Experiment No:

Date:

Aim:- Write a MapReduce program that mines weather data sensors collecting data every hour at many locations across the globe gather a large volume of log data, which is good candidate for analysis with MapReduce, since it is semi structured and record oriented.

Description:- MapReduce programme: Find Highest Temperature for each year in NCDC dataset

MapReduce is a programming model designed for processing large volumes of data in parallel by dividing the work into a set of independent tasks. Our previous traversal has given as introduction about MapReduce.

This traversal explains how to design a MapReduce program. The aim of the program is to find the Maximum Temperature recorded for each year of NCDC data.

The input for our program is weather data files for each year. This weather data is collected by National Climatic Data center - NCDC from weather

sensors at all over the world. You can find weather data for each year from "ftp://ftp.ncdc.noaa.gov/pub/data/noaa/". All files are zipped by year and the weather station. For each year there are multiple files for different weather stations. Here is an example for 1990 (ftp://ftp.ncdc.noaa.gov/pub/data/noaa/1901/)

→ 010080-99999-1990.gz
→ 010100-99999-1990.gz
→ 010150-99999-1990.gz
- - - - - - - - - - -

In our traversal, we consider only one weather data file for each year. NCDC input data used in our program can be downloaded from the link "https://github.com/lmsamaranawickrama/Hadoop-Reduce"

If we consider the details mentioned in the file, each file has entries that look like this.

0029029070999991950101060004+64333+023450FM-12+000599999V020230IN0082199999999N00000
01N9-01391+99999102641ADDGF102991999999
9999999999

when we consider the highlighted fields, the first one (029070) is the USAF weather station identifier. The next (19050101) represents the observation data. The third highlighted one (-0139) represents the air temperature in celsius times ten. So the reading of -0139 equates to -13.9 degree celsius. The next highlighted an Italic item indicates a reading quality code.

MapReduce is based on set of key value pairs. So first we have to decide on the types for the key/value pairs for the input.

Map Phase :- The input for Map phase is set of weather data files as shown. The types of input key value pairs as LongWritable and Text and the types of output key value pairs are Text and IntWritable. Each map task extracts the temperature data from the given year file. The output of the major phase is set to key value pairs. set of keys are the years. values are the temperature of each year.

Reduce phase:- Reduce phase takes all the values associated with a particular key. That is all the temperature values belong to a particular year is fed to a same reducer. Then each reducer find the highest recorded temperature for each year. The types of output key value pairs in Map phase is same for the types of input key value pairs in reduce phase (Text and IntWritable). The types of output key value pairs in reduce phase is too Text and IntWritable.

So, in this example we write the java classes

(1) HighestMapper.java

(2) HighestReducer.java

(3) HighestDriver.java

HighestMapper.java :-

```
import java.io.IOException;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
Public class HighestMapper Extends MapReduceBase
implements Mapper<longWritable, Text, Text, IntWritable>
{
  Public static final int MISSING=9999;
  Public void map(LongWritable key, Text value,
      Content.content) throws IOException,
      InterruptedException
  {
    String line = value.toString();
    String year = line.substring(15,19);
    int temperature;
    if(line.charAt(87)=='+')
      temperature = Integer.parseInt(line.substring
                                    (88,92));
    else
      temperature = Integer.parseInt(line.substring
                                    (87,92));
```

```
String quality = line.substring(92,93);
if(temperature != MISSING && quality.matches
                                ("[01459]")):
output.collect(new Text(year), new IntWritable
                                (temperature));
  }
}
```

