VISVESVARAYA TECHNOLOGICAL UNIVERSITY

JNANA SANGAMA, BELAGAVI-590018

A Project Report
on

"Weather Forecasting using Big Data and R"

Submitted in partial fulfillment of the requirements for the VIII Semester degree of
Bachelor of Engineering in Computer Science and Engineering
of Visvesvaraya Technological University, Belagavi

Submitted by:

Amruth Skanda Murthy V	**(1RN12CS011)**
Ramesh Sidaray Mudalagi	**(1RN12CS074)**
Sameer	**(1RN12CS085)**
Shashi Kiran S	**(1RN12CS094)**

Under the Guidance of

Dr. G T Raju
Dean,Prof and HOD
Dept of CSE

Department of Computer Science and Engineering
RNS Institute of Technology
Channasandra, Dr. Vishnuvardhan Road, Bengaluru-560 098

2015-2016

RNS Institute of Technology

Channasandra, Dr. Vishnuvardhan Road, Bengaluru-98

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CERTIFICATE

Certified that the Project on topic **"*Weather Forecasting using Big Data and R*"** has been successfully presented at **RNS Institute of Technology by Mr. Amruth Skanda Murthy V (1RN12CS011), Mr. Ramesh Sidaray Mudalagi (1RN12CS074), Mr. Sameer (1RN12CS085), Mr. Shashi Kiran S (1RN12CS094),** in partial fulfillment of the requirements for the *VIII Semester degree of* ***Bachelor of Engineering in Computer Science and Engineering*** *of Visvesvaraya Technological University, Belagavi* during academic year 2015-2016. It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the report deposited in the departmental library. The Project report has been approved as it satisfies the academic requirements in respect of Project work for the said degree.

Internal Guide:
Dr. G T Raju
Dean, Prof. and HOD
Dept of CSE

Dr. M K Venkatesha
Principal

External Viva:

Name of the Examiners	**Signature with Date**
1……..	
2……..	

ACKNOWLEDGMENT

The satisfaction and euphoria that accompany the successful completion of any task would be incomplete without the mention of the people who made it possible, whose constant guidance and encouragement crowned the efforts with success.

We would like to profoundly thank **Management** of **RNS Institute of Technology** for providing such a healthy environment for the successful completion of the Project work.

We would like to express our thanks to the Director **Dr. H N Shivashankar** and the Principal **Dr. M K Venkatesha** for their encouragement that motivated us to successfully complete the Project work.

It gives us immense pleasure to thank **Dr. G T Raju** Dean, Professor and Head of Department CSE, for his constant support and encouragement. Also for being the guide of our project and guiding us in executing our ideas.

We would also like to thank the Project Coordinator **Mr. Devaraju B M** Assistant Professor, Department of Computer Science & Engineering and all other teaching and non-teaching staff of Computer Science Department who has directly or indirectly helped us in the completion of the Project work.

Last, but not the least, we would hereby acknowledge and thank our parents and our friends who have been a source of inspiration and also instrumental in the successful completion of the Project work.

ABSTRACT

India is an emerging country. Now most of the cities have become smart city. Different sensors employed in smart city can be used to measure weather parameters. Weather forecast department has begun collect and analysis massive amount of data like temperature. They use different sensor values like temperature, humidity to predict the rain fall etc. When the number of sensors increases, the data becomes high volume and the sensor data have high velocity data. Thus, sensor data is a kind of Big Data. There is a need of a scalable analytics tool to process massive amount of data.

The traditional approach of process the data is very slow. We propose leveraging MapReduce with Hadoop to process the massive amount of data. Hadoop is an open source framework suitable for large scale data processing. MapReduce programming model helps to process large data sets in parallel, distributed manner. Processing the sensor data with MapReduce in Hadoop framework removes the scalability bottleneck. The speed of processing data can increase rapidly when across multi cluster distributed network. This project aims to build a data analytical engine for high velocity, huge volume temperature data from sensors using MapReduce on Hadoop for forecasting the weather.

CONTENTS

LIST OF FIGURES

FIG.NO.	TITLE	PAGE NO

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Introduction to Big Data

1.1.1 Exploding Data! :

We live in the data age. It's not easy to measure the total volume of data stored electronically, but an IDC estimate put the size of the "digital universe" at 0.18 zettabytes in 2006, and is forecasting a tenfold growth by 2011 to 1.8 zettabytes. A zettabyte is 10^{21} bytes, or equivalently one thousand exabytes, one million petabytes, or one billion terabytes. That's roughly the same order of magnitude as one disk drive for every person in the world. [1]

This flood of data is coming from many sources. Consider the following:

- The New York Stock Exchange generates about one terabyte of new trade data per day.
- Facebook hosts approximately 10 billion photos, taking up one petabyte of storage.
- Ancestry.com, the genealogy site, stores around 2.5 petabytes of data.
- The Internet Archive stores around 2 petabytes of data, and is growing at a rate of 20 terabytes per month.
- The Large Hadron Collider near Geneva, Switzerland, will produce about 15 petabytes of data per year. [2]

Figure 1.1 Data activity across globe in a minute

Data is forever. Think about it – it is indeed true. Are you using any application as it is which was built 10 years ago? Are you using any piece of hardware which was built 10 years ago? The answer is most certainly No. However, if I ask you – are you using any data which were captured 50 years ago, the answer is most certainly Yes. For example, look at the history of India. We have documented history which goes back as over 1000s of year. Data never gets old and it is going to stay there forever. Application which interprets and analysis data got changed but the data remained in its purest format in most cases.

As organizations have grown the data associated with them also grew exponentially and today there are lots of complexities to their data. Most of the big organizations have data in multiple applications and in different formats. The data is also spread out so much that it is hard to categorize with a single algorithm or logic. The mobile revolution which we are experimenting right now has completely changed how we capture the data and build intelligent systems. Big organizations are indeed facing challenges to keep all the data on a platform which give them a single consistent view of their data. This unique challenge to make sense of all the data coming in from different sources and deriving the useful actionable information out of is the revolution Big Data world is facing.

1.1.2 Defining Big Data

Figure 1.2 shows the 3Vs that define Big Data are *Variety*, *Velocity* and *Volume*. [3]

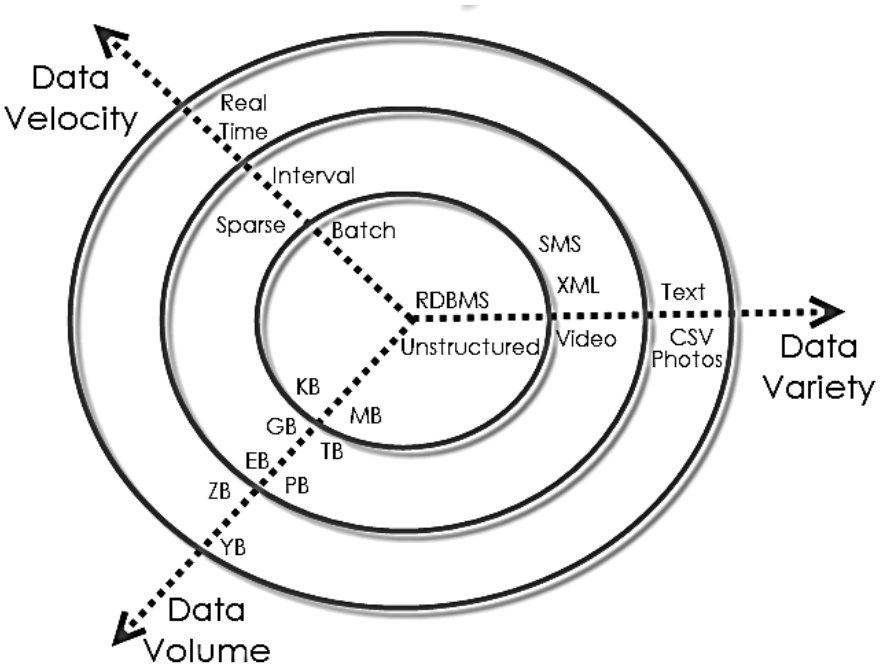


Figure 1.2 The 3 V's of Big Data

- **Volume**

We currently see the exponential growth in the data storage as the data is now more than text data. We can find data in the format of videos, music and large images on our social media channels. It is very common to have Terabytes and Petabytes of the storage system for enterprises. As the database grows the applications and architecture built to support the data needs to be re-evaluated quite often. Sometimes the same data is re-evaluated with multiple angles and even though the original data is the same the new found intelligence creates explosion of the data. The big volume indeed represents *Big Data*.

- **Variety**

Data can be stored in multiple format. For example database, excel, csv, access or for the matter of the fact, it can be stored in a simple text file. Sometimes the data is not even in the traditional format as we assume, it may be in the form of video, SMS, PDF or something we might have not thought about it. It is the need of the organization to arrange it and make it meaningful. It will be easy to do so if we have data in the same format, however it is not the case most of the time. The real world have data in many different formats and that is the challenge we need to overcome with the Big Data. This variety of the data represents *Big Data*.

- **Velocity**

The data growth and social media explosion have changed how we look at the data. There was a time when we used to believe that data of yesterday is recent. The matter of the fact newspapers is still following that logic. However, news channels and radios have changed how fast we receive the news. Today, people reply on social media to update them with the latest happening. On social media sometimes a few seconds old messages (a tweet, status updates etc.) is not something interests users. They often discard old messages and pay attention to recent updates. The data movement is now almost real time and the update window has reduced to fractions of the seconds. This high velocity data represent *Big Data*.

Further the data in it will be of three types:

- **Structured data**: Relational data.
- **Semi Structured data**: XML data.
- **Unstructured data**: Word, PDF, Text, Media Logs.

Big Data, in simple words, is not just about lots of data, it is actually a concept providing an opportunity to find new insight into your existing data as well guidelines to capture and analysis your future data. The real challenge with the big organization is to get maximum out of the data already available and predict what kind of data to collect in the future. How to take the existing data and make it meaningful that it provides us accurate insight in the past data is one of the key discussion points in many of the executive meetings in organizations. With the explosion of the data the challenge has gone to the next level and now a Big Data is becoming the reality in many organizations. It makes any business more agile and robust so it can adapt and overcome business challenges.

It has been said that "More data usually beats better algorithms" which is to say that for some problems (such as recommending movies or music based on past preferences), however fiendish your algorithms are, they can often be beaten simply by having more data (and a less sophisticated algorithm). [4]

1.2 Big Data Architecture

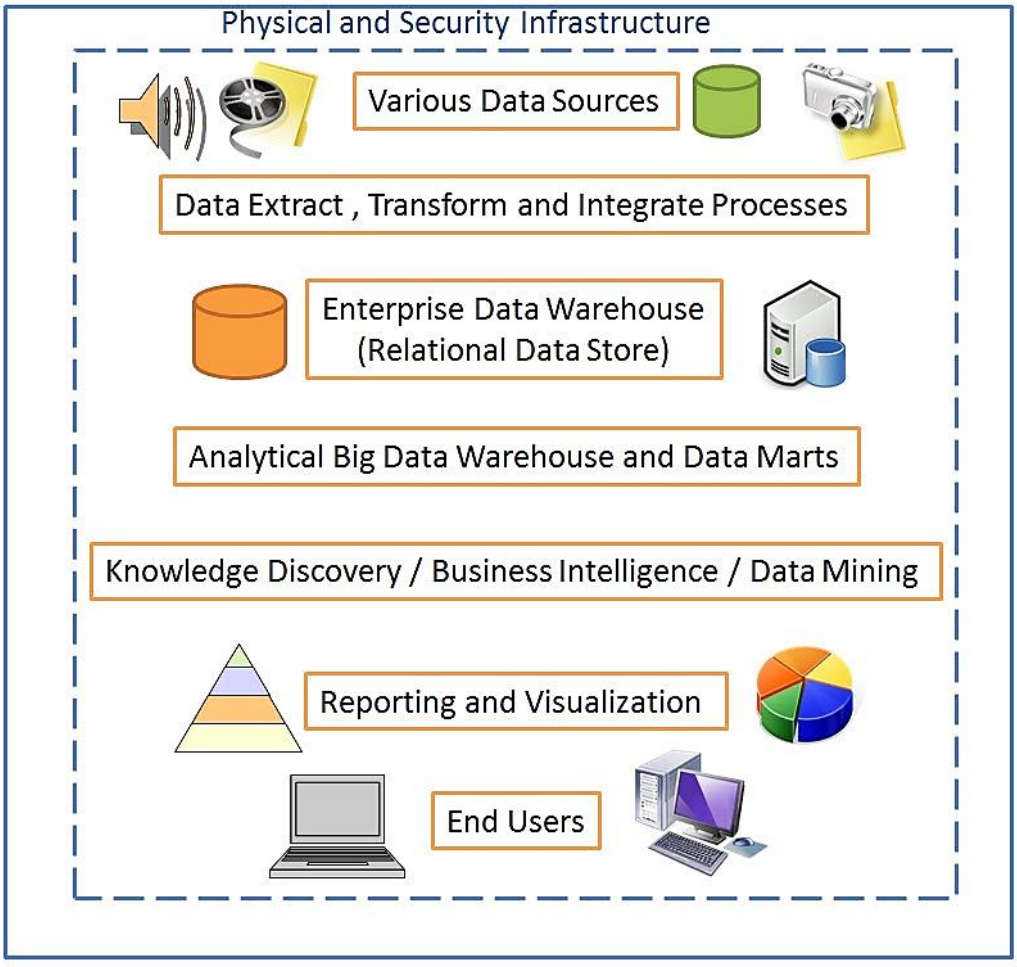


Figure 1.3 Big Data Architecture

Figure 1.3 gives good overview of how in Big Data Architecture various components are associated with each other. In Big Data various different data sources are part of the architecture hence extract, transform and integration are one of the most essential layers of the architecture. Most of the data is stored in relational as well as non-relational data marts and data warehousing solutions. As per the business need various data are processed as well converted to proper reports and visualizations for end users. Just like software the hardware is almost the most important part of the Big Data Architecture. In the big data architecture hardware infrastructure is extremely important and failure over instances as well as redundant physical infrastructure is usually implemented.

CHAPTER 2

HADOOP

2.1 Introduction to Hadoop

2.1.1 A Brief History of Hadoop

Hadoop was created by Doug Cutting, the creator of Apache Lucene, the widely used text search library. Hadoop has its origins in Apache Nutch, an open source web search engine, itself a part of the Lucene project.The name Hadoop is not an acronym; it's a made-up name. The project's creator, Doug Cutting, explains how the name came about:

"The name my kid gave a stuffed yellow elephant. Short, relatively easy to spell and pronounce, meaningless, and not used elsewhere: those are my naming criteria. Kids are good at generating such. Googol is a kid's term".

Subprojects in Hadoop also tend to have names that are unrelated to their function, often with an elephant or other animal theme ("Pig" for example). Smaller components are given more descriptive (and therefore more mundane) names. This is a good principle, as it means you can generally work out what something does from its name.

2.1.2 Problems with Data Storage and Analysis

The problem is simple: while the storage capacities of hard drives have increased massively over the years, access speeds−the rate at which data can be read from drives have not kept up. One typical drive from 1990 could store 1,370 MB of data and had a transfer speed of 4.4 MB/s, so you could read all the data from a full drive in around five minutes. Over 20 years later, one terabyte drives are the norm, but the transfer speed is around 100 MB/s, so it takes more than two and a half hours to read all the data off the disk. This is a long time to read all data on a single drive and writing is even slower. The obvious way to reduce the time is to read from multiple disks at once. Imagine if we had 100 drives, each holding one hundredth of the data.

