WEATHER DATA ANALYSIS

USING SPARK AND HADOOP MAPREDUCE

MS of Data Science - Project Report

Version 1.1

Course: Big Data Analytics (Class:2688)

Submitted By:
Ramsha Arif (23395)
Sarosh Aamir (23390)

Project Link: https://drive.google.com/drive/folders/1qDjZtDQ25br-Zvqw_YPEKEzMBgOLQhQL?usp=sharing

IBA Institute of Business Administration Karachi
Leadership and Ideas for Tomorrow

Abstract

Climate directly affects human existence. Each individual is straightforwardly or in a roundabout way influenced by weather. Because of this climate information investigation is a pivotal space for study. Agriculture area is most needy area on climate expectation.

Additionally, the travel industry area gets influenced by climate. Heaps of government bodies are keen on climate information investigation for their essential arranging if there should arise an occurrence of flood and dry season. Human disposition and wellbeing can likewise be influenced by climate.

A portion of the pivotal boundaries are temperature, weight, stickiness and wind speed. The information is gathered on hourly premise with a recurrence of 3-4 times each hour. Ordinarily, this information is put away in the unstructured arrangement. The structure of this information design is a plain content record where each field is isolated by a comma or a tab or might be by the semicolon.

This tremendous measure of information has amassed from last numerous years and it will keep on developing. Direct preparing of this gigantic unstructured information utilizing traditional strategies and apparatuses is troublesome and wasteful. This has brought about the difficulties of capacity and preparing of gigantic climate information. One of such information is put away at NCDC, USA. It has the store for climate information from last numerous years till today.

Table of Contents

ACKNOWLEDGEMENT

Big Data is a broad term for data sets so large or complex that they are difficult to process using traditional data processing applications. There is a plethora of challenges from learning to implementing solutions for examining gigantic amount of data and extracting information from it. Before taking this course, we were unaware of such challenges and the importance of big data in present and near future as every data producing application is somehow going to be a part of it.

We take this opportunity to express our sincere gratitude to our course instructor, our supervisor, Professor Dr. Tariq Mahmood, for his patience, insightful comments, invaluable suggestions, helpful information, practical advice and unceasing ideas which have helped us tremendously to understand big data as a layman's perspective and impart practical knowledge to implement a solution that can assess big data. His immense enthusiasm, profound experience and professional expertise in Big Data Analytics has inspired us to complete this project in a short span of time. We are thankful to him for his precious time in teaching us diligently, answering our queries on time and most of all encouraging us to participate in class discussions. We could not have imagined having a better instructor for this course.

Introduction

Data is growing at a large scale with high speed by various domains like social media, share market etc.; these domains may produce data in any of the forms: structured, semi structured or unstructured. Big Data provides various tools and techniques for efficient storing and processing of any kind of data. It is traditional data analysis. Big data can handle data sets with sizes beyond the ability of commonly used software tools to capture and process the data.

Weather prediction is one of the applications to predict the atmosphere of a given location. It has always been a big challenge for meteorologists to predict the status of the atmosphere and climatic conditions that may be expected. It remains quite obvious that knowing the climatic conditions earlier can play an important role for individuals and organizations. Accurate weather forecasts can help farmers to know the best time to plant, an airport control tower to send signals to planes that are landing and taking off etc.

In this project we are dealing with huge amount of unstructured weather data which has been collected from NCDC data center, here we can be able to work on historical as well as real world data where the Hadoop distributed file system is used for faster processing and compared that with the latest technique like Spark to know the processing speed. Hadoop MapReduce is one of the most widely used models for Big Data processing. Hadoop is an open source largescale data processing framework that supports distributed processing of large amount of data using simple programming models. The Apache Hadoop consists of the HDFS and MapReduce.

Apache Spark is an emerging technology in the Big Data field. It is 100 times faster than Hadoop MapReduce in most of the cases. Spark supports in-memory computing and it performs well with iterative algorithms, where the same code is executed multiple times.

Objective

Various technologies like Hadoop, Spark, Storm, NoSQL has evolved to address the challenges of Big Data. Out of this technology Hadoop Map Reduce which is efficient for batch processing and Spark which is efficient for iterative in-memory computing are the most prominent. It is important to study their relative performance and usefulness in various domains. In the current project, the weather data analytics will be done by calculation of minimum, maximum and average values of temperature. The code for analysis will be implemented using both Map Reduce and Spark. The benchmarking will be compared between these two methods on datasets of various sizes.

Weather Dataset

In this project we are dealing with huge amount of unstructured weather data which has been collected from NCDC data center. The format of dataset supports a rich set of meteorological elements, which are good candidate for analysis with big data because it is semi-structured and record oriented.

NCDC data center enables us to work on historical as well as real world data where the Hadoop distributed file system is used for faster processing and is compared to the latest technique like Apache Spark to know the processing speed.

To incorporate Big Data, we have analyzed the data of year 2011(~900MB).

Dataset can downloaded directly from: FTP -- ftp://ftp.ncdc.noaa.gov/pub/data/gsod

The following is a description of the global surface summary of day product produced by the National Climatic Data Center (NCDC) in Asheville, NC. The input data used in building these daily summaries are the Integrated Surface Data (ISD), which includes global data obtained from the USAF Climatology Center, located in the Federal Climate Complex with NCDC. The latest daily summary data are normally available 1-2 days after the date-time of the observations used in the daily summaries.

The online data files begin with 1929, and are now at the Version 7 software level. Over 9000 stations' data are typically available. The data are strictly ASCII, with a mixture of character data, real values, and integer values. Further, there is also a probability of error and missing values. For temperature, missing values are replaced with (9999.99 or -9999.99).

There are 28 daily elements included in the dataset for each station.

For details, visit https://www7.ncdc.noaa.gov/CDO/GSOD_DESC.txt

Following are the important fields with their data format taken from a set of 28 elements:

First record-- header record.
For eg. ("STATION","DATE","LATITUDE","LONGITUDE","ELEVATION","NAME","TEMP" etc.)

Field	**Position**	**Sub-string**	**Type**	**Description**
STATION	0	1-6	Int	Station number WMO/DATSAV3 number) for the location
DATE	1	15-18	Int	The date
TEMP	6	25-30	Real	Mean temperature for the day in degrees Fahrenheit to tenths. Missing = 9999.9
MAX	20	103-108	Real	Maximum temperature reported during the day in Fahrenheit. Missing = 9999.9
MIN	22	111-116	Real	Minimum temperature reported during the day in Fahrenheit. Missing = 9999.9

Analysis Usecase

The main goal is to present the analysis of weather data by calculating maximum, minimum and average value of temperatures. There are various features given in above mentioned dataset. We would like to find average temperature of a year 2011 by adding all the temperatures of each day of the year and then dividing it with the no. of counts.

