalability.

Operating under Apache license, Hadoop provides a full ecosystem. Hadoop alone cannot do amazing work but with its 'friends', it becomes a perfect match with Big Data. Hadoop Eco system contains 4 different layers but in my project I am focusing in first two layers.

- **Data Storage Layer:** This is where the data is stored in a distributed file system; consist of HDFS and HBase ColumnDB Storage.
- **Data Processing Layer:** In where the scheduling, resource management and cluster management to be calculated here. YARN job scheduling and cluster resource management with Map Reduce are located in this layer.
- **Data Access Layer:** This is the layer where the request from Management layer was sent to Data Processing Layer. Some projects have been setup for this layer, Some of them are: Hive, A data warehouse infrastructure that provides data summarization and ad hoc querying; Pig, A high-level data-flow language and execution framework for parallel computation; Mahout, A Scalable machine learning and data mining library; Avro, data serialization system.
- **Management Layer**: This is the layer that meets the user. User access the system through this layer which has the components like: Chukwa, A data collection system for managing large distributed system and ZooKeeper, high-performance coordination service for distributed applications.

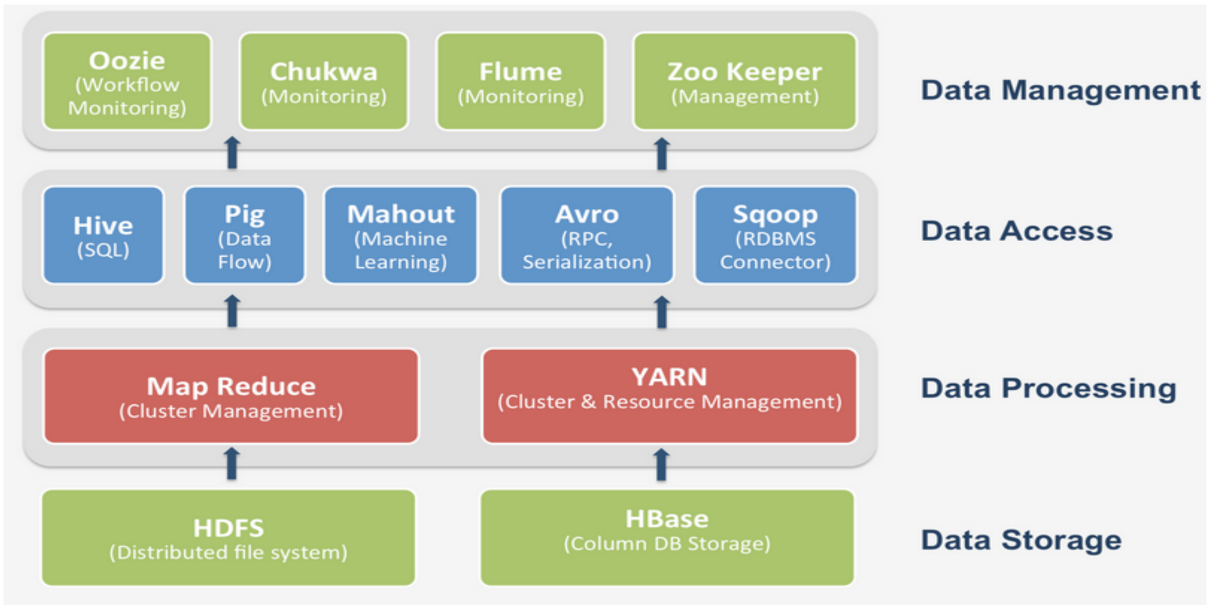

Oozie (Workflow Monitoring)
Chukwa (Monitoring)
Flume (Monitoring)
Zoo Keeper (Management)
Data Management
Hive (SQL)
Pig (Data Flow)
Mahout (Machine Learning)
Avro (RPC, Serialization)
Sqoop (RDBMS Connector)
Data Access
Map Reduce (Cluster Management)
YARN (Cluster & Resource Management)
Data Processing
HDFS (Distributed file system)
HBase (Column DB Storage)
Data Storage

Fig: Hadoop Eco System

3.1: Hadoop Configuration

It is very important to understand how to configure Hadoop Common, HDFS, Map Reduce, and YARN using the standard procedure suggested by Apache Software Foundation. After installing the Java and Hadoop we have to set the environment variables for both. Each component in Hadoop is configured using an XML file. Common properties go in core-site.xml, and properties pertaining to HDFS, Map Reduce, and YARN go into the appropriately named file: hdfs-site.xml, mapred-site.xml, and yarn-site.xml. In my case these files are all located in the usr/hadoop subdirectory. As I am working with single cluster (Standalone) Hadoop, there are no daemons running and everything runs in a single JVM. Standalone mode is suitable for running Map Reduce programs during development, since it is easy to test and debug them. After properly configuring Hadoop, we will follow the standard process to start our system

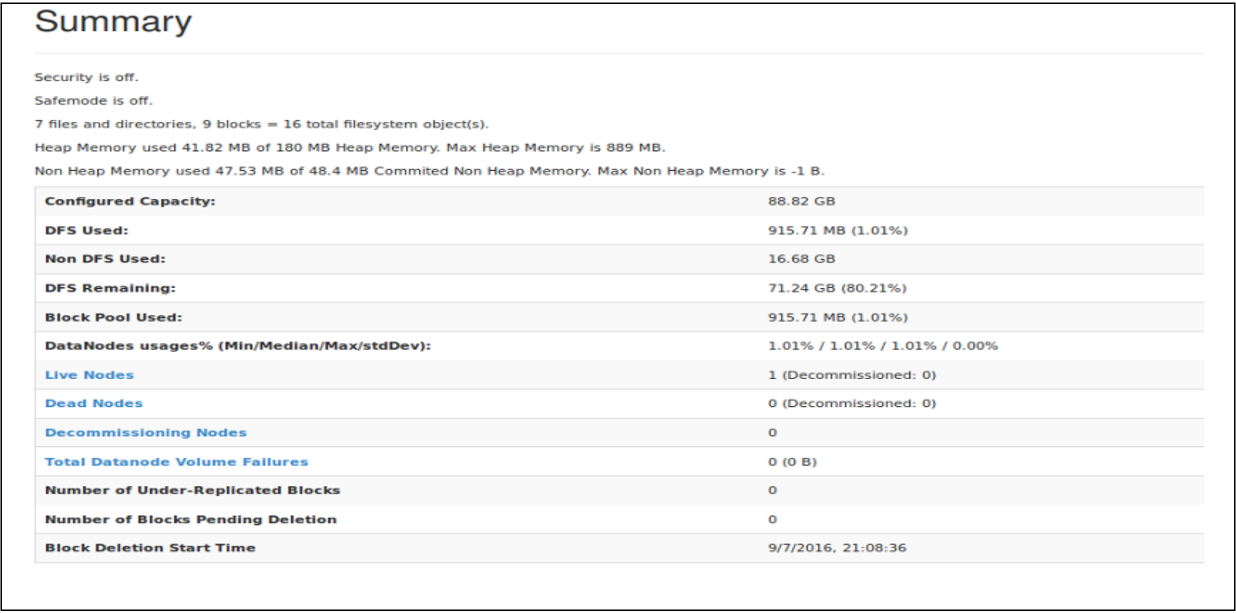
```
hduser@PROJECT:~$ ssh localhost
hduser@PROJECT:~$ hadoop namenode -format
hduser@PROJECT:~$ start-all.sh
hduser@PROJECT:~$ jps
```

Table: Commands to start Hadoop

```
5315 ResourceManager
4917 DataNode
5146 SecondaryNameNode
13595 Jps
5452 NodeManager
4782 NameNode
```

Table: Response after starting dfs and YARN

We can also check all the basic properties from the web interface for the Name Node; by default which is available at: http://localhost:50070/



Summary

Security is off.
Safemode is off.
7 files and directories, 9 blocks = 16 total filesystem object(s).
Heap Memory used 41.82 MB of 180 MB Heap Memory. Max Heap Memory is 889 MB.
Non Heap Memory used 47.53 MB of 48.4 MB Commited Non Heap Memory. Max Non Heap Memory is -1 B.

Configured Capacity:	88.82 GB
DFS Used:	915.71 MB (1.01%)
Non DFS Used:	16.68 GB
DFS Remaining:	71.24 GB (80.21%)
Block Pool Used:	915.71 MB (1.01%)
DataNodes usages% (Min/Median/Max/stdDev):	1.01% / 1.01% / 1.01% / 0.00%
Live Nodes	1 (Decommissioned: 0)
Dead Nodes	0 (Decommissioned: 0)
Decommissioning Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion	0
Block Deletion Start Time	9/7/2016, 21:08:36

Fig: Summary of Hadoop Name node

3.2: HDFS File System

HDFS is the primary distributed storage used by Hadoop applications. A HDFS cluster primarily consists of a Name Node that manages the file system metadata and Data Nodes that store the actual data. The HDFS Architecture Guide describes HDFS in detail. This user guide primarily deals with the interaction of users and administrators with HDFS clusters. The HDFS architecture diagram depicts basic interactions among Name Node, the Data Nodes, and the clients. Clients contact Name Node for file metadata or file modifications and perform actual file I/O directly with the Data Nodes.

HDFS is a file system designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware. In the below section I will explain about the features of HDFS File System:

Very large files:

“Very large” in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size. There are Hadoop clusters running today that store petabytes of data.

Streaming data access:

HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. A dataset is typically generated or copied from source, and then various analyses are performed on that dataset over time. Each analysis will involve a large proportion, if not all, of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record.

Commodity hardware:

Hadoop doesn’t require expensive, highly reliable hardware. It’s designed to run on clusters of commodity hardware (commonly available hardware that can be obtained from multiple vendors) for which the chance of node failure across the cluster is high, at least for large clusters. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

Low-latency data access:

Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS. Remember, HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency. HBase is currently a better choice for low latency access.

Lots of small files:

Because the name node holds file system metadata in memory, the limit to the number of files in a file system is governed by the amount of memory on the name node. As a rule of thumb, each file, directory, and block takes about 150 bytes. So, for example, if you had one million files, each taking one block, you would need at least 300 MB of memory. Although storing millions of files is feasible, billions is beyond the capability of current hardware.

3.2.1: HDFS Concepts

Blocks:

A disk has a block size, which is the minimum amount of data that it can read or write. File systems for a single disk build on this by dealing with data in blocks, which are an integral multiple of the disk block size. File system blocks are typically a few kilobytes in size, whereas disk blocks are normally 512 bytes. This is generally transparent to the file system user who is simply reading or writing a file of whatever length. However, there are tools to perform file system maintenance, such as df and fsck, that operate on the file system block level.

HDFS, too, has the concept of a block, but it is a much larger unit 128 MB by default. Like in a file system for a single disk, files in HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a file system for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of under-lying storage. (For example, a 1 MB file stored with a block size of 128 MB uses 1 MB of disk space, not 128 MB.) When unqualified, the term "block" in this book refers to a block in HDFS.

HDFS blocks are large compared to disk blocks, and the reason is to minimize the cost of seeks. By making a block large enough, the time to transfer the data from the disk can be significantly longer than the time to seek to the start of the block. Thus the time to transfer a large file made of multiple blocks operates at the disk transfer rate.

A quick calculation shows that if the seek time is around 10 ms and the transfer rate is 100 MB/s, to make the seek time 1% of the transfer time, we need to make the block size around 100 MB. The default is actually 128 MB, although many HDFS installations use larger block sizes. This figure will continue to be revised upward as transfer speeds grow with new generations of disk drives.

This argument shouldn't be taken too far, however. Map tasks in Map Reduce normally operate on one block at a time, so if you have too few tasks, your jobs will run slower than they could otherwise.

Name nodes and Data nodes:

An HDFS cluster has two types of nodes operating in a master-worker pattern: a name node and a number of data nodes. The name node manages the file system namespace. It maintains the file system tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log. The name node also knows the data nodes on which all the blocks for a given file are located; however, it does not store block locations persistently, because this information is reconstructed from data nodes when the system starts.

Data nodes are the workhorses of the file system. They store and retrieve blocks when they are told to (by clients or the name node), and they report back to the name node periodically with lists of blocks that they are storing. Without the name node, the file system cannot be used. In fact, if the machine running the name node was obliterated, all the files on the files system would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the data nodes. For this reason, it is important to make the name node resilient to failure, and Hadoop provides two mechanisms for this. The first way is to back up the files that make up the persistent state of the file system metadata. Hadoop can be configured so that the name node writes its persistent state to multiple file systems. These writes are synchronous and atomic. The usual configuration choice is to write to local disk as well as a remote NFS mount.

