**LAB CODES**

**WORDCOUNT**

**import** java.io.IOException;

**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.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** wordcount {

**public** **static** **class** Map **extends** Mapper<LongWritable, Text, Text, IntWritable>

{

IntWritable one = **new** IntWritable(1);

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

String[] line = value.toString().split(",");

**for**(String lines:line) {

context.write(**new** Text(lines),one);

} }

}

**public** **static** **class** Reduce **extends** Reducer<Text, IntWritable, Text, IntWritable> {

**public** **void** reduce(Text key, Iterable<IntWritable> values, Context context)

**throws** IOException, InterruptedException { **int** sum = 0;

**for** (IntWritable val : values) {

sum += val.get();

}

context.write(key, **new** IntWritable(sum));

} }

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "wordcount");

job.setJarByClass(wordcount.**class**);

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(IntWritable.**class**);

job.setMapperClass(Map.**class**);

job.setReducerClass(Reduce.**class**);

job.setInputFormatClass(TextInputFormat.**class**);

job.setOutputFormatClass(TextOutputFormat.**class**);

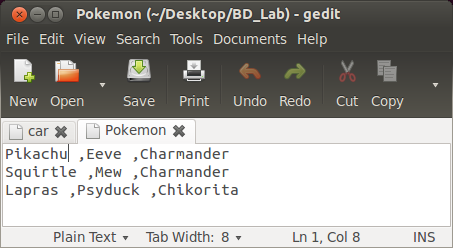
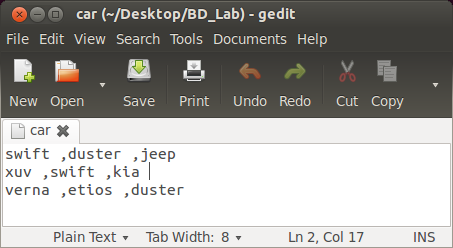
FileInputFormat.*addInputPath*(job, **new** Path(args[0]));

FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));

job.waitForCompletion(**true**);

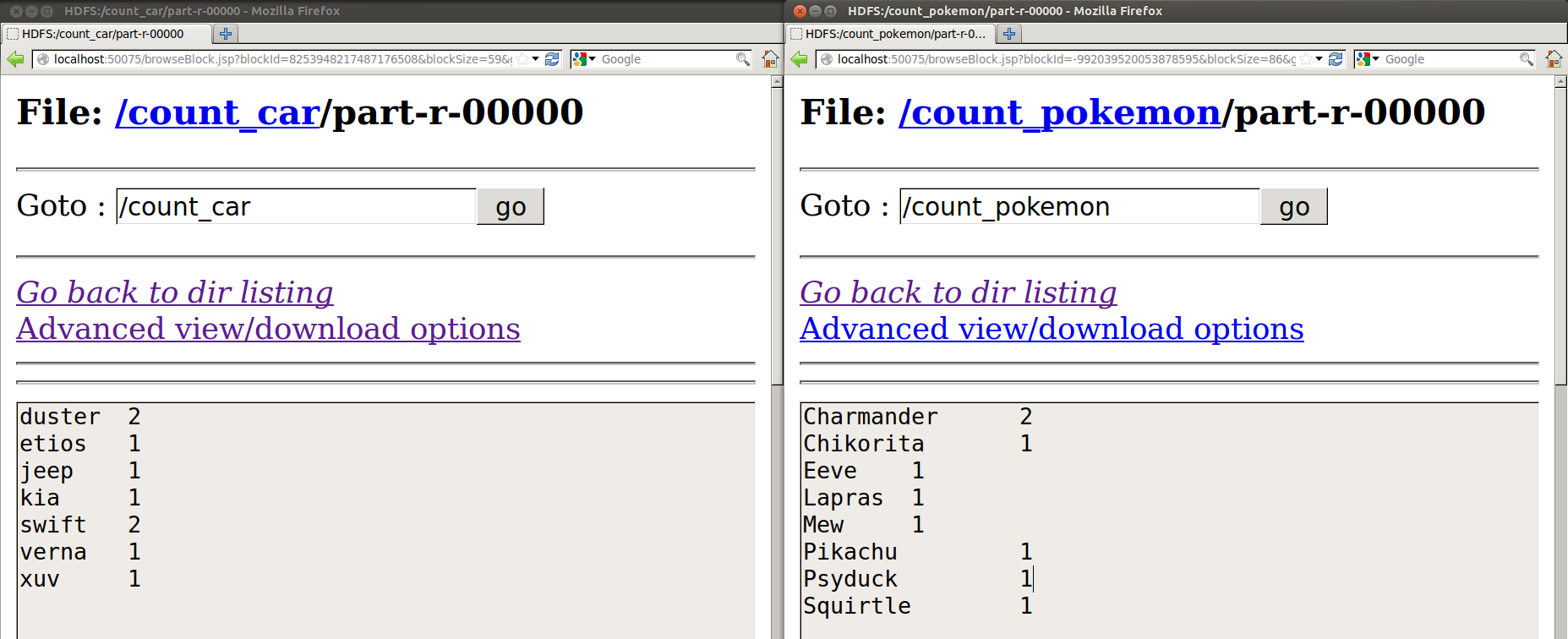
}

}

**INPUT**

**OUTPUT**

hadoop jar /home/ponny/Desktop/BD\_Lab/wc.jar wordcount /Pokemon /count\_pokemon



**Identify the word that occurs maximum number of times from a file from HDFS**

Wait until the reduce function gets complete to print the final result

**setup task, cleanup task** => runs for every map-reduce task internally

**setup task** => tasks to be performed before map

**cleanup task** =>

* tasks to be performed after map-reduce
* aggregate the final result

**MAX\_WORDCOUNT**

**import** java.io.IOException;

**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.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** max\_wordcount {

**public** **static** **class** Map **extends** Mapper<LongWritable, Text, Text, IntWritable>

{

IntWritable one = **new** IntWritable(1);

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

String[] line = value.toString().split(",");

**for**(String lines:line) {

context.write(**new** Text(lines),one);

}

}

}

**public** **static** **class** Reduce **extends** Reducer<Text, IntWritable, Text, IntWritable> {

**public** IntWritable res1 = **new** IntWritable();

**public** Text wrd1 = **new** Text();

**int** max=0;

**public** **void** reduce(Text key, Iterable<IntWritable> values, Context context)

**throws** IOException, InterruptedException {

**int** sum =0;

**for** (IntWritable val : values) {

sum += val.get();

**if**(sum>max)

{

max = sum;

wrd1.set(key);

res1.set(max);

}

}

}

**public** **void** cleanup(Context context) **throws** IOException, InterruptedException {

context.write(wrd1,res1);

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "max\_wordcount");

job.setJarByClass(max\_wordcount.**class**);

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(IntWritable.**class**);

job.setMapperClass(Map.**class**);

job.setReducerClass(Reduce.**class**);

job.setInputFormatClass(TextInputFormat.**class**);

job.setOutputFormatClass(TextOutputFormat.**class**);

FileInputFormat.*addInputPath*(job, **new** Path(args[0]));

FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));

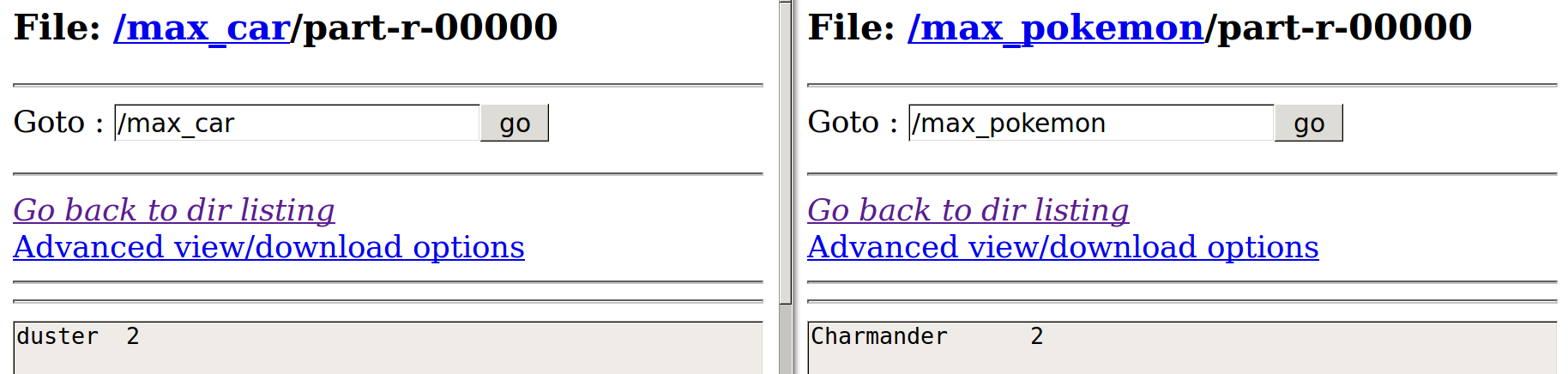
job.waitForCompletion(**true**);

}

}

hadoop jar /home/ponny/Desktop/BD\_Lab/max\_wc.jar max\_wordcount /car /max\_car

hadoop jar /home/ponny/Desktop/BD\_Lab/max\_wc.jar max\_wordcount /Pokemon /max\_pokemon



**TITANIC\_DIED**

Find the average age of males and females who died in the Titanic tragedy from the given input file. Also find the **maximum average age of female and male**.

**import** java.io.IOException;

**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.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** titanic\_died {

**public** **static** **class** Map **extends** Mapper<LongWritable, Text, Text, IntWritable>

{

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

String[] line = value.toString().split(",");

**int** a= Integer.*parseInt*(line[1]);

**if**(a==1){ **//If Died**

**try**

{

context.write(**new** Text(line[4]),**new** IntWritable(Integer.*parseInt*(line[5])));

}

**catch**(Exception e)

{

context.write(**new** Text(line[4]),**new** IntWritable(Integer.*parseInt*("0")));

}

}

}

}

**public** **static** **class** Reduce **extends** Reducer<Text, IntWritable, Text, FloatWritable> {

**public** **void** reduce(Text key, Iterable<IntWritable> values, Context context)

**throws** IOException, InterruptedException {

**int** sum =0 , n=0;;

**float** avg;

**for** (IntWritable val : values)

{

n+=1;

sum += val.get();

}

avg= sum/n;

context.write(key , **new** FloatWritable(avg));

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "titanic\_died");

job.setJarByClass(titanic\_died.**class**);

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(IntWritable.**class**);

job.setMapperClass(Map.**class**);

job.setReducerClass(Reduce.**class**);

job.setInputFormatClass(TextInputFormat.**class**);

job.setOutputFormatClass(TextOutputFormat.**class**);

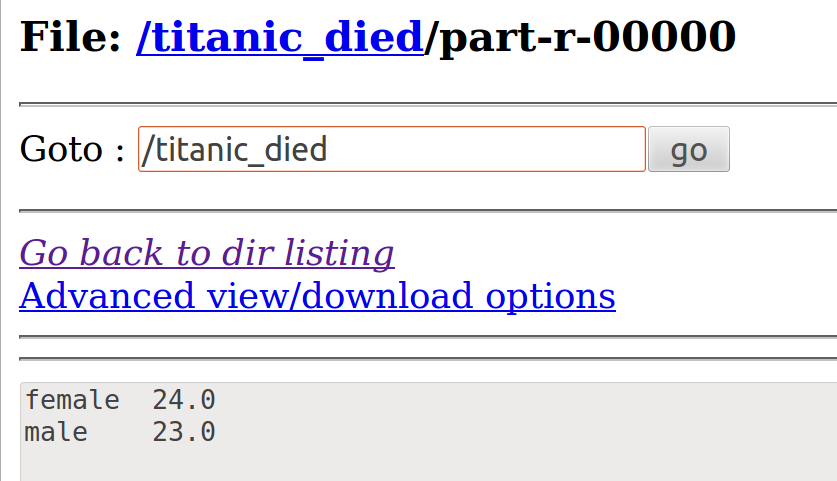
FileInputFormat.*addInputPath*(job, **new** Path(args[0]));

FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));

job.waitForCompletion(**true**);

}

}



**SALARY1**

Write a map reduce programming to find the details of the persons with the salary > 60,000 euros and years of experience >15. Also display the **total number of records (hint: Use Cleanup)** satisfying this condition. The output of the HDFS is:

country name, years of experience, salary

**import** java.io.IOException;

**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.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** salary1 {

**public** **static** **class** Map **extends** Mapper<LongWritable, Text, Text, Text>

{

IntWritable one = **new** IntWritable(1);

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

String[] line = value.toString().split(",");

**if (Float.*parseFloat*(line[1])>60000 && Float.*parseFloat*(line[0])>15){**

//Key: country, Value: Salary , Years of experience

context.write(**new** Text(line[3]),**new** Text(" , "+line[1]+" , "+line[0]));

}

}

}

**public** **static** **class** Reduce **extends** Reducer<Text, Text, Text, Text> {

**int** sum = 0;

**public** **void** reduce(Text key, Iterable<Text> values, Context context) **throws** IOException, InterruptedException {

**for** (Text val : values) {

context.write(key,val);

sum++;

}

}

**public** **void** cleanup(Context context) **throws** IOException, InterruptedException

{

context.write(**new** Text("Total Records: "),**new** Text(Integer.*toString*(sum)));

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "salary1");

job.setJarByClass(salary1.**class**);

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(Text.**class**);

job.setMapperClass(Map.**class**);

job.setReducerClass(Reduce.**class**);

job.setInputFormatClass(TextInputFormat.**class**);

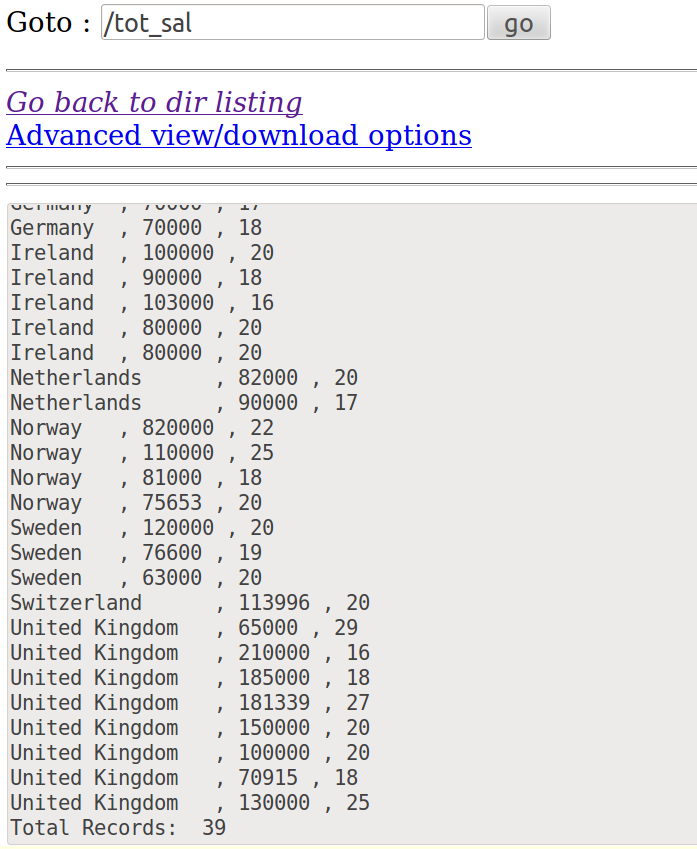
job.setOutputFormatClass(TextOutputFormat.**class**);

FileInputFormat.*addInputPath*(job, **new** Path(args[0]));

FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));

job.waitForCompletion(**true**);

}

}

**Input/Output Formats**

**KeyValueTextInputFormat**

**Use the car details file(occurrences of the car, i.e, car name and its count) that you have created in the hdfs..**

In HDFS by default the tab is the separator. Search a particular car and display its count.