HighestReducer.java :-

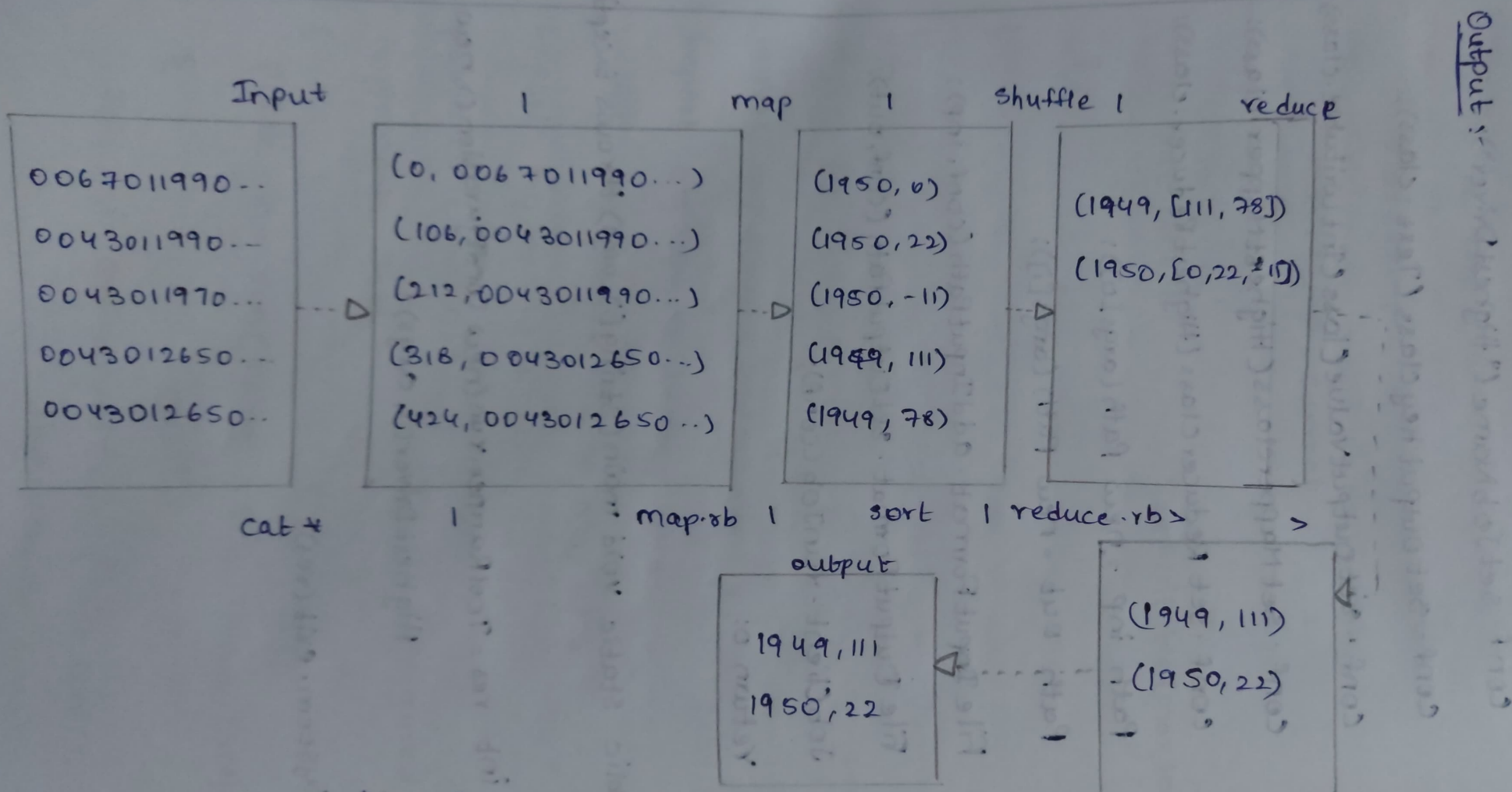
```
import java.io.IOException;
import java.util.Iterator;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
Public class HighestReducer Extends MapReduceBase
 implements Reduce < Text, IntWritable, Text, IntWritable>
{
  Public void reduce (Text Key, Iterator <IntWritable>
  values, OutputCollector < Text, IntWritable> context
  context)
  {
    int max-temp = 0
    while (values.hasNext())
    {
```

```
    int current = values.next().get();
    if (max_temp < current)
        max_temp = current;
    }
    output.collect(key, new IntWritable(max_temp/10));
  }
}
```

HighestDriver.java :

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;
public class HighestDriver Extends Configured
implements Tool
{
  Public int run(String[] args) throws Exception
  {
    JobConf conf = new JobConf(getConf(),
                    HighestDriver.class);
```