There's more to being able to read and write data in parallel to or from multiple disks, though.

The first problem to solve is hardware failure: as soon as you start using many pieces of hardware, the chance that one will fail is fairly high. A common way of voiding data loss is through replication: redundant copies of the data are kept by the system so that in the event of failure, there is another copy available. This is how RAID works, for instance, although Hadoop's filesystem, the Hadoop Distributed Filesystem (HDFS), takes a slightly different approach, as you shall see later.

The second problem is that most analysis tasks need to be able to combine the data in some way; data read from one disk may need to be combined with the data from any of the other 99 disks. Various distributed systems allow data to be combined from multiple sources, but doing this correctly is notoriously challenging. MapReduce provides a programming model that abstracts the problem from disk reads and writes, transforming it into a computation over sets of keys and values. We will look at the details of this model in later chapters, but the important point for the present discussion is that there are two parts to the computation, the map and the reduce, and it's the interface between the two where the "mixing" occurs. Like HDFS, MapReduce has built-in reliability.

This, in a nutshell, is what Hadoop provides: a reliable shared storage and analysis system. The storage is provided by HDFS and analysis by MapReduce. There are other parts to Hadoop, but these capabilities are its kernel. [1]

2.2 How Hadoop Offer Better Big Data Solutions

2.2.1 Traditional Approach

In this approach, an enterprise will have a computer to store and process big data. Here data will be stored in an RDBMS like Oracle Database, MS SQL Server or DB2 and sophisticated softwares can be written to interact with the database, process the required data and present it to the users for analysis purpose.

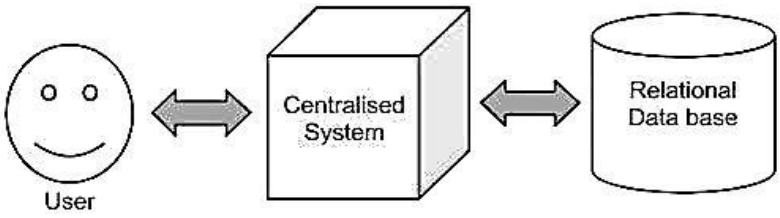


Figure 2.1 RDBMS based solution

Limitations

This approach works well where we have less volume of data that can be accommodated by standard database servers, or up to the limit of the processor which is processing the data. But when it comes to dealing with huge amounts of data, it is really a tedious task to process such data through a traditional database server.

2.2.2 Google's Solution

Google solved this problem using an algorithm called MapReduce. This algorithm divides the task into small parts and assigns those parts to many computers connected over the network, and collects the results to form the final result dataset. Figure 2.2 shows various commodity hardwares which could be single CPU machines or servers with higher capacity.

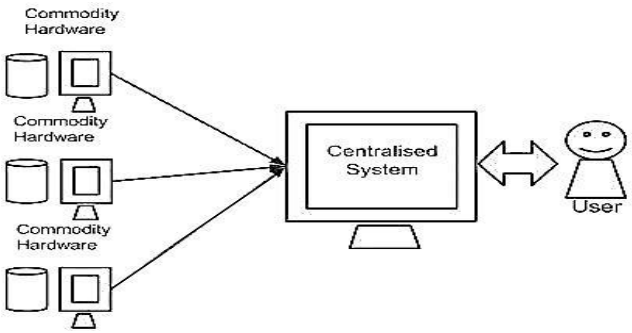


Figure 2.2 MapReduce based solution

2.2.3 Hadoop's Solution

Doug Cutting, Mike Cafarella and team took the solution provided by Google and started an Open Source Project called HADOOP in. Now Apache Hadoop is a registered trademark of the Apache Software Foundation.

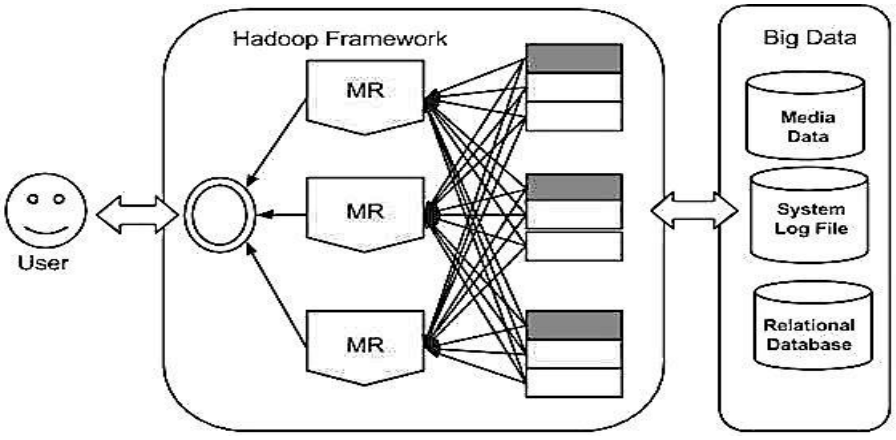


Figure 2.3 Hadoop based solution

Hadoop runs applications using the MapReduce algorithm, where the data is processed in parallel on different CPU nodes. In short, Hadoop framework is capabale enough to develop applications capable of running on clusters of computers and they could perform complete statistical analysis for a huge amounts of data. [5]

2.3 Hadoop Architecture

Hadoop framework includes following four modules as in figure 2.4:

- **Hadoop Common**: These are Java libraries and utilities required by other Hadoop modules. These libraries provide filesystem and OS level abstractions and contains the necessary Java files and scripts required to start Hadoop.
- **Hadoop YARN**: This is a framework for job scheduling and cluster resource management.
- **Hadoop Distributed File System (HDFS™)**: A distributed file system that provides high-throughput access to application data.
- **Hadoop MapReduce**: This is YARN-based system for parallel processing of large data sets. [5]

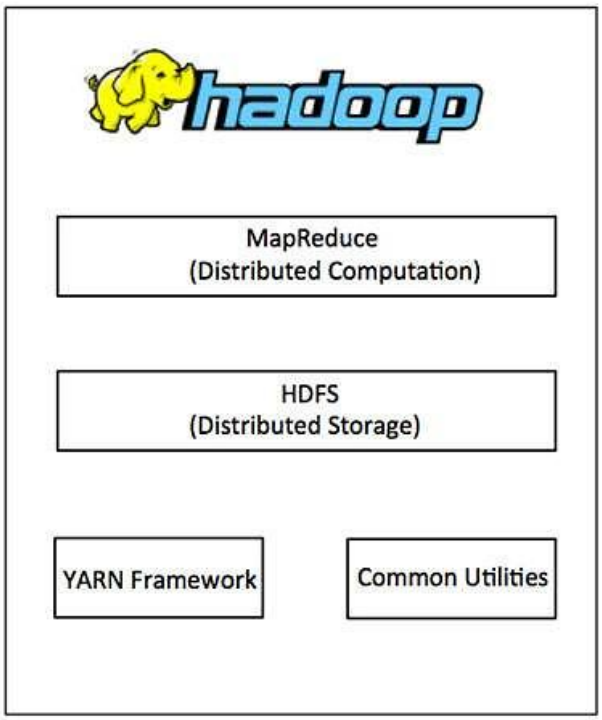


Figure 2.4 Hadoop Architecture with its 4 Modules

CHAPTER 3

MAPREDUCE

3.1 Introduction to MapReduce

3.1.1 Defining MapReduce

MapReduce is a programming model for data processing. The model is simple, yet not too simple to express useful programs in. Hadoop can run MapReduce programs written in various languages. Most important, MapReduce programs are inherently parallel, thus putting very large-scale data analysis into the hands of anyone with enough machines at their disposal.

Hadoop MapReduce is a software framework for easily writing applications which process big amounts of data in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner. [6]

The term MapReduce actually refers to the following two different tasks that Hadoop programs perform:

- **The Map Task**: This is the first task, which takes input data and converts it into a set of data, where individual elements are broken down into tuples (key/value pairs).
- **The Reduce Task**: This task takes the output from a map task as input and combines those data tuples into a smaller set of tuples. The reduce task is always performed after the map task.

Typically both the input and the output are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

The MapReduce framework consists of a single master JobTracker and one slave TaskTracker per cluster-node. The master is responsible for resource management, tracking resource consumption/availability and scheduling the jobs component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves TaskTracker execute the tasks as directed by the master and provide task-status information to the master periodically.

The JobTracker is a single point of failure for the Hadoop MapReduce service which means if JobTracker goes down, all running jobs are halted. [5]

3.2 Comparison between Traditional Approach and MapReduce

In many ways, MapReduce can be seen as a complement to an RDBMS. The differences between the two systems are shown in Table 2.1. MapReduce is a good fit for problems that need to analyze the whole dataset, in a *batch* fashion, particularly for ad hoc analysis. An RDBMS is good for point queries or updates, where the dataset has been indexed to deliver low-latency retrieval and update times of a relatively small amount of data. MapReduce suits applications where the data is written once, and read many times, whereas a relational database is good for datasets that are continually updated.

Another difference between MapReduce and an RDBMS is the amount of *structure* in the datasets that they operate on. Structured data is data that is organized into entities that have a defined format, such as XML documents or database tables that conform to a particular predefined schema. This is the realm of the RDBMS. Semi-structured data, on the other hand, is looser, and though there may be a schema, it is often ignored, so it may be used only as a guide to the structure of the data: for example, a spreadsheet, in which the structure is the grid of cells, although the cells themselves may hold any form of data. Unstructured data does not have any particular internal structure: for example, plain text or image data. MapReduce works well on *unstructured* or *semistructured* data, since it is designed to interpret the data at processing time. In other words, the input keys and values for MapReduce are not an intrinsic property of the data, but they are chosen by the person analysing the data.

	Traditional RDBMS	**MapReduce**
Data size	Gigabytes	Petabytes
Access	Interactive and batch	Batch
Updates	Read and write many times	Write once, read many times
Structure	Static schema	Dynamic schema
Integrity	High	Low
T **Scaling**	Nonlinear	Linear

Table 2.1 RDBMS compared to MapReduce

Relational data is often *normalized* to retain its integrity and remove redundancy. Normalization poses problems for MapReduce, since it makes reading a record a nonlocal operation, and one of the central assumptions that MapReduce makes is that it is possible to perform (high-speed) streaming reads and writes.

MapReduce is a *linearly scalable* programming model. The programmer writes two functions, a map function and a reduce function, each of which defines a mapping from one set of key-value pairs to another. These functions are oblivious to the size of the data or the cluster that they are operating on, so they can be used unchanged for a small dataset and for a massive one. More important, if you double the size of the input data, a job will run twice as slow. But if you also double the size of the cluster, a job will run as fast as the original one. This is not generally true of SQL queries. [1]

3.3 Data Flow

3.3.1 Terminologies

- **job:** A unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration information.
- **task:** Hadoop runs the job by dividing it into tasks, of which there are two types: *map* tasks and *reduce* tasks**.**
- **jobtracker:** Coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers.
- **tasktracker:** Run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job. If a task fails, the jobtracker can reschedule it on a different tasktracker.
- **Input splits:** Hadoop divides the input to a MapReduce job into fixed-size pieces called input splits, or just splits. Hadoop creates one map task for each split, which runs the userdefined map function for each *record* in the split. A good split size tends to be the size of an HDFS block, 64 MB by default, although this can be changed for the cluster.
- **Record:** A single tuple or line from the input split.
- **Data local :** Map task operated on HDFS block both residing on same nodes of a rack.
- **Rack local map task:** Map task operated on HDFS block both residing in different nodes of same rack.
- **Off-rack map task:** Map task operated on HDFS block which is residing in different rack other than the one map task resides in. [1]

3.3.2 Data Flow Diagram

Map tasks write their output to the local disk, not to HDFS. Why is this? Map output is intermediate output: it's processed by reduce tasks to produce the final output, and once the job is complete the map output can be thrown away. So storing it in HDFS, with replication, would be overkill. If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to re-create the map output. Reduce tasks don't have the advantage of data locality, the input to a single reduce task is normally the output from all mappers.

In the present example, we have a single reduce task that is fed by all of the map tasks. Therefore, the sorted map outputs have to be transferred across the network to the node where the reduce task is running, where they are merged and then passed to the user-defined reduce function. The output of the reduce is normally stored in HDFS for reliability. As explained in Chapter 3, for each HDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes. Thus, writing the reduce output does consume network bandwidth, but only as much as a normal HDFS write pipeline consumes.

The whole data flow with a single reduce task is illustrated in Figure 3.1. The dotted boxes indicate nodes, the light arrows show data transfers on a node, and the heavy arrows show data transfers between nodes. [1]

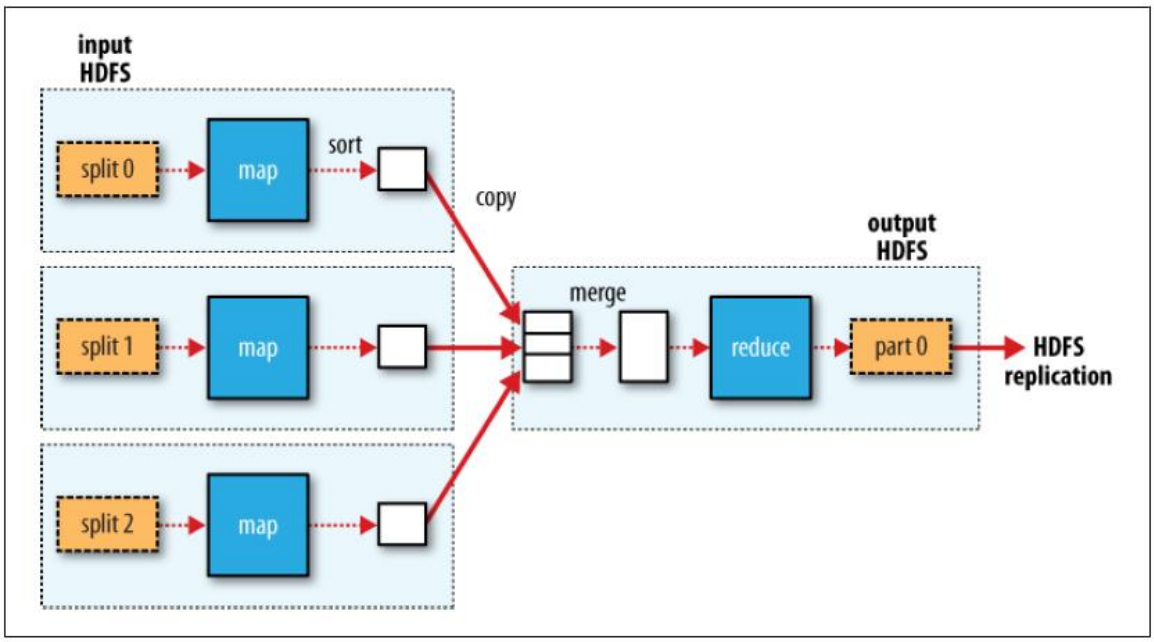


Figure 3.1 MapReduce data flow with a single reduce task

When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for any given key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner, which buckets keys using a hash function, works very well.