**Given**

* **[car\_count] wordcount output as input**
* Separator ⇒ Tab
* Display the key along with its value
* Car names ⇒ Key ( Text DataType)

import java.io.IOException;

import java.util.\*;

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.Mapper.Context;

import org.apache.hadoop.mapreduce.lib.input.\*;

import org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** keyval {

**public** **static** **class** Map **extends** Mapper<Text, Text, Text, Text>

{

String s =**"duster "**; **// fetch only the duster count**

**public** **void** map(Text key, Text value, Context context) **throws** IOException, InterruptedException {

String line=key.toString();

**if**(s.equalsIgnoreCase(line))

context.write(key, value);

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

**conf.set("mapreduce.input.keyvaluelinerecordreader.key.value.separator", "\t");**

**Job job = new Job(conf, "keyval");**

job.setJarByClass(keyval.**class**);

**job.setOutputKeyClass(Text.class);**

**job.setOutputValueClass(Text.class);**

job.setMapperClass(Map.**class**);

**job.setInputFormatClass(KeyValueTextInputFormat.class);**

job.setOutputFormatClass(TextOutputFormat.**class**);

FileInputFormat.*addInputPath*(job, **new** Path(args[0]));

FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));

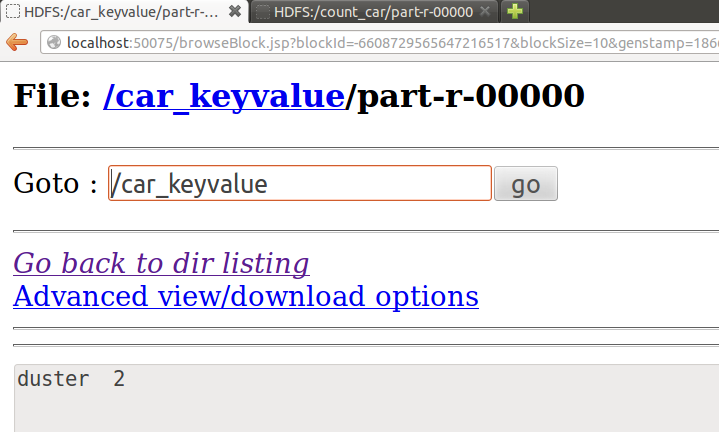
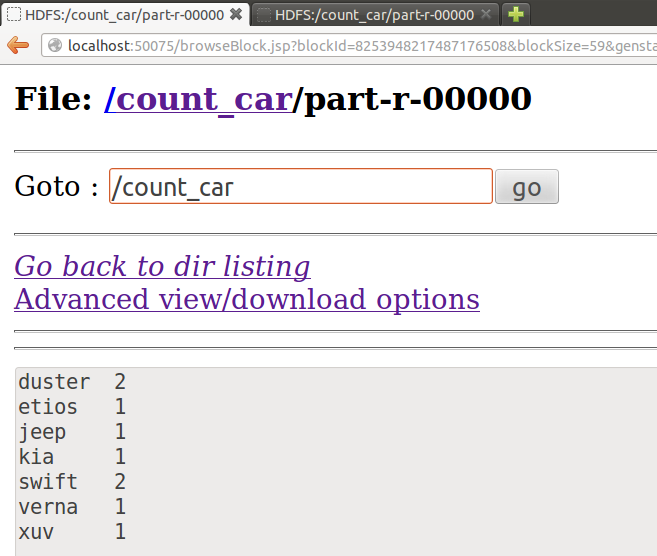
job.waitForCompletion(**true**);

}

}

hadoop jar /home/ponny/Desktop/BD\_Lab/key\_val.jar keyval /count\_car/part-r-00000 /car\_keyvalue

**Input**

**Output**

**NLineInputFormat**

**N** ⇒ No. of Records

**I/P file** ⇒ 6 Records

**Map Task** ⇒ Has to Read only 3 Records from I/P File

Thus we need **2** Map Tasks

To see the Mapper O/P , **Set ⇒ Reducer(0)**

**import** java.io.IOException;

**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.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** nline {

**public** **static** **class** Map **extends** Mapper<LongWritable, Text, Text, Text>

{

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

String[] line = value.toString().split(",");

context.write(**new** Text(line[0]),**new** Text(" , "+line[1]+" , "+line[2]+" , "+line[3]+" , "+line[4]));

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

conf.setInt(NLineInputFormat.***LINES\_PER\_MAP***,**3**);

Job job = **new** Job(conf, "nline");

job.setJarByClass(nline.**class**);

job.setOutputKeyClass(Text.**class**);

job.setMapperClass(Map.**class**);

job.setInputFormatClass(NLineInputFormat.**class**);

job.setNumReduceTasks(0);

job.setOutputFormatClass(TextOutputFormat.**class**);

FileInputFormat.*addInputPath*(job, **new** Path(args[0]));

FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));

job.waitForCompletion(**true**);

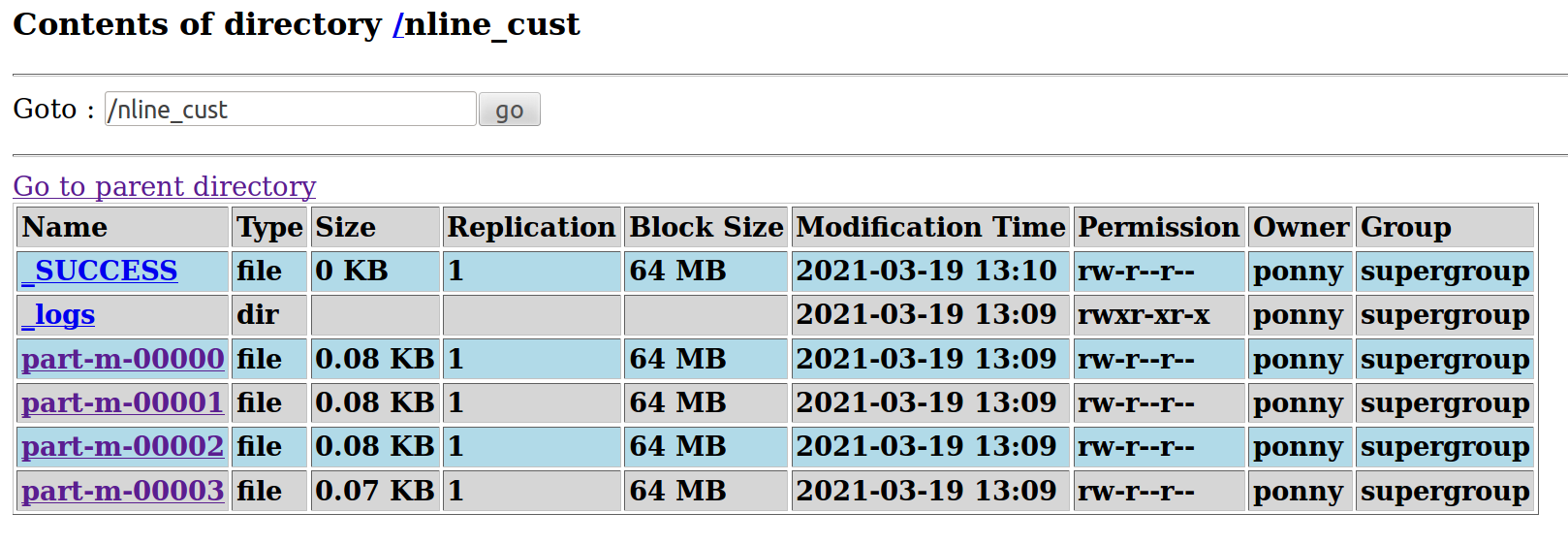
}

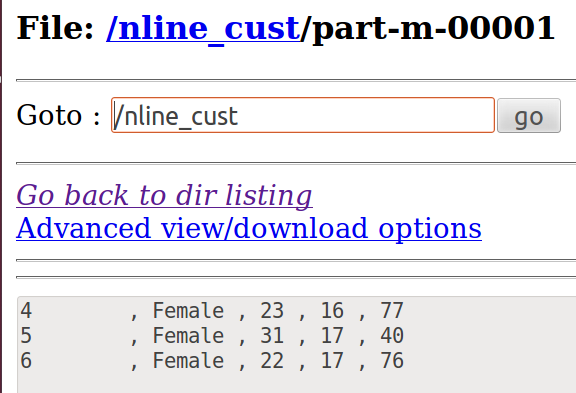
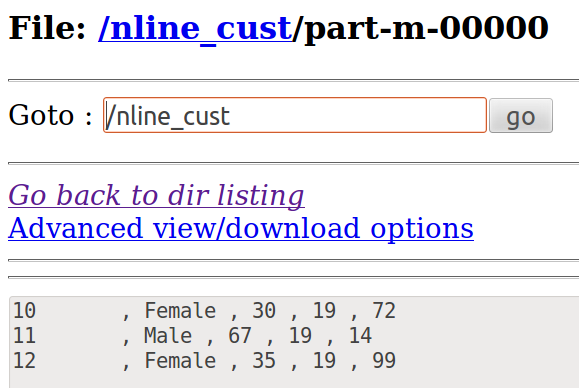
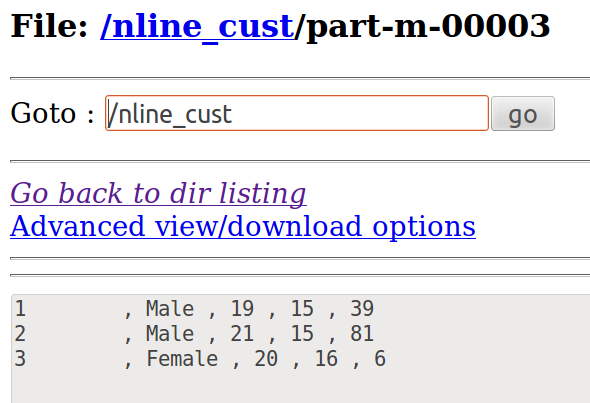
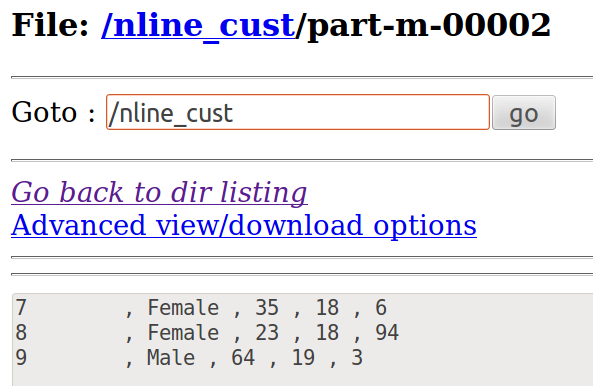
}

**Input**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CustomerID | Genre | Age | Annual\_Income\_(k$) | Spending\_Score |
|  | | | | |
| 1 | Male | 19 | 15 | 39 |
| 2 | Male | 21 | 15 | 81 |
| 3 | Female | 20 | 16 | 6 |
| 4 | Female | 23 | 16 | 77 |
| 5 | Female | 31 | 17 | 40 |
| 6 | Female | 22 | 17 | 76 |
| 7 | Female | 35 | 18 | 6 |
| 8 | Female | 23 | 18 | 94 |
| 9 | Male | 64 | 19 | 3 |
| 10 | Female | 30 | 19 | 72 |
| 11 | Male | 67 | 19 | 14 |
| 12 | Female | 35 | 19 | 99 |

hadoop jar /home/ponny/Desktop/BD\_Lab/nline.jar nline /cust.csv /nline\_cust

**Output**

****

### NLine Tool Runner

During Runtime we pass the number of mapper inputs

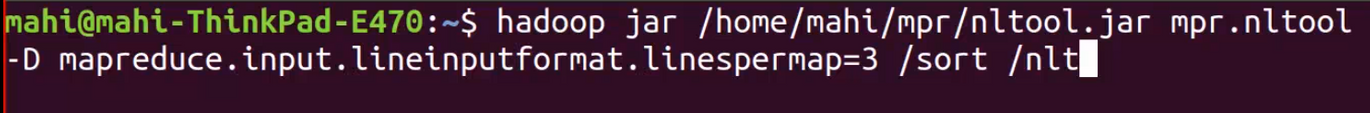
NLine Input format code

nltool.java

use run

from main ⇒ call run using ToolRunner

While running



**SequenceFileInputFormat**

* Text / anything to Sequence File
* The sequence file format can be used to store an image in the binary format.
* They store key-value pairs in a Binary container format and are more efficient (Compressed) than a text file.
* Sequence files are not human-readable.