Block Caching:

Normally a data node reads blocks from disk, but for frequently-accessed files the blocks may be explicitly cached in the data node's memory, in an off-heap block cache. By default a block is cached in only one data node's memory, although the number is configurable on a per-file basis. Job schedulers (for MapReduce, Spark and other frameworks) can take advantage of cached blocks by running tasks on the data node where a block is cached, for increased read performance. A small lookup table used in a join is a good candidate for caching, for example.

HDFS Federation:

The name node keeps a reference to every file and block in the file system in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling. HDFS Federation, introduced in the 2.x release series, allows a cluster to scale by adding name nodes, each of which manages a portion of the file system namespace. For example, one name node might manage all the files rooted under /user, say, and a second name-node might handle files under /share.

Under federation, each name node manages a namespace volume, which is made up of the metadata for the namespace, and a block pool containing all the blocks for the files in the namespace. Namespace volumes are independent of each other, which mean name nodes do not communicate with one another, and furthermore the failure of one name node does not affect the availability of the namespaces managed by other name nodes. Block pool storage is not partitioned, however, so data nodes register with each name node in the cluster and store blocks from multiple block pools. To access a federated HDFS cluster, clients use client-side mount tables to map file paths to name nodes. This is managed in configuration using View File System and the view fs:// URIs.

HDFS High-Availability:

The combination of replicating name node metadata on multiple file systems and using the secondary name node to create checkpoints protects against data loss, but it does not provide high-availability of the file system. The name node is still a single point of failure (SPOF). If it did fail, all clients including Map Reduce jobs would be unable to read, write, or list files, because the name node is the sole repository of the metadata and the file-to-block mapping. In such an event the whole Hadoop system would effectively be out of service until a new name node could be brought online.

To recover from a failed name node in this situation, an administrator starts a new primary name node with one of the file system metadata replicas and configures data nodes and clients to use this new name node. The new name node is not able to serve requests until it has

i) Loaded its namespace image into memory,
ii) Replayed its edit log, and
iii) Received enough block reports from the data nodes to leave safe mode.

3.2.2: HDFS Operations

After properly installation and configuration of Hadoop, it is important to create working environment for HDFS File system. We have to keep some place where we can work with Big HDFS files. We can use HDFS commands to do different kinds of file operations. In our project first we will create HDFS directory and then we will transfer some big files to that directory and will do some operations.

1. Creating the HDFS Directory:

```
hduser@PROJECT:~$ hadoop fs -mkdir /usr/
hduser@PROJECT:~$ hadoop fs -mkdir /usr/hduser
```

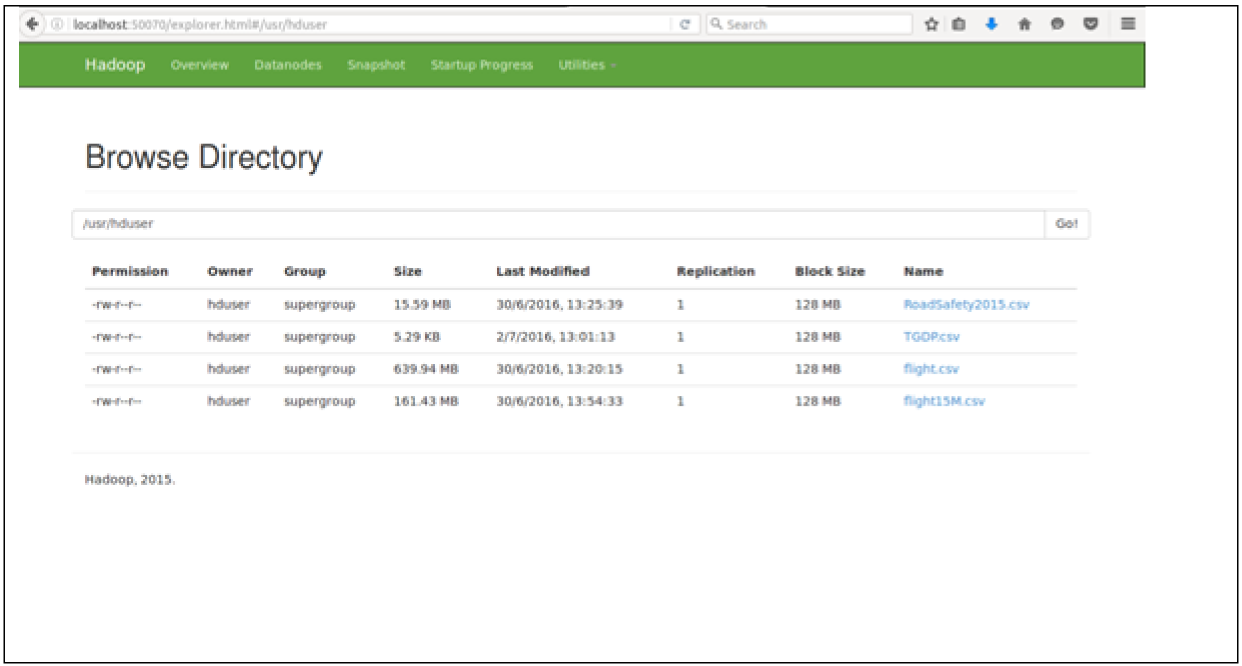
2. Transfer (put) Big files to the directory:

```
hduser@PROJECT:~$ hadoop fs -put /home/hduser/Downloads/flight15M.csv /usr/hduser
hduser@PROJECT:~$ hadoop fs -put /home/hduser/Downloads/TGDP.csv /usr/hduser
```

3. Basic Commands on files:

```
hduser@PROJECT:~$ hadoop fs -ls /usr/hduser
hduser@PROJECT:~$ hadoop fs –cat /usr/hduser/TGDP.csv
```

We can also check all the HDFS files from the web interface; by default which is available at: `http://localhost:50070/`



Browse Directory

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	hduser	supergroup	15.59 MB	30/6/2016, 13:25:39	1	128 MB	RoadSafety2015.csv
-rw-r--r--	hduser	supergroup	5.29 KB	2/7/2016, 13:01:13	1	128 MB	TGDP.csv
-rw-r--r--	hduser	supergroup	639.94 MB	30/6/2016, 13:20:15	1	128 MB	flight.csv
-rw-r--r--	hduser	supergroup	161.43 MB	30/6/2016, 13:54:33	1	128 MB	flight15M.csv

Hadoop, 2015.

Fig: Browsing Files using Local host

3.3: YARN

Apache YARN is Hadoop's cluster resource management system. YARN was introduced in Hadoop 2 to improve the Map Reduce implementation, but it is general enough to support other distributed computing paradigms as well.

YARN provides APIs for requesting and working with cluster resources, but these APIs are not typically used directly by user code. Instead, users write to higher-level APIs provided by distributed computing frameworks, which themselves are built on YARN and hide the resource management details from the user. The distributed computing frameworks like Map Reduce, Spark, and so on running as YARN applications on the cluster compute layer (YARN) and the cluster storage layer (HDFS and HBase).

How YARN works:

YARN provides its core services via two types of long-running daemon: a resource manager (one per cluster) to manage the use of resources across the cluster, and node managers running on all the nodes in the cluster to launch and monitor containers. A container executes an application-specific process with a constrained set of resources (memory, CPU, and so on). Depending on how YARN is configured, a container may be a Unix process, or a Linux cgroup.

To run an application on YARN, a client contacts the resource manager and asks it to run an application master process (step 1 in Figure 3.1). The resource manager then finds a node manager that can launch the application master in a container (step 2a and 12b). Precisely what the application master does once it is running depends on the application. It could simply run a computation in the container it is running in and return the result to the client. Or it could request more containers from the resource managers (step 3), and use them to run a distributed computation (step 4a and 4b).

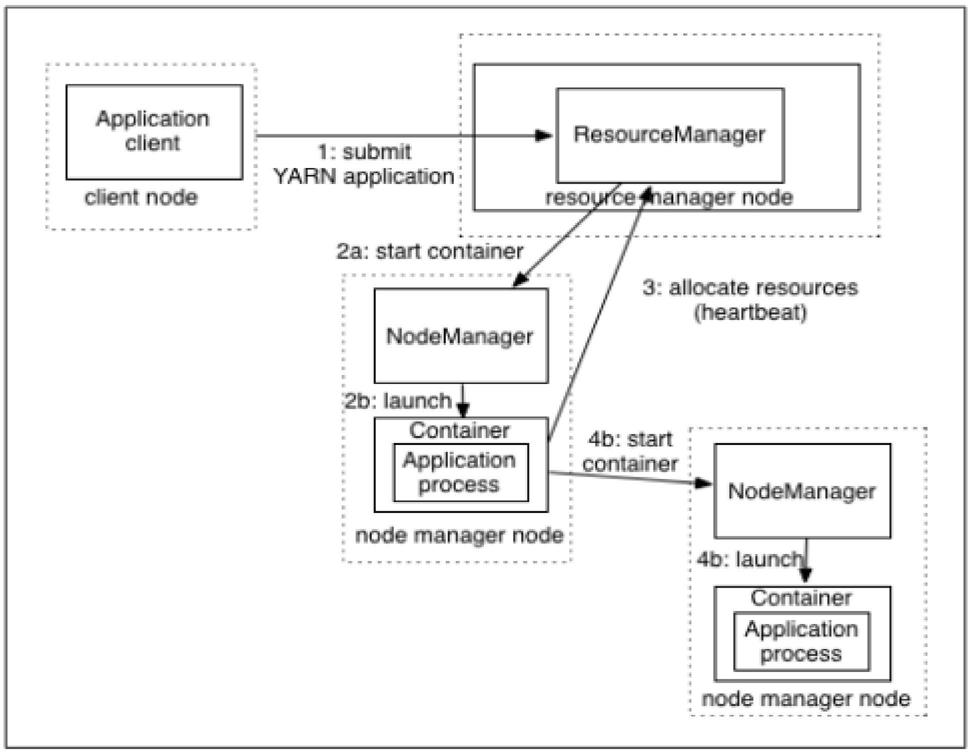
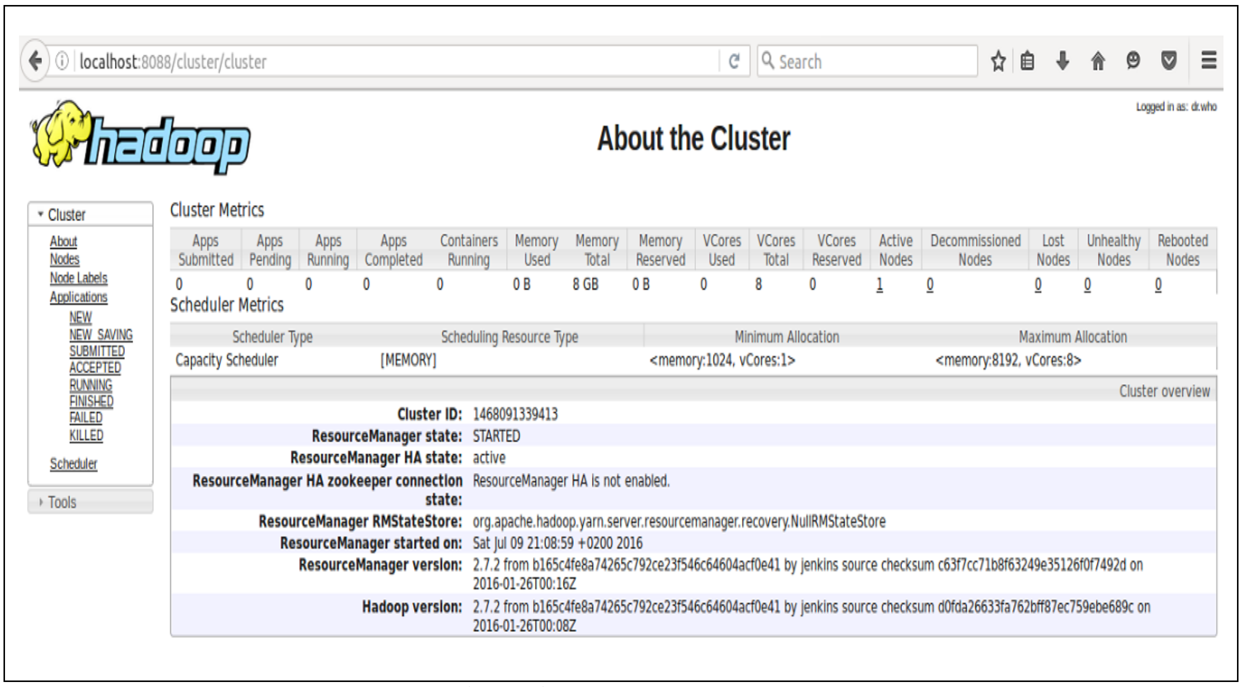


Fig 3.2: How YARN works over Hadoop

We can also check the Resource Manager from the web interface; by default which is available at: h`ttp:` `http://localhost:8088/`



About the Cluster

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
0	0	0	0	0	0 B	8 GB	0 B	0	8	0	1	0	0	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation
Capacity Scheduler	[MEMORY]	<memory:1024, vCores:1>	<memory:8192, vCores:8>

Cluster overview

Cluster ID:	1468091339413
ResourceManager state:	STARTED
ResourceManager HA state:	active
ResourceManager HA zookeeper connection state:	ResourceManager HA is not enabled.
ResourceManager RMStateStore:	org.apache.hadoop.yarn.server.resourcemanager.recovery.NullRMStateStore
ResourceManager started on:	Sat Jul 09 21:08:59 +0200 2016
ResourceManager version:	2.7.2 from b165c4fe8a74265c792ce23f546c64604acf0e41 by jenkins source checksum c63f7cc71b8f63249e35126f0f7492d on 2016-01-26T00:16Z
Hadoop version:	2.7.2 from b165c4fe8a74265c792ce23f546c64604acf0e41 by jenkins source checksum d0fda26633fa762bff87ec759ebe689c on 2016-01-26T00:08Z

Fig: Hadoop Resource Manager

3.4: Map Reduce Jobs

Map Reduce is a processing technique and a program model for distributed computing based on java. The Map Reduce algorithm contains two important tasks, namely Map and Reduce. Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller set of tuples. As the sequence of the name Map Reduce implies, the reduce task is always performed after the map job.