Similarly, maximum temperature and minimum temperature fields are also given in dataset. Here, we find maximum temperature and minimum temperature in Fahrenheit respectively.

We perform this analysis on both Hadoop MapReduce and Apache Spark for two purposes. First, to obtain information and second, to identify the suitable technique for the operation.

System Architecture

The system architecture is divided into three parts for easy analysis:

Step 1: Data Acquisition

This unit involves two main phases:

i) Data Selection: This unit involves collection of data i.e., Real time data and historical Data from data centers like NCDC (National Climatic Data Center) which indeed store the data from satellite and different base stations. It is important to choose the data that can be used appropriately in your selected technique.

ii) Data Loading: After downloading the data from data center. We need to load it into Hadoop Distributed File System environment. Hadoop is the great tool to predict the climatic conditions, with processing of large and dynamic climate data and also it is feed to Apache Spark.

Step 2: Data Preprocessing

The collected data set contains inconsistent data, hence if weather analysis is performed on this data, it will produce wrong outcomes. Therefore, necessary preprocessing techniques are applied before analyzing and required features are extracted from the data sets. This will need to understand the data format and process it in a certain manner that the algorithms can be easily applied.

Step 3: Data Analysis

The acquired data set is now analyzed using Hadoop MapReduce Algorithms and Apache spark algorithms on historical and real data to predict the weather condition in the located areas. After which, we can infer that which technology gives us optimum results.

MapReduce Implementation

MapReduce Process

Mapper is used to run the block and perform simultaneous processing of each block. Mapper filters the matching records of particular location id or year and all the parameters are extracted and get saved into HDFS (Hadoop distributed file system) as key-value pairs. In this Mapper phase the memory is allocated only once for each record of key-value and the memory space is reused resulting in optimized memory allocation.

The combiner phase is used after mapper so that it can calculate local calculations like finding maximum, minimum and average temperatures based on parameters, resulting in reduction of network traffic and load on reducer.

The reducer phase is used to calculate global Maxima, Minima and average from different parameter fields like temperature, pressure, humidity and wind speed.

The resultant data is stored back to HDFS in sorted format.

System Requirements

To run mapreduce operation on a standalone system, there are some pre-requisites:

1) Ubuntu 18.04 or 20.04
2) Install OpenJDK v1.8
3) Install OpenSSH
4) Install Hadoop 3.2.1 in a Pseudo Environment

Following are the initial steps that are taken to setup Hadoop and Java on Ubuntu System:

```
Step 1: Check java version by running following command in your terminal.
If it says that "java variable is not defined"
then run command: "sudo apt install openjdk-8-jdk -y"
```

```
ramsha@ramsha:/$ java -version; javac -version
openjdk version "1.8.0_275"
OpenJDK Runtime Environment (build 1.8.0_275-8u275-b01-0ubuntu1~20.04-b01)
OpenJDK 64-Bit Server VM (build 25.275-b01, mixed mode)
javac 1.8.0_275
```

Step 2: Install OpenSSH server by running following command:
sudo apt install openssh-server openssh-client -y

```
ramsha@ramsha:~$ sudo apt install openssh-server openssh-client -y
Reading package lists... Done
Building dependency tree
Reading state information... Done
openssh-client is already the newest version (1:8.2p1-4ubuntu0.1).
The following additional packages will be installed:
  ncurses-term openssh-sftp-server ssh-import-id
```

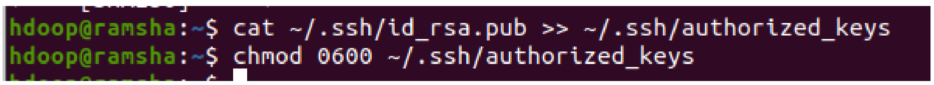
Step 3: Create a new non-root user by using command:
sudo adduser hdoop

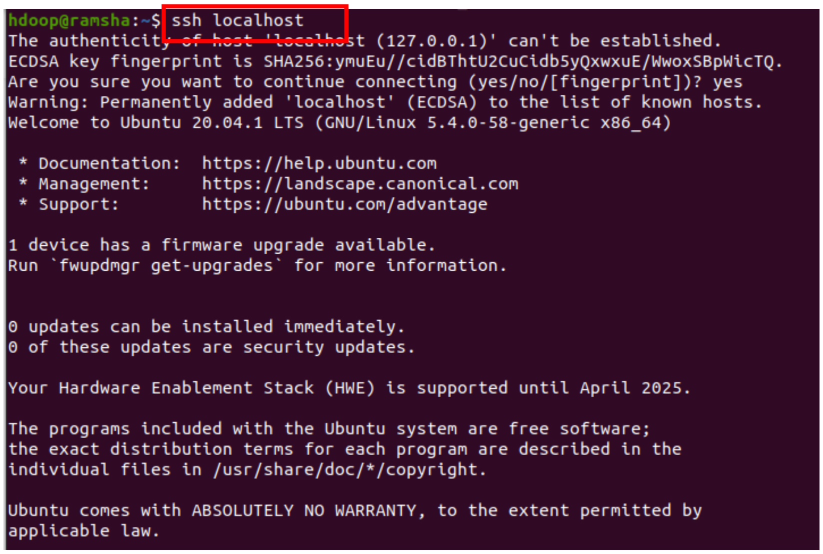
```
ramsha@ramsha:~$ sudo adduser hdoop
Adding user `hdoop' ...
Adding new group `hdoop' (1001) ...
Adding new user `hdoop' (1001) with group `hdoop' ...
Creating home directory `/home/hdoop' ...
Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for hdoop
Enter the new value, or press ENTER for the default
        Full Name []:
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n] Y
```

Step 4: Switch to newly created user "hdoop" then generate SSH key.
ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
Step 5: Use the cat command to store the public key as authorized_keys in the ssh directory:

```
ramsha@ramsha:~$ su - hdoop
Password:
hdoop@ramsha:~$ ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
Generating public/private rsa key pair.
Created directory '/home/hdoop/.ssh'.
Your identification has been saved in /home/hdoop/.ssh/id_rsa
Your public key has been saved in /home/hdoop/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:Uce77H28EmaSAkqNLcY+57Fxzo5A+GUF2W58YWB/ctU hdoop@ramsha
The key's randomart image is:
+---[RSA 3072]----+
|        .ooo..  ..|
|        .oo.+. . E|
|     . + +..o.+   |
|     .* +.= .=    |
|    .+.ooS ....   |
|     o+o+ o oo+   |
|      o+ B ..+... |
|       .o.o  ... o|
|        ...    .o.|
+----[SHA256]-----+
```