By Default Input and Output is set as Text

**import** java.io.IOException;

**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.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** seq\_in {

**public** **static** **class** FormatConverterMapper **extends** Mapper<LongWritable, Text, LongWritable, Text> {

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException

{

context.write(key, value);

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf);

job.setJarByClass(seq\_in.**class**);

job.setMapperClass(FormatConverterMapper.**class**);

job.setInputFormatClass(SequenceFileInputFormat.**class**);

FileInputFormat.*setInputPaths*(job, **new** Path(args[0]));

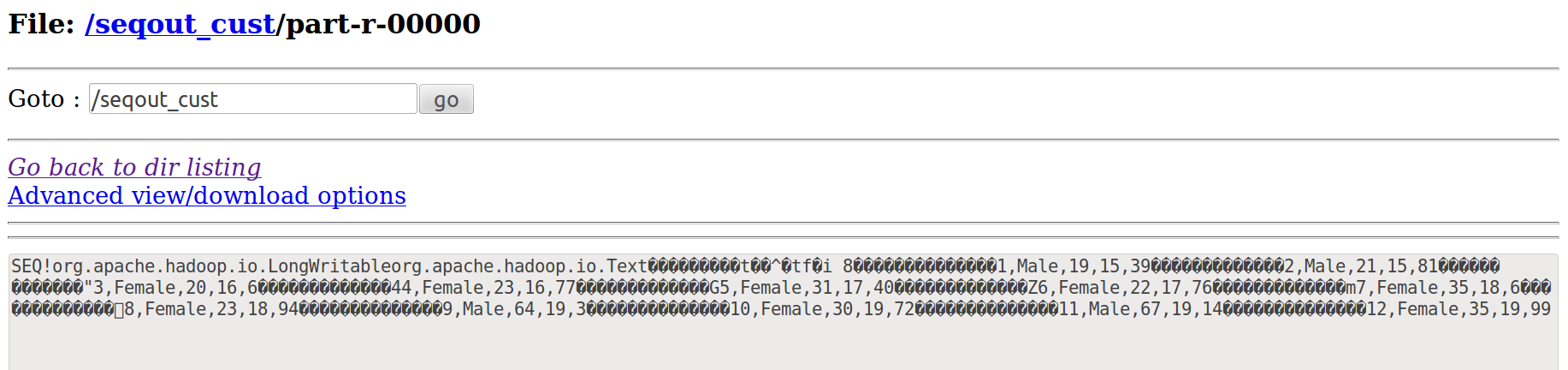
FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));

job.waitForCompletion(**true**);

}

}

hadoop jar /home/ponny/Desktop/BD\_Lab/seq\_out.jar seq\_out /cust.csv /seqout\_cust

**Output**

**SequenceFileOutputFormat**

**key ⇒ Count of characters in the whole record including “ , “**

Sequence File to Text / anything

**import** java.io.IOException;

**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.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** seq\_out {

**public** **static** **class** FormatConverterMapper **extends** Mapper<LongWritable, Text, LongWritable, Text> {

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException

{

context.write(key, value);

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf);

job.setJarByClass(seq\_out.**class**);

job.setMapperClass(FormatConverterMapper.**class**);

job.setOutputFormatClass(SequenceFileOutputFormat.**class**);

FileInputFormat.*addInputPath*(job, **new** Path(args[0]));

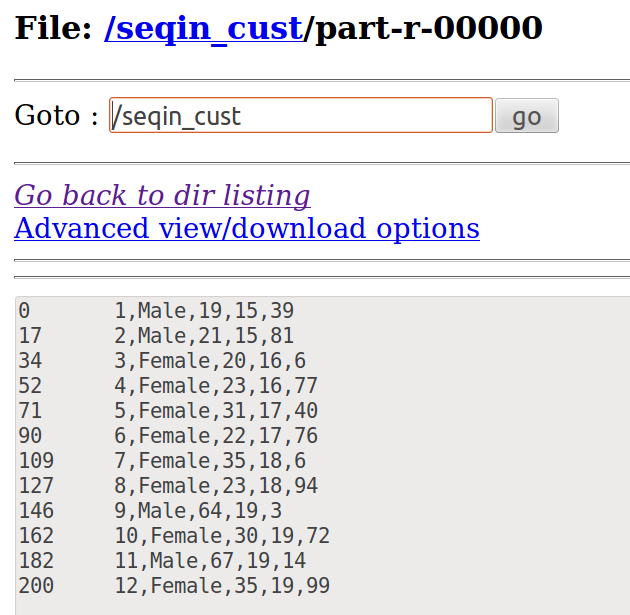
FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));

job.waitForCompletion(**true**);

}

}

hadoop jar /home/ponny/Desktop/BD\_Lab/seq\_in.jar seq\_in /seqout\_cust/part-r-00000 /seqin\_cust

**Output**

**NAME AS PARTITION NAME / Multiple Output Format**

It allows writing data to files whose names are derived from the output keys and values, or in fact from an arbitrary string.

Setup Task ⇒ Used to initialize the MultipleOutputs

Cleanup Task ⇒ Used to close the MultipleOutputs

o/p file is under the name of key

multipleOutputs.write(key, new IntWritable(sum), key.toString( ) );

Give a name in double quotes instead of key.toString( ) , so the entire o/p will be stored in a particular file

**import java.io.IOException;**

**import java.util.\*;**

**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.\*;**

**import org.apache.hadoop.mapreduce.lib.output.\*;**

**public class mulout {**

**public static class Map extends Mapper<LongWritable, Text, Text, IntWritable>**

**{**

**IntWritable one = new IntWritable(1);**

**public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {**

**String[] line = value.toString().toUpperCase().split(",");**

**for(String lines:line) {**

**context.write(new Text(lines),one);**

**}**

**}**

**}**

**public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {**

**public MultipleOutputs multipleOutputs;**

**public void setup(Context context) throws IOException, InterruptedException**

**{**

**multipleOutputs = new MultipleOutputs(context);**

**}**

**public void reduce(Text key, Iterable<IntWritable> values, Context context)**

**throws IOException, InterruptedException {**

**int sum = 0;**

**for (IntWritable val : values) {**

**sum += val.get();**

**}**

**multipleOutputs.write(key, new IntWritable(sum),key.toString());**

**}**

**public void cleanup(Context context) throws IOException, InterruptedException**

**{**

**multipleOutputs.close();**

**}**

**}**

**public static void main(String[] args) throws Exception {**

**Configuration conf = new Configuration();**

**Job job = new Job(conf, "mulout");**

**job.setJarByClass(mulout.class);**

**job.setOutputKeyClass(Text.class);**

**job.setOutputValueClass(IntWritable.class);**

**job.setMapperClass(Map.class);**

**job.setReducerClass(Reduce.class);**

**job.setInputFormatClass(TextInputFormat.class);**

**LazyOutputFormat.setOutputFormatClass(job, TextOutputFormat.class);**

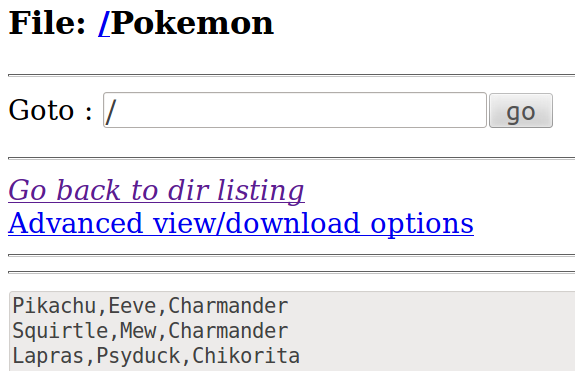
**FileInputFormat.addInputPath(job, new Path(args[0]));**

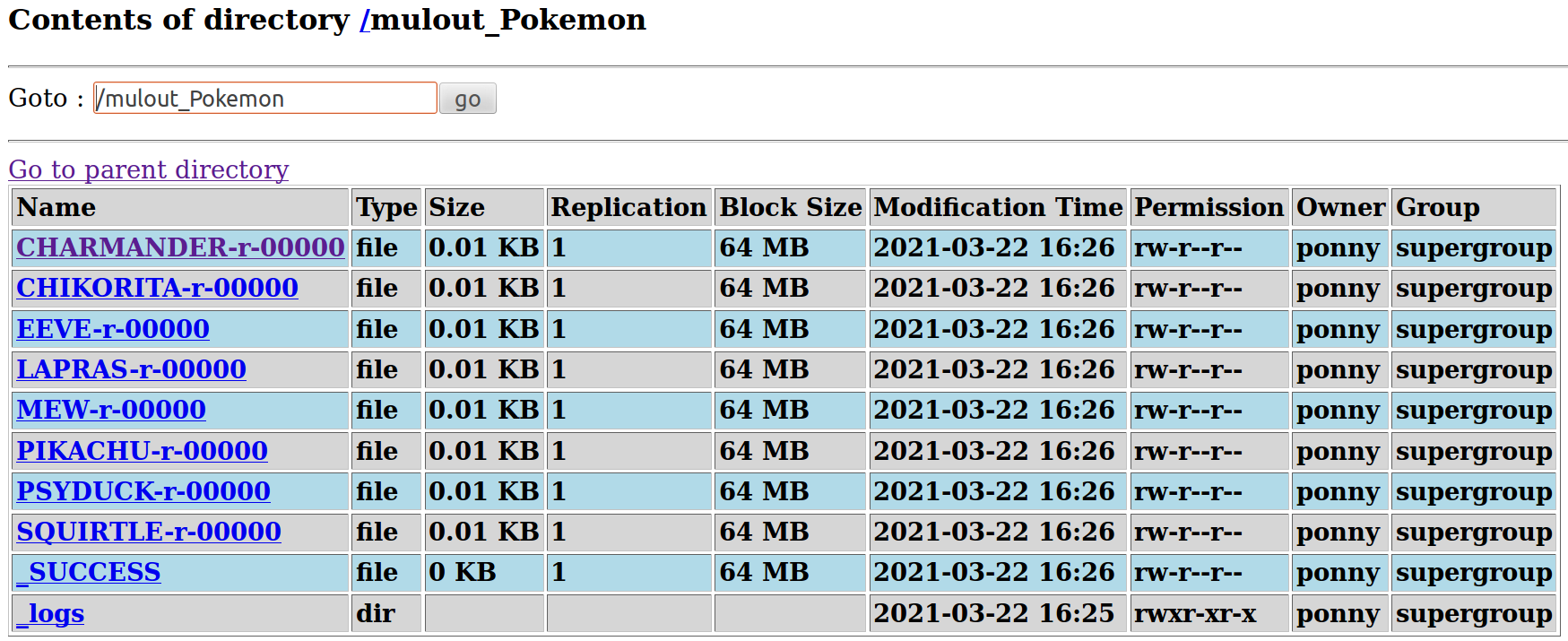
**FileOutputFormat.setOutputPath(job, new Path(args[1]));**

**job.waitForCompletion(true);**

**}**

**}**

****

**Each “Key” from the Reducer is partioned separately**

**Lazy o/p can be used to avoid empty o/p being generated in part-r-00000**

**LazyOutputFormat**

Sometimes FileOutputFormat will create output files, even if they are empty.

LazyOutputFormat is a wrapper OutputFormat which ensures that the output file will be created only when the record is emitted for a given partition.

**Hash Partitioning / Custom Partitioning**

Mapper (out) ⇒ partitioner ⇒ Reducer

gerPartition( key , value , "No of reducers: int nr ")

An organization maintains the salary details that contains years of experience,salary, gender and country name. Partition the file based on the salary. Create 3 partitions with salary <30000, salary<50000 and salary details other than the above two conditions. Write the country name and the salary details in the files. Also display the count of the records satisfying the above conditions.

**Partition based on salary**

sal<30k

sal<50k

sal⇒others

count the records in every partitioner

**CODE**

**import** java.io.IOException;

**import** java.util.\*;

**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.Reducer.Context;

**import** org.apache.hadoop.mapreduce.lib.input.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** custpart {

**public** **static** **class** Map **extends** Mapper<LongWritable, Text, Text, IntWritable>

{

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

String[] line = value.toString().split(",");

//Key: country, Value: Salary

context.write(**new** Text(line[3]),**new** IntWritable(Integer.*parseInt*(line[1])));

}

**public** **static** **class** dpart **extends** Partitioner<Text,IntWritable>

{

//"nr" receives the value from setNumReduceTasks( \_ )

**public** **int** getPartition(Text key,IntWritable value,**int** nr)

{

**if**(value.get()<30000)

**return** 0;

**if**(value.get() < 50000)

**return** 1;

**else**

**return** 2;

}

}

**public** **static** **class** Reduce **extends** Reducer<Text, IntWritable, Text, IntWritable> {

**int** count=0;

**public** **void** reduce(Text key, Iterable<IntWritable> values, Context context) **throws** IOException, InterruptedException {

**for** (IntWritable val : values) {

context.write(key,val);

count++;

}

}

**public** **void** cleanup(Context context) **throws** IOException, InterruptedException

{

context.write(**new** Text("Record Count : "),**new** IntWritable(count));

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "custpart");

job.setJarByClass(custpart.**class**);

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(IntWritable.**class**);

job.setMapperClass(Map.**class**);

job.setReducerClass(Reduce.**class**);

// give the no. of reducer

job.setPartitionerClass(dpart.**class**);

job.setNumReduceTasks(3);

job.setInputFormatClass(TextInputFormat.**class**);

job.setOutputFormatClass(TextOutputFormat.**class**);

FileInputFormat.*addInputPath*(job, **new** Path(args[0]));