Map Reduce is mainly used for parallel processing of large sets of data stored in Hadoop cluster. Initially, it is a hypothesis specially designed by Google to provide parallelism, data distribution and fault tolerance. MR processes data in the form of key value pairs. A key value (KV) pair is a mapping element between two linked data items key and its value. The key (K) acts as an identifier to the value. An example of a key-value (KV) pair is a pair where the key is the node Id and the value is its properties including neighbor nodes, predecessor node, etc. MR API provides the following features like batch processing, parallel processing of huge amounts of data and high availability.

For processing large sets of data MR comes into the picture. The programmers will write MR applications that could be suitable for their business scenarios. Programmers have to understand the MR working flow and according to the flow, applications will be developed and deployed across Hadoop clusters. Hadoop built on Java APIs and it provides some MR APIs that is going to deal with parallel computing across nodes. The MR work flow undergoes different phases and the end result will be stored in hdfs with replications. Job tracker is going to take care of all MR jobs that are running on various nodes present in the Hadoop cluster. Job tracker plays vital role in scheduling jobs and it will keep track of the entire map and reduce jobs. Actual map and reduce tasks are performed by Task tracker.

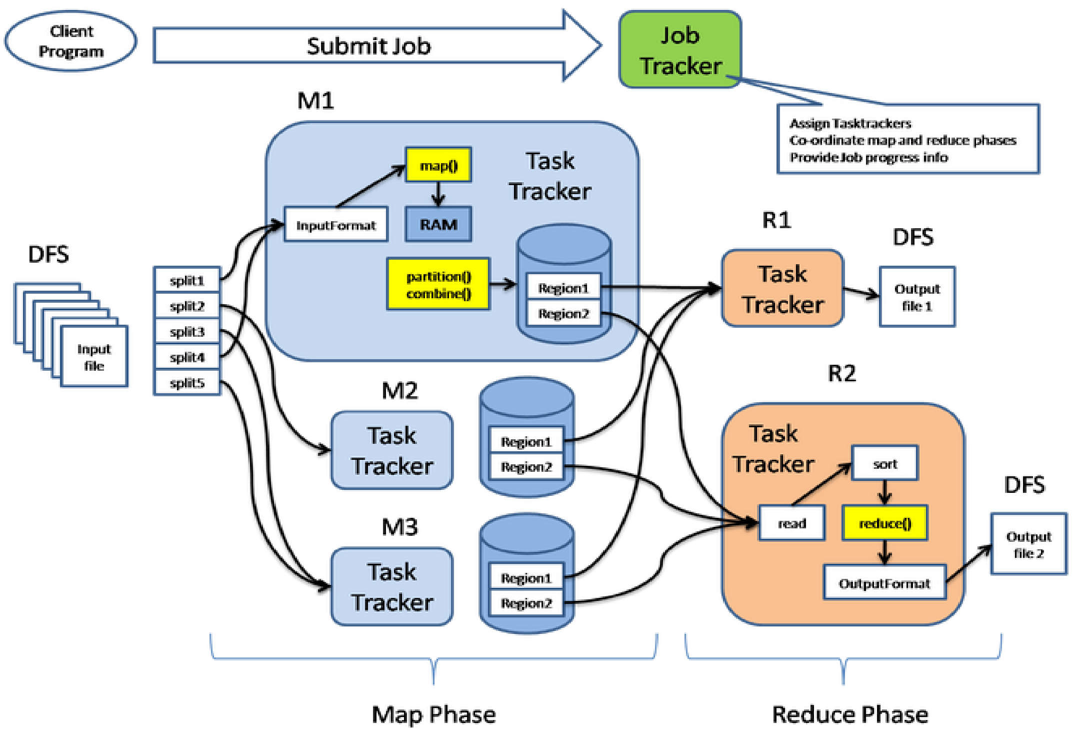


Fig: Map Reduce Phase of Hadoop

Map reduce architecture consists of mainly two processing stages. First one is the map stage and the second one is reduce stage. The actual MR process happens in task tracker. In between map and reduce stages, Intermediate process will take place. Intermediate process will do operations like shuffle and sorting of the mapper output data. The Intermediate data is going to get stored in local file system.

3.4.1 Mapper Phase

In Mapper Phase the input data is going to split into 2 components, Key and Value. The key is writable and comparable in the processing stage. Value is writable only during the processing stage. Suppose, client submits input data to Hadoop system, the Job tracker assigns tasks to task tracker. The input data that is going to get split into several input splits.

Input splits are the logical splits in nature. Record reader converts these input splits in Key-Value (KV) pair. This is the actual input data format for the mapped input for further processing of data inside Task tracker. The input format type varies from one type of application to another. So the programmer has to observe input data and to code according.

Suppose we take Text input format, the key is going to be byte offset and value will be the entire line. Partition and combiner logics come in to map coding logic only to perform special data operations. Data localization occurs only in mapper nodes.

Combiner is also called as mini reducer. The reducer code is placed in the mapper as a combiner. When mapper output is a huge amount of data, it will require high network bandwidth. To solve this bandwidth issue, we will place the reduced code in mapper as combiner for better performance. Default partition used in this process is Hash partition.

A partition module in Hadoop plays a very important role to partition the data received from either different mappers or combiners. Petitioner reduces the pressure that builds on reducer and gives more performance. There is a customized partition which can be performed on any relevant data on different basis or conditions.

Also, it has static and dynamic partitions which play a very important role in hadoop as well as hive. The partitioner would split the data into numbers of folders using reducers at the end of map reduce phase. According to the business requirement developer will design this partition code. This partitioner runs in between Mapper and Reducer. It is very efficient for query purpose.

3.4.2 Intermediate Process

The mapper output data undergoes shuffle and sorting in intermediate process. The intermediate data is going to get stored in local file system without having replications in Hadoop nodes. This intermediate data is the data that is generated after some computations based on certain logics. Hadoop uses a Round-Robin algorithm to write the intermediate data to local disk. There are many other sorting factors to reach the conditions to write the data to local disks.

3.4.3 Reducer Phase

Shuffled and sorted data is going to pass as input to the reducer. In this phase, all incoming data is going to combine and same actual key value pairs are going to write into hdfs system. Record writer writes data from reducer to hdfs. The reducer is not so mandatory for searching and mapping purpose.

Reducer logic is mainly used to start the operations on mapper data which is sorted and finally it gives the reducer outputs like part-r-0001etc,. Options are provided to set the number of reducers for each job that the user wanted to run. In the configuration file mapred-site.xml, we have to set some properties which will enable to set the number of reducers for the particular task.

Speculative Execution plays an important role during job processing. If two or more mappers are working on the same data and if one mapper is running slow then Job tracker assigns tasks to the next mapper to run the program fast. The execution will be on FIFO (First In First Out).

Chapter 4: R and Its Environment

R is an extremely versatile open source programming language for statistics and data science. It is widely used in every field where there is data; business, industry, government, medicine, academia, and so on. R provides a wide variety of statistical (linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering,) and graphical techniques, and is highly extensible. The S language is often the vehicle of choice for research in statistical methodology, and R provides an Open Source route to participation in that activity. One of R's strengths is the ease with which well-designed publication quality plots can be produced, including mathematical symbols and formulae where needed. Great care has been taken over the defaults for the minor design choices in graphics, but the user retains full control.

Object Oriented Programming:

R has the properties of object oriented programming g. The advantages of object orientation can be explained by example. Consider statistical regression. When you perform a regression analysis with other statistical packages, such as SAS or SPSS, you get a mountain of output on the screen. By contrast, if you call the lm() regression function in R, the function returns an object containing all the results the estimated co-efficient, their standard errors, residuals, and so on. You then pick and choose, programmatically, which parts of that object to extract. You will see that R's approach makes programming much easier, partly because it offers a certain uniformity of access to data. This uniformity stems from the fact that R is polymorphic, which means that a single function can be applied to different types of inputs, which the function processes in the appropriate way. Such a function is called a generic function. (If you are a C++ programmer, you have seen a similar concept in virtual functions.) For instance, consider the plot() function. If you apply it to a list of numbers, you get a simple plot. But if you apply it to the output of a regression analysis, you get a set of plots representing various aspects of the analysis. Indeed, you can use the plot() function on just about any object produced by R. This is nice, since it means that you, as a user, have fewer commands to remember.

Functional Programming:

As is typical in functional programming languages, a common theme in R programming is avoidance of explicit iteration. Instead of coding loops, you exploit R's functional features, which let you express iterative behavior implicitly. This can lead to code that executes much more efficiently, and it can make a huge timing difference when running R on large data sets. As you will see, the functional programming nature of the R language offers many advantages:

- Clearer, more compact code
- Potentially much faster execution speed
- Less debugging, because the code is simpler
- Easier transition to parallel programming

4.1 Performance of R:

R was purposely designed to make data analysis and statistics easier for you to do. It was not designed to make life easier for your computer. While R is slow compared to other programming languages, for most purposes, it's fast enough. To understand R's performance, it helps to think about R as both a language and as an implementation of that language. The R-language is abstract: it defines what R code means and how it should work. The implementation is concrete: it reads R code and computes a result. The R-language is mostly defined in terms of how GNU-R works. This is in contrast to other languages, like C++ and javascript that make a clear distinction between language and implementation by laying out formal specifications that describe in minute detail how every aspect of the language should work. Beyond performance limitations due to design and implementation, it has to be said that a lot of R code is slow simply because it's poorly written. Few R users have any formal training in programming or software development. Fewer still write R code for a living. Most people use R to understand data: it's more important to get an answer quickly than to develop a system that will work in a wide variety of situations.

Code optimization is an effective approach to improve the performance of the R code. Optimizing code to make it run faster is an iterative process and it follows the 3 step approach.

1. Find the biggest bottleneck (the slowest part of your code).
2. Try to eliminate it (you may not succeed but that's ok).
3. Repeat until your code is "fast enough."

This sounds easy, but it's not.

Even experienced programmers have a hard time identifying bottlenecks in their code. Instead of relying on your intuition, you should profile your code: use realistic inputs and measure the run time of each individual operation. Only once you've identified the most important bottlenecks can you attempt to eliminate them. It's difficult to provide general advice on improving performance, but if we initiate the steps carefully we can at least understand which the potential spots are. It's easy to get caught up in trying to remove all bottlenecks. Don't! Your time is valuable and is better spent analyzing your data, not eliminating possible inefficiencies in your code. Be pragmatic: don't spend hours of your time to save seconds of computer time. To enforce this advice, you should set a goal time for your code and optimize only up to that goal. This means you will not eliminate all bottlenecks. Some you will not get to because you've met your goal. Others you may need to pass over and accept either because there is no quick and easy solution or because the code is already well optimized and no significant improvement is possible. Accept these possibilities and move on to the next candidate.

Another important feature related to performance is micro benchmarking. A micro benchmark is a measurement of the performance of a very small piece of code, something that might take microseconds (µs) or nanoseconds (ns) to run. This intuition, by-and-large, is not useful for increasing the speed of real code. The observed differences in micro benchmarks will typically be dominated by higher order effects in real code; a deep understanding of subatomic physics is not very helpful when baking. Don't change the way you code because of these micro benchmarks. Instead wait until you've read the practical advice in the following chapters. The best tool for micro benchmarking in R is the microbenchmark package. It provides very precise timings, making it possible to compare operations that only take a tiny amount of time.