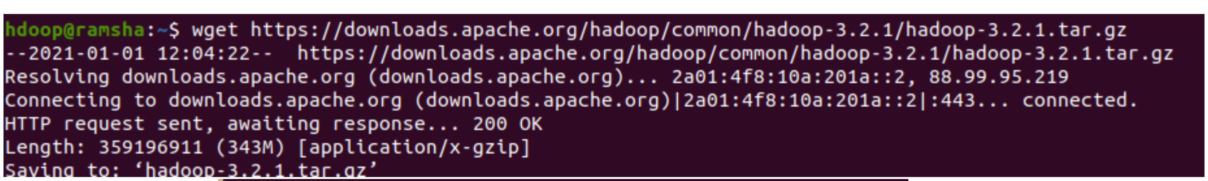
```
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
chmod 0600 ~/.ssh/authorized_keys


Then Verify everything is set up correctly by using the hdoop user t
o SSH to localhost:
Ssh localhost
```





Step 6: Now Download and install hadoop. It may take a while. After that, you need to unzip Hadoop folder.

```
wget https://downloads.apache.org/hadoop/common/hadoop-3.2.1/hadoop
-3.2.1.tar.gz
```



Step 7: Configure following hadoop files.

```
hdoop@ramsha:~$ nano .bashrc
hdoop@ramsha:~$ source ~/.bashrc
hdoop@ramsha:~$ nano $HADOOP_HOME/etc/hadoop/hadoop-env.sh
hdoop@ramsha:~$ nano $HADOOP_HOME/etc/hadoop/core-site.xml
hdoop@ramsha:~$ nano $HADOOP_HOME/etc/hadoop/hdfs-site.xm


Use "fg" to return to nano.

[1]+  Stopped                 nano $HADOOP_HOME/etc/hadoop/hdfs-site.xm
hdoop@ramsha:~$ nano $HADOOP_HOME/etc/hadoop/hdfs-site.xml
hdoop@ramsha:~$ nano $HADOOP_HOME/etc/hadoop/mapred-site.xml
hdoop@ramsha:~$ nano $HADOOP_HOME/etc/hadoop/yarn-site.xml
hdoop@ramsha:~$ hdfs namenode -format
```

Step 8: Navigate to the hadoop-3.2.1/sbin directory and execute the following commands to start the NameNode and DataNode:

./start-dfs.sh

Once the namenode, datanodes, and secondary namenode are up and running, start the YARN resource and nodemanagers by typing:

./start-yarn.sh

You may run command "jps" to ensure that required nodes are running.

```
hdoop@ramsha:~$ cd hadoop-3.2.1/sbin
hdoop@ramsha:~/hadoop-3.2.1/sbin$ ./start-dfs.sh
Starting namenodes on [localhost]
Starting datanodes
Starting secondary namenodes [ramsha]
ramsha: Warning: Permanently added 'ramsha' (ECDSA) to the list of known hosts.
hdoop@ramsha:~/hadoop-3.2.1/sbin$ ./start-yarn.sh
Starting resourcemanager
Starting nodemanagers
hdoop@ramsha:~/hadoop-3.2.1/sbin$ jps
27588 ResourceManager
27382 SecondaryNameNode
27752 NodeManager
28090 Jps
27164 DataNode
26991 NameNode
```

Step 9: Run on browser.

http://localhost:9870

localhost:9870/dfshealth.html#tab-overview

Summary

Security is off.

Safemode is off.

1 files and directories, 0 blocks (0 replicated blocks, 0 erasure coded block groups) = 1 total filesystem object(s).

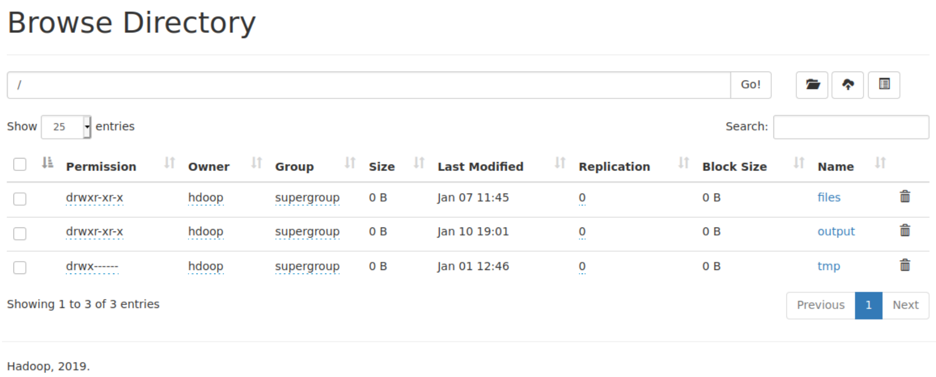
Heap Memory used 80.76 MB of 277.5 MB Heap Memory. Max Heap Memory is 1.7 GB.

Non Heap Memory used 47.33 MB of 48.46 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	14.58 GB
Configured Remote Capacity:	0 B
DFS Used:	24 KB (0%)
Non DFS Used:	3.88 GB
DFS Remaining:	9.94 GB (68.19%)
Block Pool Used:	24 KB (0%)
DataNodes usages% (Min/Median/Max/stdDev):	0.00% / 0.00% / 0.00% / 0.00%
Live Nodes	1 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion (including replicas)	0
Block Deletion Start Time	Fri Jan 01 12:22:24 +0500 2021
Last Checkpoint Time	Fri Jan 01 12:23:30 +0500 2021
Enabled Erasure Coding Policies	RS-6-3-1024k

Step 10: Once your Hadoop is running on browser.

Put your dataset in hdfs.
$ hdfs dfs – put <location/filename>
Here, my data has been uploaded in files folder.



Browse Directory

/ Go!

Show 25 entries Search:

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
drwxr-xr-x	hdoop	supergroup	0 B	Jan 07 11:45	0	0 B	files
drwxr-xr-x	hdoop	supergroup	0 B	Jan 10 19:01	0	0 B	output
drwx------	hdoop	supergroup	0 B	Jan 01 12:46	0	0 B	tmp

Showing 1 to 3 of 3 entries Previous 1 Next

Hadoop, 2019.