The data flow for the general case of multiple reduce tasks is illustrated in Figure 3.2. This diagram makes it clear why the data flow between map and reduce tasks is colloquially known as "the shuffle" as each reduce task is fed by many map tasks. The process by which the system performs the sort and transfers the map outputs to the reducers as inputs is known as the *shuffle.* The shuffle is more complicated than this diagram suggests, and tuning it can have a big impact on job execution time. In many ways, the shuffle is the heart of MapReduce and is where the "magic" happens.

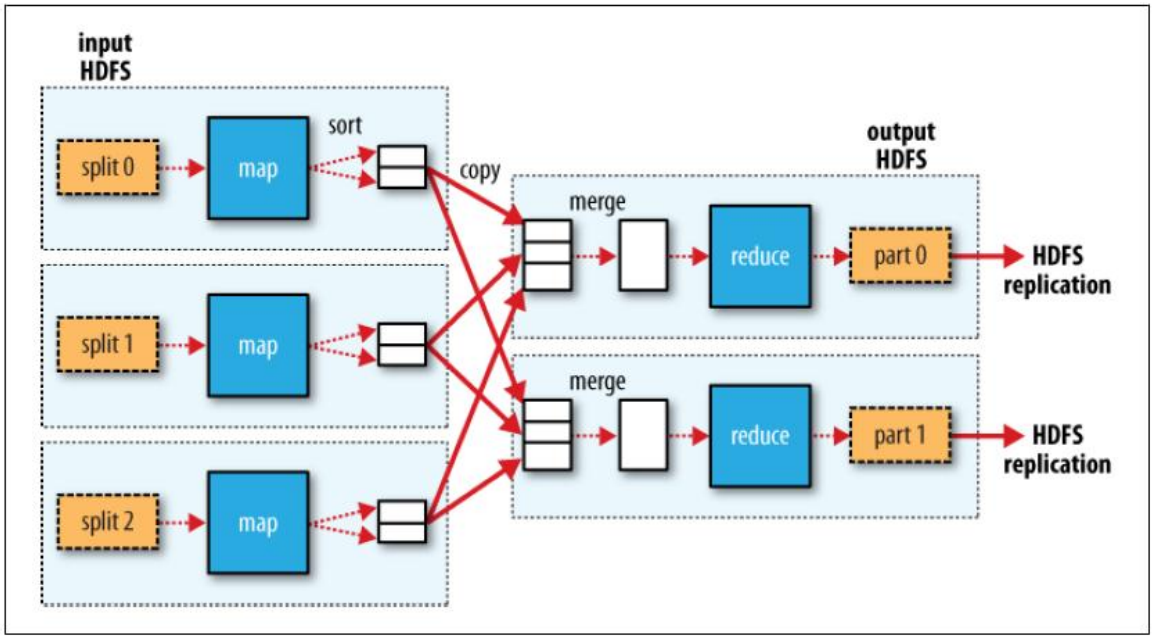


Figure 3.2 MapReduce data flow with multiple reduce tasks

3.4 The MapReduce Web UI

Hadoop comes with a web UI for viewing information about your jobs. It is useful for following a job's progress while it is running, as well as finding job statistics and logs after the job has completed. You can find the UI at http://jobtracker-host:50030/

3.4.1 The jobtracker page

A screenshot of the home page is shown in Figure 3.3. The first section of the page gives details of the Hadoop installation, such as the version number and when it was compiled, and the current state of the jobtracker (in this case, running), and when it was started.

Next is a summary of the cluster, which has measures of cluster capacity and utilization. This shows the number of maps and reduces currently running on the cluster, the total number of job submissions, the number of tasktracker nodes currently available, and the cluster's capacity: in terms of the number of map and reduce slots available across the cluster ("Map Task Capacity" and "Reduce Task Capacity"), and the number of available slots per node, on average. The number of tasktrackers that have been blacklisted by the jobtracker is listed as well. Below the summary, there is a section about the job scheduler that is running (here the default). You can click through to see job queues.

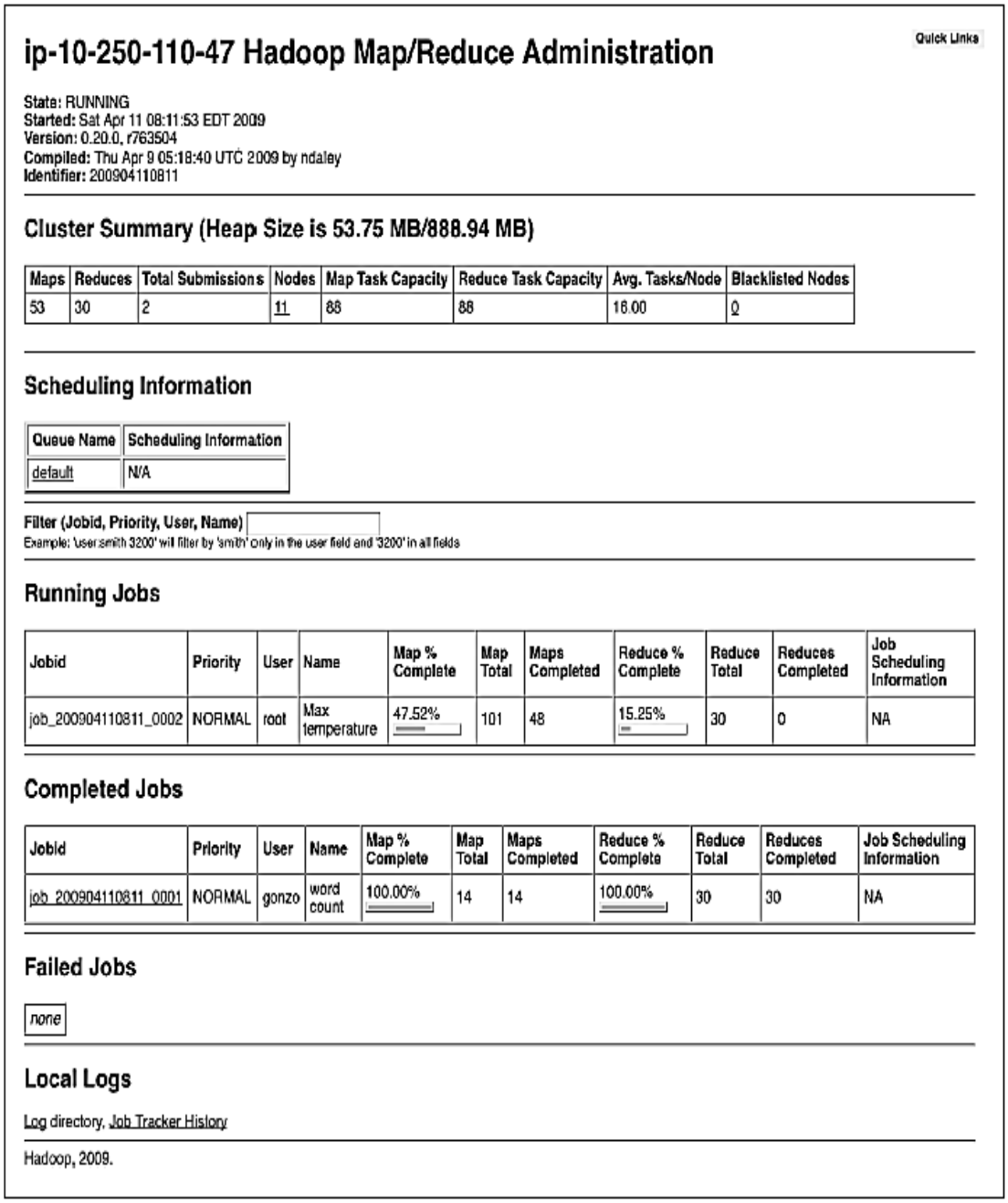
Further down, we see sections for running, (successfully) completed, and failed jobs. Each of these sections has a table of jobs, with a row per job that shows the job's ID, owner, name (as set in the Job constructor or **setJobName()** method, both of which internally set the **mapred.job.name** property) and progress information. Finally, at the foot of the page, there are links to the jobtracker's logs, and the jobtracker's history: information on all the jobs that the jobtracker has run. The main view displays only 100 jobs (configurable via the **mapred.jobtracker.completeuserjobs.maximum** property), before consigning them to the history page. Note also that the job history is persistent, so you can find jobs here from previous runs of the jobtracker.

3.4.2 Job History

Job history refers to the events and configuration for a completed job. It is retained whether the job was successful or not, in an attempt to provide interesting information for the user running a job.

Job history files are stored on the local filesystem of the jobtracker in a *history* subdirectory of the *logs* directory. It is possible to set the location to an arbitrary Hadoop filesystem via the **hadoop.job.history.location** property. The jobtracker's history files are kept for 30 days before being deleted by the system.

A second copy is also stored for the user in the _logs/history subdirectory of the job's output directory. This location may be overridden by setting **hadoop.job.history.user.location**. By setting it to the special value none, no user job history is saved, although job history is still saved centrally. A user's job history files are never deleted by the system. The history log includes job, task, and attempt events, all of which are stored in a plaintext file. The history for a particular job may be viewed through the web UI, or via the command line, using hadoop job -history (which you point at the job's output directory).



ip-10-250-110-47 Hadoop Map/Reduce Administration

Quick Links

State: RUNNING
Started: Sat Apr 11 08:11:53 EDT 2009
Version: 0.20.0, r763504
Compiled: Thu Apr 9 05:18:40 UTC 2009 by ndaley
Identifier: 200904110811

Cluster Summary (Heap Size is 53.75 MB/888.94 MB)

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes
53	30	2	11	88	88	16.00	0

Scheduling Information

Queue Name	Scheduling Information
default	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0002	NORMAL	root	Max temperature	47.52%	101	48	15.25%	30	0	NA

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0001	NORMAL	gonzo	word count	100.00%	14	14	100.00%	30	30	NA

Failed Jobs

none

Local Logs

Log directory, Job Tracker History

Hadoop, 2009.

Figure 3.3 Screenshot of the jobtracker page

3.4.3 Job Page

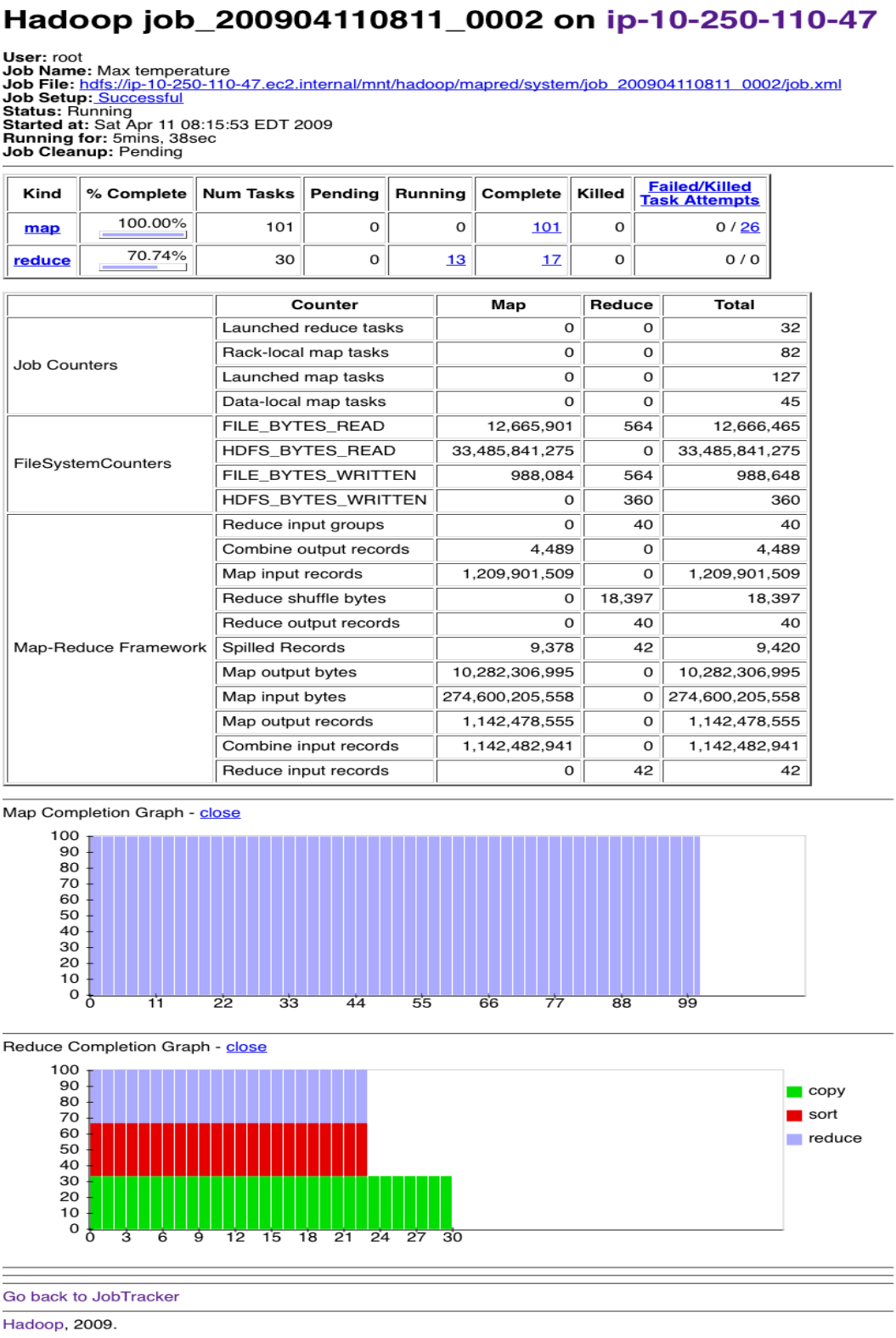
Clicking on a job ID brings you to a page for the job, illustrated in Figure 3.4. At the top of the page is a summary of the job, with basic information such as job owner and name, and how long the job has been running for. The job file is the consolidated configuration file for the job, containing all the properties and their values that were in effect during the job run. If you are unsure of what a particular property was set to, you can click through to inspect the file.

While the job is running, you can monitor its progress on this page, which periodically updates itself. Below the summary is a table that shows the map progress and the reduce progress. "Num Tasks" shows the total number of map and reduce tasks for this job (a row for each). The other columns then show the state of these tasks: "Pending" (waiting to run), "Running," "Complete" (successfully run), "Killed" (tasks that have failed this column would be more accurately labeled "Failed").

The final column shows the total number of failed and killed task attempts for all the map or reduce tasks for the job (task attempts may be marked as killed if they are a speculative execution duplicate, if the tasktracker they are running on dies or if they are killed by a user).

Further down the page, you can find completion graphs for each task that show their progress graphically. The reduce completion graph is divided into the three phases of the reduce task: copy (when the map outputs are being transferred to the reduce's tasktracker), sort (when the reduce inputs are being merged), and reduce (when thereduce function is being run to produce the final output).

In the middle of the page is a table of job counters. These are dynamically updated during the job run, and provide another useful window into the job's progress and general health.