4.2 The R environment

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. It includes

- an effective data handling and storage facility,
- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either on-screen or on hardcopy, and
- a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.

The term "environment" is intended to characterize it as a fully planned and coherent system, rather than an incremental accretion of very specific and inflexible tools, as is frequently the case with other data analysis software.

R, like S, is designed around a true computer language, and it allows users to add additional functionality by defining new functions. Much of the system is itself written in the R dialect of S, which makes it easy for users to follow the algorithmic choices made. For computationally-intensive tasks, C, C++ and Fortran code can be linked and called at run time. Advanced users can write C code to manipulate R objects directly.

Many users think of R as a statistics system. We prefer to think of it of an environment within which statistical techniques are implemented. R can be extended (easily) via *packages*. There are about eight packages supplied with the R distribution and many more are available through the CRAN family of Internet sites covering a very wide range of modern statistics.

R has its own LaTeX-like documentation format, which is used to supply comprehensive documentation, both on-line in a number of formats and in hardcopy.

4.3 R Packages:

The capabilities of R are extended through user created packages, which allow specialized statistical techniques, graphical devices, import/export capabilities, reporting tools, etc. These packages are developed primarily in R, and sometimes in Java, C, C++, and Fortran. A core set of packages is included with the installation of R, with more than 7,801 additional packages (as of January 2016) available at the Comprehensive R Archive Network (CRAN), Bioconductor, Omegahat, GitHub, and other repositories. The "Task Views" page on the CRAN website lists a wide range of tasks (in fields such as Finance, Genetics, High Performance Computing, Machine Learning, Medical Imaging, Social Sciences and Spatial Statistics) to which R has been applied and for which packages are available. R has also been identified by the FDA as suitable for interpreting data from clinical research.

Other R package resources include Crantastic, a community site for rating and reviewing all CRAN packages, and R-Forge, a central platform for the collaborative development of R packages, R-related software, and projects. R-Forge also hosts many unpublished beta packages, and development versions of CRAN packages. The Bioconductor project provides R packages for the analysis of genomic data, such as Affymetrix and cDNA microarray object-oriented data-handling and analysis tools, and has started to provide tools for analysis of data from next generation high throughput sequencing methods.

Chapter 5: R + Hadoop , An Integration

In the beginning, big data and R were not natural friends. R programming requires that all objects be loaded into the main memory of a single machine. The limitations of this architecture are quickly realized when big data becomes a part of the equation. In contrast, distributed file systems such as Hadoop are missing strong statistical techniques but are ideal for scaling complex operations and tasks. Vertical scaling solutions requiring investment in costly supercomputing hardware often cannot compete with the cost-value return offered by distributed, commodity hardware clusters. To conform to the in memory, single machine limitations of the R language, data scientists often had to restrict analysis to only a subset of the available sample data. Prior to deeper integration with Hadoop, R language programmers offered a scale out strategy for overcoming the in memory challenges posed by large data sets on single machines.

Fig: RHADOOP Integration

This was achieved using message-passing systems and paging. This technique is able to facilitate work over data sets too large to store in main memory simultaneously; however, its low-level programming approach presents a steep learning curve for those unfamiliar with parallel programming paradigms. Alternative approaches seek to integrate R’s statistical capabilities with Hadoop’s distributed clusters using integration with Hadoop Streaming. For programmers wishing to program MapReduce jobs in languages (including R) other than Java, this option is to make use of Hadoop’s Streaming API. User-submitted MapReduce jobs undergo data transformations with the assistance of UNIX standard streams and serialization, guaranteeing Java-compliant input to Hadoop regardless of the language originally inputted by the programmer. Developers continue to explore various strategies to leverage the distributed computation capability of MapReduce and the almost limitless storage capacity of HDFS in ways that can be exploited by R. Integration of Hadoop with R is ongoing, with Bridging solutions that integrate high-level programming and querying languages with Hadoop, which is RHadoop. I also use this technique to use all the analysis power of R over Big Hadoop files.

5.1 RHADOOP Packages:

RHadoop is a collection of five R packages that allow users to manage and analyze data with Hadoop. The packages have been tested (and always before a release) on recent releases of the Cloudera and Hortonworks Hadoop distributions and should have broad compatibility with open source Hadoop and mapR's distribution. We normally test on recent Revolution R/Microsoft R and CentOS releases, but we expect all the RHadoop packages to work on a recent release of open source R and Linux.

RHadoop consists of the following packages:

5.1.1 rJava: Low-Level R to Java Interface:

Modern programming languages that are mainly used to develop enterprise software systems include: Java. These platforms have rich functionality to write business logic, however they are not much efficient when it comes to statistical or mathematical modeling. In the field of modeling, the major contributors are: R, Weka, Octave etc. Out of these most work as simulation environments, however R could be used both, for simulation as well as for production level systems. From the above discussion it is clear that intelligence based software could not be developed just by using a single technology. To overcome this obstacle a combination of technologies should be used. The figure below shows a high level view of such an intelligent software system and where each technology fits.

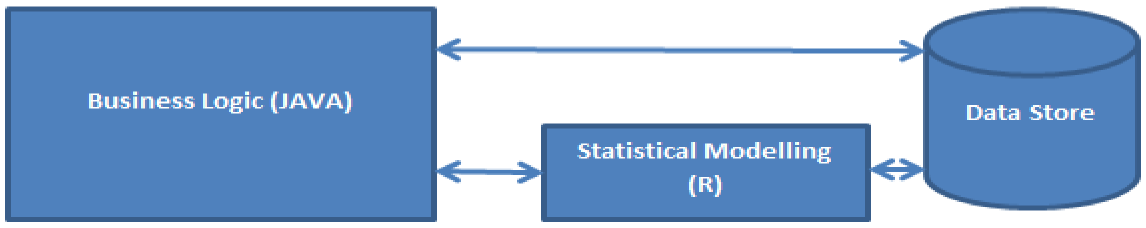


Fig. R with Java

From the figure it is clear that a hybrid system has to be created. In the current scenario the hybrid system consists of JAVA for business logic programming and R for statistical programming. This shows that we have a need to integrate R with Java, which is the main theme of this project.

5.1.2 rhdfs: Integrate R with HDFS

This R package provides basic connectivity to the Hadoop Distributed File System. R programmers can browse, read, write, and modify files stored in HDFS. The following functions are part of this package:

- File Manipulations
  hdfs.copy, hdfs.move, hdfs.rename, hdfs.delete, hdfs.rm, hdfs.del, hdfs.chown, hdfs.put, hdfs.get
- File Read/Write
  hdfs.file, hdfs.write, hdfs.close, hdfs.flush, hdfs.read, hdfs.seek, hdfs.tell, hdfs.line.reader, hdfs.read.text.file
- Directory
  hdfs.dircreate, hdfs.mkdir
- Utility
  hdfs.ls, hdfs.list.files, hdfs.file.info, hdfs.exists
- Initialization
  hdfs.init, hdfs.defaults

5.1.3 rmr2: Mapreduce job in R:

Package that allows R developer to perform statistical analysis in R via Hadoop MapReduce functionality on a Hadoop cluster. One of the important prerequisites of installing this package is rJava. We can install rmr2 if rJava is already in the R. rmr2 package is a good way to perform a data analysis in the Hadoop ecosystem. Its advantages are the flexibility and the integration within an R environment. Its disadvantages are the necessity of having a deep understanding of the MapReduce paradigm and the high amount of required time for writing code. I think that it's very useful to customize the algorithms only after having used some current ones first. For instance, the first stage of the analysis may consist in aggregating data through Hive and perform Machine Learning through Mahout. Afterwards, rmr2 allows modifying the algorithms in order to improve the performances and fit better the problems. The goals of rmr2 package are:

- To provide map-reduce programmers the easiest, most productive, most elegant way to write map reduce jobs. Programs written using the rmr package may need one-two orders of magnitude less code than Java, while being written in a readable, reusable and extensible language.
- To give R programmers a way to access the map-reduce programming paradigm and way to work on big data sets in a way that is natural for data analysts working in R.

5.1.4 plyrmr: Data Manipulation with mapreduce job

This R package enables the R user to perform common data manipulation operations, as found in popular packages such as plyr and reshape2, on very large data sets stored on Hadoop. Like rmr, it relies on Hadoop mapreduce to perform its tasks, but it provides a familiar plyr like interface while hiding many of the mapreduce details. plyrmr provides:

- Hadoop-capable equivalents of well known data.frame functions: transmute and bind.cols generalize over transform and summarize; select from dplyr; melt and dcast from reshape2; sampling, quantiles, counting and more.
- Simple but powerful ways of applying many functions operating on data frames to Hadoop data sets: gapply and magic.wand.
- Simple but powerful ways to group data: group, group.f, gather and ungroup.
- All of the above can be combined by normal functional composition: delayed evaluation helps mitigating any performance penalty of doing so by minimizing the number of Hadoop jobs launched to evaluate an expression.

5.1.5 rhbase: Integrate HBase with R

This R package provides basic connectivity to HBASE, using the Thrift server. R programmers can browse, read, write, and modify tables stored in HBASE. Installing the package requires that you first install and build Thrift. Once we have the libraries built, be sure they are in a path where the R client can find them (i.e. /usr/lib). The following functions are part of this package:

- Table Maninpulation
  hb.new.table, hb.delete.table, hb.describe.table, hb.set.table.mode, hb.regions.table
- Read/Write
  hb.insert, hb.get, hb.delete, hb.insert.data.frame, hb.get.data.frame, hb.scan, hb.scan.ex
- Utility : hb.list.tables
- Initialization : hb.defaults, hb.init

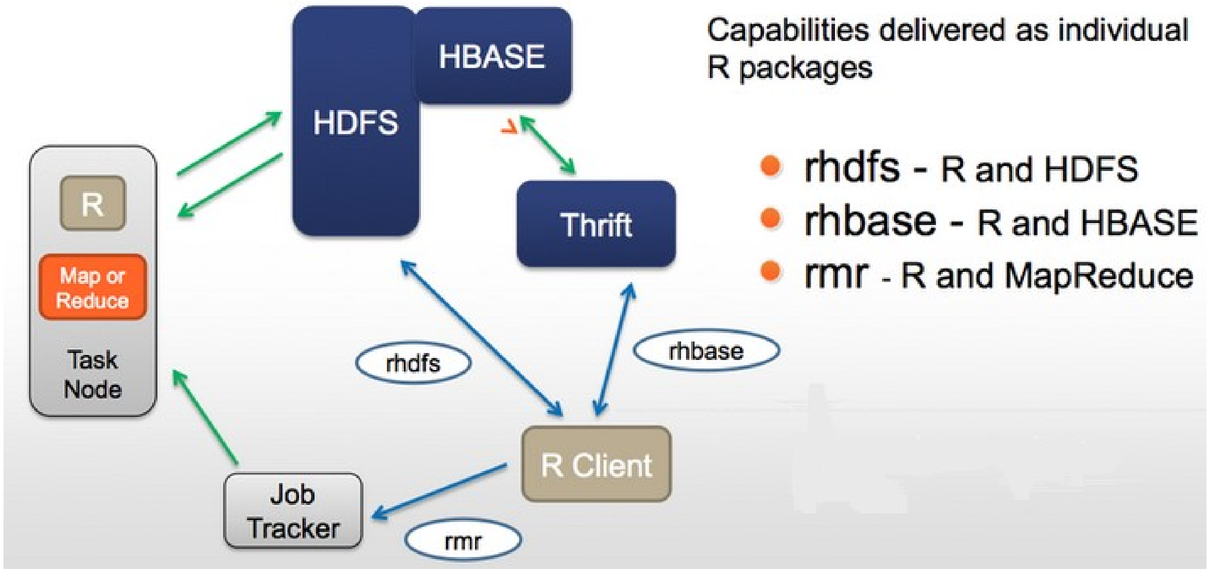


Fig: R packages for RHADOOP

5.2 Environment setup for RHADOOP:

Environment setting is an important task before executing any code. We use different libraries in our R code, so we have to set our environment based on those libraries. Generally in our RHADOOP operations we use library (rmr2) and library (rhdfs). In rmr2 library we use Java , Hadoop streaming and in rhdfs we use Hadoop HDFS, so we have to set the environmental variable for Java and Hadoop.