```
conf.setJobName("HighestDriver");
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);
conf.setMapperclass(HighestMapper.class);
conf.setReducerClass(HighestReducer.class);
Path inp = new Path(args[0]);
Path out = new Path(args[1]);
FileInputFormat.addInputPath(conf, inp);
FileOutputFormat.setOutputPath(conf, out);
JobClient.runJob(conf);
Return 0;
}
Public static void main(String[], args) throws Exception
{
  int res = ToolRunner.run(new Configuration(), new
          HighestDriver(), args);
  System.exit(res);
}
}
```



Map Reduce' Logical 'data Flow

$hadoop jar maxtemp1.jar com.tem.MaxTemperature /user/nagaakhil1519/nc.txt /usr/nagaakhil1519/output

$hadoop fs -ls /usr/nagaakhil1519/output

found 2 items

-rw-r--r-- 3 /usr/nagaakhil1519/output/_success

-rw-r--r-- 3 /usr/nagaakhil1519/output/part-r-00000

$hadoop fs -cat /usr/nagaakhil1519/output/part-r-00000

1901 317

1902 244

1903 289

1904 256

1905 286

1906 294

1907 283

1908 289

1909 298

1910 294

Experiment No:

Date:

Aim:- Implement Matrix Multiplication with Hadoop MapReduce

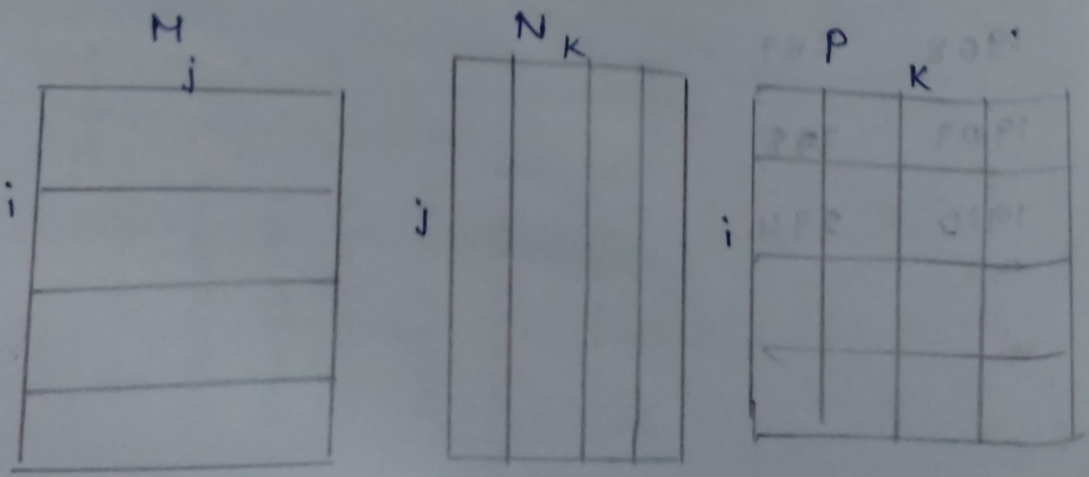
Description:-

Matrix Multiplication with MapReduce:-

Matrix-vector and matrix-matrix calculations fit nicely into the MapReduce style of computing. In this post I will only examines matrix-matrix calculations as.

Suppose we have a $p \times q$ matrix M, whose element in row i and column j will be denoted m_{ij} and a $q \times r$ matrix N whose element in row j and column k is denoted by n_{jk} then the product $P = MN$ will be $p \times r$ matrix P whose element in row i and column k will be denoted by p_{ik} where

$P(i,k) = m_{ij} * n_{jk}$



Matrix Data Model for MapReduce :- We represent matrix M as a relation M(I, J, V) with tuples (i, j, m_{ij}) and matrix N as a relation N(J, K, W) with tuples (j, K, n_{jK}). Most matrices are sparse so large amount of cells have zero value. When we represent matrices in this form, we donot need to keep entries for the cells that have values of zero to save large amount of disk space. As input data files, we store matrix M and N on HDFS in following format:

M, i, j, m_{ij}	N, j, K, n_{jK}
M, 0, 0, 10.0	N, 0, 0, 1.0
M, 0, 2, 9.0	N, 0, 2, 3.0
M, 0, 3, 9.0	N, 0, 4, 2.0
M, 1, 0, -1.0	N, 1, 0, 2.0
M, 1, 1, 3.0	N, 3, 2, -1.0
M, 1, 2, 18.0	N, 3, 6, 4.0
M, 1, 3, 25.2	N, 4, 0, -1.0

MapReduce :- We will write, Map and Reduce functions to process i/p files. Map function will produce key, value pairs from the input data as it is described in Algorithm1. Reduce function uses the output of

Map function and performs the calculations and produces key, value pairs as described in Algorithm 2

Algorithm 1 :- The Map function

1) for each element m_{ij} of M do

2) Produce (key, value) pairs as $((i,k),(M,j,m_{ij}))$, for $k = 1,2,3...$ up to the number of columns of N

3) for each element n_{jk} of N do

4) Produce (key, value) pairs as $((i,k),(N,j,n_{jk}))$, for $i = 1,2,3...$ upto the number of rows of M

5) Return set of (key, value) pairs that each key (i,k) has a list with values (M,j,m_{ij}) and (N,j,n_{jk}) for all possible values of j.

Algorithm 2 :- The Reduce function

1) for each key (i,k) do