Step 11. Then run jar file to start mapreduce operation.
$ Hadoop jar <location/jarfile> <datasetOnHDFS> <output>

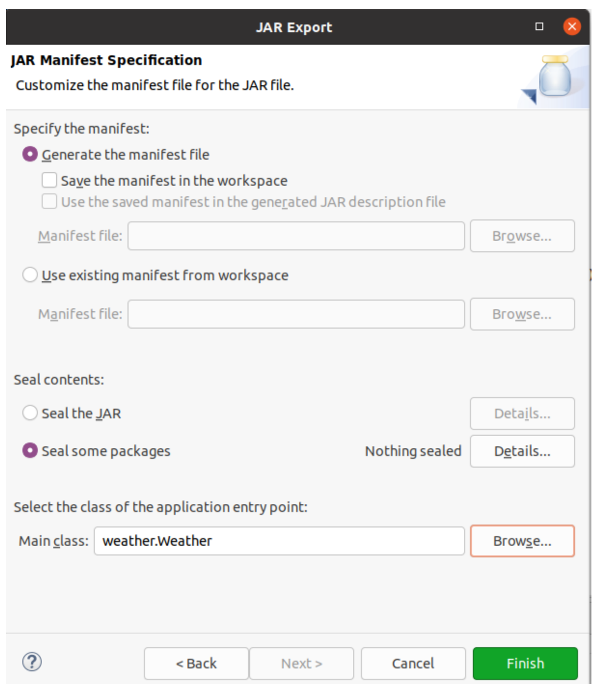
```
hdoop@ramsha:~/hadoop-3.2.1/sbin$ hadoop jar /home/ramsha/eclipse-workspace/weat
her.jar /files/2011/ /output
```

MapReduce Program

To perform this operation, we need to write a script where we will perform data preprocessing and mapreduce functions. Here, we are writing this script in Java using Eclipse IDE. After which we create .jar file that will be executed in Hadoop.

Steps to create a .jar file using Eclipse:

1. First Open **Eclipse** -> then select **File -> New -> Java Project** ->Name it **MyProject** -> then select **use an execution environment** -> choose **JavaSE-1.8** then **next** -> **Finish.**
2. In this Project Create Java class with name **Weather** -> then click **Finish.**
3. Copy the **code** given below to this **Weather** class
4. Now we need to add external jar files for the packages that we have to import. Download the jar package Hadoop Common and Hadoop MapReduce Core according to your Hadoop version. In my case, I have Hadoop-3.2.1 which is stable with JavaSE-1.8.
5. Now we add these external jars to our **MyProject**. Right Click on **MyProject** -> then then select **Build Path**-> Click on **Configure Build Path** and select **Add External jars** and add jars from its download location then click -> **Apply and Close**.
6. Now export the project as a jar file. Right-click on **MyProject** choose **Export.** Then go to **Java -> JAR file** click -> **Next** and choose your export destination then click -> **Next**.
   Choose Main Class as **Weather** by clicking -> **Browse** and then click -> **Finish** -> **Ok**.



Java Code

```
public class Weather extends Configured implements Tool {

  public static class MapClass extends MapReduceBase implements
Mapper<LongWritable, Text, Text, Text> {

      private Text word = new Text();
      private Text values = new Text();

      public void map(LongWritable key, Text value, OutputCollector<Text, Text> output,
Reporter reporter)
          throws IOException {
        String line = value.toString();
        /** NOTE: Each and every string replacement is important as there can be different
formatting or missing value in any file. */
        line = line.replaceAll("\\s+", "");  //removes extra spaces
        line = line.replaceAll("\"\"", "-"); //replace empty value "" to -
        line = line.replaceAll(",", "");    //replaces , to none
        line = line.replaceAll("\"", " ");  //replaces single " to single space
        line = line.replaceAll("\\s+", " "); //replaces further extra spaces to one single space
        StringTokenizer itr = new StringTokenizer(line);
        int counter = 0;
        String key_out = " ------------  Weather Data Analysis  ------------ ";
        String value_str = "";
        boolean skip = false;
        // This loop will read each line which 28 attribute. Each attribute can be read
through switch case.
        loop: while (itr.hasMoreTokens() && counter < 27) {
          String str = itr.nextToken();
          if (str.contains("STATION")) {            // removing header line
            return;
          }
          switch (counter) {
          case 6:// Temperature
            if (str.equals("9999.9")) {// Ignoring rows where temperature is all 9
              return;
            } else {
              value_str = str.concat(" ");
              break;
            }
          case 20:// max temp
            if (str.equals("9999.9") || str.equals("*")) {
              skip = true;
              break loop;
            } else {
              value_str = value_str.concat(str).concat(" ");
              break;
            }
          case 22:// min temp
            if (str.equals("9999.9") || str.equals("*")) {
              skip = true;
              break loop;
            } else {
              value_str = value_str.concat(str).concat(" ");
              break;
            }
          default:
            break;
          }
          counter++;
        }
        if (!skip) {
          word.set(key_out);
          values.set(value_str);
          output.collect(word, values);
        }
      }
    }
```

This is preprocessing of data. The data we got from ncdc was a little bit of unstructured and messy. Values were missing, additional spaces and extra characters like comma and inverted commas.

Before cleaning:

"00701899999","2011-03-09","0.0","0.0","7018.0","WXPOD 7018"," 64.2"," 9"," 50.2"," 9","9999.9"," 0","999.9"," 0"," 1.9"," 9"," 2.3"," 9"," 6.0","999.9"," 68.0","*"," 55.4","*","

After cleaning:

00701899999 2011-03-09 0.0 0.0 7018.0 WXPOD7018 64.2 9 50.2 9 9999.9 0 999.9 0 1.9 9 2.3 9 6.0 999.9 68.0 * 55.4 * 0.00 I 999.9 000000

Mapper Function:
A Mapper is used to run the block and perform simultaneous processing of each block.
Mapper filters the matching required record and stores it in a variable.