1. Java Setup:

```
Sys.setenv("JAVA_HOME"="/usr/lib/java/1.8.0_77")
Sys.setenv("JAVAC"="/usr/lib/java/1.8.0_77/bin/javac")
Sys.setenv("JAR"="/usr/lib/java/1.8.0_77/bin/jar")
Sys.setenv("JAVAH"="/usr/lib/java/1.8.0_77/bin/javah")
```

2. Hadoop setup:

```
Sys.setenv(HADOOP_HOME="/usr/lib/hadoop")
Sys.setenv(HADOOP_CMD="/usr/lib/hadoop/bin/hadoop")
Sys.setenv(HADOOP_STREAMING="/usr/lib/hadoop/share/hadoop/tools/lib/hadoop-
streaming-2.7.2.jar")
```

5.3 Getting Started with RHADOOP:

After successfully install all the required RHADOOP packages we will set the environmental variables and then our system will be ready for first Code execution. To test our RHADOOP we will execute the code that will generate the square of a list of value range (1 to 10). In the first section I will explain how I have organized the code and in the second part I will explain how the code will execute and generate result.

5.3.1 First Code Analysis:

In this section, I will explain the first RHADOOP code (code 1) that will help us to test our integration. We are using the hdfs and mapreduce functionality in our example. We are also using time functions to calculate the execution time, which will help us to understand the performance of the code.

```
start.time <- Sys.time()
library(rmr2)
library(rhdfs)
hdfs.init()
sample<-1:10
sints<-to.dfs(sample)
out<-mapreduce(input = sints, map = function(k,v) keyval(v,v^2))
from.dfs(out)
df<-as.data.frame(from.dfs(out))
print(df)
end.time <- Sys.time()
time.taken <- end.time - start.time
time.taken
```

Code 1: Code for Square operation

I have started my coding with assigning the rmr2 and rfdfs libraries and initialized rhdfs using hdfs.init() function. Then I have assigned a sequence of values (1 to 10) in sample variable. The next lines are for mapreduce jobs.

```
sints<-to.dfs(sample)
out<-mapreduce(input = sints, map = function(k,v) keyval(v,v^2))
from.dfs(out)
```

This will be the requirement to write our first mapreduce job in rmr. The first line puts the data into HDFS, where the bulk of the data has to reside for mapreduce to operate on. It is not possible to write out big data with to.dfs, not in a scalable way. to.dfs is nonetheless very useful for a variety of uses like writing test cases, learning and debugging. to.dfs can put the data in a file of our own choosing, but if we don't specify one, it will create temp files and clean them up when done. The return value is something we call a big data object. We can assign it to variables, pass it to other rmr functions, mapreduce jobs or read it back in. It is a stub, that is, the data is not in memory, only some information that helps finding and managing the data. This way we can refer to very large data sets, whose size exceeds memory limits.

Now onto the second line, it has mapreduce with map function. We prefer named arguments with mapreduce, because there are quite a few possible arguments, but it's not mandatory. The input is the variable sints which contains the output of to.dfs, that is a stub for our small number data set in its HDFS version, but it could be a file path or a list containing a mix of both. The function to apply, which is called a map function as opposed to the reduce function, which we are not using here, is a regular R function with a few constraints:

1. It's a function of two arguments, a collection of keys and one of values.
2. It returns key value pairs using the function keyval, which can have vectors, lists, matrices or data.frames as arguments; we can also return NULL. We can avoid calling keyval explicitly but the return value x will be converted with a call to keyval(NULL,x). This is not allowed in the map function when the reduce function is specified and under no circumstance in the combine function, since specifying the key is necessary for the shuffle phase.

In my example, the return value is big data object, and we can pass it as input to other jobs or read it into memory with from.dfs. from.dfs is complementary to to.dfs and returns a key-value pair collection. We use as.data.frame to return the output as frame from.dfs which is useful in defining map reduce algorithms whenever a mapreduce job produces something of reasonable size, like a summary, that can fit in memory and needs to be inspected to decide on the next steps, or to visualize it. It is much more important than to.dfs in production work.

I have also used time functions (Sys.time()) in my R code. At the beginning, I have used time function to initialize the time as start.time and after the execution of all code section, I kept track of the time and kept it in end.time. From these two variables, I could easily calculate the code execution time.

```
start.time <- Sys.time()
.
.
end.time <- Sys.time()
time.taken <- end.time - start.time
time.taken
```

5.3.2 Code Execution and Log Analysis:

In this section, I will explain RHADOOP code execution steps with hdfs, mapreduce and Hadoop streaming functionality. The execution starts with loading the required packages. As we are using libraries, the associated packages need to activate at the beginning. In my case, the required packages are methods, rmr2, rJava and rhdfs. It also loads related objects in the execution environment.

```
Loading required package: methods
Loading required package: rmr2
Loading required package: rJava
Loading required package: rhdfs
```

When the hadoop command is invoked with a classname as the first argument, it launches a Java Virtual Machine (JVM) to run the class. The hadoop command adds the Hadoop libraries (and their dependencies) to the classpath and picks up the Hadoop configuration, too. To add the application classes to the classpath, we've defined an environment variable called HADOOP_CMD=/usr/lib/hadoop/bin/hadoop, which the hadoop script picks up.

At the beginning, the mapreduce job assigns a job name (**local1534009156_0001**) for an entire session and initiate the task. There is one map task (**local1534009156_0001_m_000000_0**) assigned for this mapreduce operation. After the initiation of the job, the map and reduce procedure runs separately. In my first code, I have not used any reduce function, so there will be no reducer (**numReduceTasks: 0)** operations. Hadoop streaming will be executed through pipe MapRed operations and it will complete the map and finish the job. An output file will be generated to default location (hdfs://localhost:9000/tmp/..).

```
16/07/09 21:18:46 INFO mapreduce.Job: Running job: job_local1534009156_0001
16/07/09 21:18:46 INFO mapred.LocalJobRunner: Waiting for map tasks
16/07/09 21:18:46 INFO mapred.LocalJobRunner: Starting task:
attempt_local1534009156_0001_m_000000_0
16/07/09 21:18:47 INFO mapred.Task:  Using ResourceCalculatorProcessTree : [ ]
16/07/09 21:18:47 INFO mapred.MapTask: Processing split:
hdfs://localhost:9000/tmp/file132a2d72e8fa:0+417
16/07/09 21:18:47 INFO mapreduce.Job: Job job_local1534009156_0001 running in uber
mode : false
16/07/09 21:18:47 INFO mapreduce.Job:  map 0% reduce 0%
16/07/09 21:18:47 INFO zlib.ZlibFactory: Successfully loaded & initialized native-zlib
library
16/07/09 21:18:47 INFO compress.CodecPool: Got brand-new decompressor [.deflate]
16/07/09 21:18:47 INFO mapred.MapTask: numReduceTasks: 0
16/07/09 21:18:47 INFO streaming.PipeMapRed: PipeMapRed exec [/usr/bin/Rscript, --
vanilla, ./rmr-streaming-map132a75f36d3b]
16/07/09 21:18:47 INFO Configuration.deprecation: map.input.length is deprecated. Instead,
use mapreduce.map.input.length
16/07/09 21:18:47 INFO Configuration.deprecation: mapred.job.id is deprecated. Instead, use
mapreduce.job.id
16/07/09 21:18:47 INFO Configuration.deprecation: user.name is deprecated. Instead, use
mapreduce.job.user.name
16/07/09 21:18:47 INFO Configuration.deprecation: mapred.task.partition is deprecated.
Instead, use mapreduce.task.partition
16/07/09 21:18:47 INFO streaming.PipeMapRed: R/W/S=1/0/0 in:NA [rec/s] out:NA [rec/s]
16/07/09 21:18:49 INFO streaming.PipeMapRed: Records R/W=3/1
16/07/09 21:18:49 INFO streaming.PipeMapRed: MRErrorThread done
16/07/09 21:18:49 INFO streaming.PipeMapRed: mapRedFinished
16/07/09 21:18:49 INFO mapred.LocalJobRunner:
16/07/09 21:18:50 INFO mapred.Task: Task: attempt_local1534009156_0001_m_000000_0
is done. And is in the process of committing
16/07/09 21:18:50 INFO mapred.LocalJobRunner:
16/07/09 21:18:50 INFO mapred.Task: Task attempt_local1534009156_0001_m_000000_0
```

```
is allowed to commit now
16/07/09 21:18:50 INFO output.FileOutputCommitter: Saved output of task
'attempt_local1534009156_0001_m_000000_0' to
hdfs://localhost:9000/tmp/file132a15fcee89/_temporary/0/task_local1534009156_0001_m
_000000
16/07/09 21:18:50 INFO mapred.LocalJobRunner: Records R/W=3/1
16/07/09 21:18:50 INFO mapred.Task: Task 'attempt_local1534009156_0001_m_000000_0'
done.
16/07/09 21:18:50 INFO mapred.LocalJobRunner: Finishing task:
attempt_local1534009156_0001_m_000000_0
16/07/09 21:18:50 INFO mapred.LocalJobRunner: map task executor complete.
16/07/09 21:18:50 INFO mapreduce.Job:  map 100% reduce 0%
16/07/09 21:18:51 INFO mapreduce.Job: Job job_local1534009156_0001 completed
successfully
```

The next section of the output, which deals with Counters, shows the statistics that Hadoop generates for each job it runs. These are very useful for the purpose of checking whether the amount of data processed matches with our expectation. The system will assign 20 counters for the mapreduce job. It will start with file system counters. There will be a counter for each file system (Local, HDFS, etc). We can receive general idea about the reading, writing, and their operations from this counter.

```
16/07/09 21:18:51 INFO mapreduce.Job: Counters: 20

File System Counters
        FILE: Number of bytes read=119969
        FILE: Number of bytes written=412415
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=420
        HDFS: Number of bytes written=1886
        HDFS: Number of read operations=9
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=3
```

The final section of counter deals with Map Reduce Framework. The number of input records consumed by all the maps in the job is denoted by Map input record and in this case it is 3. The number of Map output records is 21. As there is no reduce operation in our code, Reduce input/ output events are absent. Hadoop mapreduce job only works with mapper and generate result based on it.

```
Map-Reduce Framework

        Map input records=3
        Map output records=21
        Input split bytes=94
        Spilled Records=0
        Failed Shuffles=0
        Merged Map outputs=0
        GC time elapsed (ms)=0
        Total committed heap usage (bytes)=158334976
```

Result:

The final section is designated for the result. System will generate the square of the series of value (1 to 10). In the result system, the time of code execution will also be shown. Total execution time is 1.089565.

```
   key val
1    1   1
2    2   4
3    3   9
4    4  16
5    5  25
6    6  36
7    7  49
8    8  64
9    9  81
10  10 100
> end.time <- Sys.time()
> time.taken <- end.time - start.time
> time.taken
Time difference of 1.089565 mins
```

Chapter 6: Case Study 1: GDP of a Country:

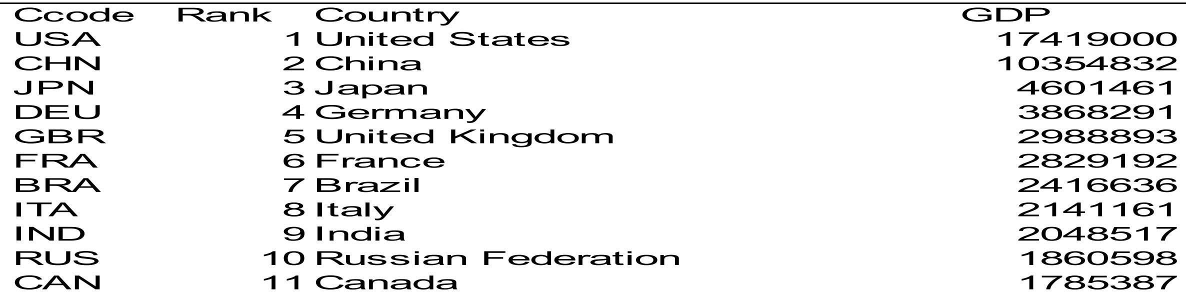
In the first case study, I am going to demonstrate Gross domestic product (GDP) data analysis using RHadoop. This Data set is available in World Bank data catalog (http://datacatalog.worldbank.org) site, which represents the present (2015) GDP of 195 countries. We will compare all the GDP data with a fixed value and finally attempt to know how many countries are in Poor and Rich (using set a random GDP) category. In this case study, I am going to use functionality of HDFS and MapReduce.

6.1 GDP Data set

This data set contains four columns with different variables. First column is the Country code of the individual countries and the assigned data type is Character. 2nd one is the Ranking of the country (1 to 195) and they are Numeric Integers. 3nd one is the full name of the country, whose data type is Character, and final one is the GDP value given is million USD and used data type is Numeric Integer.

Ccode: Character type
Rank: Numeric Integer (1 to 195)
Country: Character type (Full Name of the country)
GDP: Numeric Integer (GDP Value)

The data, as it is, is not worthy of processing. The data needs to be adjusted first, to make it suitable for MapReduce algorithm. The final format that we have used for data analysis is as follows (where the last column is the GDP of the given country in millions USD). As we using CSV format, all the data is separated by comma (,).