2) sort values begin with M by j in list M

3) sort values begin with N by j in list N

4) multiply m_{ij} and n_{jk} for jth value of each list

5) sum up $m_{ij} * n_{jk}$

6) return $(i,k), \sum_{j=1} m_{ij} * n_{jk}$

.

.

.

The value in row i and column K of product matrix P will be:

$$P(i,k) = \sum_{j=1} m_{ij} * n_{jk}$$

Suppose we have two matrices, M, 2×3 matrix & N, 3×2 matrix as follows.

The product P of MN will be as follows

$$\begin{bmatrix} 1a+2c+3e & 1b+2d+3f \\ 4a+5c+6e & 4b+5d+6f \end{bmatrix}$$ where

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad N = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$$

The Map Task:- For matrix M, map task (algorithm) produce key, value as follows (i,k), (M,j,m,j)

$m_{11} = 1 \Rightarrow (1,1), (M,1,1)$ for $k=1$

$(1,2), (M,1,1)$ for $k=2$

$m_{12} = 2 \Rightarrow (1,1), (M,2,2)$ for $K=1$

$(1,2), (M,2,2)$ for $K=2$

.....

$m_{23} = 6 \Rightarrow (2,1), (M,3,6)$ for $K=1$

$(2,2), (M,3,6)$ for $K=2$

For Matrix N, map task (algorithm 2) produce key, value pairs as follows (i,k), (N,j, n_{jk})

$n_{11} = a \Rightarrow$ (1,1), (N,1,a) for i=1

(2,1), (N,1,a) for i=2

$n_{21} = c \Rightarrow$ (1,1), (N,2,c) for i=1

(2,1), (N,2,c) for i=2

- - - - - - - - -

$n_{32} = f \Rightarrow$ (1,2), (N,3,f) for i=1

(2,2), (N,3,f) for i=2

After combine operation the map task will return key, value pairs as follows

((i,k), [(M,j,m_{ij}); (M,j,m_{ij}) ... (N,j,n_{jk}) (N,j,n_{jk}) ...])

((1,1), [(M,1,1) (M,2,2), (M,3,3), (N,1,b), (N,2,d), (N,3,f)])

((1,2), [(M,1,1) (M,2,2), (M,3,3), (N,1,b), (N,2,d), (N,3,f)])

((2,1), [(M,1,4), (M,2,5), (M,3,6), (N,1,a), (N,2,c), (N,3,e)])

((2,2), [(M,1,4), (M,2,5), (M,3,6), (N,1,b), (N,2,d), (N,3,f)])

The Reduce Task : Reduce Task takes the key, value pairs as the input and process one key at a time For each key it divides the values in two seperate lists for M and N. As an example it

will create the following list for key

(1,1): [(M,1,1), (M,2,2), (M,3,3), (N,1,a), (N,2,c), (N,3,e)]

Reduce task sorts the values begin with M in one list and values begin with N in another list as follows:

I_M = [(M,1,1), (M,2,2), (M,3,3)]

I_N = [(N,1,a), (N,2,c), (N,3,e)]

then sums up the multiplication of m_{ij} and n_{jx} for each j as follows

$$P(1,1) = 1a + 2c + 3e$$

The same computation applied to all input entries of reduce task

$$P(i,k) = \sum_{j=1} m_{ij} * n_{jk}$$

for all i and k is then calculated as follows

$P(1,1) = 1a + 2c + 3e$

$P(1,2) = 1b + 2d + 3f$

$P(2,1) = 4a + 5c + 6e$

$P(2,2) = 4b + 5d + 6f$

The product of matrix P of MN is then generated as:

$$\begin{bmatrix} 1a+2c+3e & 1b+2d+3f \\ 4a+5c+6e & 4b+5d+6f \end{bmatrix} \quad \cdots (2)$$

So, in this we write three java classes

- MatrixMultiply.java
- Map.java
- Reduce.java

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat
Public class MatrixMultiplication
{
  Public static void main(String[] args) throws
  Exception
  {
    Configuration conf = new Configuration();
    conf.set("m", "2");
```

```
conf.set("m","5");
conf.set("p","3");
Job job = new Job(conf,"MatrixMultiplication");
job.setJarByClass(MatrixMultiplication.class);