Hadoop job_200904110811_0002 on ip-10-250-110-47

User: root
Job Name: Max temperature
Job File: hdfs://ip-10-250-110-47.ec2.internal/mnt/hadoop/mapred/system/job_200904110811_0002/job.xml
Job Setup: Successful
Status: Running
Started at: Sat Apr 11 08:15:53 EDT 2009
Running for: 5mins, 38sec
Job Cleanup: Pending

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	101	0	0	101	0	0 / 26
reduce	70.74%	30	0	13	17	0	0 / 0

	Counter	Map	Reduce	Total
Job Counters	Launched reduce tasks	0	0	32
	Rack-local map tasks	0	0	82
	Launched map tasks	0	0	127
	Data-local map tasks	0	0	45
FileSystemCounters	FILE_BYTES_READ	12,665,901	564	12,666,465
	HDFS_BYTES_READ	33,485,841,275	0	33,485,841,275
	FILE_BYTES_WRITTEN	988,084	564	988,648
	HDFS_BYTES_WRITTEN	0	360	360
Map-Reduce Framework	Reduce input groups	0	40	40
	Combine output records	4,489	0	4,489
	Map input records	1,209,901,509	0	1,209,901,509
	Reduce shuffle bytes	0	18,397	18,397
	Reduce output records	0	40	40
	Spilled Records	9,378	42	9,420
	Map output bytes	10,282,306,995	0	10,282,306,995
	Map input bytes	274,600,205,558	0	274,600,205,558
	Map output records	1,142,478,555	0	1,142,478,555
	Combine input records	1,142,482,941	0	1,142,482,941
	Reduce input records	0	42	42

Map Completion Graph - close

Reduce Completion Graph - close

Go back to JobTracker

Hadoop, 2009.

Figure 3.4 Screenshot of the job page

CHAPTER 4

HADOOP DISTRIBUTED FILE SYSTEM (HDFS)

4.1 Introduction to HDFS

4.1.1 Defining HDFS

When a dataset outgrows the storage capacity of a single physical machine, it becomes necessary to partition it across a number of separate machines. Filesystems that manage the storage across a network of machines are called *distributed filesystems*. Since they are network-based, all the complications of network programming kick in, thus making distributed filesystems more complex than regular disk filesystems. For example, one of the biggest challenges is making the filesystem tolerate node failure without suffering data loss. [1]

Hadoop comes with a distributed filesystem called HDFS, which stands for *Hadoop Distributed filesystem*. (You may sometimes see references to "DFS", informally or in older documentation or configurations, which is the same thing.) HDFS is Hadoop's flagship filesystem and is the focus of this chapter, but Hadoop actually has a general purpose filesystem abstraction, so we'll see along the way how Hadoop integrates with other storage systems (such as the local filesystem and Amazon S3).

Hadoop File System was developed using distributed file system design. It is run on commodity hardware. Unlike other distributed systems, HDFS is highly fault-tolerant and designed using low-cost hardware. HDFS holds very large amount of data and provides easier access. To store such huge data, the files are stored across multiple machines. These files are stored in redundant(replication) fashion to rescue the system from possible data losses in case of failure. HDFS also makes applications available to parallel processing.

4.1.2 Features of HDFS

- It is suitable for the distributed storage and processing.
- Hadoop provides a command interface to interact with HDFS.
- The built-in servers of namenode and datanode help users to easily check the status of cluster.
- Streaming access to file system data.
- HDFS provides file permissions and authentication.

4.2 HDFS Architecture

Figure 4.1 is the architecture of a Hadoop File System.

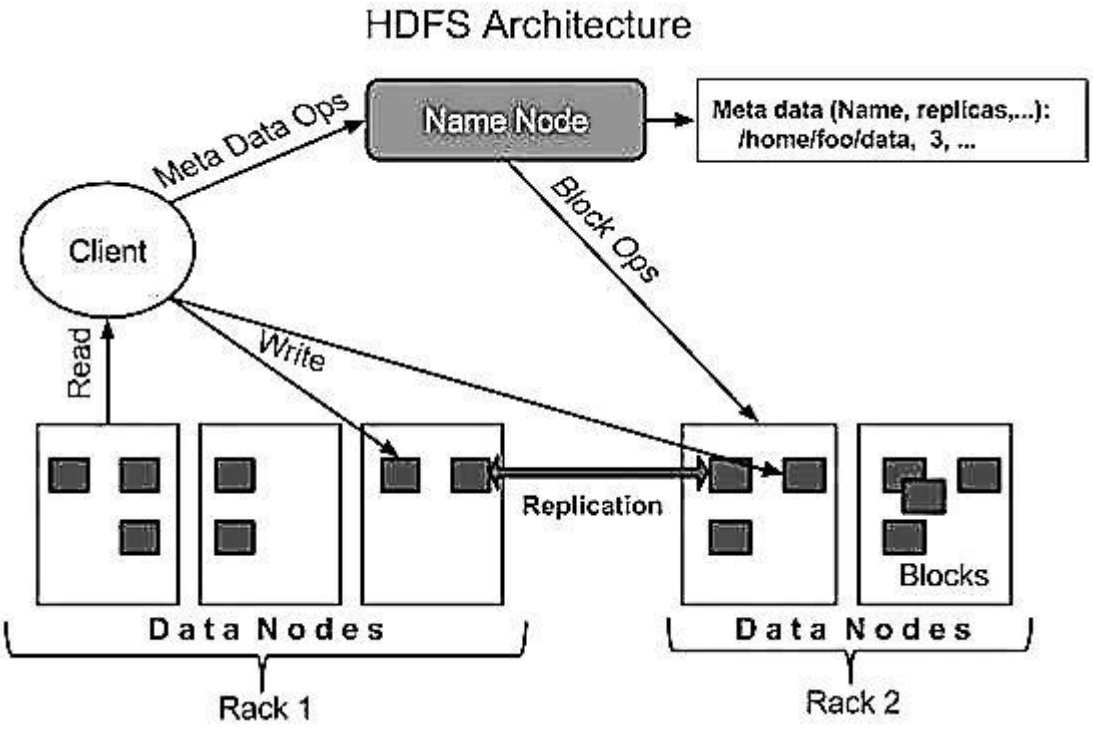


Figure 4.1 HDFS Architecture

HDFS follows the master-slave architecture and it has the following elements.

4.2.1 Blocks

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

HDFS, too, has the concept of a block, but it is a much larger unit-64 MB by default. Like in a filesystem for a single disk, files in HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage. When unqualified, the term "block" in this book refers to a block in HDFS.

Why Is a Block in HDFS So Large?

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 made to be significantly larger 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, then to make the seek time 1% of the transfer time, we need to make the block size around 100 MB. The default is actually 64 MB, although many HDFS installations use 128 MB blocks. 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 MapReduce normally operate on one block at a time, so if you have too few tasks (fewer than nodes in the cluster), your jobs will run slower than they could otherwise.

Advantages of Blocks

Having a block abstraction for a distributed filesystem brings several benefits. The first benefit is the most obvious: a *file can be larger* than any single disk in the network. There's nothing that requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks in the cluster. In fact, it would be possible, if unusual, to store a single file on an HDFS cluster whose blocks filled all the disks in the cluster.

Second, making the *unit of abstraction* a block rather than a file simplifies the storage subsystem. Simplicity is something to strive for all in all systems, but is especially important for a distributed system in which the failure modes are so varied. The storage subsystem deals with blocks, simplifying storage management (since blocks are a fixed size, it is easy to calculate how many can be stored on a given disk) and eliminating metadata concerns (blocks are just a chunk of data to be stored—file metadata such as permissions information does not need to be stored with the blocks, so another system can handle metadata separately).

Furthermore, blocks fit well with replication for providing *fault tolerance* and *availability*. To insure against corrupted blocks and disk and machine failure, each block is replicated to a small number of physically separate machines (typically three). If a block becomes unavailable, a copy can be read from another location in a way that is transparent to the client.

A block that is no longer available due to corruption or machine failure can be replicated from its alternative locations to other live machines to bring the replication factor back to the normal level. Similarly, some applications may choose to set a high replication factor for the blocks in a popular file to spread the read load on the cluster.

4.2.2 Namenode

The namenode manages the filesystem namespace. It maintains the filesystem 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 namenode also knows the datanodes on which all the blocks for a given file are located, however, it does not store block locations persistently, since this information is reconstructed from datanodes when the system starts.

The namenode is the commodity hardware that contains the GNU/Linux operating system and the namenode software. It is a software that can be run on commodity hardware. The system having the namenode acts as the master server and it does the following tasks:

- Manages the file system namespace.
- Regulates client's access to files.
- It also executes file system operations such as renaming, closing, and opening files and directories.

Without the namenode, the filesystem cannot be used. In fact, if the machine running the namenode were obliterated, all the files on the filesystem would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the datanodes. For this reason, it is important to make the namenode 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 filesystem metadata. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. These writes are synchronous and atomic. The usual configuration choice is to write to local disk as well as a remote *NFS mount*.

It is also possible to run a *secondary namenode*, which despite its name does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. The secondary namenode usually runs on a separate physical machine, since it requires plenty of CPU and as much memory as the

namenode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing. However, the state of the secondary namenode lags that of the primary, so in the event of total failure of the primary, data loss is almost certain. The usual course of action in this case is to copy the namenode's metadata files that are on NFS to the secondary and run it as the new primary.

4.2.3 Datanode

Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that they are storing.

The datanode is a commodity hardware having the GNU/Linux operating system and datanode software. For every node (Commodity hardware/System) in a cluster, there will be a datanode. These nodes manage the data storage of their system:

- Datanodes perform read-write operations on the file systems, as per client request.
- They also perform operations such as block creation, deletion, and replication according to the instructions of the namenode.

CHAPTER 5

R

5.1 Introduction to R

R is a language and environment for statistical computing and graphics. It is a GNU project which is similar to the S language and environment which was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues. R can be considered as a different implementation of S. There are some important differences, but much code written for S runs unaltered under R.

R provides a wide variety of statistical (linear and nonlinear modelling, 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.

R is available as Free Software under the terms of the Free Software Foundation's GNU General Public License in source code form. It compiles and runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and MacOS.[9]

5.2 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.

CHAPTER 6

PROBLEM STATEMENT

6.1 Weather Forecasting so far

Weather forecasting is the application of science and technology to predict the state of the atmosphere for a given location. Human beings have attempted to predict the weather informally for millennia, and formally since the nineteenth century. Weather forecasts are made by collecting quantitative data about the current state of the atmosphere at a given place and using scientific understanding of atmospheric processes to project how the atmosphere will change. [8]

Ancient weather forecasting methods usually relied on observed patterns of events, also termed pattern recognition. For example, it might be observed that if the sunset was particularly red, the following day often brought fair weather. This experience accumulated over the generations to produce weather lore. However, not all of these predictions prove reliable, and many of them have since been found not to stand up to rigorous statistical testing.

It was not until the 20th century that advances in the understanding of atmospheric physics led to the foundation of modern numerical weather prediction. The first computerised weather forecast was performed by a team led by the mathematician John von Neumann; von Neumann publishing the paper *Numerical Integration of the Barotropic Vorticity Equation in 1950*. Practical use of numerical weather prediction began in 1955, spurred by the development of programmable electronic computers.

To make an accurate forecast, we must first understand what processes are occurring in the atmosphere to produce the current weather at the location for which the we are forecasting. This is done by measuring certain elements (making observations) of the atmosphere; i.e., temperature, pressure, wind direction and speed, humidity, cloud cover, precipitation, etc. The more complete measurement coverage across the earth's surface and vertically through the atmosphere of the elements which affect the weather we experience, the better picture we have of the processes producing the weather we are currently experiencing. By observing the changes which take place to these elements over time and comparing the changing patterns with historical patterns, an understanding of expected weather conditions can be made.

If we can understand how the atmosphere changes over time in response to various factors; i.e., differences in warming across the earth's surface from solar radiation, radiational cooling at night, warming of the atmosphere due to latent heat release during condensation, etc., and can write mathematical equations to express these changes, then a useful tool becomes available to the forecaster - computer models - which can be constructed to express how the atmosphere is changing and will appear at some future time. The output from these models can be used as an aid to forecasters in preparing the forecasts.

Major research and development effort is ongoing in improving all areas of the process, from development of better observational techniques (both surface systems, upper air systems, and satellite systems), development of forecasting techniques to be used by forecasters, to development of better mathematical equations and computer models, to procedures to communicate weather information to users in a timely and reliable manner.

6.2 Our Solution

India is an emerging country. Now most of the cities have become smart. Different sensors employed in smart city can be used to measure weather parameters. Weather forecast department has begun collect and analysis massive amount of data like temperature. They use different sensor values like temperature, humidity to predict the rain fall etc. When the number of sensors increases, the data becomes high volume and the sensor data have high velocity data. Thus, The sensor data is a kind of Bigdata. There is a need of a scalable analytics tool to process massive amount of data.

The traditional approach of process the data is very slow. We propose leveraging MapReduce with Hadoop to process the massive amount of data. Hadoop is an open source framework suitable for large scale data processing. MapReduce programming model helps to process large data sets in parallel, distributed manner. Processing the sensor data with MapReduce in Hadoop framework removes the scalability bottleneck. The speed of processing data can increase rapidly when across multi cluster distributed network. This project aims to build a data analytical engine for high velocity, huge volume temperature data from sensors using MapReduce on Hadoop for forecasting the weather.

CHAPTER 7

IMPLEMENTATION

In this project we have implemented our solution using 3 different algorithms, collected results and compared them to decide on better and efficient algorithm. They are :

- Normals method or mean calculation in MapReduce on Hadoop [7]
- Linear Regression Algorithm in MapReduce on Hadoop
- Sliding Window Algorithm in R

7.1 NCDC Input Data

National Climatic Data Center (NCDC) have provide weather datasets. Daily Global Weather Measurements 1929-2009 (NCDC, GSOD) dataset is one of the biggest dataset available for weather forecast. Its total size is around 20 GB. It is available on amazon web services. The United States National Climatic Data Center (NCDC), previously known as the National Weather Records Center (NWRC), in Asheville, North Carolina is the world's largest active archive of weather data. The Center has more than 150 years of data on hand with 224 gigabytes of new information added each day. NCDC archives 99 percent of all NOAA data, including over 320 million paper records; 2.5 million microfiche records; over 1.2 petabytes of digital data residing in a mass storage environment. NCDC has satellite weather images back to 1960.

Proposed System used the temperature dataset of NCDC, GHCN (Global Historical Climatology Network)-Daily. It is an integrated database of daily climate summaries from land surface stations across the globe. This input data is split to records and the records are stored in HDFS. They are split and each split goes to unique mappers. Finally all results go to reducer. MapReduce Framework execution is shown in Chapter Snapshots. The results shows that adding more number of systems to the network will speed up the entire data processing. That is the major advantage of MapReduce with Hadoop framework.

7.2 The Normals Method (Mean)

The input weather dataset contain the values of temperature, time, place etc. The input weather dataset file on the left of Figure 3 is split into chunks of data. The size of these splits is controlled by the InputSplit method within the FileInputFormat class of the Map Reduce job. The number of splits is influenced by the HDFS block size, and one mapper job is created for each split data chunk. Each split data chunk that is, "a record," is sent to a Mapper process on a Data Node server.

In the proposed method, the Map process creates a series of key-value pairs where the key is the Plain Old Java Object(POJO), comprising of data fields like place, date,etc., and the value is the temperature. These key-value pairs are then shuffled into lists by key type. The shuffled lists are input to Reduce tasks, which reduce the dataset volume by the values. The Reduce output is then a simple list of averaged key-value pairs. The Map Reduce framework takes care of all other tasks, like scheduling and resources.