Ccode	Rank	Country	GDP
USA	1	United States	17419000
CHN	2	China	10354832
JPN	3	Japan	4601461
DEU	4	Germany	3868291
GBR	5	United Kingdom	2988893
FRA	6	France	2829192
BRA	7	Brazil	2416636
ITA	8	Italy	2141161
IND	9	India	2048517
RUS	10	Russian Federation	1860598
CAN	11	Canada	1785387

6.2 Code Analysis for GDP

I have started my code with assigning the rmr2 and rfdfs libraries and initialized rhdfs using hdfs.init() function. Theses libraries are associated with HDFS file system operation and mapreduce jobs.

```
library(rmr2)
library(rhdfs)
hdfs.init()
```

Next section is for data insertion and pre processing. hdfs.file command is used to specifying the hadoop HDFS file in read mode. This function can be used to read and write files both on the local file system and the HDFS. If the object is a raw vector, it is written directly to the HDFS connection object, otherwise it is serialized and the bytes written to the connection. An environment will be assigned to ashd after hdfs.file operation. Using the hdfs.read command,we can read the ashd environment. rawToChar converts raw bytes(already in m) either to a single character string or a character vector of single bytes. Finally we will read this character as table format, so our data will now be available in gdp variable as tabular format.

```
ashd <- hdfs.file("/usr/hduser/TGDP.csv",mode="r")
m = hdfs.read(ashd)
c = rawToChar(m)
gdp = read.table(textConnection(c),header = TRUE, sep = ",")
```

In this GDP case study, hdfs data is stored to gdp.values using to.dfs, which can put the data in a file but if we don't specify one, it will create temp files and clean them up when done. The return value is something we call a big data object. We can assign it to variables, pass it to other rmr functions, mapreduce jobs or read it back in. It is a stub, that is the data is not in memory, only some information that helps finding and managing the data. This way we can refer to very large data sets, whose size exceeds memory limits.

The next part is dedicated for logical operation. I have assigned a random value to the variable and compared gdp (v[4]) value of the data set with it. If the GDP is less than the variable, the GDP will be either in the Poor group, otherwise in the Rich group.

The final section is for the mapreduce job. In the mapreduce function, we have used gdp.values as input. Both map and reduce options are using keyval pairs to assign the specific job. from.dfs is complementary to to.dfs and returns a key-value pair collection which will generate the final result for this program.

```
gdp.values <- to.dfs(gdp)
Minavg = 50000
gdp.map.fn <- function(k,v) {
key <- ifelse(v[4] < Minavg, "Poor", "Rich")
            keyval(key, 1)
            }

count.reduce.fn <- function(k,v) {  keyval(k, length(v)) }
count <- mapreduce(input=gdp.values, map = gdp.map.fn, reduce = count.reduce.fn)
from.dfs(count)
```

I have also used time function (Sys.time()) in my R code. At the beginning, I have used time function to initialize the time as start.time and after the execution of all code section, I have kept track of the time and keep it in end.time. From these two variables, I can easily calculate the code execution time.

```
start.time <- Sys.time()
.
.
end.time <- Sys.time()
time.taken <- end.time - start.time
time.taken
```

6.3 Code Execution and Log Analysis

In this section, I am going to explain the code execution steps with hdfs, mapreduce and Hadoop streaming functionality. The execution will start with loading the required packages. As we are using libraries, the associated packages need to be activated first. In my case the required packages are methods, rmr2, rJava and rhdfs. It also loads related objects in the execution environment.

```
Loading required package: methods
Loading required package: rmr2
Loading required package: rJava
Loading required package: rhdfs
```

When the hadoop command is invoked with a classname as the first argument, it launches a Java Virtual Machine (JVM) to run the class. The hadoop command adds the Hadoop libraries (and their dependencies) to the classpath and picks up the Hadoop configuration, too. To add the application classes to the classpath, we've defined an environment variable called HADOOP_CMD=/usr/lib/hadoop/bin/hadoop, which the hadoop script picks up.

Initially, the mapreduce job assigns a job id (**local20849418_0001**) for an entire session and start the task. There is one map task (**local20849418_0001_m_000000_0**) and one reduce task (**local20849418_0001_r_000000_0**) is assigned for this mapreduce operation. After starting the job, the map and reduce procedure runs separately. Hadoop streaming will start execution through pipe MapRed operations, first complete the map, and then finish reduce and finally complete the entire job. Mapreduce operation can be explained as:

- Assign id for map (**local20849418_0001_m_000000_0**) and reduce (**local20849418_0001_r_000000_0**) task.
- Initialize mapreduce.Job: **map 0% reduce 0%**
- Complete the map task and start the reduce task. **map 100% reduce 0%**
- Complete the reduce task. **map 100% reduce 100%**

An output file will be generated to a default location (hdfs://localhost:9000/tmp/..).

```
16/07/09 21:39:12 INFO mapreduce.Job: The url to track the job: http://localhost:8080/
16/07/09 21:39:12 INFO mapreduce.Job: Running job: job_local20849418_0001
16/07/09 21:39:12 INFO mapred.LocalJobRunner: Waiting for map tasks
16/07/09 21:39:12 INFO mapred.LocalJobRunner: Starting task:
attempt_local20849418_0001_m_000000_0
16/07/09 21:39:12 INFO mapred.MapTask: numReduceTasks: 1
16/07/09 21:39:13 INFO mapreduce.Job:  map 0% reduce 0%
16/07/09 21:39:14 INFO streaming.PipeMapRed: Records R/W=3/1
16/07/09 21:39:14 INFO mapred.Task: Task:attempt_local20849418_0001_m_000000_0 is
done. And is in the process of committing
16/07/09 21:39:14 INFO mapred.LocalJobRunner: Records R/W=3/1
16/07/09 21:39:14 INFO mapred.Task: attempt_local20849418_0001_m_000000_0' done.
16/07/09 21:39:14 INFO mapred.LocalJobRunner: Finishing task:
attempt_local20849418_0001_m_000000_0
16/07/09 21:39:14 INFO mapred.LocalJobRunner: map task executor complete.
16/07/09 21:39:14 INFO mapred.LocalJobRunner: Waiting for reduce tasks
16/07/09 21:39:14 INFO mapred.LocalJobRunner: Starting task:
attempt_local20849418_0001_r_000000_0
16/07/09 21:39:14 INFO reduce.EventFetcher: attempt_local20849418_0001_r_000000_0
Thread started: EventFetcher for fetching Map Completion Events
16/07/09 21:39:15 INFO mapreduce.Job:  map 100% reduce 0%
16/07/09 21:39:16 INFO streaming.PipeMapRed: Records R/W=5/1
16/07/09 21:39:16 INFO streaming.PipeMapRed: MRErrorThread done
16/07/09 21:39:16 INFO streaming.PipeMapRed: mapRedFinished
16/07/09 21:39:16 INFO mapred.Task: Task:attempt_local20849418_0001_r_000000_0 is
done. And is in the process of committing
16/07/09 21:39:16 INFO mapred.Task: Task attempt_local20849418_0001_r_000000_0 is
allowed to commit now
16/07/09 21:39:16 INFO output.FileOutputCommitter: Saved output of task
'attempt_local20849418_0001_r_000000_0' to
hdfs://localhost:9000/tmp/file132a1a7fec10/_temporary/0/task_local20849418_0001_r_0000
00
16/07/09 21:39:16 INFO mapred.LocalJobRunner: Records R/W=5/1 > reduce
16/07/09 21:39:16 INFO mapred.Task: Task 'attempt_local20849418_0001_r_000000_0'
done.
16/07/09 21:39:16 INFO mapred.LocalJobRunner: Finishing task:
attempt_local20849418_0001_r_000000_0
16/07/09 21:39:16 INFO mapred.LocalJobRunner: reduce task executor complete.
16/07/09 21:39:17 INFO mapreduce.Job:  map 100% reduce 100%
16/07/09 21:39:17 INFO mapreduce.Job: Job job_local20849418_0001 completed
```

The next section of the log deals with Counters, which projects the statistics that Hadoop generates for every job it runs. These are very useful for ascertaining whether the amount of data processed is what we expected beforehand. System assigns 36 counters for mapreduce task. It starts with the file system counters. There is a designated counter for each file system (Local, HDFS, etc). We can have general ideas about the read, write, and their operations from this counter.

```
16/07/09 21:39:17 INFO mapreduce.Job: Counters: 36
	File System Counters
		FILE: Number of bytes read=265302
		FILE: Number of bytes written=849258
		FILE: Number of read operations=0
		FILE: Number of large read operations=0
		FILE: Number of write operations=0
		HDFS: Number of bytes read=8670
		HDFS: Number of bytes written=1623
		HDFS: Number of read operations=17
		HDFS: Number of large read operations=0
		HDFS: Number of write operations=4
```

The final section of counter deals with Map Reduce Framework. The number of input records consumed by all the maps in the job is denoted by Map input record and for this case it is 3. The number of Map output records is 5. As there is no combiner, the input records of Reduce will be the same as the output records of Map. In this case, Hadoop mapreduce job handles both the mapper and reducer to generate result.

```
Map-Reduce Framework
		Map input records=3
		Map output records=5
		Map output bytes=2705
		Map output materialized bytes=2728
		Input split bytes=94
		Combine input records=0
		Combine output records=0
		Reduce input groups=2
		Reduce shuffle bytes=2728
		Reduce input records=5
		Reduce output records=6
		Spilled Records=10
		Shuffled Maps =1
		Failed Shuffles=0
		Merged Map outputs=1
		GC time elapsed (ms)=17
		Total committed heap usage (bytes)=528482304
```

Result:

The result is separated into two groups: Rich and Poor. In our simplified analysis, 81 countries are Rich and the remaining 114 are Poor. The total program execution time is 33.60417.

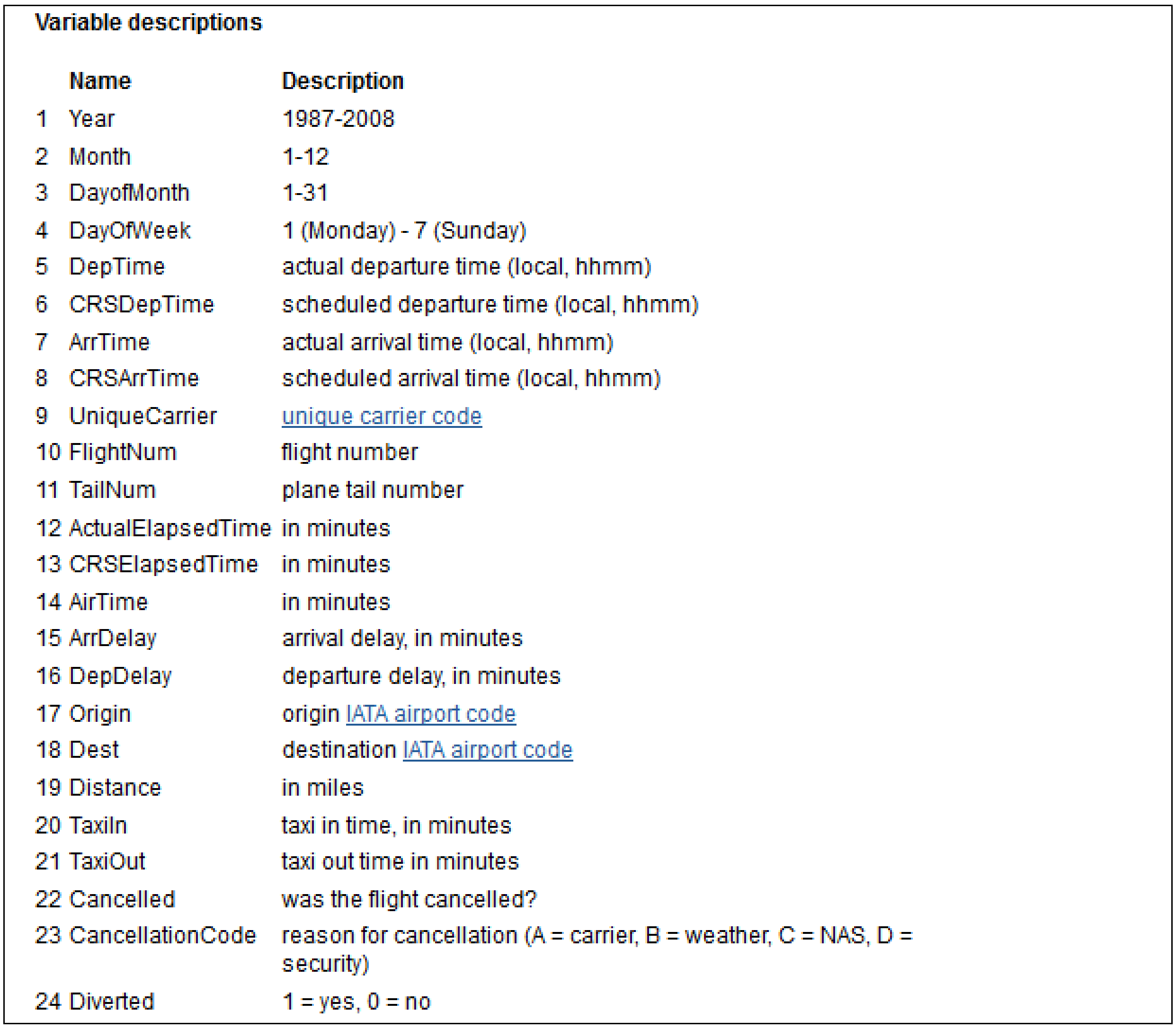
```
> from.dfs(count)
$key   GDP
 "Rich"
 "Poor"
$val
[1]  81 114
Time difference of 33.60417 secs
```

Chapter 7: Case Study 2: Operation on Flight Data set

In my second case study, I am going to analyze Flight data set using RHadoop. This Data set is available in Statistical computing (http://stat-computing.org/dataexpo/2009/the-data.html) web site, which contains flight data of different years. In this case study, I am using the data of 2005 to find out the number of flight were cancelled in that year.