job.setOutputKeyClass(Text.class);
job.setMapperClass(MatrixMapper.class);
job.setReducerClass(MatrixReducer.class);
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
job.waitForCompletion(true);
}
}
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
Public class MatrixMapper Extends Mapper <Long Writable,
  Text, Text, Text>
{
```

```
public void map (LongWritable key, Text value, Context
context) throws IOException, InterruptedException
{
  Configuration conf = context.getConfiguration();
  int m = Integer.parseInt(conf.get("m"));
  int p = Integer.parseInt(conf.get("p"));
  String line = value.toString();
  String[] indicesAndValue = line.split(",");
  Text Output Key = new Text();
  Text Output Value = new Text();
  if (indicesAndValue[0].equals("A"))
  {
    for(int k=0; k<p; k++)
    {
      OutputKey.set(indicesAndValue[1]+",",+k);
      OutputValue.set("A",+indicesAndValue[2]+
      ",",+ indicesAndValue[3]);
      context.write(output key, outputvalue);
    }
  }
  else
  {
    for(int i=0 ; i<m; i++)
    {
```

```
        Outputkey.set(i+",",+indicesAndValue[2]);
        Output key.set("B,"+ indicesAndValue[i]+
        ","+ indicesAndValues[3]);
        context.write(Output key, OutputValue);
        }
      }
    }
}
import java.io.IOException;
import java.util.HashMap;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.Mapreduce.Reducer;
Public class MatrixReducer Extends Reducer<Text,
Text, Text, Text>
{
  Public void reduce (Text key, Iterable <Text>values,
  Context context) throws IOException, Interputed
  Exception
  {
    String[] value;
    HashMap < Integer, Float > hashA = new HashMap
    <Integer, Float>();
    HashMap<Integer, Float> hashB = new HashMap
    <Integer, Float>();
```

```
for (Text val : values)
{
    value = val.toString().split(",");
    if (value[0].equals("A"))
    {
        hashA.put(Integer.parseInt(value[1]), Float.
        ParseFloat(value[2]));
    }
    else
    {
        hashB.put(Integer.parseInt(value[1]),
            Float.parseFloat(value[2]));
    }
}
int n= Integer.parseInt(context.getConfiguration().
        get("n"));
float result = 0.0f;
float a_ij;
float b_jk;
for(int j=0; j<n; j++)
{
    a_ij = hashA.containsKey(j)? hashA.get(j):0.0f;
    b_jk = hashB.containsKey(j)? hashB.get(j): 0.0f;
```

```
        result += a-ijk * b-jk;
    }
    if (result != 0.0f)
    {
        context.write(null, new Text(key.toString()+","+
        Float.toString(result)));
    }
}
}
```

Input:-

```
$hadoop fs -cat/user/nagaakhil1519/mm.txt
A,0,0,4
A,0,1,5
A,0,2,6
A,0,3,7
A,0,4,8
A,1,0,1
A,1,1,3
A,1,2,4
A,1,3,5
A,1,4,6
B,0,0,6
B,0,1,3
```

B,0,2,4

B,1,0,1

B,1,1,2

B,2,2,7

B,2,0,8

B,2,1,6

B,2,2,5

B,3,0,2

B,3,1,3

B,3,2,7

B,4,0,7

B,4,1,4

B,4,2,2

output:-

$hadoop jar matrix.jar MatrixMultiplication/user/nagaakhil1519/mm.txt /user/nagaakhil1519/output2

$hadoop fs -ls/user/nagaakhil1519/output2

found 2 items

-rw-r--r-- 3 nagaakhil1519 /user/nagaakhil1519/output2/-SUCCESS

-rw-r--r-- 3 nagaakhil1519 /user/nagaakhil1519/output2/part-r-00000

$hadoop fs -cat /user/nagaakhil1519/output2/part-r-00000

0, 0, 147.0

0, 1, 111.0

0, 2, 104.0

1, 0, 93.0

1, 1, 72.0

1, 2, 62.0