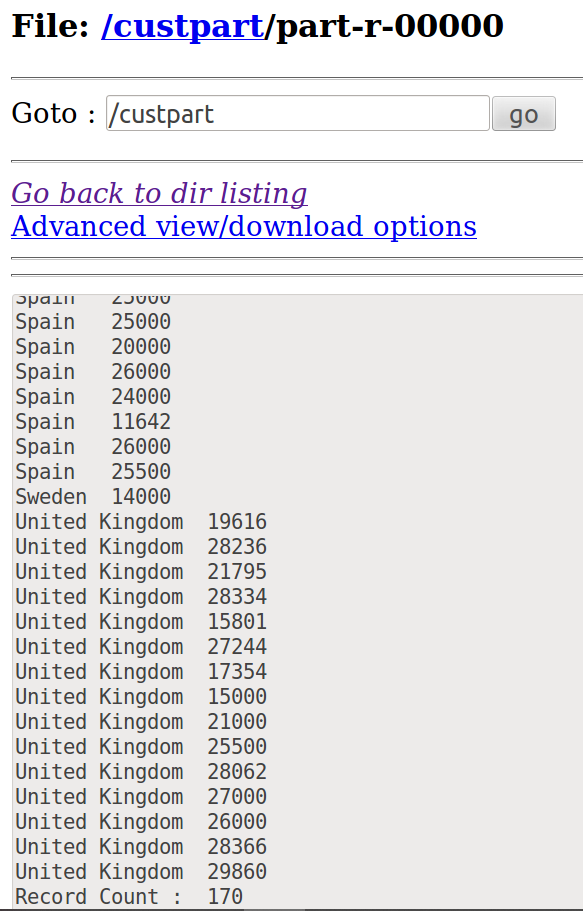
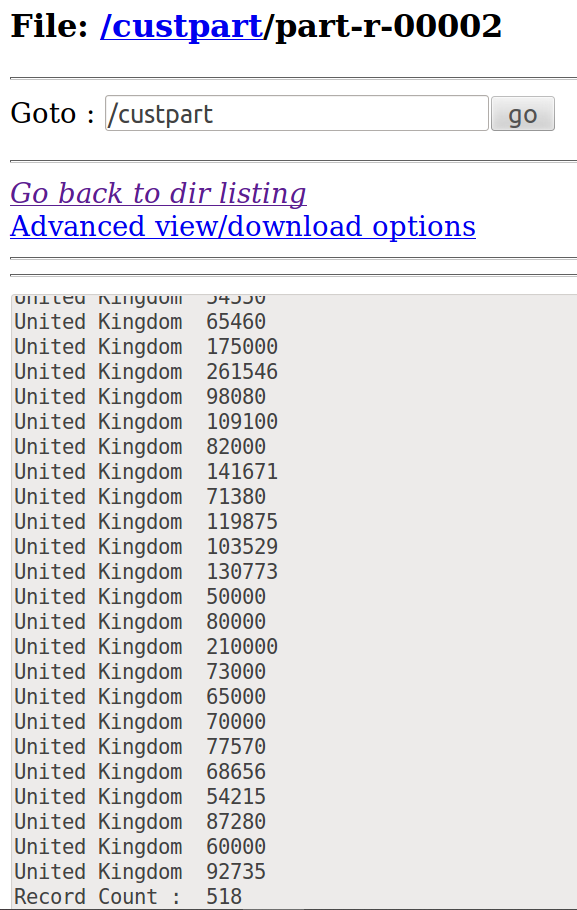
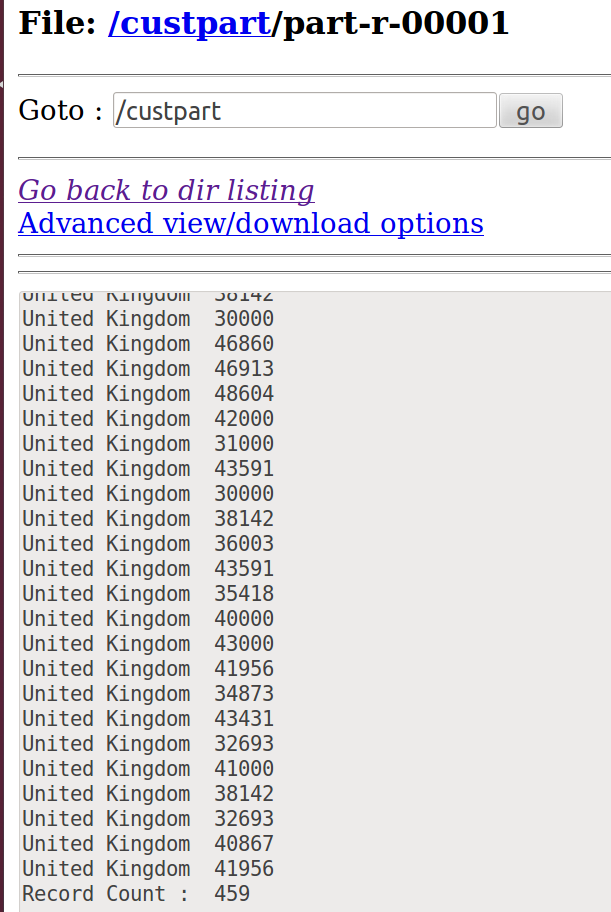
FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));

job.waitForCompletion(**true**);

}

}

hadoop jar /home/ponny/Desktop/BD\_Lab/cust\_part.jar custpart /salary1.csv /custpart

****

**Custom Counters**

Salary details are maintained in the text file. Create the user defined counters. Count the number of persons having 10 years of experience and write the details of country name and the salary with 10 years of experience in HDFS. Also count the number of persons earning the salary greater than 50,000.

**import** java.io.IOException;

**import** java.util.\*;

**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.Reducer.Context;

**import** org.apache.hadoop.mapreduce.lib.input.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** custcount {

**public** **enum** ct

{

*cnt*,*nt* //cnt => Condition 1 ; nt => Condition 2

};

**public** **static** **class** Map **extends** Mapper<LongWritable, Text, Text, FloatWritable>

{

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

String[] line = value.toString().split(",");

**if**(Float.*parseFloat*(line[0])==10.0)

{

context.getCounter(ct.*cnt*).increment(1);

// country name , salary

context.write(**new** Text(line[3]),**new** FloatWritable(Float.*parseFloat*(line[1])));

}

**if**(Float.*parseFloat*(line[1])>50000)

{

context.getCounter(ct.*nt*).increment(1);

}

}

}

**public** **static** **class** Reduce **extends** Reducer<Text, FloatWritable, Text, FloatWritable>

{

**public** **void** reduce(Text key, Iterable<FloatWritable> values, Context context) **throws** IOException, InterruptedException

{

**for** (FloatWritable val : values)

{

context.write(key,val);

}

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "custcount");

job.setJarByClass(custcount.**class**);

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(FloatWritable.**class**);

job.setMapperClass(Map.**class**);

job.setReducerClass(Reduce.**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**);

**Counters cn = job.getCounters();**

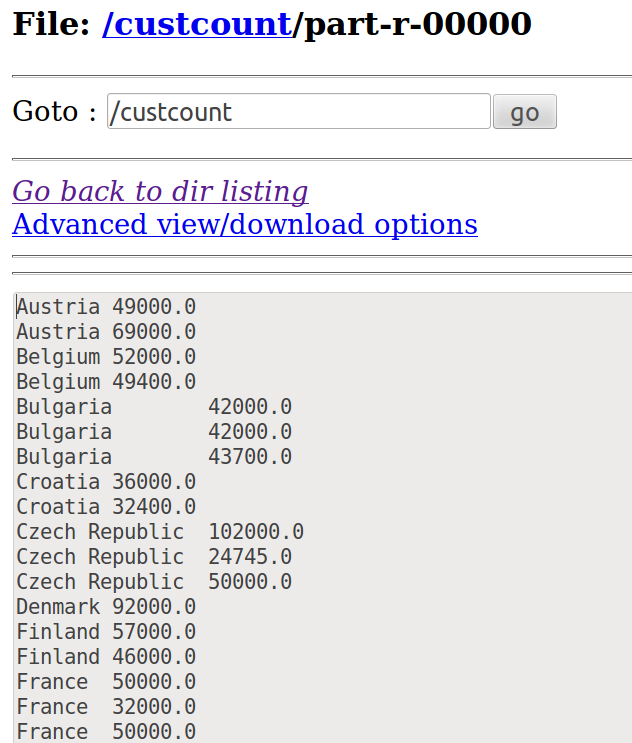
**cn.findCounter(ct.*cnt*).getValue();**

**cn.findCounter(ct.*nt*).getValue();**

}

}

**hadoop jar /home/ponny/Desktop/BD\_Lab/custcount.jar custcount /salary1.csv /custcount**

****

21/03/31 12:37:08 INFO mapred.JobClient:  **nt=476**

21/03/31 12:37:08 INFO mapred.JobClient:  **cnt=131**

**Side Data**

Create a text file for students details(student id,student name, marks-5 courses) in HDFS. Get a particular student details using side data configuration. Display the student details in hdfs. Also display the total and average of the student’s marks in hdfs.

**import** java.io.IOException;

**import** java.util.\*;

**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.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** sidedata {

**public** **static** **class** Map **extends** Mapper<LongWritable, Text, Text, Text>

{

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

String[] line = value.toString().split(",");

String name1;

name1 = context.getConfiguration().get("name");

String studname = line[1];

**if**(studname.equals(name1)){

{

String[] marks= {line[4],line[5],line[6],line[7],line[8]};

**float** n=marks.length;

**float** sum =0;

**float** avg=0;

**for**(**int** i=0; i< marks.length; i++)

{

sum = sum + Float.*parseFloat*(marks[i]);

avg= sum/n;

}

context.write(**new** Text(line[0]+" , "+line[1]+" , "+line[2]+" , "+line[3]+" , "+line[4]+" , "+line[5]+" , "+line[6]+" , "+line[7]+" , "+line[8]),**new** Text("\nSum : "+sum+"\nAvg : "+avg));

}

} } }

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

conf.set("name",args[2]);

Job job = **new** Job(conf, "sidedata");

job.setJarByClass(sidedata.**class**);

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(Text.**class**);

job.setMapperClass(Map.**class**);

job.setInputFormatClass(TextInputFormat.**class**);

job.setOutputFormatClass(TextOutputFormat.**class**);

FileInputFormat.*addInputPath*(job, **new** Path(args[0]));

FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));

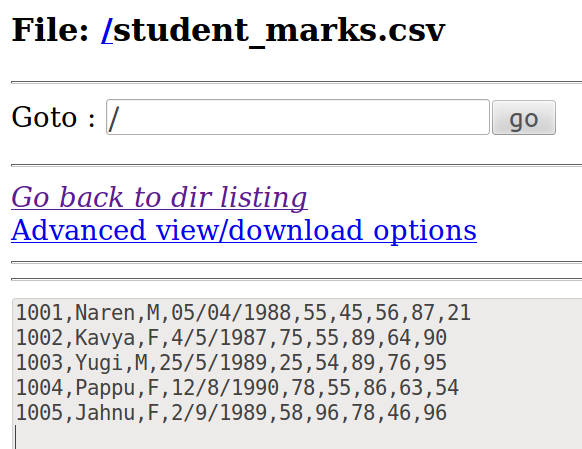
job.waitForCompletion(**true**);

}

}

hadoop jar /home/ponny/Desktop/BD\_Lab/sidedata.jar sidedata /student\_marks.csv /Naren\_mark Naren

## INPUT



**OUTPUT**

****

**Mapside Join**

**Distributed Cache**

Create the student details(student id,student name,CGPA) file and the department details(student id,department) file in HDFS. Let the department details can be stored using distributed cache. Write the student id, name,department and CGPA in the HDFS. Use map side join.

**DISTRIBUTED CACHE** ⇒ **DEPARTMENT**

**import** java.io.IOException;

**import** java.util.\*;

**import** org.apache.hadoop.filecache.DistributedCache;

**import** org.apache.hadoop.fs.Path;

**import** java.io.BufferedReader;

**import** java.io.FileReader;

**import** org.apache.hadoop.conf.\*;

**import** org.apache.hadoop.io.\*;

**import** org.apache.hadoop.mapreduce.\*;

**import** org.apache.hadoop.mapreduce.lib.input.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** distributedcache {

**public** **static** **class** Map **extends** Mapper<LongWritable, Text, Text, Text>

{

Path[ ] cfile=**new** Path[0]; //cfile ⇒ path variable//cfile ⇒ path variable

ArrayList<Text> dep=**new** ArrayList<Text>(); //ArrayList class ⇒ Expandable : no size restriction

**public** **void** setup(Context context)

{

Configuration conf=context.getConfiguration();

**try**

{

cfile = DistributedCache.*getLocalCacheFiles*(conf);

@SuppressWarnings("resource")

//BufferedReader class is used to read the text from a character-based input stream

BufferedReader reader=**new** BufferedReader(**new** FileReader(cfile[0].toString()));

String line;

**while** ((line=reader.readLine())!=**null**)

{

Text tt=**new** Text(line);

dep.add(tt);

}

}

**catch**(IOException e)

{

e.printStackTrace();

}

}

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

String line2 = value.toString(); //stud\_cgpa

String[ ] elements=line2.split(",");

**for**(Text e:dep)

{

String[ ] line1 = e.toString().split(","); //dept

**if**(elements[0].equals(line1[0]))

{

context.write(**new** Text(elements[0]),**new** Text(elements[1]+" , "+elements[2]+" , "+line1[1]));

}

} } }

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "distributedcache");

job.setJarByClass(distributedcache.**class**);

DistributedCache.*addCacheFile*(**new** Path(args[0]).toUri(),job.getConfiguration());

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(Text.**class**);

job.setMapperClass(Map.**class**);

job.setInputFormatClass(TextInputFormat.**class**);

job.setOutputFormatClass(TextOutputFormat.**class**);

FileInputFormat.*addInputPath*(job, **new** Path(args[1])); //DIST\_CACHE : DEPT

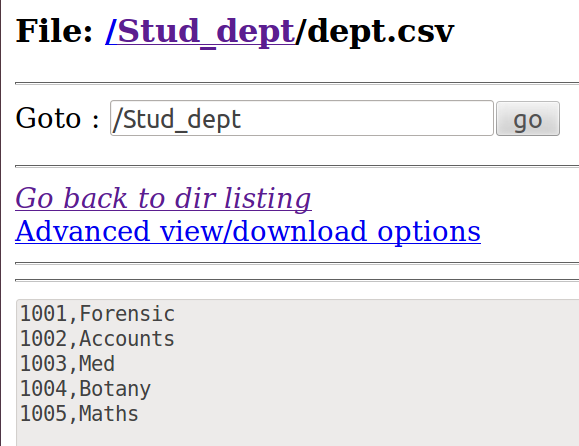
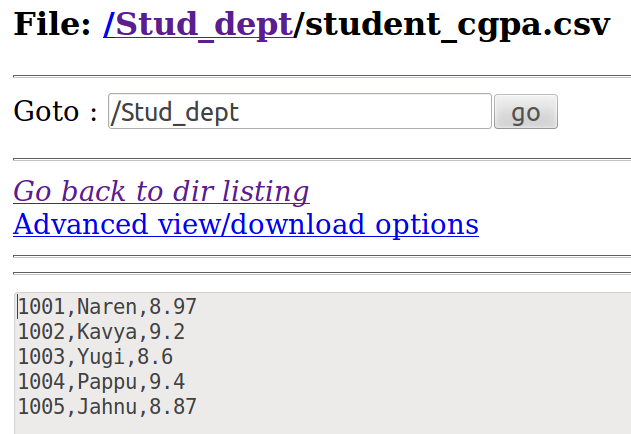
FileOutputFormat.*setOutputPath*(job, **new** Path(args[2]));

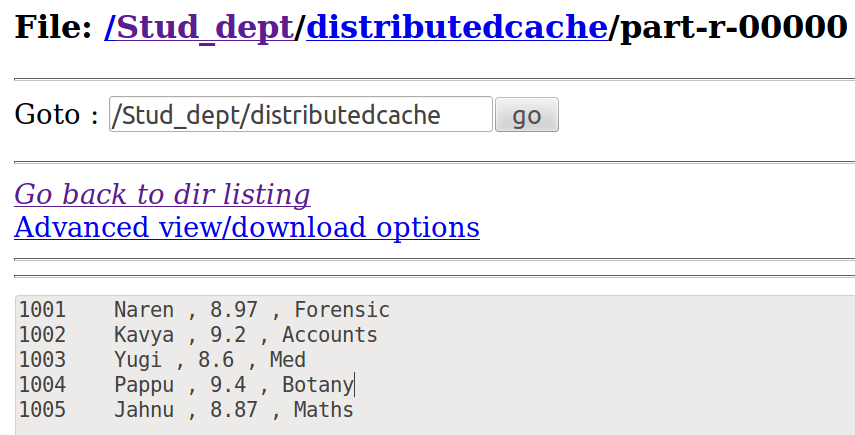
job.waitForCompletion(**true**);

}

}

hadoop jar jarfile classname distributedcache\_path Input\_path Output\_path

hadoop jar /home/ponny/Desktop/BD\_Lab/distributedcache.jar distributedcache /Stud\_dept/dept.csv /Stud\_dept/student\_cgpa.csv /Stud\_dept/distributedcache



**Reduce Side Join**

Create the customer details(cid,name) and the transaction details(cid, transaction

amount) file in hdfs. Write the cname, number of transactions and transaction amount in

hdfs. Use reduce side join.