7.1 Flight Data set

This data set contains 29 variables and our focus is on 22nd variable which is labeled as cancelled. This data set contains millions of rows, and for our experiment we have created 2 different data sets containing 1 million and 1.5 million rows. For both data sets , we will find out number of flights were cancelled in the year 2005.



Variable descriptions

	Name	Description
1	Year	1987-2008
2	Month	1-12
3	DayofMonth	1-31
4	DayOfWeek	1 (Monday) - 7 (Sunday)
5	DepTime	actual departure time (local, hhmm)
6	CRSDepTime	scheduled departure time (local, hhmm)
7	ArrTime	actual arrival time (local, hhmm)
8	CRSArrTime	scheduled arrival time (local, hhmm)
9	UniqueCarrier	unique carrier code
10	FlightNum	flight number
11	TailNum	plane tail number
12	ActualElapsedTime	in minutes
13	CRSElapsedTime	in minutes
14	AirTime	in minutes
15	ArrDelay	arrival delay, in minutes
16	DepDelay	departure delay, in minutes
17	Origin	origin IATA airport code
18	Dest	destination IATA airport code
19	Distance	in miles
20	TaxiIn	taxi in time, in minutes
21	TaxiOut	taxi out time in minutes
22	Cancelled	was the flight cancelled?
23	CancellationCode	reason for cancellation (A = carrier, B = weather, C = NAS, D = security)
24	Diverted	1 = yes, 0 = no

7.2 Code Analysis for Flight Operations

I have started my code with assigning the rmr2 and rfdfs libraries and initialized rhdfs using hdfs.init() function. Theses libraries are associated with HDFS file system operation and mapreduce jobs.

```
library(rmr2)
library(rhdfs)
hdfs.init()
```

Next section is for data insertion and pre processing. hdfs.file command is used to specifying the hadoop HDFS file in read mode. This function can be used to read and write files both on the local file system and the HDFS. If the object is a raw vector, it is written directly to the HDFS connection object, otherwise it is serialized and the bytes written to the connection. An environment will be assigned to ashd after hdfs.file operation. Using the hdfs.read command, we can read the ashd environment. rawToChar converts raw bytes(already in m) either to a single character string or a character vector of single bytes. Finally we will read this character as table format, so our data will now be available in data variable as tabular format. As this hdfs file contains millions of rows, it's really time consuming to insert the data into the variable.

```
ashd <- hdfs.file("/usr/hduser/flight1M.csv",mode="r")
m = hdfs.read(ashd)
c = rawToChar(m)
data = read.table(textConnection(c),header = TRUE, sep = ",")
```

In this case study, hdfs data is stored to data.values using to.dfs, which can put the data in a file but if we don't specify one, it will create temp files and clean them up when done. The return value is something we call a big data object. We can assign it to variables, pass it to other rmr functions, mapreduce jobs or read it back in. It is a stub, that is the data is not in memory, only some information that helps finding and managing the data. This way we can refer to very large data sets, whose size exceeds memory limits.

The next part is dedicated for logical operation. We are dealing with the variable "Cancelled" (v[23]) which contains Boolean value(0/1). If it is 1, the flight was delayed and otherwise its ok. I used this logic in my code.

The final section is for the mapreduce job. In the mapreduce function, we have used data.values as input. Both map and reduce options are using keyval pairs to assign the specific job. from.dfs is complementary to to.dfs and returns a key-value pair collection which will generate the final result for this program.

```
data.values <- to.dfs(data)
data.map.fn <- function(k,v) {
key <- ifelse(v[23] ==1, "Delay", "Ok")
            keyval(key, 1)
            }

count.reduce.fn <- function(k,v) {  keyval(k, length(v)) }
count <- mapreduce(input=data.values, map = data.map.fn, reduce = count.reduce.fn)
from.dfs(count)
```

I have also used time function (Sys.time()) in my R code. At the beginning, I have used time function to initialize the time as start.time and after the execution of all code section, I have kept track of the time and keep it in end.time. From these two variables, I can easily calculate the code execution time.

```
start.time <- Sys.time()
.
.
end.time <- Sys.time()
time.taken <- end.time - start.time
time.taken
```

7.3 Code Execution and Log Analysis

In this section, I am going to explain the code execution steps with hdfs, mapreduce and Hadoop streaming functionality. The execution will start with loading the required packages. As we are using libraries, the associated packages need to be activated first. In my case the required packages are methods, rmr2, rJava and rhdfs. It also loads related objects in the execution environment.

```
Loading required package: methods
Loading required package: rmr2
Loading required package: rJava
Loading required package: rhdfs
```

When the hadoop command is invoked with a classname as the first argument, it launches a Java Virtual Machine (JVM) to run the class. The hadoop command adds the Hadoop libraries (and their dependencies) to the classpath and picks up the Hadoop configuration, too. To add the application classes to the classpath, we've defined an environment variable called HADOOP_CMD=/usr/lib/hadoop/bin/hadoop, which the hadoop script picks up.

Initially, the mapreduce job assigns a job id (**local789849698_0001**) for an entire session and start the task. There is one map task (**local789849698_0001_m_000000_0**) and one reduce task (**local789849698_0001_r_000000_0**) is assigned for this mapreduce operation. After starting the job, the map and reduce procedure runs separately. Hadoop streaming will start execution through pipe MapRed operations, first complete the map, and then finish reduce and finally complete the entire job. Mapreduce operation can be explained as:

- Assign id for map (**local789849698_0001_m_000000_0**) and reduce (**local789849698_0001_r_000000_0**) task.
- Initialize mapreduce.Job: **map 0% reduce 0%**
- Complete the map task and start the reduce task. **map 100% reduce 0%**
- Complete the reduce task. **map 100% reduce 100%**

An output file will be generated to a default location (hdfs://localhost:9000/tmp/..).

```
16/07/09 22:47:00 INFO mapreduce.JobSubmitter: Submitting tokens for job:
job_local789849698_0001
16/07/09 22:47:03 INFO mapred.LocalJobRunner: Waiting for map tasks
16/07/09 22:47:03 INFO mapred.LocalJobRunner: Starting task:
attempt_local789849698_0001_m_000000_0
16/07/09 22:47:03 INFO mapreduce.Job:  map 0% reduce 0%
16/07/09 22:47:05 INFO mapred.MapTask: numReduceTasks: 1
16/07/09 22:47:09 INFO mapreduce.Job:  map 1% reduce 0%
16/07/09 22:47:18 INFO mapreduce.Job:  map 5% reduce 0%
16/07/09 22:47:19 INFO streaming.PipeMapRed: Records R/W=10/1
16/07/09 22:47:24 INFO mapred.LocalJobRunner: Records R/W=10/1 > map
16/07/09 22:47:24 INFO mapreduce.Job:  map 21% reduce 0%
16/07/09 22:47:27 INFO mapred.LocalJobRunner: Records R/W=10/1 > map
16/07/09 22:47:28 INFO mapreduce.Job:  map 23% reduce 0%
16/07/09 22:47:30 INFO streaming.PipeMapRed: Records R/W=50/24
16/07/09 22:47:30 INFO mapred.LocalJobRunner: Records R/W=50/24 > map
16/07/09 22:47:31 INFO mapreduce.Job:  map 32% reduce 0%
16/07/09 22:47:33 INFO mapred.LocalJobRunner: Records R/W=50/24 > map
16/07/09 22:47:34 INFO mapreduce.Job:  map 36% reduce 0%
16/07/09 22:47:36 INFO mapred.LocalJobRunner: Records R/W=50/24 > map
16/07/09 22:47:37 INFO mapreduce.Job:  map 49% reduce 0%
16/07/09 22:47:38 INFO streaming.PipeMapRed: R/W/S=100/58/0 in:3=100/31 [rec/s]
out:1=58/31 [rec/s]
16/07/09 22:47:39 INFO mapred.LocalJobRunner: Records R/W=50/24 > map
16/07/09 22:47:40 INFO mapreduce.Job:  map 63% reduce 0%
16/07/09 22:47:42 INFO streaming.PipeMapRed: Records R/W=123/69
16/07/09 22:47:42 INFO mapred.LocalJobRunner: Records R/W=123/69 > map
16/07/09 22:47:43 INFO mapreduce.Job:  map 67% reduce 0%
16/07/09 22:47:45 INFO streaming.PipeMapRed: MRErrorThread done
16/07/09 22:47:45 INFO streaming.PipeMapRed: mapRedFinished
16/07/09 22:47:45 INFO mapred.LocalJobRunner: Records R/W=123/69 > map
16/07/09 22:47:45 INFO mapred.MapTask: Starting flush of map output
16/07/09 22:47:45 INFO mapred.MapTask: Spilling map output
16/07/09 22:47:47 INFO mapred.Task: Task 'attempt_local789849698_0001_m_000000_0'
done.
16/07/09 22:47:47 INFO mapred.LocalJobRunner: Finishing task:
attempt_local789849698_0001_m_000000_0
16/07/09 22:47:47 INFO mapred.LocalJobRunner: map task executor complete.
16/07/09 22:47:47 INFO mapreduce.Job:  map 100% reduce 0%
16/07/09 22:47:47 INFO mapred.LocalJobRunner: Waiting for reduce tasks
16/07/09 22:47:47 INFO mapred.LocalJobRunner: Starting task:
attempt_local789849698_0001_r_000000_0
16/07/09 22:47:54 INFO mapreduce.Job:  map 100% reduce 69%
```

```
16/07/09 22:48:12 INFO mapred.LocalJobRunner: reduce > reduce
16/07/09 22:48:12 INFO mapreduce.Job:  map 100% reduce 100%
16/07/09 22:48:16 INFO streaming.PipeMapRed: Records R/W=75/1
16/07/09 22:48:16 INFO streaming.PipeMapRed: MRErrorThread done
16/07/09 22:48:16 INFO streaming.PipeMapRed: mapRedFinished
16/07/09 22:48:20 INFO mapred.Task: Task:attempt_local789849698_0001_r_000000_0 is
done. And is in the process of committing
16/07/09 22:48:20 INFO mapred.LocalJobRunner: reduce > reduce
16/07/09 22:48:20 INFO mapred.Task: Task attempt_local789849698_0001_r_000000_0 is
allowed to commit now
16/07/09 22:48:20 INFO output.FileOutputCommitter: Saved output of task
'attempt_local789849698_0001_r_000000_0' to
hdfs://localhost:9000/tmp/file132a4e48f65/_temporary/0/task_local789849698_0001_r_000000
16/07/09 22:48:20 INFO mapred.LocalJobRunner: Records R/W=75/1 > reduce
16/07/09 22:48:20 INFO mapred.Task: Task 'attempt_local789849698_0001_r_000000_0' done.
16/07/09 22:48:20 INFO mapred.LocalJobRunner: Finishing task:
attempt_local789849698_0001_r_000000_0
16/07/09 22:48:20 INFO mapred.LocalJobRunner: reduce task executor complete.
16/07/09 22:48:21 INFO mapreduce.Job: Job job_local789849698_0001 completed successfully
```