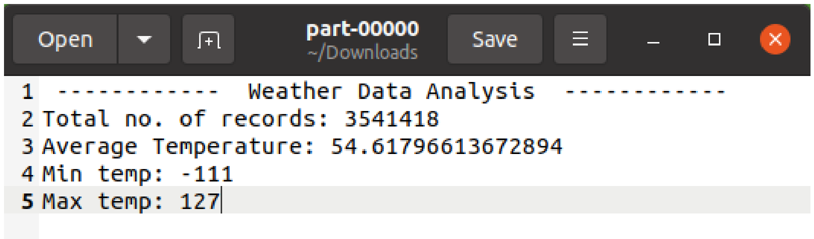
```
/**
   * Reducer Class for Job
   * A reducer class that just emits 3 attribute vector with average temperature ,
   * max temp, min temp for each input
   */
   public static class Reduce extends MapReduceBase implements Reducer<Text, Text,
Text, Text> {
      private Text value_out_text = new Text();
      private int max_temp = Integer.MIN_VALUE;
      private int min_temp = Integer.MAX_VALUE;
      private int temp1 = 0;
      private int temp2 = 0;

      public void reduce(Text key, Iterator<Text> values, OutputCollector<Text, Text>
output, Reporter reporter)
          throws IOException {
        double sum_temp = 0;
        int count = 0;

        while (values.hasNext()) {

          String str = values.next().toString();
          StringTokenizer itr = new StringTokenizer(str);
          int count_vector = 0;
          while (itr.hasMoreTokens()) {
            String nextToken = itr.nextToken(" ");
            if (count_vector == 0) {
              sum_temp += Double.parseDouble(nextToken);
            }
            if (count_vector == 1) {
              double val1 = Double.parseDouble(nextToken);
              temp1 = (int) (val1);
              if (temp1 > max_temp) {
                max_temp = temp1;
              }
            }
            if (count_vector == 2) {
              double val2 = Double.parseDouble(nextToken);
              temp2 = (int) (val2);
              if (temp2 < min_temp) {
                min_temp = temp2;
              }
            }
            count_vector++;
          }
          count++;
        }

        if (sum_temp == 0) {
          return;
        } else {
          double avg_tmp = sum_temp / count;
          System.out.println(key.toString() + " count is " + count + " sum of temp is " +
sum_temp + "");

          String value_out = "\nTotal no. of records: " + count + "\nAverage Temperature: "
              + String.valueOf(avg_tmp) + "\nMin temp: " + String.valueOf(min_temp) +
"\nMax temp: "
              + String.valueOf(max_temp);
          value_out_text.set(value_out);
          output.collect(key, value_out_text);
        }

      }
   }
```

Reducer Function:
A Reducer is used to calculate average temperature. Also, finds out maximum and minimum temperature. Stores it in particular variables then writes an output.

```
static int printUsage() {
    System.out.println("weather [-m <maps>] [-r <reduces>] <job_1 input> <job_1
output>");
    ToolRunner.printGenericCommandUsage(System.out);
    return -1;
  }

  /**
   * The main driver for weather map/reduce program. Invoke this method to submit
   * the map/reduce job.
   *
   * @throws IOException When there is communication problems with the job
   *                     tracker.
   */
  public int run(String[] args) throws Exception {
    Configuration config = getConf();
    JobConf conf = new JobConf(config, Weather.class);

    conf.setJobName("Weather Job1");

    // the keys are words (strings)
    conf.setOutputKeyClass(Text.class);
    // the values are counts (ints)
    conf.setOutputValueClass(Text.class);

    conf.setMapOutputKeyClass(Text.class);
    conf.setMapOutputValueClass(Text.class);

    conf.setMapperClass(MapClass.class);
    conf.setReducerClass(Reduce.class);
    List<String> other_args = new ArrayList<String>();

    for (int i = 0; i < args.length; ++i) {
      try {
        if ("-m".equals(args[i])) {
          conf.setNumMapTasks(Integer.parseInt(args[++i]));
        } else if ("-r".equals(args[i])) {
          conf.setNumReduceTasks(Integer.parseInt(args[++i]));
        } else {
          other_args.add(args[i]);
        }
      } catch (NumberFormatException except) {
        System.out.println("ERROR: Integer expected instead of " + args[i]);
        return printUsage();
      } catch (ArrayIndexOutOfBoundsException except) {
        System.out.println("ERROR: Required parameter missing from " + args[i - 1]);
        return printUsage();
      }
    }

    FileInputFormat.setInputPaths(conf, other_args.get(0));
    FileOutputFormat.setOutputPath(conf, new Path(other_args.get(1)));

    JobClient.runJob(conf);

    return 0;
  }

  public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new Configuration(), new Weather(), args);
    System.exit(res);
  }

}
```

MapReduce Output

Below given is an output file downloaded from HDFS after the completion of MapReduce job.




```
 ------------  Weather Data Analysis  ------------
Total no. of records: 3541418
Average Temperature: 54.61796613672894
Min temp: -111
Max temp: 127
```

Spark Implementation

Apache Spark is a fast and powerful framework that provides an API to perform massive distributed processing over resilient sets of data. The main abstraction Spark provides is a resilient distributed data set (RDD), which is the fundamental and backbone data type of this engine. Spark SQL is Apache Spark's module for working with structured data and MLlib is Apache Spark's scalable machine learning library. Apache Spark is written in Scala programming language. To support Python with Spark, the Apache Spark community released a tool, PySpark. PySpark has similar computation speed and power as Scala. PySpark is a parallel and distributed engine for running big data applications. Using PySpark, you can work with RDDs in Python programming language.

Spark and Hadoop environment setup

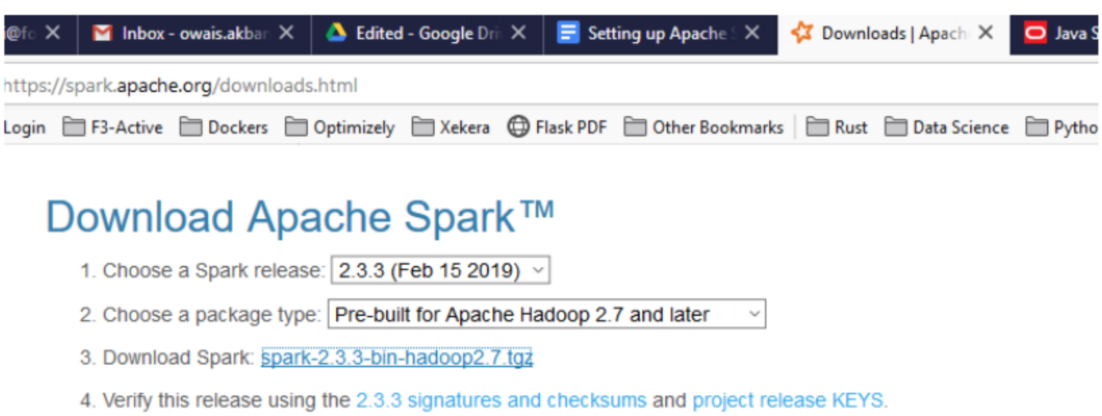
Install JDK. Java 8 is a prerequisite for working with Apache Spark. Spark runs on top of Scala and Scala requires Java Virtual Machine to execute.

Download JDK 8 based on your system requirements and run the installer. Ensure to install Java to a path that doesn't contains spaces. For the purpose of this blog, we change the default installation location to c:\jdk (Earlier versions of spark cause trouble with spaces in paths of program files). The same applies when the installer proceeds to install JRE. Change the default installation location to c:\jre.

Install Spark

Download the pre-built version of Apache Spark 2.4.07. The package downloaded will be packed as tgz file. Please extract the file using any utility such as WinRar.

Once unpacked, copy all the contents of unpacked folder and paste to a new location: c:\spark.