**NOTE:**

**I/P Records**

Cust ⇒ Cust id , name

trans ⇒ Cust id , Transaction amnt

**O/P : cname , no of trans , trans amnt**

**2 Mappers needed,**

1. Customer Mapper

2. Transaction mapper

To Differentiate add a word " cust," before cust name & " trans," before transaction amount

**import** java.io.IOException;

**import** java.util.\*;

**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.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** redsidejoin {

**public** **static** **class** custmapper **extends** Mapper<LongWritable, Text, Text, Text>

{

IntWritable one = **new** IntWritable(1);

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

String[] line = value.toString().split(",");

context.write(**new** Text(line[0]), **new** Text("cust"+","+line[1]));

}

}

**public** **static** **class** transmapper **extends** Mapper<LongWritable,Text,Text,Text>

{

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

String[] line = value.toString().split(",");

context.write(**new** Text(line[0]), **new** Text("trans"+","+line[1]));

}

}

**public** **static** **class** Reduce **extends** Reducer<Text,Text,Text,Text> {

String st1;

**public** **void** reduce(Text key, Iterable<Text> values, Context context ) **throws** IOException, InterruptedException {

**int** sum = 0;

**int** c=0,amt=0;

**for**(Text val:values)

{

String[] line = val.toString().split(",");

**if** (line[0].equals("trans"))

{

amt += Integer.*parseInt*(line[1]);

c++;

}

**else** **if** (line[0].equals("cust"))

{

st1=line[1].toString();

}

}

context.write(**new** Text(st1), **new** Text(c+" , "+amt));

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "redsidejoin");

job.setJarByClass(redsidejoin.**class**);

job.setOutputKeyClass(Text.**class**);

**MultipleInputs**.*addInputPath*(job,**new** Path(args[0]), TextInputFormat.**class**, custmapper.**class**);

MultipleInputs.*addInputPath*(job,**new** Path(args[1]), TextInputFormat.**class**, transmapper.**class**);

job.setReducerClass(Reduce.**class**);

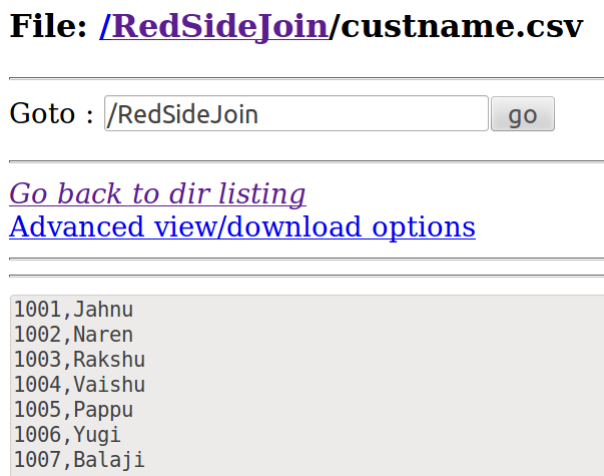
job.setOutputFormatClass(TextOutputFormat.**class**);

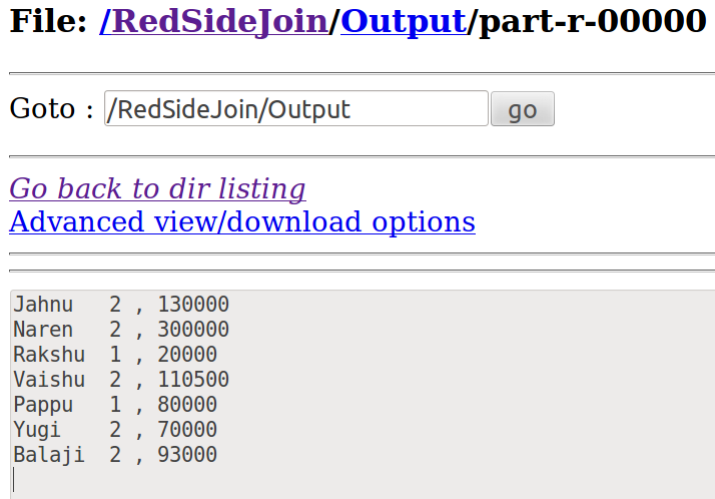
FileOutputFormat.*setOutputPath*(job, **new** Path(args[2]));

job.waitForCompletion(**true**);

}

}



**OUTPUT**

**TreeMAP**

* Red-Black Tree format / not Hash function
* Won't accept Key as NULL unlike HashMap
* Always considers unique values
* **Result is sorted** based on the KEY

**Salary ⇒ Key , Entire Record ⇒ Values**

Allows to use multiple methods like ,

* put( ) ⇒ Add values
* remove( ) ⇒ Remove values
* size( ) ⇒ no of records in Tree Map

**Display highest 10 salary records,**

* To display that we reduce our records to 10 by removing the least salary records that is in the beginning
* Common Key as NullWritable ⇒ Easy to Retrieve and group (Reducer)
* Declare TreeMap again in Reducer ⇒ To sort all the records got from various Mappers

**Create a text file with employee name, department, designation, salary. Identify the top 10 salary details from the file. Write the details in HDFS. Create 20 records.**

**Retrieve Top 10 Records**

**CODE**

**import** java.io.IOException;

**import** java.util.\*;

**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.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** Treemapp {

**public** **static** **class** Map **extends** Mapper<LongWritable, Text,NullWritable,Text>

{

**private** TreeMap<Integer, Text> salary = **new** TreeMap<Integer, Text>();

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException

{

String[] line = value.toString().split(",");

salary.put(Integer.*parseInt*(line[3]),**new** Text(value));

**if** (salary.size() > 10) {

salary.remove(salary.firstKey());

}

}

**protected** **void** cleanup(Context context) **throws** IOException, InterruptedException

{ **for** ( Text name : salary.values() ) {

context.write(NullWritable.*get*(), name);

}

}

}

**public** **static** **class** Reduce **extends** Reducer<NullWritable, Text, NullWritable, Text>

{

**public** **void** reduce(NullWritable key, Iterable<Text> values, Context context) **throws** IOException, InterruptedException {

TreeMap<Integer, Text> salary = **new** TreeMap< Integer, Text>();

**for** (Text value : values) {

String line = value.toString();

String[] elements=line.split(",");

**int** i= Integer.*parseInt*(elements[3]);

salary.put(i, **new** Text(value));

**if** (salary.size() > 10) {

salary.remove(salary.firstKey());

}

}

**for** (Text t : salary.values()) {

context.write(NullWritable.*get*(), t);

}

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "Treemapp");

job.setJarByClass(Treemapp.**class**);

job.setOutputKeyClass(NullWritable.**class**);

job.setOutputValueClass(Text.**class**);

job.setNumReduceTasks(1);

job.setMapperClass(Map.**class**);

job.setReducerClass(Reduce.**class**);

job.setInputFormatClass(TextInputFormat.**class**);

job.setOutputFormatClass(TextOutputFormat.**class**);

FileInputFormat.*addInputPath*(job, **new** Path(args[0]));

FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));

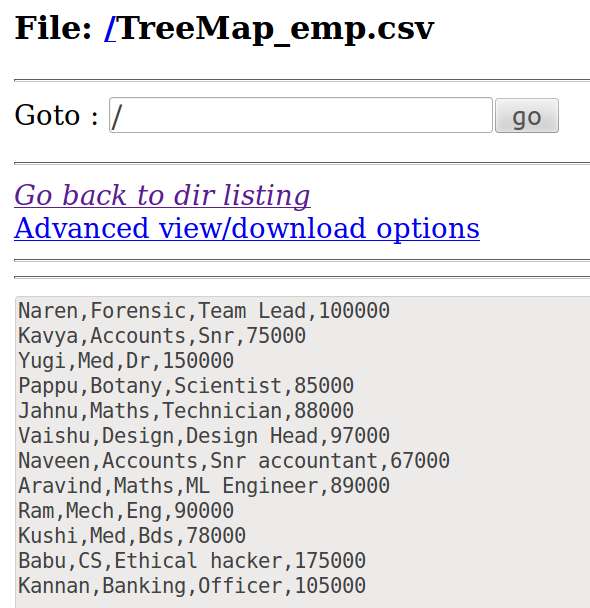
job.waitForCompletion(**true**);

}

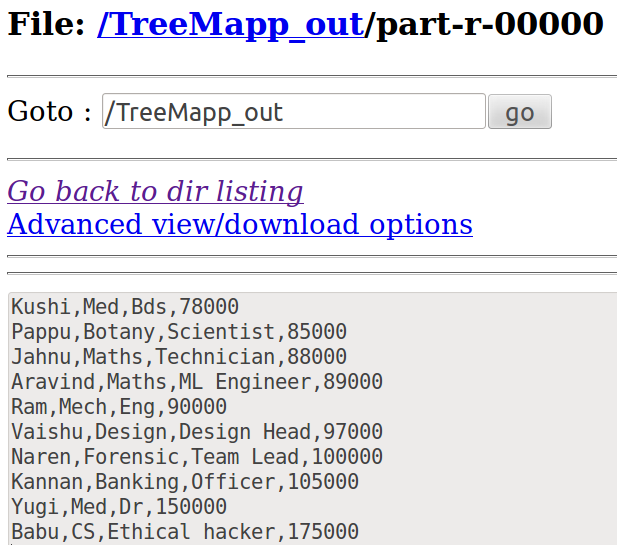
**}**

hadoop jar /home/ponny/Desktop/BD\_Lab/Treemapp.jar Treemapp /TreeMap\_emp.csv /TreeMapp\_out

**Input file**

****

**OUTPUT**



### Secondary Sort - Composite Key

### Unlike Key sort , we can Sort using 1/ more keys

Composite Key ⇒ Combine 2Keys

implements Writable,WritableComparable<CompositeKeyWritable>

Dept AO , Emp DO

Override everywhere with the COMPOSITE key

-state ⇒ sort in D.O

**import** java.util.\*;

**import** java.io.\*;

**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.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** secsort {

**public** **static** **class** CompositeKeyWritable **implements** Writable,

WritableComparable<CompositeKeyWritable> {

**private** String deptNo;

**private** String emp;

**public** CompositeKeyWritable() {

}

**public** CompositeKeyWritable(String deptNo, String emp) {

**this**.deptNo = deptNo;

**this**.emp = emp;

}

@Override

**public** String toString() {

**return** (**new** StringBuilder().append(deptNo).append("\t")

.append(emp)).toString();

}

**public** **void** readFields(DataInput dataInput) **throws** IOException {

deptNo = WritableUtils.*readString*(dataInput);

emp = WritableUtils.*readString*(dataInput);

}

**public** **void** write(DataOutput dataOutput) **throws** IOException {

WritableUtils.*writeString*(dataOutput, deptNo);

WritableUtils.*writeString*(dataOutput, emp);

}

**public** **int** compareTo(CompositeKeyWritable objKeyPair) {

**int** result = deptNo.compareTo(objKeyPair.deptNo);

**if** (0 == result) {

result = emp.compareTo(objKeyPair.emp);

}

**return** result;

}

}

**public** **static** **class** mapper1 **extends**

Mapper<LongWritable, Text, CompositeKeyWritable, NullWritable> {

@Override

**public** **void** map(LongWritable key, Text value, Context context)

**throws** IOException, InterruptedException {

**if** (value.toString().length() > 0) {

String arrEmpAttributes[] = value.toString().split(",");

context.write(

**new** CompositeKeyWritable(

arrEmpAttributes[1].toString(),

(arrEmpAttributes[0].toString())), NullWritable.*get*());

}

}

}

**public** **static** **class** SecondarySortBasicPartitioner **extends**

Partitioner<CompositeKeyWritable, NullWritable> {

@Override

**public** **int** getPartition(CompositeKeyWritable key, NullWritable value,

**int** numReduceTasks) {

**return** (key.deptNo.hashCode() % numReduceTasks);

}

}

**public** **static** **class** SecondarySortBasicCompKeySortComparator **extends** WritableComparator {

**protected** SecondarySortBasicCompKeySortComparator() {

**super**(CompositeKeyWritable.**class**, **true**);

}

@Override

**public** **int** compare(WritableComparable w1, WritableComparable w2) {

CompositeKeyWritable key1 = (CompositeKeyWritable) w1;

CompositeKeyWritable key2 = (CompositeKeyWritable) w2;

**int** cmpResult = key1.deptNo.compareTo(key2.deptNo);

**if** (cmpResult == 0)

{

**return** -key1.emp.compareTo(key2.emp);

}

**return** cmpResult;

}

}

**public** **static** **class** SecondarySortBasicGroupingComparator **extends** WritableComparator {

**protected** SecondarySortBasicGroupingComparator() {

**super**(CompositeKeyWritable.**class**, **true**);

}

@Override

**public** **int** compare(WritableComparable w1, WritableComparable w2) {

CompositeKeyWritable key1 = (CompositeKeyWritable) w1;

CompositeKeyWritable key2 = (CompositeKeyWritable) w2;

**return** key1.deptNo.compareTo(key2.deptNo);

}

}

**public** **static** **class** SecondarySortBasicReducer **extends**

Reducer<CompositeKeyWritable, NullWritable, CompositeKeyWritable, NullWritable> {

@Override

**public** **void** reduce(CompositeKeyWritable key, Iterable<NullWritable> values,

Context context) **throws** IOException, InterruptedException {

**for** (NullWritable value : values) {

context.write(key, NullWritable.*get*());

}

}

}

**public** **static** **void** main(String[] args) **throws** Exception{

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "secsort");

job.setJarByClass(secsort.**class**);

job.setMapperClass(mapper1.**class**);

job.setPartitionerClass(SecondarySortBasicPartitioner.**class**);

job.setSortComparatorClass(SecondarySortBasicCompKeySortComparator.**class**);

job.setGroupingComparatorClass(SecondarySortBasicGroupingComparator.**class**);

job.setReducerClass(SecondarySortBasicReducer.**class**);

job.setMapOutputKeyClass(CompositeKeyWritable.**class**);

job.setMapOutputValueClass(NullWritable.**class**);

job.setOutputKeyClass(CompositeKeyWritable.**class**);

job.setOutputValueClass(NullWritable.**class**);

job.setNumReduceTasks(1);

FileInputFormat.*setInputPaths*(job, **new** Path(args[0]));

FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));

job.waitForCompletion(**true**);

}

}

**FOR UNDERSTANDING ONLY**

-----

import java.util.\*;

import java.io.\*;

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.\*;

import org.apache.hadoop.mapreduce.lib.output.\*;

public class secsort {

//Build-in compare( ) => sorts the key of the Reducer

//override that compare( )

public static class CompositeKeyWritable implements Writable,

WritableComparable<CompositeKeyWritable> {

private String deptNo;

private String emp;

//Empty constructor

public CompositeKeyWritable() {

}

//Constructor with 2 String Args

//this => Accesses Current classes variables

public CompositeKeyWritable(String deptNo, String emp) {

this.deptNo = deptNo;

this.emp = emp;

}

//Merge 2 fields => Using StringBuilder => append(key1 \t key2)

@Override

public String toString() {

//append( ) ⇒ Helps to combine the 2 fields

return (new StringBuilder().append(deptNo).append("\t")

.append(emp)).toString();

}

//Read and Write the data

//Reads the content from the i/p file in deserialized form

public void readFields(DataInput dataInput) throws IOException {

deptNo = WritableUtils.readString(dataInput);

emp = WritableUtils.readString(dataInput);

}

//Writes in Serialized form

public void write(DataOutput dataOutput) throws IOException {

WritableUtils.writeString(dataOutput, deptNo);

WritableUtils.writeString(dataOutput, emp);

}

//Inbuilt func()in MapReduce

public int **compareTo**(CompositeKeyWritable objKeyPair) {

//if the department record is available sort it , internally it sorts the employee details

int result = deptNo.compareTo(objKeyPair.deptNo);

if (0 == result) {

result = emp.compareTo(objKeyPair.emp);

}

return result;

}

}

//O/P Key => CompositeKey

public static class mapper1 extends

Mapper<LongWritable, Text, CompositeKeyWritable, NullWritable> {

@Override

public void map(LongWritable key, Text value, Context context)

throws IOException, InterruptedException {

if (value.toString().length() > 0) {

String arrEmpAttributes[] = value.toString().split(",");

// Creating object for CompositeKeyWritable Class

**//arr{1] Dept**

**//arr[0] Emp**

context.write(

new CompositeKeyWritable( arrEmpAttributes[1].toString(),

arrEmpAttributes[0].toString())), NullWritable.get());

}

}

}

public static class SecondarySortBasicPartitioner extends

Partitioner<CompositeKeyWritable, NullWritable> {

//Return the no.of partitions

@Override

public int getPartition(CompositeKeyWritable key, NullWritable value,

int numReduceTasks) {

//hashkey is created based on hash function on the key-value , based on it partitioners will be created

return (key.deptNo.hashCode() % numReduceTasks);

}

}

public static class SecondarySortBasicCompKeySortComparator extends WritableComparator {

protected SecondarySortBasicCompKeySortComparator() {

//super – access the parent class – compositekeywritable

super(CompositeKeyWritable.class, true);

}

@Override

public int compare(WritableComparable w1, WritableComparable w2) {

CompositeKeyWritable key1 = (CompositeKeyWritable) w1;

CompositeKeyWritable key2 = (CompositeKeyWritable) w2;

int cmpResult = key1.deptNo.compareTo(key2.deptNo);

if (cmpResult == 0)

{

// - => Desc order , remove(-) => Acs Order

return **-**key1.emp.compareTo(key2.emp);

}

return cmpResult;

}

}

//Group the Dept records together => Shuffling

public static class SecondarySortBasicGroupingComparator extends WritableComparator {

protected SecondarySortBasicGroupingComparator() {

super(CompositeKeyWritable.class, true);

}

@Override

public int compare(WritableComparable w1, WritableComparable w2) {

CompositeKeyWritable key1 = (CompositeKeyWritable) w1;

CompositeKeyWritable key2 = (CompositeKeyWritable) w2;

return key1.deptNo.compareTo(key2.deptNo);

}

}

public static class SecondarySortBasicReducer extends

Reducer<CompositeKeyWritable, NullWritable, CompositeKeyWritable, NullWritable> {

//Write in the reducer

@Override

public void reduce(CompositeKeyWritable key, Iterable<NullWritable> values, Context context) throws IOException, InterruptedException {

for (NullWritable value : values) {

context.write(key, NullWritable.get());

}

}

}

public static void main(String[] args) throws Exception{

Configuration conf = new Configuration();

Job job = new Job(conf, "secsort");

job.setJarByClass(secsort.class);

job.setMapperClass(mapper1.class);

job.setPartitionerClass(SecondarySortBasicPartitioner.class);

job.setSortComparatorClass(SecondarySortBasicCompKeySortComparator.class);

job.setGroupingComparatorClass(SecondarySortBasicGroupingComparator.class);

job.setReducerClass(SecondarySortBasicReducer.class);

// If Map and Reduce DataTypes are different use below

job.set**Map**OutputKeyClass(CompositeKeyWritable.class);

job.set**Map**OutputValueClass(NullWritable.class);

// Same Map and Reduce DataType

job.setOutputKeyClass(CompositeKeyWritable.class);

job.setOutputValueClass(NullWritable.class);

job.setNumReduceTasks(1);

FileInputFormat.setInputPaths(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[1]));

job.waitForCompletion(true);

}

}

**DICTIONARY PROGRAM - REVERSE, UPPERCASE**

**ONLY REVERSE**

**import** java.io.IOException;

**import** java.util.\*;

**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.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** revv {

**public** **static** **class** Map **extends** Mapper<LongWritable, Text, Text, IntWritable>

{

IntWritable one = **new** IntWritable(1);

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

String[] line = value.toString().split(",");

**int** n=line.length;

**for**(String i:line)

{

StringBuffer sb = **new** StringBuffer();

sb.append(i);

i = sb.reverse().toString();

context.write(**new** Text(i),one);

}

}

}

**public** **static** **class** Reduce **extends** Reducer<Text, IntWritable, Text, IntWritable> {

**public** **void** reduce(Text key, Iterable<IntWritable> values, Context context)

**throws** IOException, InterruptedException { **int** sum = 0;

**for** (IntWritable val : values) {

sum += val.get();

}

context.write(key, **new** IntWritable(sum));

} }

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "wordcount");

job.setJarByClass(revv.**class**);

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(IntWritable.**class**);

job.setMapperClass(Map.**class**);

job.setReducerClass(Reduce.**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**);

}

}

**REVERSE & UPPERCASE**

**import** java.io.IOException;

**import** java.util.\*;

**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.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** dictrev {

**public** **static** **class** Map **extends** Mapper<Text, Text, Text, Text>

{

String s ="Create" ;

String[] lines;

**public** **void** map(Text key, Text value, Context context) **throws** IOException, InterruptedException

{

**if**(s.equalsIgnoreCase(key.toString()))

{

**int** n;

String[] line = value.toString().split(",");

**int** len;

n=line.length;

**for**(String i:line)

{

StringBuffer s=**new** StringBuffer();

s.append(i);

s.reverse();

len=s.length();

context.write(**new** Text(s.toString()), **new** Text(Integer.*toString*(len)));

}

context.write(key,value);

context.write(**new** Text("uppercase word:"), **new** Text(key.toString().toUpperCase()));

//context.write(new Text("Count"), new Text(Integer.toString(n)));

}

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

conf.set("mapreduce.input.keyvaluelinerecordreader.key.value.separator", "-");

Job job = **new** Job(conf, "dictionary");

job.setJarByClass(dictrev.**class**);

job.setOutputValueClass(Text.**class**);

job.setOutputKeyClass(Text.**class**);

job.setMapperClass(Map.**class**);

job.setInputFormatClass(KeyValueTextInputFormat.**class**);

job.setOutputFormatClass(TextOutputFormat.**class**);

FileInputFormat.*addInputPath*(job, **new** Path(args[0]));

FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));

job.waitForCompletion(**true**);

}

}

**SET MAXIMUM SPLIT – MAXSPLIT / MINSPLIT**

import java.io.IOException;

import java.util.\*;

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.\*;

import org.apache.hadoop.mapreduce.lib.output.\*;

public class maxsplit {

public static class Map extends Mapper<LongWritable, Text, LongWritable, Text> {

public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException

{

context.write(key, value);

}

}

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

// Following Statement sets the min split size...

**conf.set("mapred.min.split.size","10000");**

Job job = new Job(conf, "inputsplit");

job.setJarByClass(maxsplit.class);

job.setOutputKeyClass(LongWritable.class);

job.setOutputValueClass(Text.class);

job.setMapperClass(Map.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);

}

}

**PREVENTING SPLITS**

Some applications don’t want files to be split, so that a single mapper can process each input file in its entirety.

**Entire I/P file goes to a single mapper**

**import** java.io.IOException;

**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.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** splitf {

**public** **static** **class** Map **extends** Mapper<LongWritable, Text, LongWritable, Text> {

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

context.write(key, value);

} }

**public** **class** splitfalse **extends** TextInputFormat {

**protected** **boolean** isSplitable(JobContext context, Path file) {

**return** **false**;

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "splitfalse");

job.setJarByClass(splitf.**class**);

job.setOutputKeyClass(LongWritable.**class**);

job.setOutputValueClass(Text.**class**);

job.setMapperClass(Map.**class**);

**job.setInputFormatClass(splitfalse.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**);

}

}

**PRACTICE PROBLEM LAB**

**The File “customer” has the customer id , Account type (savings/current) and customer name are attributes. The File “amount” has the customer id, balance amount are attributes.**

**a). If the type of account is savings , then add Rs. 10000 in the balance. Write the customer id ,name, balance in HDFS.**

**b) If the type of account is current , then add Rs. 20000 in the balance. Write the customer id ,name, balance in HDFS.**

**Use map reduce programming.**

**REDUCE SIDE JOIN**

**import** java.io.\*;

**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.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** assessr{

**public** **static** **class** custmapper **extends** Mapper<LongWritable,Text,Text,Text>{

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException{

//key => cust\_id

String[] line = value.toString().split(",");

context.write(**new** Text(line[0]), **new** Text(line[2]+","+"account:"+","+line[1])); //key:cust\_id , val:name , account , current/saving

}

}

**public** **static** **class** transmapper **extends** Mapper<LongWritable,Text,Text,Text>

{

**public** **void** map(LongWritable key, Text value, Context context)**throws** IOException, InterruptedException{

String[] line = value.toString().split(",");

//key => cust\_id

context.write(**new** Text(line[0]), **new** Text("amnt:"+","+line[1])); //key:cust\_id val: amnt , bal

}

}

**public** **static** **class** jreducer **extends** Reducer<Text,Text,Text,Text>

{

String st1;

**public** **void** reduce(Text key, Iterable<Text> values, Context context )

**throws** IOException, InterruptedException

{

**int** amt=0;

**for**(Text val:values)

{

String[] line = val.toString().split(",");

**if** (line[1].equals("account:")){

**if** (line[2].equals("current")){

amt+=20000;

st1=line[0].toString();

}

**else** **if** (line[2].equals("savings")){

amt+=10000;

st1=line[0].toString();

}

}

**else** **if** (line[0].equals("amnt:"))

{

amt+=Integer.*parseInt*(line[1]);

}

}

context.write(**new** Text(key), **new** Text(st1+" , "+amt));

} }

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "redjoin");

job.setJarByClass(assessr.**class**);

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(Text.**class**);

job.setReducerClass(jreducer.**class**);

job.setInputFormatClass(TextInputFormat.**class**);

job.setOutputFormatClass(TextOutputFormat.**class**);

MultipleInputs.*addInputPath*(job,**new** Path(args[0]),TextInputFormat.**class**,custmapper.**class**);

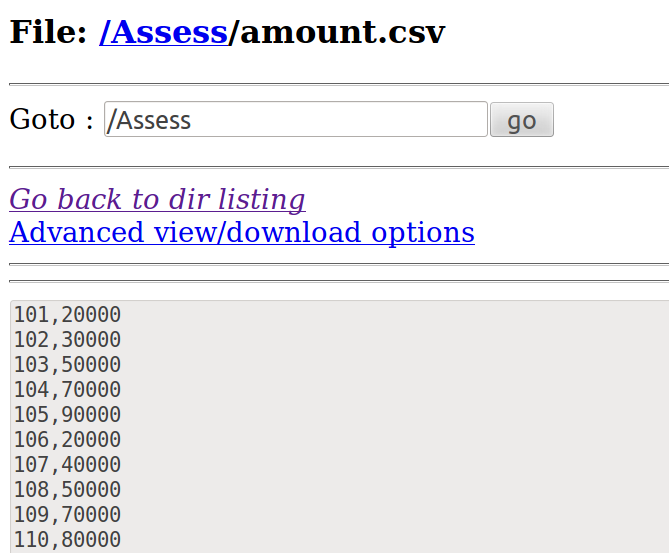
MultipleInputs.*addInputPath*(job,**new** Path(args[1]),TextInputFormat.**class**,transmapper.**class**);

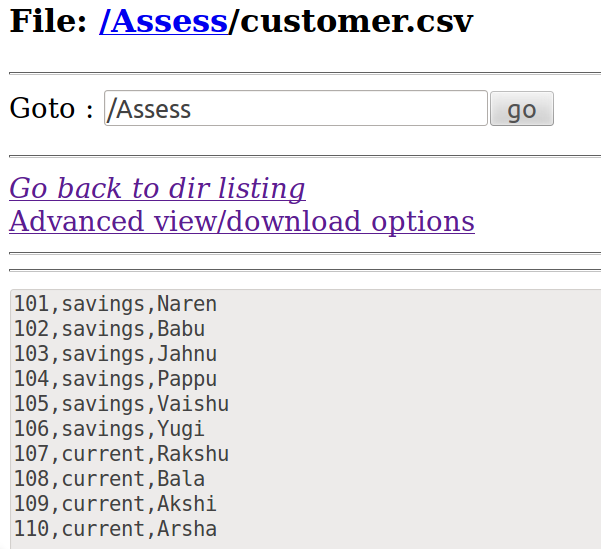
FileOutputFormat.*setOutputPath*(job, **new** Path(args[2]));

job.waitForCompletion(**true**);

}

}

**INPUT**

****

**OUTPUT**

101 Naren , 30000

102 Babu , 40000

103 Jahnu , 60000

104 Pappu , 80000

105 Vaishu , 100000

106 Yugi , 30000

107 Rakshu , 60000

108 Bala , 70000

109 Akshi , 90000

110 Arsha , 100000

***MAP SIDE JOIN – Distributed Cache***

**import** java.io.IOException;

**import** java.util.\*;

**import** org.apache.hadoop.filecache.DistributedCache;

**import** org.apache.hadoop.fs.Path;

**import** java.io.BufferedReader;

**import** java.io.FileReader;

**import** org.apache.hadoop.conf.\*;

**import** org.apache.hadoop.io.\*;

**import** org.apache.hadoop.mapreduce.\*;

**import** org.apache.hadoop.mapreduce.lib.input.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** cust {

**public** **static** **class** Map **extends** Mapper<LongWritable, Text, Text, Text>

{

Path[ ] cfile=**new** Path[0];