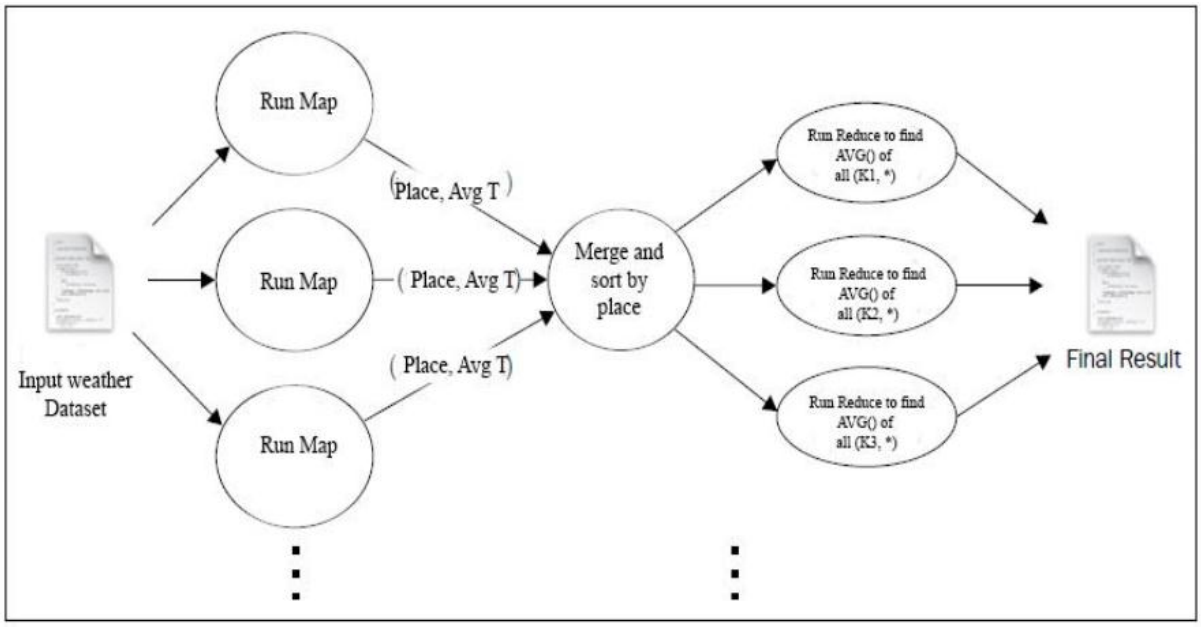


Figure 7.1 Proposed MapReduce Framework for Normals method

7.2.1 Driver Operation

The driver is what actually sets up the job, submits it, and waits for completion. The driver is driven from a configuration file for ease of use for things like being able to specify the input/output directories. It can also accept Groovy script based mappers and reducers without recompilation.

7.2.2 Mapper Operation

The actual mapping routine acts as filter so that only records that matched certain criteria would be sent to the reducer. The input to map routine is directly fed from NCDC dataset. First selection of the weather parameter (tmax, tmin, etc.,) from input data upon which prediction is to be made. The filtered records are then sent for *parsing* which does the following tasks. The input consists of 31-32 columns. The first column has concatenated set of data such as stationId (length=11), followed by year and month (in 'yyyymm' format) and then parameter name (tmax, tmin, prcp). The subsequent columns in each record hold the values corresponding to parameter for every day of that month. It also includes certain data flags(S,I,E,etc) which are filtered out as well. Example of one record as shown below:

AG000060590200501TMAX 200 E 180 E 175 E 182 E 179 E 190 E 205 E 180 E 190 E 186 E 234 E 264 E 214 E 178 E 150 E 170 E 168 E 146 E 150 E 158 E 190 E 172 E 184 E 162 E 180 E 170 E 164 E 176 E 180 E 138 E 150 E

The temperature values here are expressed in multiples of 10. So we need to take $1/10^{th}$ of the actual value mentioned in records. The parser returns the essential formatted input data that is then used to initialise a POJO, a plain old java object, which acts as *key* type as well as *value* type in our implementation, "<WFDataPojo key, WFDataPojo value>". This Object holds all the data such as stationId, date, temperature max value, temperature min value, etc., This <key,value> pair is what forwarded to reducer.

7.2.3 Reducer Operation

With the sequencing and mapping complete, the resulting <key, value> pairs that matched the criteria to be analysed were forwarded to the reducer. While the actual simple averaging operation was straightforward and relatively simple to set up, the reducer turned out to be more complex than expected. Once a <key, value> object has been created, a comparator is also needed to order the keys. If the data is to be grouped, a group comparator is also needed. In addition, a partitioner must be created in order to handle the partitioning of the data into groups of sorted keys. With all these components in place, Hadoop takes the <key, value> pairs generated by the mappers and groups and sorts them as specified as shown in figure 7.2. By default, Hadoop assumes that all values that share a key will be sent to the same reducer. Hence, a single operation over a very large data set will only employ one reducer, i.e., one node. By using partitions, sets of keys to group can be created to pass these grouped keys to different reducers and parallelize the reduction operation. This may result in multiple output

files so that an additional combination step may be needed to handle the consolidation of all results.

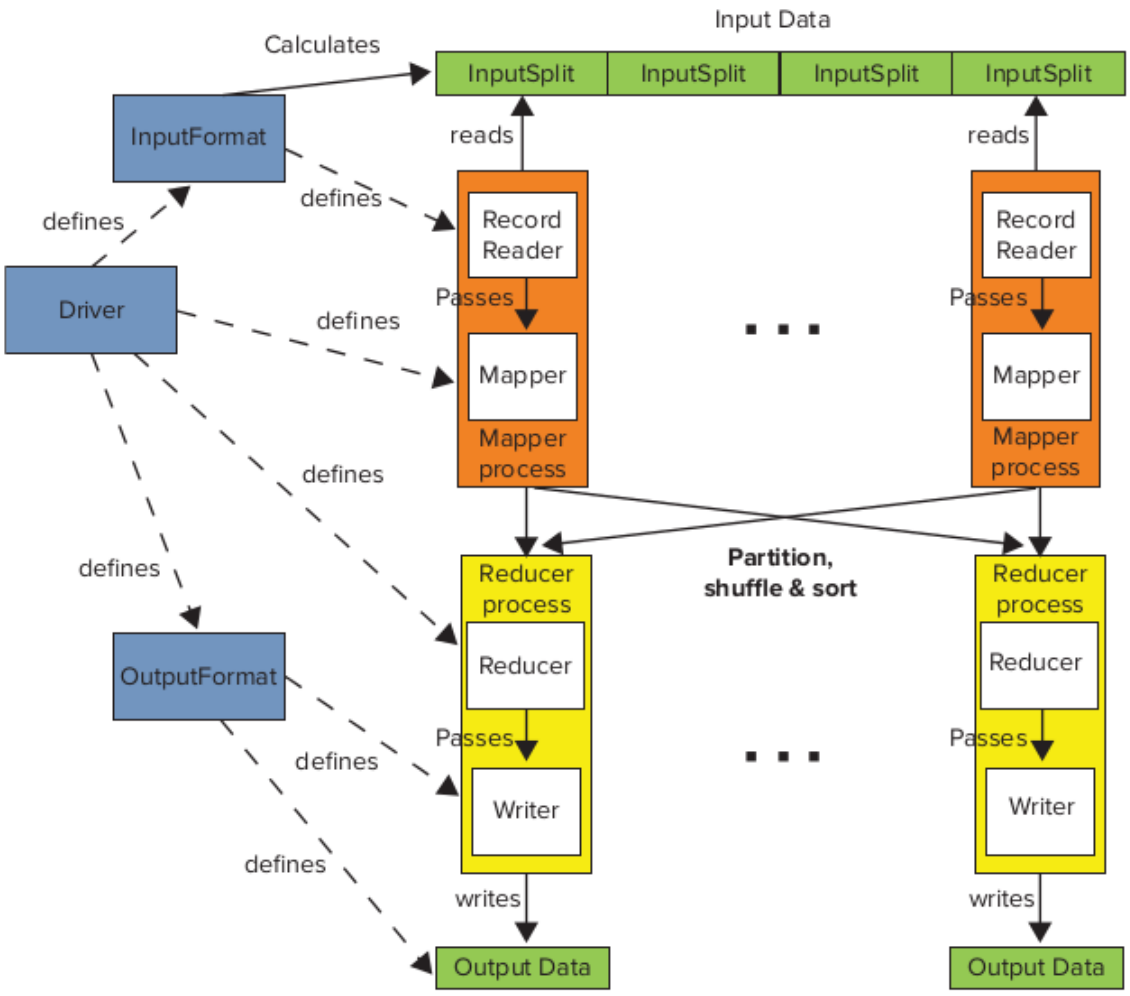


Figure 7.2 Splits, Partioner, Shuffle & Sort,etc., Inside MapReduce

In brief,

- The NCDC input files were ingested into the Hadoop file system with the default replica factor of three and, initially, the default block size of 64 MB.
- The job containing the actual MapReduce operation was submitted to the Head Node to be run.
- Along with the JobTracker, the Head Node schedules and runs the job on the cluster.
- Hadoop distributes all the mappers across all data nodes that contain the data to be analyzed.
- On each data node, the input format reader opens up each file for reading and parses all data into meaning full form for the mapping function.

- The mapper saves the <key, value> pair for delivery back to the reducer.
- All keys and values within a file are read and analyzed by the mapper. Once the mapper is done, all the <key, value> pairs that have required data are sent back to the reducer.
- The reducer then performs the desired averaging operation on the sorted <key, value> pairs to create a final <key, value> pair result.
- This final result is then stored as a sequence file within the HDFS.

The Results of this algorithm are mentioned in Chapter 9.

7.3 Linear Regression Algorithm

7.3.1 Defining Simple Linear Regression

Simple linear regression(SLR) is a statistical method that allows us to summarize and study relationships between two continuous (quantitative) variables.

- One variable, denoted x, is regarded as the *predictor*, explanatory, or independent variable.
- The other variable, denoted y, is regarded as the *response*, outcome, or dependent variable.

Simple linear regression gets its adjective "simple," because it concerns the study of only one predictor variable.

7.3.2 Simple Linear Regression Algorithm for Weather prediction

Step 1: Create a training set with 'x' values representing years (1892-2016) of same day of some particular station and 'y' values representing corresponding temperature readings of that particular weather station.

X-values (years)	**Y-values** (TMAX)
1892	20.0
1893	17.2
…	…

Table 7.1 Section of training set

Let 'm' indicate the size of training set i.e., the number of rows in above table.

Step 2: Plot this in a graph of x versus y indicated by a figure 7.3

Step 3: Establish a hypothesis function

$$y = h_\theta(x) = \theta_0 + \theta_1 x$$

This represents a straight line through the x-y plot from Step 2 as shown in figure 7.3. Our goal is to determine θ_0 & θ_1 values that forms a best fitting line through the plot. This is achieved as follows,

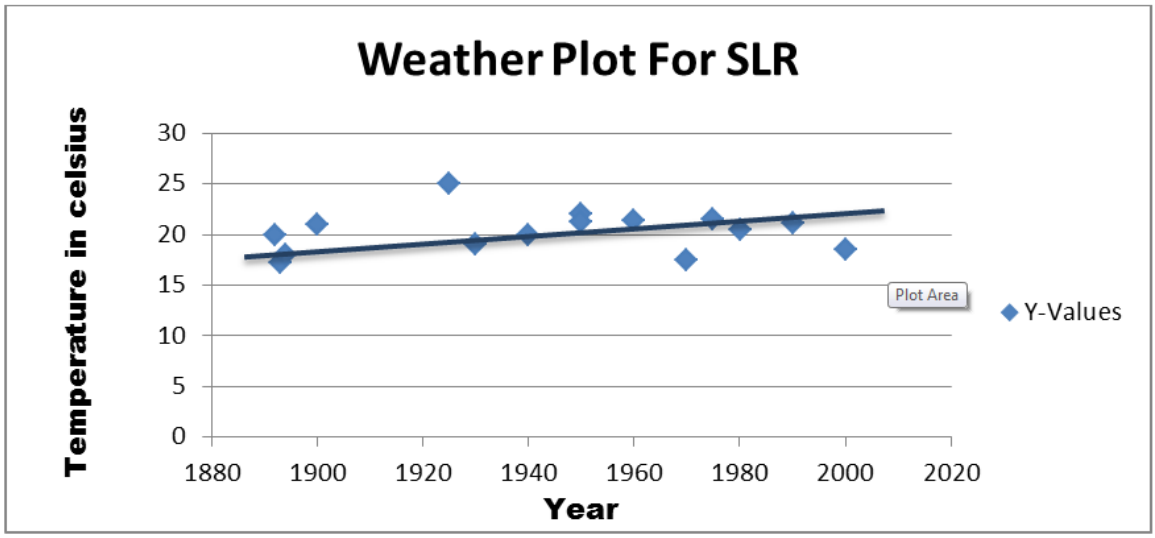


Figure 7.3 Regression Line plot

Step 4: Compute Cost Function

$$J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x)^{(i)} - y^{(i)})^2$$

This equation gives the mean of squared difference between a point in the graph and line i.e., the vertical distance between the two. The lesser the value the more accurate is the hypothesis.

Step 5: Iterate step 4 for range of values of θ_0 & θ_1 until you find the hypothesis that minimises the cost function to the least possible value. This hypothesis constitutes for the Best fitting line or is called the Regression Equation.

repeat until {

$$\underset{\theta_0, \theta_1}{minimize} \quad J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x)^{(i)} - y^{(i)})^2$$

}

Step 6: With Regression Equation available we substitute for 'x' in hypothesis representing year (past or future) and the curve gives the corresponding 'y' value representing temperature of that particular curve, which represents a particular day of a particular station. This algorithm is similarly applied for 365 days of whole year giving 365 different curves each capable of predicting temperature of the days they represent. In this project we predict for 5 upcoming years(2016-2020).

7.3.3 Implementation in MapReduce

The Driver Operation and Mapper Operation are exactly the same as explained in Mean calculation Method. <key, value> from mapper output is fed to reducer.

The Reduce Operation is going to receive <key, list(value)> after Shuffle and sorting mapper output. This list consist of temperature values of all possible years from input dataset for a specific date. This year and value data combined are passed on to make a call to *linear regression* routine. This routine accepts two operands and performs above mentioned computations. The resultant Regression Equation is returned to Reducer that computes temperatures for 5 upcoming years and writes the resulting <key, value> pairs to HDFS.

This ouput file can be copied to local file system for plotting comparison graphs and check accuracy and consistency with actual data.

7.4 Sliding Window Algorithm in R

To predict the future's weather condition, the variation in the conditions in past years must be utilized. The probability that the weather condition of the day in consideration will match the same day in previous year is very less. But the probability that it will match within the span of adjacent fortnight of previous year is very high. So, for the fortnight considered for previous year a sliding window is selected of size equivalent to a week. Every week of sliding window is then matched with that of current year's week in consideration. The window best matched is made to participate in the process of predicting weather conditions. The prediction is made based on sliding window algorithm. The month-wise results are being computed for three years to check the accuracy. The results of the approach suggested that the method used for weather condition prediction is quite efficient with an average accuracy of 92.2%.