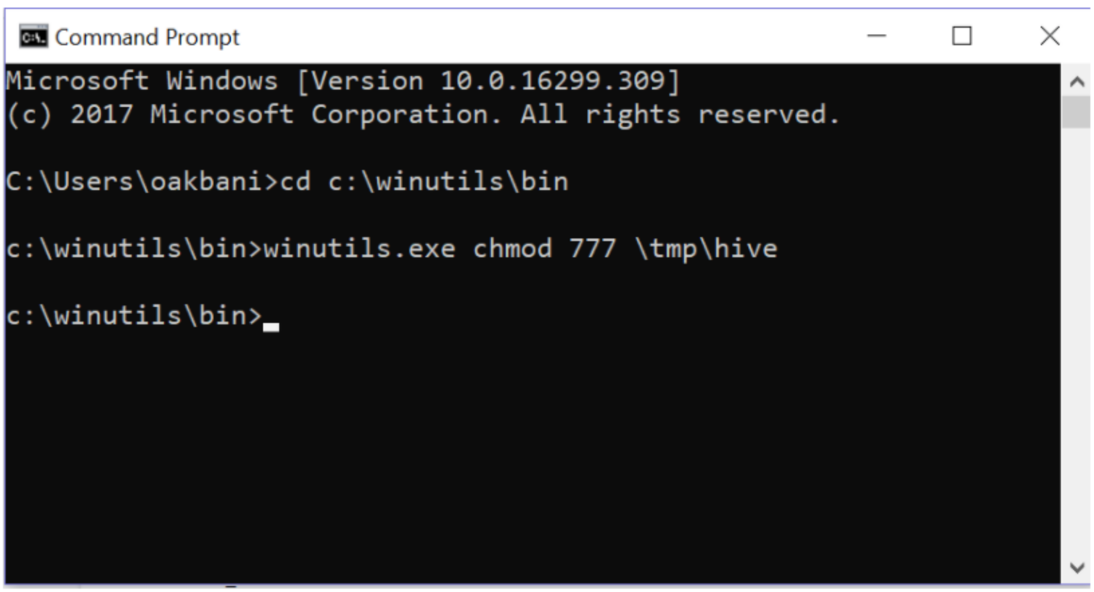


Install winutils.exe

Spark uses Hadoop internally for file system access. Even if you are not working with Hadoop (or only using Spark for local development), Windows still needs Hadoop to initialize "Hive" context, otherwise Java will throw java.io.IOException. This can be fixed by adding a dummy Hadoop installation that tricks Windows to believe that Hadoop is actually installed.

Download Hadoop 2.7 winutils.exe. Create a directory winutils with subdirectory bin and copy downloaded winutils.exe into it such that its path becomes: c:\winutils\bin\winutils.exe.

Spark SQL supports Apache Hive using HiveContext. Apache Hive is a data warehouse software meant for analyzing and querying large datasets, which are principally stored on Hadoop Files using SQL-like queries. HiveContext is a specialized SQLContext to work with Hive in Spark. The next step is to change access permissions to c:\tmp\hive directory using winutils.exe.



Setting up Environment Variables

The final step is to set up some environment variables.

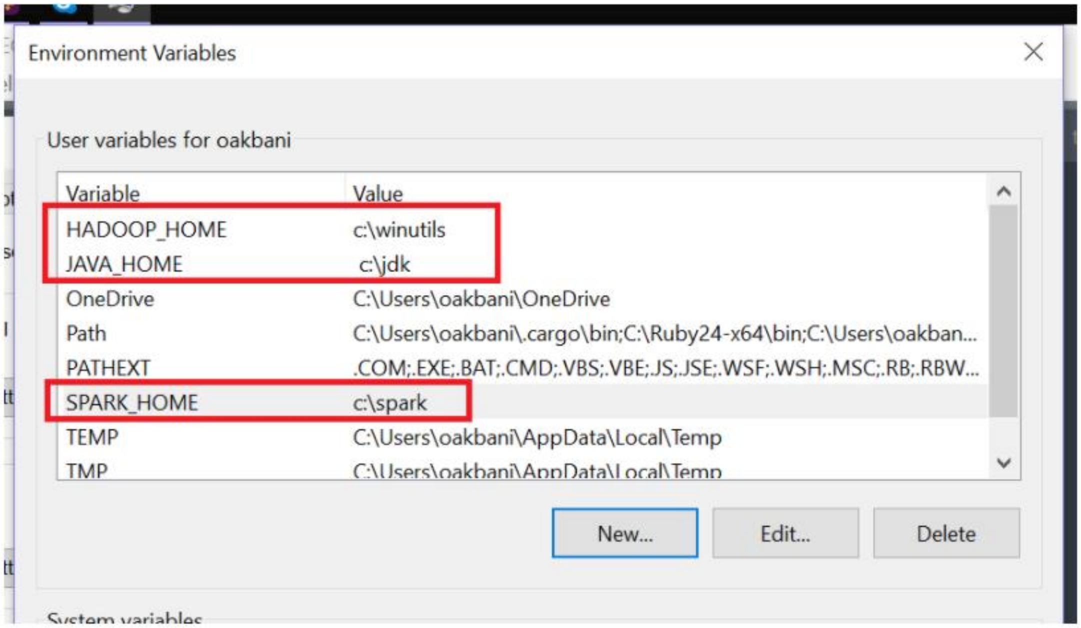
From start menu, go to Control Panel > System > Advanced System Settings and click on Environment variables button from the dialog box.