ArrayList<Text> cust\_bal=**new** ArrayList<Text>();

**public** **void** setup(Context context)

{

Configuration conf=context.getConfiguration();

**try**

{

cfile = DistributedCache.*getLocalCacheFiles*(conf);

@SuppressWarnings("resource")

BufferedReader reader=**new** BufferedReader(**new** FileReader(cfile[0].toString()));

String line;

**while** ((line=reader.readLine())!=**null**)

{

Text tt=**new** Text(line);

cust\_bal.add(tt);

}

}

**catch**(IOException e)

{

e.printStackTrace();

}

}

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

String line2 = value.toString(); //cust

String[ ] elements=line2.split(",");

**for**(Text e:cust\_bal)

{

String[ ] line1 = e.toString().split(","); //cust\_bal

**if**(elements[0].equals(line1[0])){

**if**(elements[2].equals("savings"))

{

line1[1]= String.*valueOf*(Integer.*parseInt*(line1[1]) +10000);

}

**if**(elements[2].equals("current"))

{

line1[1]= String.*valueOf*(Integer.*parseInt*(line1[1]) + 20000);

}

context.write(**new** Text(elements[0]),**new** Text(elements[1]+" , "+elements[2]+" , "+line1[1]));

}

} } } //BufferedReader class is used to read the text from a character-based input stream

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "cust");

job.setJarByClass(cust.**class**);

DistributedCache.*addCacheFile*(**new** Path(args[0]).toUri(),job.getConfiguration());

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(Text.**class**);

job.setMapperClass(Map.**class**);

job.setInputFormatClass(TextInputFormat.**class**);

job.setOutputFormatClass(TextOutputFormat.**class**);

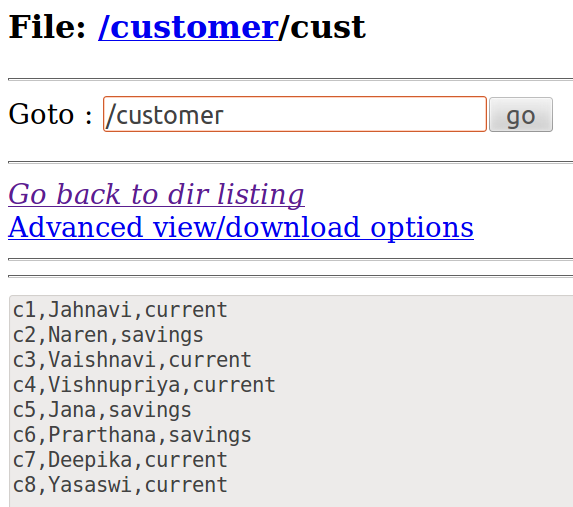
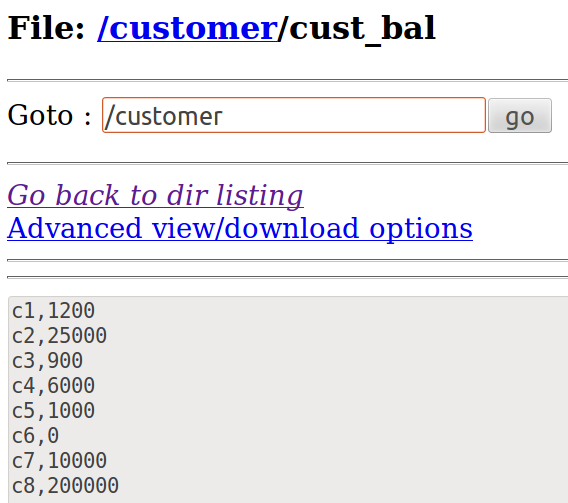
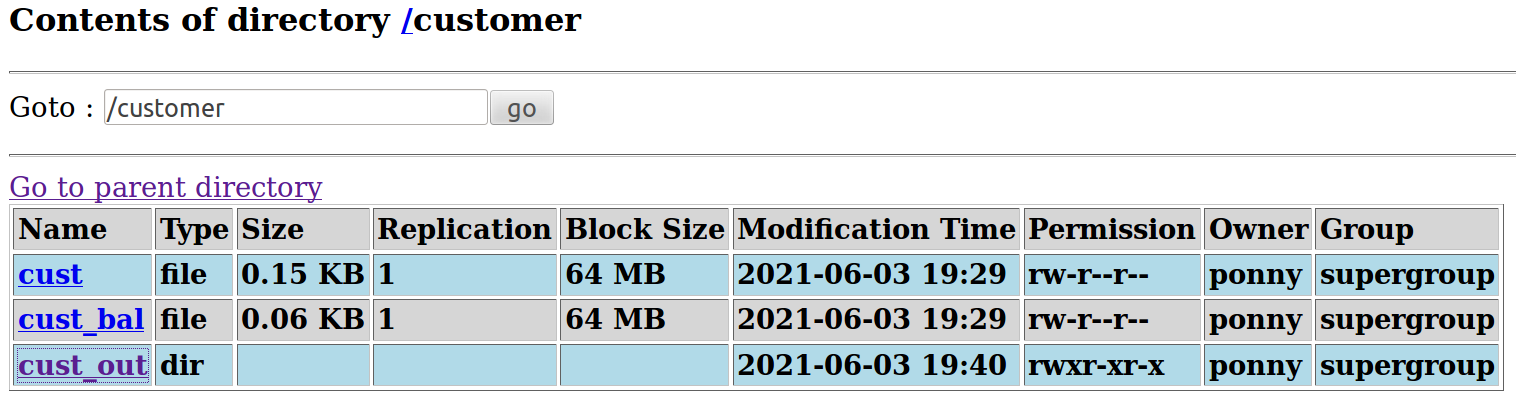
FileInputFormat.*addInputPath*(job, **new** Path(args[1]));

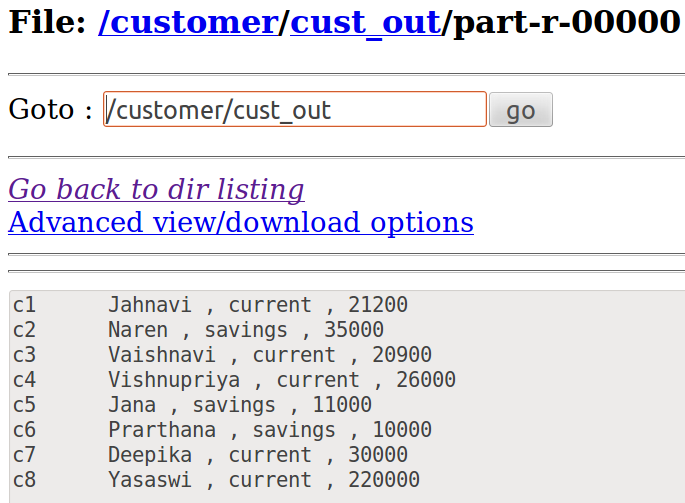
FileOutputFormat.*setOutputPath*(job, **new** Path(args[2]));

job.waitForCompletion(**true**);

}

}



****

2.**Sports** details are maintained in a text file. It has person name, age, sport name, gender and salary. Find the person receiving maximum salary in **each sports category** and write the person name, sport name and the salary in HDFS. The sports details file is in compressed format. Apply map reduce programming to perform the above job.

max(sal) each sports category => Name,sport,max(sal)

**import** java.io.IOException;

**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.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** sport\_count {

**public** **static** **class** FormatConverterMapper **extends** Mapper<LongWritable, Text, Text, Text>

{

IntWritable one = **new** IntWritable(1);

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

String[] line = value.toString().split(",");

context.write(**new** Text(line[2]), **new** Text(line[0]+","+line[1]+","+line[3]+","+line[4]));

//key : sport

//value : name, age, gender, salary

}

}

**public** **static** **class** Reduce **extends** Reducer<Text, Text, Text, Text> {

**public** Text word = **new** Text();

**public** IntWritable maxsum = **new** IntWritable();

String name,age,gender;

**public** **void** reduce(Text key, Iterable<Text> values, Context context)**throws** IOException, InterruptedException {

**int** max = 0;

**int** sum = 0;

**for** (Text val : values) {

String[] line = val.toString().split(",");

sum = Integer.*parseInt*(line[3]);

**if** (sum > max){

max = sum;

name=line[0];

age=line[1];

gender=line[2];

}

}

context.write(key, **new** Text(name+","+age+","+gender+","+max));

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "sport\_count");

job.setJarByClass(sport\_count.**class**);

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(Text.**class**);

job.setMapperClass(FormatConverterMapper.**class**);

job.setReducerClass(Reduce.**class**);

job.setInputFormatClass(SequenceFileInputFormat.**class**);

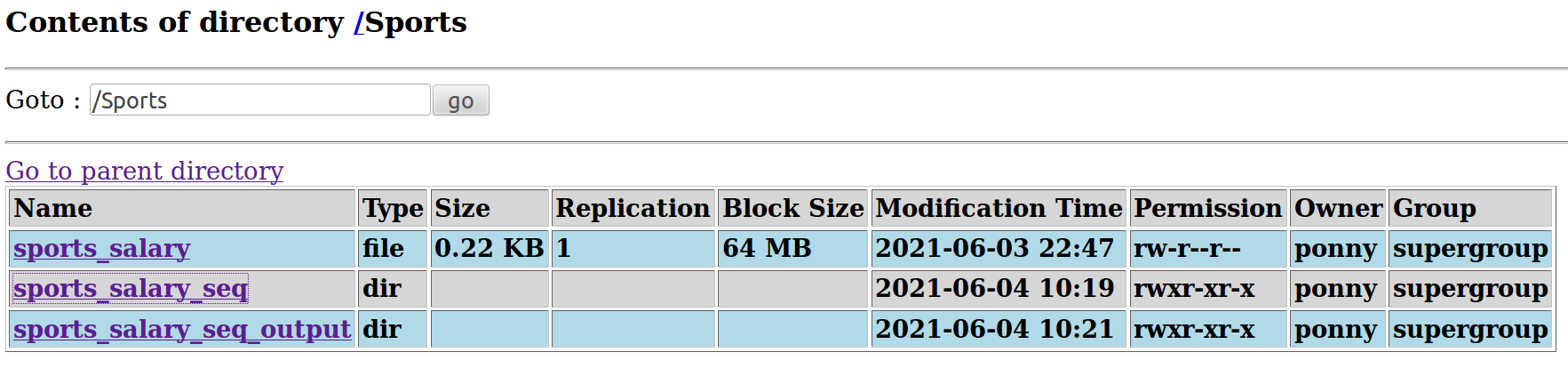
job.setOutputFormatClass(TextOutputFormat.**class**);

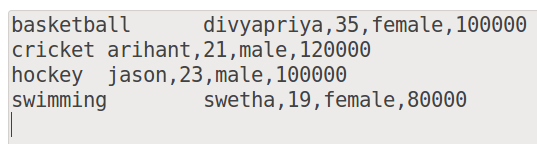
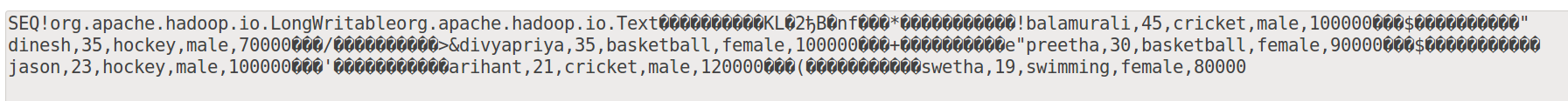
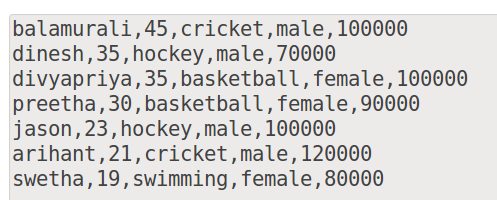
FileInputFormat.*addInputPath*(job, **new** Path(args[0]));

FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));

job.waitForCompletion(**true**);

}

**}**

****

**CAT 2**

**Me:** An organization maintains the salary details that have employee-id, employee-name, age, address, city and salary as attributes. Create 2 salary input files. Write the city name and average salary of each city in HDFS. Write a MapReduce program to produce the output files based on the name of city in a reducer.

**Multiple I/P + Multiple O/P**

import java.io.IOException;

import java.util.\*;

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.\*;

import org.apache.hadoop.mapreduce.lib.output.\*;

public class sal1 {

public static class map1 extends Mapper<LongWritable,Text,Text,IntWritable>{

public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException{

String[] line = value.toString().split(",");

context.write(new Text(line[4]),new IntWritable(Integer.parseInt(line[5])));

}

}

public static class map2 extends Mapper<LongWritable,Text,Text,IntWritable>

{

public void map(LongWritable key, Text value, Context context)throws IOException, InterruptedException{

String[] line = value.toString().split(",");

context.write(new Text(line[4]),new IntWritable(Integer.parseInt(line[5])));

}

}

public static class Reduce extends Reducer<Text, IntWritable, Text, FloatWritable> {

public MultipleOutputs multipleOutputs;

public void setup(Context context) throws IOException, InterruptedException

{

multipleOutputs = new MultipleOutputs(context);

}

public void reduce(Text key, Iterable<IntWritable> values, Context context)

throws IOException, InterruptedException {

int sum=0, n=0;

float avg;

for (IntWritable val : values) {

n+=1;

sum += val.get();

}

avg = sum/n;

multipleOutputs.write(key, new FloatWritable(avg),key.toString());

}

public void cleanup(Context context) throws IOException, InterruptedException

{

multipleOutputs.close();

}

}

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

Job job = new Job(conf, "sal1");

job.setJarByClass(sal1.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

job.setReducerClass(Reduce.class);

job.setInputFormatClass(TextInputFormat.class);

job.setOutputFormatClass(TextOutputFormat.class);

MultipleInputs.addInputPath(job,new Path(args[0]),TextInputFormat.class,map1.class);

MultipleInputs.addInputPath(job,new Path(args[1]),TextInputFormat.class,map2.class);