7.4.1 Methodology

There is always a slightly variation in weather conditions which may depend upon the last seven days or so variation. Here variation refers to difference between previous day parameter and present day's parameter. Also there exists a dependency between the weather conditions persisting in current week in consideration and those of previous years.

In this work a methodology is being proposed that could mathematically model these two types of dependency and utilize them to predict the future's weather conditions. To predict the day's weather conditions this work will take into account the conditions prevailing in previous week, that is, in last seven days which are assumed to be known. Also the weather condition of seven previous days and seven upcoming days for previous year is taken into consideration.

For instance, if the weather condition of 16 November 2012 is to be predicted then we will take into consideration the conditions from 09 November 2012 to 15 November 2012 and conditions from 09November to 22November 2011 for previous years. Now in order to model the aforesaid dependencies the current year's variation throughout the week is being matched with those of previous years by making use of sliding window. The best-matched window is selected to make the prediction. The selected window and the current year's weekly variations are together used to predict the weather condition.

The reason for applying sliding window matching is that the weather conditions prevailing in a year may not lie or fall on exactly the same date as they might have existed in previous years. That is why seven previous days and seven on-going days are being considered. Hence a total period of fortnight is checked in previous condition to find the similar one. Sliding window is quite good technique to capture the variation that could match the current year's variation.

7.4.2 Sliding Window Algorithm for Weather Prediction

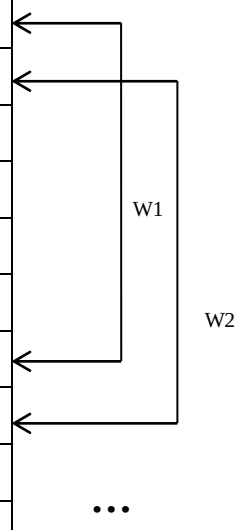
The work proposes to predict a day's weather conditions. For this the previous seven days weather is taken into consideration along with fortnight weather conditions of past years. Suppose we need to predict weather of 23rd August 2013 then we will take into consideration the weather conditions of 16th August 2013 to 22nd August 2013 along with the weather conditions prevailing in the span of 16th August to 29th August in past years. Then the day by day variation in current year is computed. The variation is also being computed from the fortnight data of previous year.

In this work the two major weather parameters will be taken into consideration, that is, maximum temperature, minimum temperature. Hence the size of the variation of the current year will be represented by matrix of size 7 × 2. And similarly for past year the matrix size would be 14 × 2. Now, the first step is to divide the matrix of size 14×2 into the sliding windows. Hence, 8 sliding windows can be made of size 7×2 each. The concept of sliding window is shown in Table 7.2

Now the next step is to compare every window with the current year's variation. The best-matched window is selected for making the prediction. The Euclidean distance approach is used for the purpose of matching.

The reason for taking Euclidean distance is its power to represent similarity in spite of its simplicity.

S. No	Max. Temp	Min Temp
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		



.

Table 7.2 Sliding window concept where $W1$ represents Window number 1
And $W2$ represent window number 2

Following are the parameters used for the weather condition prediction:

1. **Mean**: Mean of day's weather conditions, that is, maximum temperature, minimum temperature. After adding each separately, and divide by total day's number

$$Mean = \frac{Sum\ of\ Parameters}{number\ of\ Days}$$

2. **Variation**: Calculate day by day variation after taking difference of each parameter. This tells how the next day's Weather is related to previous day's weather.
3. **Euclidean distance**: It compares data variation of current year and previous year. By this we are able to mathematically model the aforesaid defined dependencies. That the relationship between previous year and previous week data is being defined mathematically can be used to predict the future conditions.

$$Euclidean\ Distance = \sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)^2}$$

7.4.3 Pseudo Code for **Sliding Window Algorithm**

Step 1: Take matrix "CD" of last seven days for current year's data of size 7 × 2.

Step 2: Take matrix "PD" of fourteen days for previous year's data of size 14 × 2.

Step 3: Make 8 sliding windows of size 7 × 2 each from the matrix "PD" as $W1$, $W2$, $W, \ldots, W8$

Step 4: Compute the Euclidean distance of each sliding window with the matrix "CD" as ED1, ED2, ED3, . . . , ED8

Step 5: Select matrix Wi as Wi = Corresponding Matrix (Min.(EDi)) $\forall i \in [1, 8]$

Step 6: For k = 1 to n

1. For WCk compute the variation vector for the matrix "CD" of size 6 × 2 as "VC".
2. For WCk compute the variation vector for the matrix "PD" of size 6 × 2 as "VP".
3. Compute the Mean of the variations found in the previous step

   3.1. Tmax_Mean1 =Mean (VC_TMAX)

   3.2. Tmax_Mean2 = Mean(VP_TMAX)

   3.3. Tmin_Mean1 =Mean (VC_TMIN)

   3.4. Tmin_Mean2 = Mean(VP_TMIN)

   3.5. Predicted Variation "V_TMAX" = (Tmax_Mean1 + Tmax_Mean2)/2

   3.6. Predicted Variation "V_TMIN" = (Tmin_Mean1 + Tmin_Mean2)/2

4. Add the resultant Means to the Current year's data to get predicted values

   4.1. Add "V_TMAX" to the previous day's weather condition in consideration to get the predicted condition of Maximum temperature.

   4.2. Add "V_TMIN" to the previous day's weather condition in consideration to get the predicted condition of Minimum temperature.

Step 7: End

The sliding window used for predicting the "n" number of weather conditions (WC1 ,WC2 ,WC3 , .WCn)

CHAPTER 8

IMPLEMENTATION<code>

8.1 The Normals Method

8.1.1 The Driver Class

```
package com.rnsit.weather.controller;
import com.rnsit.weather.*;
import org.apache.hadoop.*;
public class WFDriver extends Configured implements Tool {
    public int run(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = new Job(conf, "WFDriver");
        job.setJarByClass(WFDriver.class);
        job.setMapperClass(WFMeanCalculationMapper.class);
        job.setReducerClass(WFMeanCalculationReducer.class);
        job.setMapOutputKeyClass(WFDataPojo.class);
        job.setMapOutputValueClass(WFDataPojo.class);
        job.setOutputKeyClass(Text.class);
        job.setNumReduceTasks(160);
        job.setPartitionerClass(WeatherPartitioner.class);
        job.setOutputValueClass(DoubleWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        return job.waitForCompletion(true) ? 0 : 1;
    }
    public static void main(String[] args) throws Exception{
        int res = ToolRunner.run(new Configuration(), new
                    WFDriver(), args);
        System.exit(res);
    }
}
```

8.1.2 The Mapper Class

```
package com.rnsit.weather.mappers;
import com.rnsit.weather.*;
import org.apache.hadoop.*;
import java.io.*;
import java.util.*;

public  class WFMeanCalculationMapper extends Mapper<Object, Text,
WFDataPojo, WFDataPojo> {

    public static Map<Integer,String> mapp=new HashMap<Integer,String>() ;
    static {
        for(int i=1;i<=31;i++) {
            mapp.put(i,(Integer.valueOf(i).toString().length()>1)?
             Integer.valueOf(i).toString():"0"+i);
        }
    }
    public void map(Object key, Text value, Context context) throws
    IOException, InterruptedException {
      if (value.toString().contains("TMAX")) {
        String[] arrayValues=WFCsvParser.parse(value.toString());
        String stationId = arrayValues[0];
        String[] stationArr = stationId.split("~");
        String date = stationArr[1];
        String stationCode = stationArr[0];
        String yyyymmdd;
        for (int i = 1; i < arrayValues.length; i++) {
           if (arrayValues[i].matches("[0-9]*")) {
               if (i <= 31) {
                  yyyymmdd = date.concat(mapp.get(i));
                  WFDataPojo wfDataPojo = new WFDataPojo();
              wfDataPojo.setDate(newLongWritable(Long.parseLong(yyyymmdd)));
              wfDataPojo.setStationCode(new Text(stationCode));
              wfDataPojo.setTmax(new
                    DoubleWritable(Double.parseDouble(arrayValues[i])/10.0));
                    context.write(wfDataPojo, wfDataPojo);
            }}
         }}
      }}
```

8.1.3 The Parser Class

```
package com.rnsit.weather.parsers;

import com.rnsit.weather.com.rnsit.weather.model.WFDataPojo;

import org.apache.hadoop.io.Text;

import java.util.*;

public class WFCsvParser {

    public static final int startId=0;

    public static final int endId=11;

    public static final int dateStartIndex=11;

    public static final int dateEndIndex=17;

        public static String[] parse(String value){

        String[] data = value.toString().split("\\s+");

        List<String> list = new ArrayList<String>(Arrays.asList(data));

        List<String> ll=new ArrayList<String>();

        ll.add("E");

        ll.add("G");

        ll.add("I");

        ll.add("S");

        list.removeAll(ll);

        String[] strData=new String[list.size()];

        list.toArray(strData);

        String stationId=strData[0];

      strData[0]=stationId.substring(startId,endId).concat("~").concat(

      stationId.substring(dateStartIndex,dateEndIndex));

        return strData;
    }

}
```

8.1.4 The Reducer Class

```
package com.rnsit.weather.reducers;
import com.rnsit.weather.com.rnsit.weather.model.WFDataPojo;
import com.rnsit.weather.mappers.WFMeanCalculationMapper;
import org.apache.commons.lang.ArrayUtils;
import org.apache.hadoop.*;
import java.io.IOException;
import java.util.*;
public class WFMeanCalculationReducer extends Reducer<WFDataPojo,
DoubleWritable,Text,DoubleWritable> {
    private DoubleWritable result = new DoubleWritable();
    Double sum = 0.0; int count = 0;
    public void reduce(WFDataPojo key, Iterable<DoubleWritable > values,
                Context context) throws IOException, InterruptedException{
            for(DoubleWritable i : values){
                  sum += i.get();
                  count ++;
            }
            sum/=count;
            result.set(sum);
            context.write(
            new Text(key.getStationCode()+":"+(int)x1[i]+key.getMonthDay()),
            result);
        }
    }

}
```

8.1.5 The POJO Class

```
package com.rnsit.weather.com.rnsit.weather.model;
import org.apache.hadoop.io.*;
import java.io.*;
public class WFDataPojo implements WritableComparable<WFDataPojo > {
    private LongWritable date;
    public WFDataPojo(LongWritable date, Text city, Text stationCode,
    DoubleWritable tmax, DoubleWritable tmin, DoubleWritable precipitation)
{
        this.date = date;
        this.city = city;
        this.stationCode = stationCode;
        this.tmax = tmax;
        this.tmin = tmin;
        this.precipitation = precipitation;
    }
    public WFDataPojo(){}
    private Text city;
    private Text stationCode;
    private DoubleWritable tmax;
    private DoubleWritable tmin;
    private DoubleWritable precipitation;

    public void setDate(LongWritable date) {
        this.date = date;
    }

    public void setStationCode(Text stationCode) {
        this.stationCode = stationCode;
    }

    public void setCity(Text city) {
        this.city = city;
    }

    public void setTmax(DoubleWritable tmax) {
        this.tmax = tmax;
    }

    public void setTmin(DoubleWritable tmin) {
        this.tmin = tmin; }
```

```
public void setPrecipitation(DoubleWritable precipitation) {
    this.precipitation = precipitation;
}

public LongWritable getDate() {
    return date;
}

public DoubleWritable getPrecipitation() {
    return precipitation;
}

public DoubleWritable getTmin() {
    return tmin;
}

public DoubleWritable getTmax() {
    return tmax;
}

public Text getStationCode() {
    return stationCode;
}

public Text getCity() {
    return city;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    WFDataPojo that = (WFDataPojo) o;
    if (city != null ? !city.equals(that.city) : that.city != null)
        return false;
    if (date != null ? !date.equals(that.date) : that.date != null)
        return false;
    if (precipitation != null ?
       !precipitation.equals(that.precipitation) :
        that.precipitation != null)
              return false;
```

```
        if (stationCode != null ? !stationCode.equals(that.stationCode) :
            that.stationCode != null) return false;
        if (tmax != null ? !tmax.equals(that.tmax) : that.tmax != null)
            return false;
        if (tmin != null ? !tmin.equals(that.tmin) : that.tmin != null)
            return false;
        return true;
    }
    @Override
    public int hashCode() {
        int result = date != null ? date.hashCode() : 0;
        result = 31*result+(stationCode != null ? stationCode.hashCode():0);
        return result;
    }
    @Override
    public int compareTo(WFDataPojo o) {
       int retCode=this.getStationCode().compareTo(o.getStationCode());
       if(retCode!=0){
           return retCode;
       }
        else{
           String mmdd1 = this.getDate().toString().substring(4);
           String mmdd2 = o.getDate().toString().substring(4);
           return mmdd1.compareTo(mmdd2);
       }
    }
    @Override
    public void write(DataOutput dataOutput) throws IOException {
        dataOutput.writeLong(date.get());
        dataOutput.writeUTF(stationCode.toString());
        dataOutput.writeDouble(tmax.get());
    }
    @Override
    public void readFields(DataInput dataInput) throws IOException {
        setDate(new LongWritable(dataInput.readLong()));
        setStationCode(new Text(dataInput.readUTF()));
        setTmax(new DoubleWritable(dataInput.readDouble()));
    }
    public String getMonthDay(){
        return this.getDate().toString().substring(4);
    }
}
```

8.2 Simple Linear Regression

The Driver, Mapper and Pojo classes are same as in normal method.