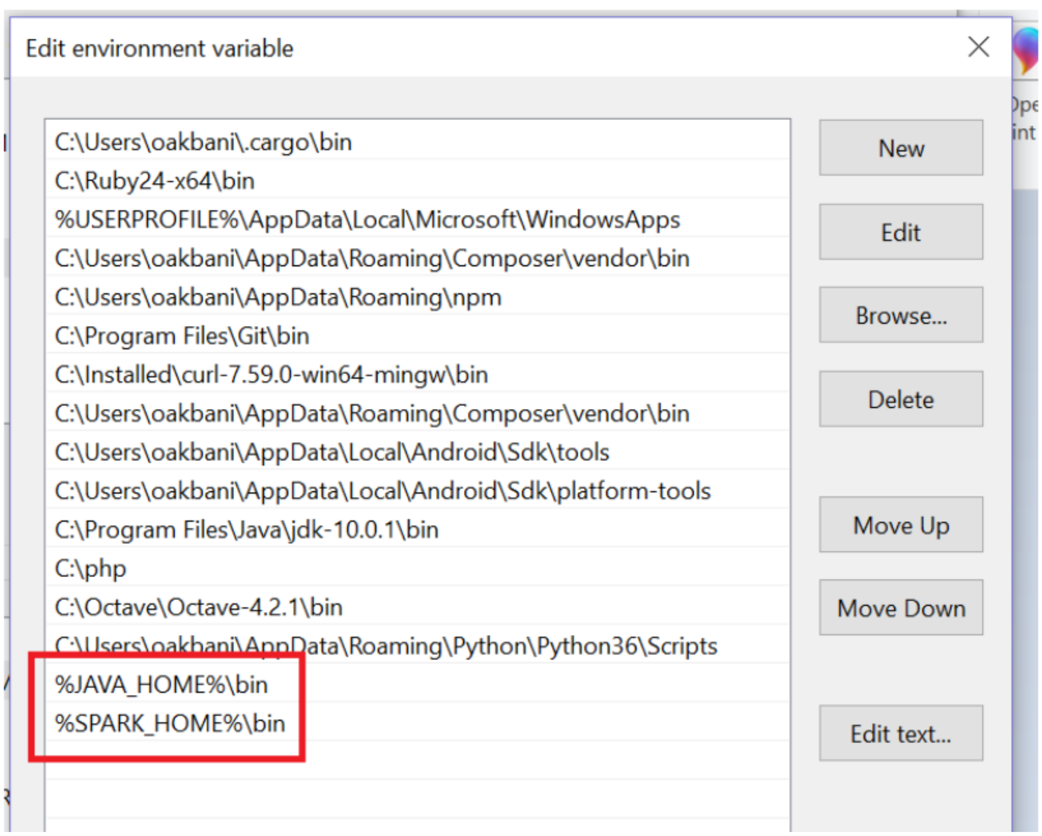
Under the user variables, add three new variables:

JAVA_HOME: c:\jdk

SPARK_HOME: c:\spark

HADOOP_HOME: c:\winutils



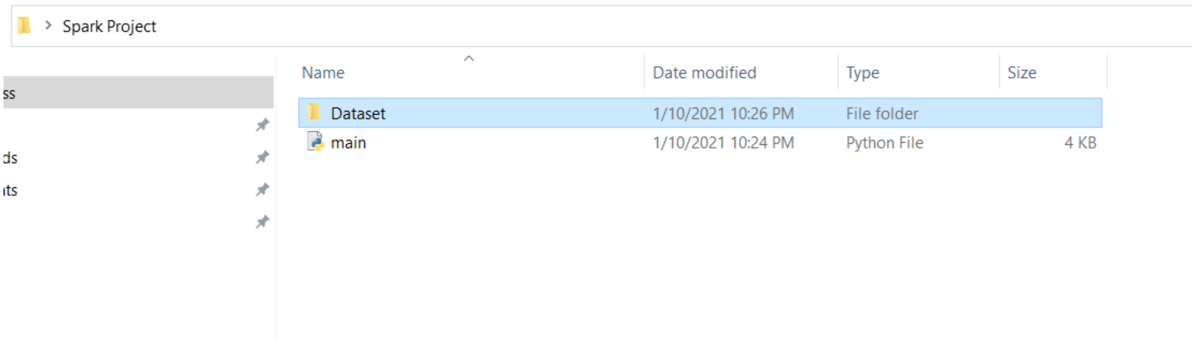


Creating Files.

Create a new folder Spark Project and add sub folder – Dataset

Under Dataset add year folder example 2011, and Add CSV files according to year in appropriate folder.

Add python file to the Spark Project Folder.

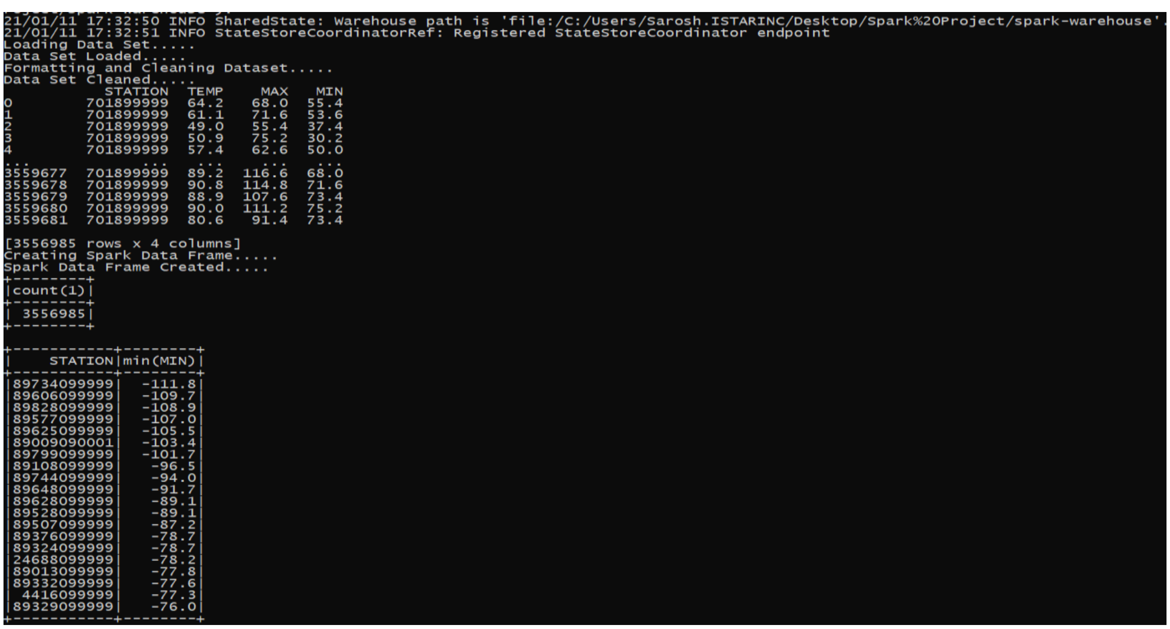


Submit Spark Job

Spark-Submit <python file>

```
C:\Users\Sarosh.ISTARINC\Desktop\Spark Project>spark-submit main.py
```