The next section of the log deals with Counters, which projects the statistics that Hadoop generates for every job it runs. These are very useful for ascertaining whether the amount of data processed is what we expected beforehand. System assigns 36 counters for mapreduce task. It starts with the file system counters. There is a designated counter for each file system (Local, HDFS, etc). We can have general ideas about the read, write, and their operations from this counter.

```
16/07/09 22:48:22 INFO mapreduce.Job: Counters: 36
	File System Counters
		FILE: Number of bytes read=51430106
		FILE: Number of bytes written=60307376
		FILE: Number of read operations=0
		FILE: Number of large read operations=0
		FILE: Number of write operations=0
		HDFS: Number of bytes read=43343196
		HDFS: Number of bytes written=1677
		HDFS: Number of read operations=17
		HDFS: Number of large read operations=0
		HDFS: Number of write operations=4
```

The final section of counter deals with Map Reduce Framework. The number of input records consumed by all the maps in the job is denoted by Map input record and for this case it is 123. The number of Map output records is 75. As there is no combiner, the input records of Reduce will be the same as the output records of Map. Reduce input group is 30 which handles 75 reduce input records and finally generate 6 reduce output record. In this case, Hadoop mapreduce job handles both the mapper and reducer to generate result.

```
Map-Reduce Framework
            Map input records=123
            Map output records=75
            Map output bytes=8018180
            Map output materialized bytes=8018456
            Input split bytes=94
            Combine input records=0
            Combine output records=0
            Reduce input groups=30
            Reduce shuffle bytes=8018456
            Reduce input records=75
            Reduce output records=6
            Spilled Records=150
            Shuffled Maps =1
            Failed Shuffles=0
            Merged Map outputs=1
            GC time elapsed (ms)=373
            Total committed heap usage (bytes)=404750336
```

Result:

The result is separated into two groups: Delay and Ok. In our simplified analysis, in 1 million flights only 32743 flights were delay. The total program execution time is 6.469971 mins.

```
$key
  Cancelled
 "Ok"
 "Delay"

$val
[1] 967257  32743

>
> end.time <- Sys.time()
> time.taken <- end.time - start.time
> time.taken
Time difference of 6.469971 mins
```

Chapter 8: Observations of Case studies

In the case studies, I used RHADOOP to analyze few set of data. The goal is to understand the RHADOOP and the working principle of Mapreduce over Hadoop files. I have successfully finished my first work and I learned a lot from my experience. In the below section I am explaining my observations for these case studies:

8.1 Managing Big Data set:

We used different size of data files in our case studies. In GDP case we used very small data set. Whatever the size of data set, we have to pre process to avoid the anomalies. In Flight analysis case, we in fact used 2 data sets with 1 million and 1.5 Million rows. In case of data analysis, maximum time spend for data initialization to variables. We used different approaches to understand the operations and performance.

Approach 1: Direct from local directory: System can read and insert all the data to variable. RHADOOP can analysis data using mapreduce functionality.

```
data <- read.csv("/home/hduser/Downloads/flight1M.csv")
```

Approach 2: Using HDFS option and hdfs.file command: System uses character conversion option and can read / insert all the data to variable. In case of missing values and special characters in the file, interruption can happen. RHADOOP can analysis data using mapreduce functionality.

```
ashd <- hdfs.file("/usr/hduser/flight1M.csv",mode="r")
m = hdfs.read(ashd)
c = rawToChar(m)
data = read.table(textConnection(c),header = TRUE, sep = ",")
```

Approach 3: Using HDFS option and hdfs.line.reader command: System uses this command to read / insert specific number of rows. Default numbers of rows are 1000. I used 100000 numbers of rows and it worked perfectly but when I used 1M rows system failed and crashed. RHADOOP can analysis upto 100000 rows of data using mapreduce functionality.

```
reader = hdfs.line.reader("/usr/hduser/flight1M.csv")
xx = reader$read()
data=read.table(textConnection(xx),header = TRUE,sep = ",")
```

Approach 4: Using HDFS option and hdfs.read.text.file command: System uses this command to read / insert the whole data file but handle 100000 numbers of rows. RHADOOP can analysis upto 100000 rows of data using mapreduce functionality.

```
red=hdfs.read.text.file("/usr/hduser/flight1M.csv")
x = red$read()
data2=read.table(textConnection(x),header = TRUE,sep = ",")
```

8.2 MapReduce Job Balance:

In all my mapreduce operation, I found balance between Mapper and Reducer. The number of input records consumed by all the maps in the job is denoted by Map input record and for this case it is 123. The number of Map output records is 75. As there is no combiner, the input records of Reduce will be the same as the output records of Map. Reduce input group is 30 which handles 75 reduce input records and finally generate 6 reduce output record. In this case, Hadoop mapreduce job handles both the mapper and reducer to generate result. If there is no reducer, there will be no Reduce input / output records.

```
Map-Reduce Framework
        Map input records=123
        Map output records=75
        Reduce input groups=30
        Reduce input records=75
        Reduce output records=6
```

8.3 Define number of Mapper & Reducer:

Number of mappers and reducers initially depends on the number of data blocks. If we define any map-reduce job, system generates mapper / reducer (or Both) id based on the primary mapreduce id. As of the definition of rmr2 package, we can define number of mappers and reducer using backend parameters. I did an experiment based on this idea. For the same data set I changed the number of mappers and reducers using rmr2 package parameters. The result was impressive, with the increase of mappers and reducers, execution time was decreasing. But I did not get any concrete information from the log files and also there has not enough documentation for this type of package parameters.

```
Count <- mapreduce(input=data.values,map = data.map.fn,reduce =
count.reduce.fn,backend.parameters = list((hadoop=list(D='mapred.reduce.tasks=1',
D='mapred.map.tasks=2'))))
```

Number of rows	Mapper	Reducer	Time
1M	2	1	6.434929
1M	3	1	5.530457
1M	3	2	5.120123
1M	5	2	4.766741

8.4 Environment and Libraries:

Setting up environmental variables for associated libraries and packages is not easy. We have to set this for each and every session. Before execution of any code, we have to assign those variables properly. When the hadoop command is invoked with a classname in the code, it launches a Java Virtual Machine (JVM) to run the class. The hadoop command adds the Hadoop libraries (and their dependencies) to the classpath and picks up the Hadoop configuration, too. To add the application classes to the classpath, we’ve defined an environment variable called HADOOP_CMD=/usr/lib/hadoop/bin/hadoop, which the hadoop script picks up.

Chapter 9: Conclusion:

Hadoop is the most widely accepted and used open source framework to compute big data analysis in an easily scalable environment. It's a fault tolerant, reliable, highly scalable, cost-effective solution, that can handle petabytes of data. Its two main components: HDFS and MapReduce contribute to the success of Hadoop. It can handle the task of storing and analyzing unstructured data. Hadoop is a tried and tested solution in the production environment and well adopted by industry leading organizations like Google, Yahoo, and Facebook. The power of Hadoop becomes even more enhanced, when we integrate R with it. After combining both, we can use Hadoop for data storage and mapreduce job and R for data analysis together.

In this project, I have successfully used RHADOOP for my case studies. Hadoop had been used for HDFS & mapreduce job and for setting logic and data analysis, R worked perfectly here. These case studies helped me to learn about the operational steps of hdfs and mapreduce jobs. I have used different ways for setting and analyzing the data and observed each and every option. I strongly feel that the success of every work depends on the proper analysis of the operational features, thus, I have tried my best to know about RHADOOP and its operational features for data analysis.

While working with this technology, the fascination and the interest on the subject matter bloomed into me. I aspire to work in detail with the performance issues of mapreduce job and I have already started working on it.

Chapter 10: References

1. Tom White, Hadoop: The Definitive Guide. Fourth Edition . O'Reilly, 2015.
2. Luis Torgo, Data Mining with R Learning with Case Studies. Chapman & Hall/CRC, 2011.
3. Norman Matloff, THE ART OF R PROGRAMMING. No starch press, 2011.
4. Scaling Hadoop to 4000 nodes at Yahoo!, https://developer.yahoo.com/blogs/hadoop/scaling-hadoop-4000-nodes-yahoo-410.html.
5. HDFS Scalability: The limits to growth by KonstantinV. Shvachko (April 2010, pp. 6–16, https://www.usenix.org/publications/login/april-2010-volume-35-number-2/hdfs-scalability-limits-growth)
6. http://www.cio.com/article/3023838/analytics/21-data-and-analytics-trends-that-will-dominate-2016.html
7. http://hortonworks.com/apache/hadoop/
8. http://www.quantide.com
9. http://www.tutorialspoint.com/hadoop/hadoop_mapreduce.htm
10. http://a4academics.com/tutorials/83-hadoop/840-map-reduce-architecture
11. https://blog.udemy.com/hadoop-ecosystem/
12. https://hadoop.apache.org
13. http://www.tutorialspoint.com/hadoop
14. http://www.fastcompany.com/3030063/why-the-r-programming-language-is-good-for-business
15. Advanced R: http://adv-r.had.co.nz/Performance.html
16. http://hadooptutorial.info/rhadoop-installation-on-ubuntu
17. https://blogs.adobe.com/digitalmarketing/analytics/3-reasons-need-big-data-analytics-strategy/

Appendix

In this section I am going to explain the installation process of RHADOOP Packages.

1. rJava : Low-Level R to Java Interface:

1.1 Install Java

To start, we need Java. We can download the Java Runtime Environment (JRE) and Java Development Kit (JDK). After properly installation of Java we can check executing the command:

```
$ java –version
java version "1.7.0_67"
```

1.2 Configure Java Parameters for R

R provides the javareconf utility to configure Java support in R. To prepare the R environment for Java, we can execute this command:

```
$ sudo R CMD javareconf
or
$ R CMD javareconf –e
```

1.3 Install rJava Package

rJava release versions can be obtained from CRAN. Assuming an internet connection is available, the install.packages command in an R session will do the trick.

```
install.packages("rJava")
```

1.4 Configure the Environment Variable CLASSPATH

The CLASSPATH environment variable must contain the directories with the jar and class files. The class files in this example will be created in /usr/lib/Java/java_1.8.0_77.

```
export CLASSPATH=$JAVA_HOME/jlib: /usr/lib/Java/java_1.8.0_77
```

2 rhdfs: Integrate R with HDFS

Setting Up Environment:

Before installing the package, we have to set the environment for Hadoop. We can execute the following command to set Hadoop Environment.

```
Sys.setenv(HADOOP_HOME="/usr/lib/hadoop")
Sys.setenv(HADOOP_CMD="/usr/lib/hadoop/bin/hadoop")
Sys.setenv(HADOOP_STREAMING="/usr/lib/hadoop/share/hadoop/tools/lib/hadoop-
streaming-2.7.2.jar")
```

Install rhdfs Package:

rhdfs release versions can be obtained from github.com. Assuming an internet connection is available, the install.packages command in an R session will do the trick.

```
> install.packages("rhdfs")
```

3 rmr2: Mapreduce job in R:

Setting Up Environment:

Before installing the package, we have to set the environment for Hadoop and Java. We can execute the following command to set Hadoop and Java Environment.

```
Sys.setenv("JAVA_HOME"="/usr/lib/java/1.8.0_77")
Sys.setenv(HADOOP_HOME="/usr/lib/hadoop")
Sys.setenv(HADOOP_CMD="/usr/lib/hadoop/bin/hadoop")
Sys.setenv(HADOOP_STREAMING="/usr/lib/hadoop/share/hadoop/tools/lib/hadoop-
streaming-2.7.2.jar")
```

Install rmr2 Package:

rmr2 release versions can be obtained from github.com. Assuming an internet connection is available, the install.packages command in an R session will do the trick.

```
  install.packages("rmr2 ")
```

4 plyrmr: Data Manipulation with mapreduce job

Setting Up Environment:

Before installing the package, we have to set the environment for Hadoop. We can execute the following command to set Hadoop Environment.

```
Sys.setenv(HADOOP_HOME="/usr/lib/hadoop")
Sys.setenv(HADOOP_CMD="/usr/lib/hadoop/bin/hadoop")
Sys.setenv(HADOOP_STREAMING="/usr/lib/hadoop/share/hadoop/tools/lib/hadoop-
streaming-2.7.2.jar")
```

Install plyrmr Package:

plyrmr release versions can be obtained from github.com. Assuming an internet connection is available, the install.packages command in an R session will do the trick.

```
> install.packages("plyrmr ")
```

5 rhbase: Integrate HBase with R

Setting Up Environment:

Before installing the package, we have to set the environment for Hadoop. We can execute the following command to set Hadoop Environment.

```
Sys.setenv(HADOOP_HOME="/usr/lib/hadoop")
Sys.setenv(HADOOP_CMD="/usr/lib/hadoop/bin/hadoop")
Sys.setenv(HADOOP_STREAMING="/usr/lib/hadoop/share/hadoop/tools/lib/hadoop-
streaming-2.7.2.jar")
```

Install rhbase Package:

rhbase release versions can be obtained from github.com. Assuming an internet connection is available, the install.packages command in an R session will do the trick.

```
  install.packages("rhbase ")
```