8.2.1 The Reducer Class

```
package com.rnsit.weather.reducers;
import com.rnsit.weather.com.rnsit.weather.model.WFDataPojo;
import com.rnsit.weather.mappers.WFMeanCalculationMapper;
import org.apache.commons.lang.ArrayUtils;
import org.apache.hadoop.*;
import java.io.IOException;
import java.util.*;

public class WFMeanCalculationReducer extends
Reducer<WFDataPojo,WFDataPojo,Text,DoubleWritable> {
    private DoubleWritable result = new DoubleWritable();
    public void reduce(WFDataPojo key, Iterable<WFDataPojo> values,
        Context context) throws IOException, InterruptedException {
        List<Double> keyList = new ArrayList<Double>();
        List<Double> valList = new ArrayList<Double>();
        for(WFDataPojo val : values){
   keyList.add(Double.parseDouble(val.getDate().toString().substring(0,4)));
            valList.add(Double.parseDouble(val.getTmax().toString()));
        }
        Double[] years=new Double[keyList.size()];
        keyList.toArray(years);
        Double[] tmaxs=new Double[valList.size()];
        valList.toArray(tmaxs);
        double x1[] = new double[] {2016,2017,2018,2019,2020};
        double y1[]= new double[5];
        double theta[]=WFRegressionCalculationReducer.lreg(
                        ArrayUtils.toPrimitive(years),
                        ArrayUtils.toPrimitive(tmaxs));
        for (int i=0;i<5;i++){
            y1[i]=theta[1]*x1[i]+theta[0];
            context.write(
            new Text(key.getStationCode()+":"+(int)x1[i]+key.getMonthDay()),
            new DoubleWritable(y1[i]));
        }
    }
}
```

8.2.2 The Regression Algorithm Class

```
package com.rnsit.weather.reducers;
public class WFRegressionCalculationReducer {
    public static double[] lreg(double[] x,double[] y) {
        int n = 0;
        double sumx = 0.0, sumy = 0.0, sumx2 = 0.0;
        while(n!=y.length) {
            sumx  += x[n];
            sumx2 += x[n] * x[n];
            sumy  += y[n];
            n++;
        }
        double xbar = sumx / n;
        double ybar = sumy / n;
        double xxbar = 0.0, yybar = 0.0, xybar = 0.0;
        for (int i = 0; i < n; i++) {
            xxbar += (x[i] - xbar) * (x[i] - xbar);
            yybar += (y[i] - ybar) * (y[i] - ybar);
            xybar += (x[i] - xbar) * (y[i] - ybar);
        }
        double theta[] = new double[2];
        theta[1] = xybar / xxbar;
        theta[0] = ybar - theta[1] * xbar;
        return theta;
    }
}
```

8.3 Sliding Window Algorithm in R

```
library(RJSONIO)
library(plyr)
library(dplyr)
library(zoo)
library(stringr)
library(plotly)

#reading the date
readDate <- function()
{
  n <- readline(prompt="Enter an Date(yyyy-mm-dd): ")
  return(n)
}

#user Input
prediction_date<-readDate()
predictionDate<-as.Date(prediction_date)
class(predictionDate)
##############################################################################
#current temperature
temp_data1 <- read.csv("G:\\Project\\Sliding Window
Algorithm\\bangalore_2016_csv.csv" , sep = ",")
total_data1<-temp_data1[, seq(1,3)]
names(total_data1)<- c('DATE','TMAX','TMIN')
total_data1
total_data1$TMAX<-as.numeric(substr(as.character(total_data1$TMAX),1,2))
total_data1$TMIN<-as.numeric(substr(as.character(total_data1$TMIN),1,2))
total_data1
cur_year_data<-total_data1[as.numeric(substr(y<-
as.character((total_data1$DATE)),1,4)) %in% 2016,]
cur_year_data

#getting the date format from the curr year data
cur_month_date<-seq(predictionDate-7, by = 'day',length.out = 7)
days1<-as.numeric(paste(substr(y<-
as.character(cur_month_date),6,7),substr(y<-
as.character(cur_month_date),9,nchar(y)),sep=''))
```

```
#matching the data
#var<- filter(prev_year_data,DATE=(DATE%/%10000==2015))
cur_year_data$DATE<-as.numeric(substr(y<-
as.character(cur_year_data$DATE),5,nchar(y)))
CD<-cur_year_data[cur_year_data$DATE %in% days1,c(2,3)]

as.data.frame.matrix(CD)
CD[1,1]

#centigarde conversion
for(i in 1:7)
{
  CD[i,1]=((as.numeric(CD[i,1])-32)*(5/9))
  CD[i,2]=((as.numeric(CD[i,2])-32)*(5/9))
}
as.data.frame.matrix(CD)
currYearData<-array(CD[1:2])
names(currYearData)<- c('TMAX','TMIN')

##############################################################################
#previous years data - actual temperature from Accuweather API
temp_data <- read.csv("G:\\Project\\Sliding Window
Algorithm\\Bangalore_2015_csv_02.csv" , sep = ",")
total_data<-temp_data[, seq(1,3)]
names(total_data)<- c('DATE','TMAX','TMIN')
total_data
total_data$TMAX<-as.numeric(substr(as.character(total_data$TMAX),1,2))
total_data$TMIN<-as.numeric(substr(as.character(total_data$TMIN),1,2))
total_data

prev_year_data<-total_data[as.numeric(substr(y<-
as.character((total_data$DATE)),1,4)) %in% 2015,]
prev_year_data

#getting the date format from the previous year data
prev_month_date<-seq(predictionDate-7, by = 'day',length.out = 14)
days<-as.numeric(paste(substr(y<-
as.character(prev_month_date),6,7),substr(y<-
as.character(prev_month_date),9,nchar(y)),sep=''))

#matching the data
#var<- filter(prev_year_data,DATE=(DATE%/%10000==2015))
```

```
prev_year_data$DATE<-as.numeric(substr(y<-
as.character(prev_year_data$DATE),5,nchar(y)))
PD<-prev_year_data[prev_year_data$DATE %in% days,c(2,3)]
as.data.frame.matrix(PD)

#############################################################################
#dividing previous year data into 8 sliding window algorithms
x<-PD
y <- rollapply(x, width=7, FUN=function(x) {print(x);},align = "right")
as.data.frame.matrix(y)

sliding_windows <- array(, dim=c(7,2,8))
j<-1
for(i in 1:8)
{
  sliding_windows[,j,i]=y[i,1:7]
  sliding_windows[,j+1,i]=y[i,8:14]
  sliding_windows[,,i]
}
#conversion to centigrade
for(i in 1:8)
{
  for(j in 1:7)
  {
    for(k in 1:2)
    {
      sliding_windows[j,k,i]=(as.numeric(sliding_windows[j,k,i]-32)*(5/9))
    }
  }
}
#############################################################################
#converting the current year dataframe to matrix
curr <- array(, dim=c(7,2,1))
for(i in 1:7)
{
  for(j in 1:2)
  {
    curr[i,j,1]=as.numeric(currYearData[i,j])
  }
}

#ED_matrices contains the euclidean distances - 8 in number
```

```
ED_matrices <- array(, dim=c(7,1,8))

#function to find the euclidian distance
euclidian_diat<- function(x1,x2,y1,y2){
  res<- sqrt(((x1-x2)^2)+((y1-y2)^2))
  return(res)
}
for(i in 1:8)
{
  for(j in 1:7)
  {
    ED_matrices[j,1,i]<-
euclidian_diat(curr[j,1,1],sliding_windows[j,1,i],curr[j,2,1],sliding_window
s[j,2,i])

  }
}
#contains the mean of the each euclidean matrix
ED_means<- array(,dim = c(8,1))
for (i in 1:8) {
  ED_means[i,1]=mean(ED_matrices[,,i])
}
#finding the min(ED_means)
matched_window_position=which(ED_means==min(ED_means))

#matched_window
sliding_windows[,,matched_window_position]
matched_window=array(,dim = c(7,2))
matched_window=sliding_windows[,,matched_window_position]
matched_window

#########################################################################
#variation
vc <- array(, dim = c(6,2))
vp <- array(, dim=c(6,2))
for(i in 1:6)
{
 for(j in 1:2)
  {
    vc[i,j]=curr[i,j,1]-curr[i+1,j,1]
  }
}
```

```
vc

for(i in 1:6)
{
  for(j in 1:2)
  {
    vp[i,j]=matched_window[i,j]-matched_window[i+1,j]
  }
}
vp
class(vp)
colnames(vc)<-c('TMAX','TMIN')
colnames(vp)<-c('TMAX','TMIN')
colnames(curr) <- c('TMAX','TMIN')

as.data.frame(vc)
vc[,1]
#mean of the variations
mean_TMAX1=mean(vc[,1])
mean_TMAX2=mean(vp[,1])

mean_TMIN1=mean(vc[,2])
mean_TMIN2=mean(vp[,2])

v_TMAX=(mean_TMAX1+mean_TMAX2)/2
v_TMIN=(mean_TMIN1+mean_TMIN2)/2

res_TMAX <- as.data.frame(matrix(curr[,1,1]+v_TMAX, ncol = 1, nrow = 7))
is.data.frame(res_TMAX)
res_TMIN <- as.data.frame(matrix(curr[,2,1]+v_TMIN, ncol = 1, nrow = 7))

res_TMAX
res_TMIN
colnames(res_TMAX)<-c('TMAX')
colnames(res_TMIN)<-c('TMIN')

#########################################################################
#Actual Temperatures

actual_temp_data<- read.csv("G:\\Project\\Sliding Window
Algorithm\\bangalore_2016_csv.csv" , sep = ",")
actual_total_data<-actual_temp_data[, seq(1,3)]
```

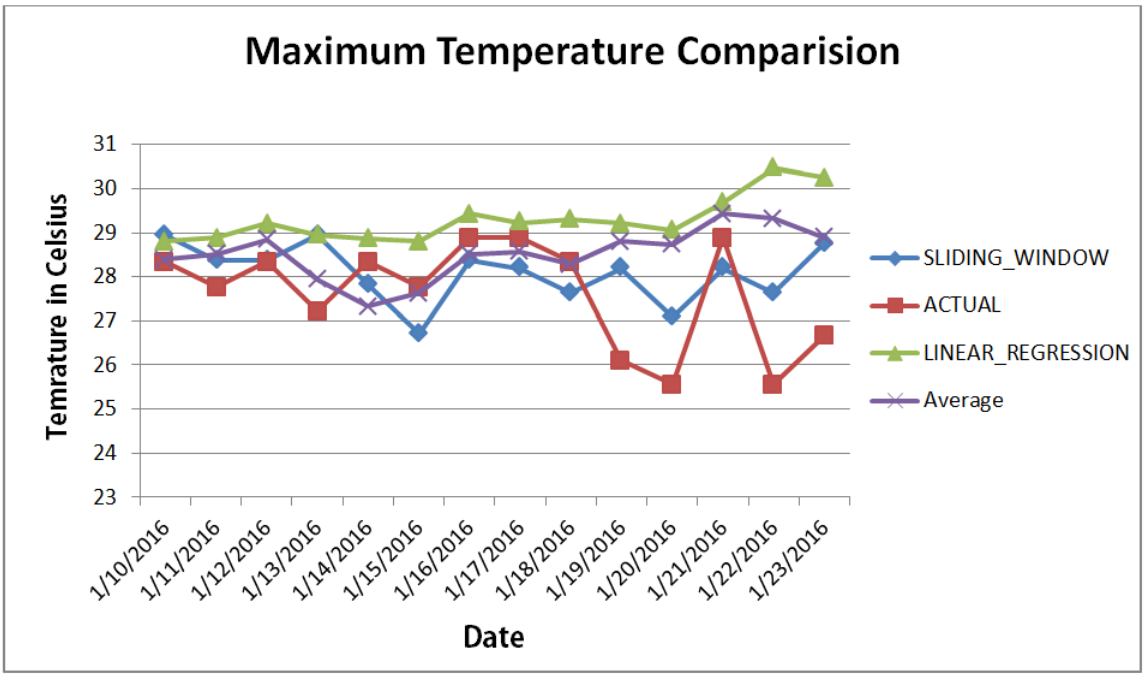
```
names(actual_total_data)<- c('DATE','TMAX','TMIN')
actual_total_data
actual_total_data$TMAX<-
as.numeric(substr(as.character(actual_total_data$TMAX),1,2))
actual_total_data$TMIN<-
as.numeric(substr(as.character(actual_total_data$TMIN),1,2))
actual_total_data

actual_cur_year_data<-actual_total_data[as.numeric(substr(y<-
as.character((actual_total_data$DATE)),1,4)) %in% 2016,]
actual_cur_year_data

#getting the date format from the curr year data
actual_cur_month_date<-seq(predictionDate, by = 'day',length.out = 7)
actual_days<-as.numeric(paste(substr(y<-
as.character(actual_cur_month_date),6,7),substr(y<-
as.character(actual_cur_month_date),9,nchar(y)),sep=''))

#matching the data
#var<- filter(prev_year_data,DATE=(DATE%/%10000==2015))
actual_cur_year_data$DATE<-as.numeric(substr(y<-
as.character(actual_cur_year_data$DATE),5,nchar(y)))
actual_CD<-actual_cur_year_data[actual_cur_year_data$DATE %in%
actual_days,c(2,3)]

as.data.frame.matrix(actual_CD)
#centigarde conversion
for(i in 1:7)
{
  actual_CD[i,1]=((as.numeric(actual_CD[i,1])-32)*(5/9))
  actual_CD[i,2]=((as.numeric(actual_CD[i,2])-32)*(5/9))
}
as.data.frame.matrix(actual_CD)
actual_currYearData<-array(actual_CD[1:2])
names(actual_currYearData)<- c('Actual_TMAX','Actual_TMIN')
actual_currYearData

#############################################################################
dates<- seq(predictionDate, by = 'day',length.out = 7)
resultant_data_frame<-
cbind(dates,res_TMAX,res_TMIN,actual_currYearData$Actual_TMAX,actual_currYea
rData$Actual_TMIN)
```

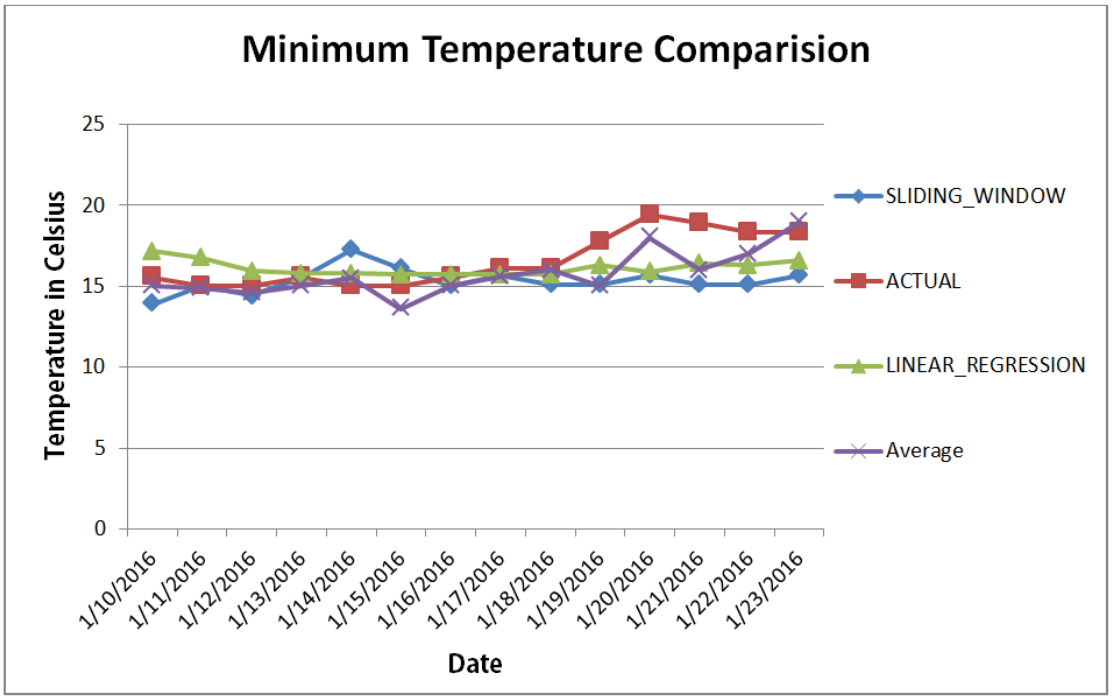
```
colnames(resultant_data_frame)<-
c('Dates','P_TMAX','P_TMIN','A_TMAX','A_TMIN')
resultant_data_frame
write.csv(resultant_data_frame, file = "G:\\Project\\Sliding Window
Algorithm\\predicted_temperature_data.csv")
predicted_temperature=read.csv("G:\\Project\\Sliding Window
Algorithm\\predicted_temperature_data.csv")
str(predicted_temperature)
predicted_temperature

#drawing the comparision Plot
f <- list(
  family = "Courier New, monospace",
  size = 18,
  color = "Blue"
)
x <- list(
  title = "Dates",
  titlefont = f
)
y <- list(
  title = "Temperature",
  titlefont = f
)
#Graph
predicted_tmax_tmin_plot<-plot_ly(predicted_temperature,x = Dates, y =P_TMAX
, name="Predicted Max Temp", line = list(shape = "spline"))  %>%
  add_trace(x = Dates,y =P_TMIN, name="Predicted Min Temp", line =
list(shape = "spline")) %>%
  layout(xaxis = x, yaxis = y)

predicted_tmax_tmin_plot

#########################################################################
#Comparision of Minimum and maximum Temperature

comparision_tmax_tmin_plot<-plot_ly(predicted_temperature,x = Dates, y
=P_TMAX , name="Predicted Max Temp", line = list(shape = "spline"))  %>%
  add_trace(x = Dates,y =P_TMIN, name="Predicted Min Temp", line =
list(shape = "spline")) %>%
  add_trace(x = Dates,y = A_TMIN, name="Actual Min Temp", line = list(shape
= "spline")) %>%
```

```
  add_trace(x = dates,y = A_TMAX, name="Actual Max Temp", line = list(shape
= "spline")) %>%
  layout(xaxis = x, yaxis = y)

comparision_tmax_tmin_plot

#############################################################################
#Comparision of Maximum Temperature
comparision_tmax_plot<-plot_ly(predicted_temperature,x = Dates, y =P_TMAX ,
name="Predicted Max Temp", line = list(shape = "spline"))  %>%
  add_trace(x = dates,y = A_TMAX, name="Actual Max Temp", line = list(shape
= "spline")) %>%
  layout(xaxis = x, yaxis = y)

comparision_tmax_plot

#############################################################################
#Comparision of Minimum Temperature

comparision_tmin_plot<-plot_ly(predicted_temperature,x = Dates, y =P_TMIN ,
name="Predicted Min Temp", line = list(shape = "spline"))  %>%
  add_trace(x = dates,y = A_TMIN, name="Actual Min Temp", line = list(shape
= "spline")) %>%
  layout(xaxis = x, yaxis = y)

comparision_tmin_plot

#############################################################################
```