Output:



```
21/01/11 17:32:50 INFO SharedState: Warehouse path is 'file:/C:/Users/Sarosh.ISTARINC/Desktop/Spark%20Project/spark-warehouse'.
21/01/11 17:32:51 INFO StateStoreCoordinatorRef: Registered StateStoreCoordinator endpoint
Loading Data Set.....
Data Set Loaded.....
Formatting and Cleaning Dataset.....
Data Set Cleaned.....
           STATION  TEMP    MAX   MIN
0        701899999  64.2   68.0  55.4
1        701899999  61.1   71.6  53.6
2        701899999  49.0   55.4  37.4
3        701899999  50.9   75.2  30.2
4        701899999  57.4   62.6  50.0
...            ...   ...    ...   ...
3559677  701899999  89.2  116.6  68.0
3559678  701899999  90.8  114.8  71.6
3559679  701899999  88.9  107.6  73.4
3559680  701899999  90.0  111.2  75.2
3559681  701899999  80.6   91.4  73.4

[3556985 rows x 4 columns]
Creating Spark Data Frame.....
Spark Data Frame Created.....
+--------+
|count(1)|
+--------+
| 3556985|
+--------+

+-----------+--------+
|    STATION|min(MIN)|
+-----------+--------+
|89734099999|  -111.8|
|89606099999|  -109.7|
|89828099999|  -108.9|
|89577099999|  -107.0|
|89625099999|  -105.5|
|89009090001|  -103.4|
|89799099999|  -101.7|
|89108099999|   -96.5|
|89744099999|   -94.0|
|89648099999|   -91.7|
|89628099999|   -89.1|
|89528099999|   -89.1|
|89507099999|   -87.2|
|89376099999|   -78.7|
|89324099999|   -78.7|
|24688099999|   -78.2|
|89013099999|   -77.8|
|89332099999|   -77.6|
| 4416099999|   -77.3|
|89329099999|   -76.0|
+-----------+--------+
```

```
only showing top 20 rows

+-----------+--------+
|    STATION|max(MAX)|
+-----------+--------+
|72029304989|   131.0|
|40635599999|   127.4|
|40833099999|   126.7|
|40672099999|   126.1|
|40551099999|   126.0|
|40689099999|   125.6|
|40794099999|   125.2|
|40831099999|   124.9|
|40811099999|   124.9|
|40550099999|   124.7|
|40676099999|   124.5|
|40664099999|   124.5|
|40587099999|   124.2|
|40586099999|   123.8|
|68300799999|   123.8|
|41749099999|   123.8|
|40650099999|   123.8|
|60640099999|   123.8|
|41715099999|   123.8|
|40780199999|   123.8|
+-----------+--------+
only showing top 20 rows

+-----------+------------------+
|    STATION|         avg(TEMP)|
+-----------+------------------+
| 1469099999| 43.30328768063082|
| 2271099999| 37.80904103991104|
| 2987099999|44.388767054146285|
| 3649099999|52.034520543764714|
| 3882099999|51.926575428165805|
| 6464099999| 52.60876703131689|
| 7255099999|55.290411032062686|
| 7610099999| 57.20904107289771|
|12100099999| 48.59716723931072|
|16022099999|38.338082242991824|
|16726099999| 62.90383563107007|
|26238099999| 45.04368118057539|
|40729099999|54.097260172073156|
|41287099999| 79.71227548793405|
|41685099999| 70.60724231858107|
|47686099999| 58.55561630039999|
|47843099999| 62.27917814907963|
|47870099999| 65.46401093032334|
|52818099999| 44.30739724930019|
|56838099999| 69.74054785950543|
+-----------+------------------+
only showing top 20 rows
```

Code:

```
from pyspark import SparkContext, SparkConf
from pyspark.sql import SparkSession
from geopy.geocoders import Nominatim
import os
import pyspark.sql.functions as sqlf
from pyspark.sql.types import *
import time


import pandas as pd
import glob
sc = SparkContext.getOrCreate(SparkConf().setMaster("local[*]"))
spark = SparkSession.builder.appName('pandasToSparkDF').getOrCreate()
sc.setLogLevel("ERROR")


def create_spark_dataframes(year):
    #loading the dataset
    print('Loading Data Set.....')
    path = f'Dataset/{year}'
    w_data_columns = ["STATION", "TEMP", "MAX", "MIN"]
    all_files = glob.glob(os.path.join(path, "*.csv"))
    df_from_each_file = (pd.read_csv(f, usecols=w_data_columns) for f in
all_files)
    df_weather = pd.concat(df_from_each_file, ignore_index=True)
    print('Data Set Loaded.....')


    #data preprocessing
    print('Formatting and Cleaning Dataset.....')

    df_weather = df_weather.dropna()
    df_weather = df_weather.drop(df_weather[df_weather.MAX ==
9999.9].index)

    print('Data Set Cleaned.....')
    print(df_weather)
    w_schema = StructType([StructField(w_data_columns[0],
StringType(),True),
                          StructField(w_data_columns[1],
FloatType(),True),
                          StructField(w_data_columns[2],
FloatType(),True),
                          StructField(w_data_columns[3], FloatType(),
True)
                           ]
                          )
    print('Creating Spark Data Frame.....')

    df_spark_weather = spark.createDataFrame(df_weather, schema=w_schema)

    print('Spark Data Frame Created.....')

    return df_spark_weather
```

```
def get_min_temp(df_weather):

    cold =
df_weather.groupBy(df_weather.STATION).agg(sqlf.min('MIN')).sort(sqlf.asc(
'min(MIN)')).show()


def get_max_temp(df_weather):
    hot =
df_weather.groupBy(df_weather.STATION).agg(sqlf.max('MAX')).sort(sqlf.desc
('max(MAX)')).show()


def avg_temp(df_weather):
    avg =
df_weather.groupBy(df_weather.STATION).agg(sqlf.avg('TEMP')).show()


if __name__ == '__main__':
    start_time = time.time()
    df_weather_spark = create_spark_dataframes('2011')
    df_weather_spark.createOrReplaceTempView("weather")
    count = spark.sql('select count(*) from weather')
    count.show()
    get_min_temp(df_weather_spark)
    get_max_temp(df_weather_spark)
    avg_temp(df_weather_spark)
    end_time = time.time()
    time_elapsed = (f'Total Time Taken {end_time - start_time}')
    print(time_elapsed)

# See PyCharm help at https://www.jetbrains.com/help/pycharm/
```

Conclusion

The conclusion of our project is that the Meteorological department of every country collects large amount of weather data everyday which has been generated by satellites, storing and processing of this large amount of data becomes very challenging. In our project we worked on major parameters like average, maximum and minimum temperature of each day using MapReduce and Spark.

Moreover, after performing a comparative analysis of both spark and Hadoop MapReduce framework with respect to time used and analyzed the performance. Spark job took few minutes to give output while, MapReduce job took hours to complete.

Below given are the results:

	MapReduce Job	Spark Job
Time	~ 4 hours	~ 5 minutes
Records read	3541418	3556985
Average	54.6	43.3
Max Temp	127	131
Min Temp	-111	-111

From aforementioned analysis of weather data carried from National Climate Data Centre, we can conclude that Spark is far more efficient in terms of producing accurate results and costing time than MapReduce. Hence, for big data analysis and running multiple jobs, Spark would be an optimum choice.