CHAPTER 9

RESULTS





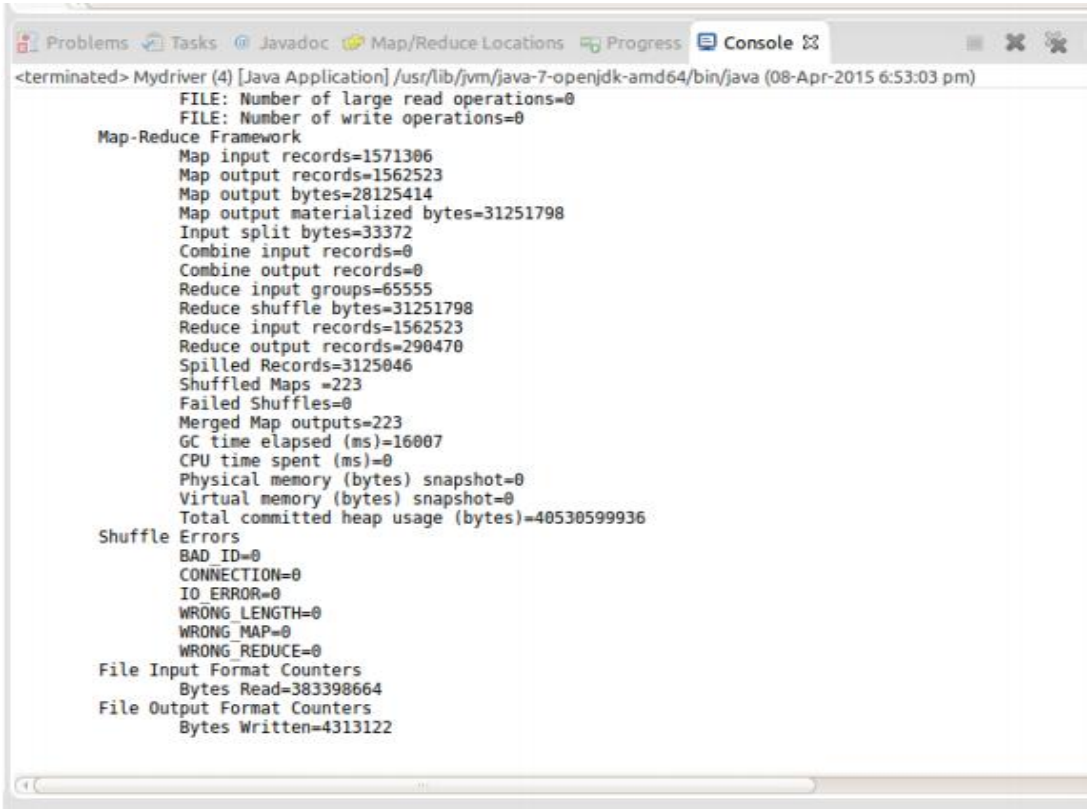


Figure 9.3 MapReduce Framework Execution (Terminal log)

1	IN021010100	20170101	12.079128857554224
2	IN021010100	20170102	10.249765866294709
3	IN021010100	20170103	11.325959552583683
4	IN021010100	20170104	12.123833454189793
5	IN021010100	20170105	10.54164785553047
6	IN021010100	20170106	10.46963889573577
7	IN021010100	20170107	11.088696992346932
8	IN021010100	20170108	11.924964028117643
9	IN021010100	20170109	11.80957642725599
10	IN021010100	20170110	9.621987424951545
11	IN021010100	20170111	8.9675372203277
12	IN021010100	20170112	9.46317782120176
13	IN021010100	20170113	8.549991060253888
14	IN021010100	20170114	9.651213814439501
15	IN021010100	20170115	10.09148787273628
16	IN021010100	20170116	11.750910783678762
17	IN021010100	20170117	11.01743237577682
18	IN021010100	20170118	10.87205901639345
19	IN021010100	20170119	11.66786061118188
20	IN021010100	20170120	11.32592534776424
21	IN021010100	20170121	10.709993350751226
22	IN021010100	20170122	9.67420987185244
23	IN021010100	20170123	10.000494658060719
24	IN021010100	20170124	9.821613561616314
25	IN021010100	20170125	9.826453875213645
26	IN021010100	20170126	10.172101959022498
27	IN021010100	20170127	9.331352718078385
28	IN021010100	20170128	9.781956677881517

Figure 9.4 Reducer output for one station for the year 2017

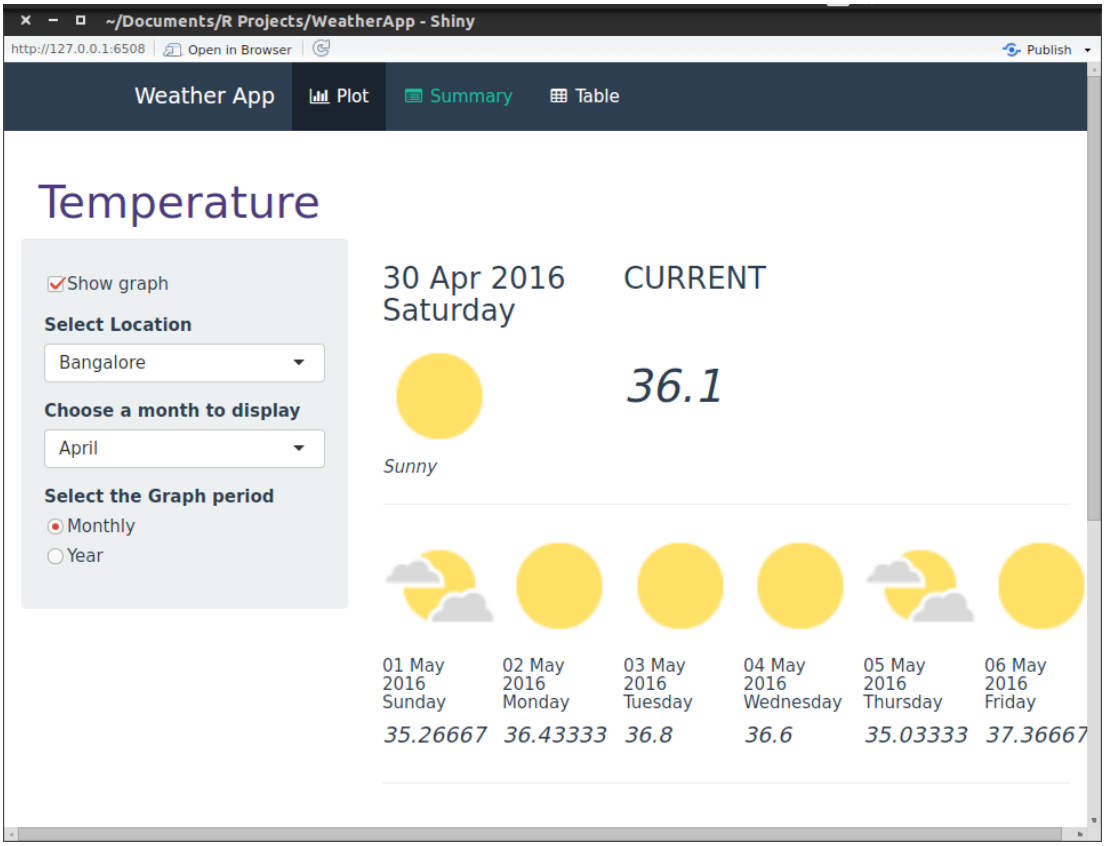


Figure 9.5 Web Application Showing the Maximum Temperature weekly prediction of Bangalore, April 2016

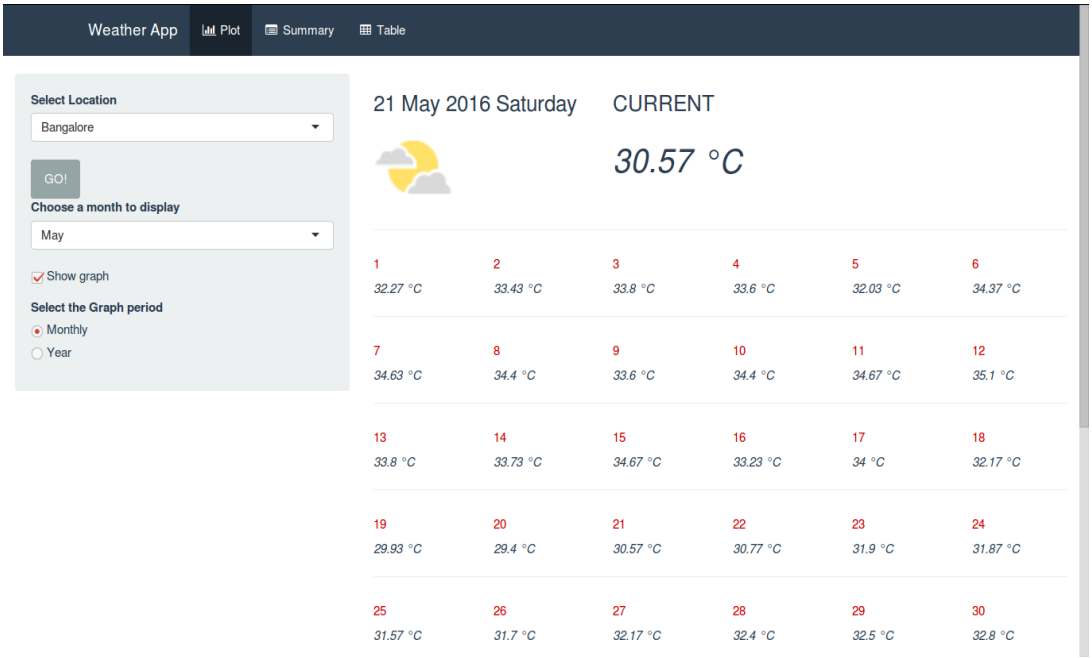


Figure 9.6 Web Application showing Maximum Temperature monthly prediction of Bangalore, May 2016

Month	Efficiency %
Jan 16	97.24
Feb 16	88.94
Mar 16	92.25
Apr 16	98.17

Table 9.1 Accuracy of prediction per month (2016) achieved in Sliding Window

Month	Efficiency %
Jan 16	95.75
Feb 16	95.95
Mar 16	96.9
Apr 16	93.48

Table 9.2 Accuracy of prediction per month (2016) achieved in Linear Regression

Month	Efficiency %
Jan 16	96.48
Feb 16	95.36
Mar 16	95.94
Apr 16	93.20

Table 9.3 Accuracy of prediction per month (2016) achieved in Normal/Mean method

CHAPTER 10

CONCLUSION

From the results we observe that each algorithm's efficiency is over 92%. The mean calculation being the simplest achieves results that compete well with other techniques discussed in this project. This goes to prove that, however fiendish your algorithms are, they can often be beaten simply by having more data (and a less sophisticated algorithm).

We can also conclude that MapReduce, if run in cluster, can achieve faster results. This demonstrates the power of distributed computing. These results were experienced when the same code was executed in single node first and then multi-node cluster. The speed of execution for multi-cluster didn't compromise with increasing dataset as single node setup did.

The Sliding Window algorithm which predicts weather with just current and previous year's data proves to have potential against Big Data, however as we can observe the variations/distortions in prediction values. Yet it achieves results that vary with actual temperature by mere +/- 2 degrees.

Future study, Weather prediction techniques have evolved since its dawn in 650 BC to current model based and Numerical weather prediction with help of super computers. These modern techniques are power and resource intensive. Still 100% efficiency is yet to achieved. An efficient low cost solution is to be designed to make very accurate predictions even with vast turbulences/unexpected changes that exist in weather conditions.

BIBLIOGRAPHY

[1] Tom White, Hadoop: The Definitive Guide, Third Edition, O'Reilly, Jan 2012

[2] From Gantz et al., "The Diverse and Exploding Digital Universe" March 2008 (*http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf*)

[3] Big Data - Learning Basics of Big Data in 21 Days, October 2013 (*http://blog.sqlauthority.com/2013/10/30/big-data-learning-basics-of-big-data-in-21-days-bookmark/*)

[4] The quote is from Anand Rajaraman writing about the Netflix Challenge (*http://anand.typepad.com/datawocky/2008/03/more-data-usual.html*). Alon Halevy, Peter Norvig, and Fernando Pereira make the same point in "The Unreasonable Effectiveness of Data," IEEE Intelligent Systems, March/April 2009.

[5] TutorialsPoint, Introduction to BigData - Hadoop (http://www.tutorialspoint.com/hadoop/hadoop_introduction.htm)

[6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. *The Google File System*. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03, pages 29–43, New York, NY, USA, October 2003. ACM. ISBN 1 -58113- 757-5. doi: 10.1145/945445.945450

[7] Riyaz P.A., Surekha Mariam Varghese, *Leveraging Map Reduce With Hadoop for Weather Data Analytic,* IOSR Journal of Computer Engineering (IOSR-JCE), May – Jun. 2015

[8] WikiPedia (https://en.wikipedia.org/wiki/Weather_forecasting)

[9] R Documentation (https://www.r-project.org/other-docs.html)

[10] Piyush Kapoor and Sarabjeet Singh Bedi, *Weather Forecasting Using Sliding Window Algorithm*, ISRN Signal Processing, Volume 2013 (2013), Article ID 156540, 5 pages. (http://dx.doi.org/10.1155/2013/156540)