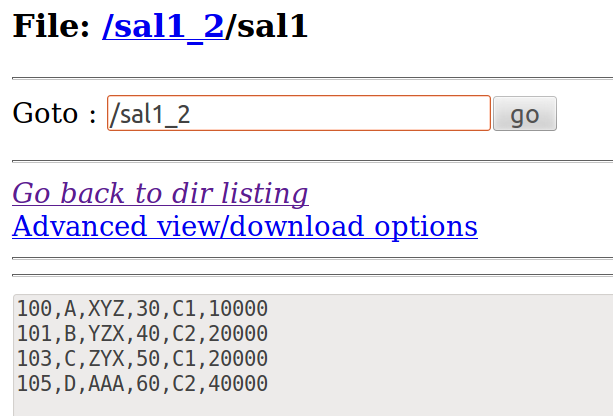
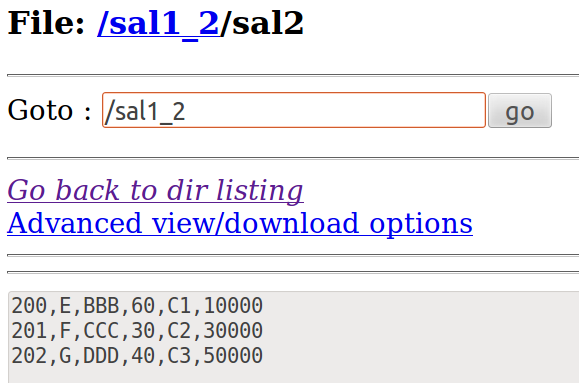
FileOutputFormat.setOutputPath(job, new Path(args[2]));

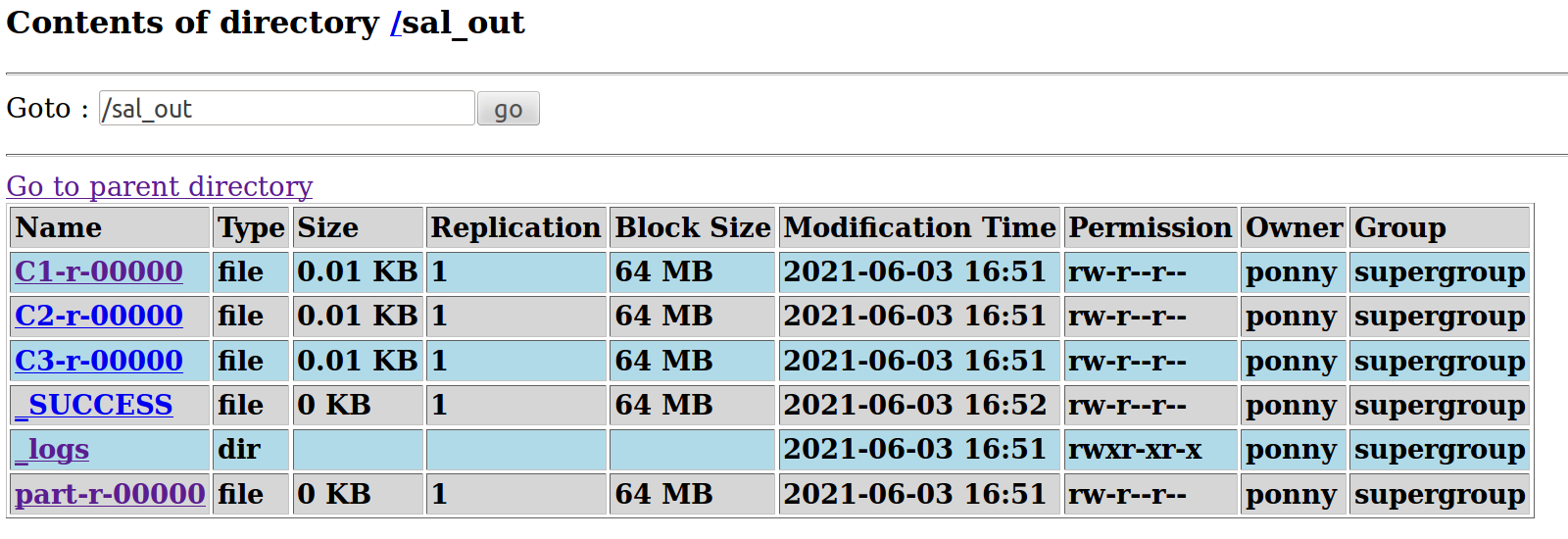
job.waitForCompletion(true);

}

}

**MULTIPLE - INPUT**



**MULTIPLE - OUTPUT**

**Me :** The mega event coordinator has a text file that has Kids-id, Kids- name, event-name and marks as attributes. Here marks attribute is used to specify their score, out of 100 in the competition. The coordinator likes to store the kids-name, marks and event-name in alphabetical order based on kids-name. Also use appropriate concept to find out the number of kids who have scored greater than 90. Develop an appropriate MapReduce program for the same.

**CUSTOM COUNTER : No.of kids + Give kids\_name to Reducer ( A.O. sort )**

**import** java.io.IOException;

**import** java.util.\*;

**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.Reducer.Context;

**import** org.apache.hadoop.mapreduce.lib.input.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** kids {

**public** **enum** ct

{

*cnt* //cnt => Condition 1

};

**public** **static** **class** Map **extends** Mapper<LongWritable, Text, Text, Text>

{

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

String[] line = value.toString().split(",");

context.write(**new** Text(line[1]), **new** Text(line[0]+" "+line[2]+" "+line[3]));

**if**(Integer.*parseInt*(line[3])>90)

{

context.getCounter(ct.*cnt*).increment(1);

}

}

}

**public** **static** **class** Reduce **extends** Reducer<Text, Text, Text, Text>

{

**public** **void** reduce(Text key, Iterable<Text> values, Context context) **throws** IOException, InterruptedException

{

**for** (Text val : values)

{

context.write(key,val);

}

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "kids");

job.setJarByClass(kids.**class**);

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(Text.**class**);

job.setMapperClass(Map.**class**);

job.setReducerClass(Reduce.**class**);

job.setInputFormatClass(TextInputFormat.**class**);

job.setOutputFormatClass(TextOutputFormat.**class**);

FileInputFormat.*addInputPath*(job, **new** Path(args[0]));

FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));

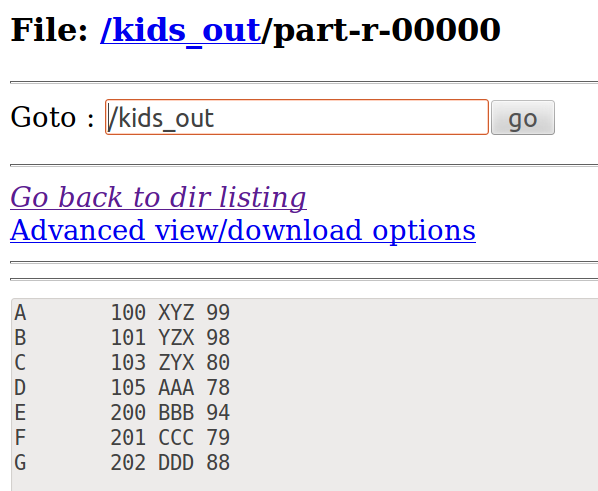
job.waitForCompletion(**true**);

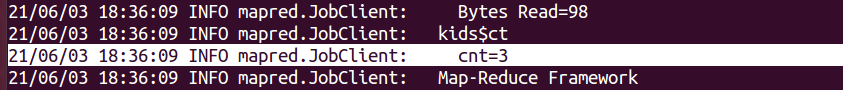
Counters cn = job.getCounters();

cn.findCounter(ct.*cnt*).getValue();

}

}





**Vaishu:** The drawing event coordinator has a text file that has kids-id, kids-name, event-name, and marks as attributes. Here marks attribute is used to specify their score, out of 100 in the competition. The coordinator likes to list out the **5 kids who scored the highest**. Develop a MapReduce program to display the kid's details with their name, marks, and event name. Also, display the **second-highest scored kids** details with the name of the kids be displayed in **uppercase**.

**import** java.io.IOException;

**import** java.util.\*;

**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.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** Treemap\_marks {

**public** **static** **class** Map **extends** Mapper<LongWritable, Text,NullWritable,Text>

{

**private** TreeMap<Integer, Text> marks = **new** TreeMap<Integer, Text>();

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException

{

String[] line = value.toString().split(",");

**int** i = Integer.*parseInt*(line[3]);

marks.put(i,**new** Text(value));

**if** (marks.size() > 5)

{

marks.remove(marks.firstKey());

}

}

**protected** **void** cleanup(Context context) **throws** IOException, InterruptedException

{ **for** ( Text name : marks.values() ) {

context.write(NullWritable.*get*(), name);

}

}

}

**public** **static** **class** Reduce **extends** Reducer<NullWritable, Text, NullWritable, Text>

{

**int** k = 1;

**public** **void** reduce(NullWritable key, Iterable<Text> values, Context context) **throws** IOException, InterruptedException {

TreeMap<Integer, Text> marks = **new** TreeMap< Integer, Text>();

**for** (Text value : values)

{

String line = value.toString();

String[] elements=line.split(",");

**int** i= Integer.*parseInt*(elements[3]);

marks.put(i, **new** Text(value));

**if** (marks.size() > 5)

{

marks.remove(marks.firstKey());

}

}

**for** (Text t : marks.values())

{

String[] line = t.toString().split(",");

**if**(k==4)

{

line[1] = line[1].toUpperCase();

}

t = **new** Text(line[0]+','+line[1]+','+line[2]+','+line[3]);

context.write(NullWritable.*get*(), t);

k++;

}

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "kids");

job.setJarByClass(Treemap\_marks.**class**);

job.setOutputKeyClass(NullWritable.**class**);

job.setOutputValueClass(Text.**class**);

job.setNumReduceTasks(1);

job.setMapperClass(Map.**class**);

job.setReducerClass(Reduce.**class**);

job.setInputFormatClass(TextInputFormat.**class**);

job.setOutputFormatClass(TextOutputFormat.**class**);

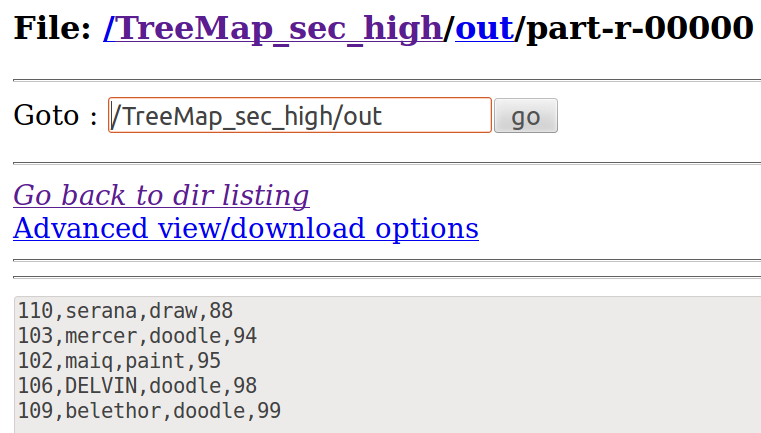
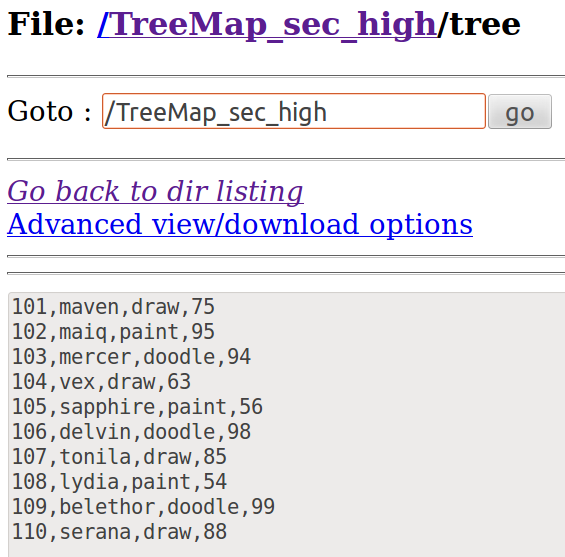
FileInputFormat.*addInputPath*(job, **new** Path(args[0]));

FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));

job.waitForCompletion(**true**);

}

}



**UG PG / UGPG**

**Student details are maintained in the files “st” and “mark”. The File “st” has the student-id , student-type (UG/PG) and student-name are attributes. The File “mark” has the student-id, (marks of 4 subjects) M1, M2, M3 and M4 are attributes.  
  
a). If the type of student belongs to ”UG” , then write the student-id, student-type, student-name and average marks of the corresponding student in HDFS.  
  
b) If the student-type is “PG”, then write the student-id, student-type, student-name and total marks of the corresponding student in HDFS.  
  
Use map reduce programming.**

import java.io.IOException;

import java.util.\*;

import org.apache.hadoop.filecache.DistributedCache;

import org.apache.hadoop.fs.Path;

import java.io.BufferedReader;

import java.io.FileReader;

import org.apache.hadoop.conf.\*;

import org.apache.hadoop.io.\*;

import org.apache.hadoop.mapreduce.\*;

import org.apache.hadoop.mapreduce.lib.input.\*;

import org.apache.hadoop.mapreduce.lib.output.\*;

public class ugpg {

public static class Map extends Mapper<LongWritable, Text, Text, Text>

{

Path[ ] cfile=new Path[0];

ArrayList<Text> ugpg=new ArrayList<Text>();

public void setup(Context context)

{

Configuration conf=context.getConfiguration();

try

{

cfile = DistributedCache.getLocalCacheFiles(conf);

@SuppressWarnings("resource")

BufferedReader reader=new BufferedReader(new FileReader(cfile[0].toString()));

String line;

while ((line=reader.readLine())!=null)

{

Text tt=new Text(line);

ugpg.add(tt);

}

}

catch(IOException e)

{

e.printStackTrace();

}

}

public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

String line2 = value.toString(); //ugpg

String[ ] elements=line2.split(",");

float ans=0;

for(Text e:ugpg)

{

String[ ] line = e.toString().split(","); //ugpg

**if(elements[0].equals(line[0])){**

if(elements[1].equals("pg"))

{

String [] marks = {line[1],line[2],line[3],line[4],line[5]};

float n = marks.length;

float sum=0;

for(int i=0; i< marks.length; i++)

{

sum = sum + Float.parseFloat(marks[i]);

ans = sum;

}

}

if(elements[1].equals("ug"))

{

String [] marks = {line[1],line[2],line[3],line[4],line[5]};

float n = marks.length;

float sum=0;

for(int i=0; i< marks.length; i++)

{

sum = sum + Float.parseFloat(marks[i]);

ans= sum/n;

}

}

context.write(new Text(elements[0]),new Text(elements[1]+" , "+elements[2]+" , "+String.valueOf(ans)));

}

} } } //BufferedReader class is used to read the text from a character-based input stream

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

Job job = new Job(conf, "ugpg");

job.setJarByClass(ugpg.class);

DistributedCache.addCacheFile(new Path(args[0]).toUri(),job.getConfiguration());

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

job.setMapperClass(Map.class);

job.setInputFormatClass(TextInputFormat.class);

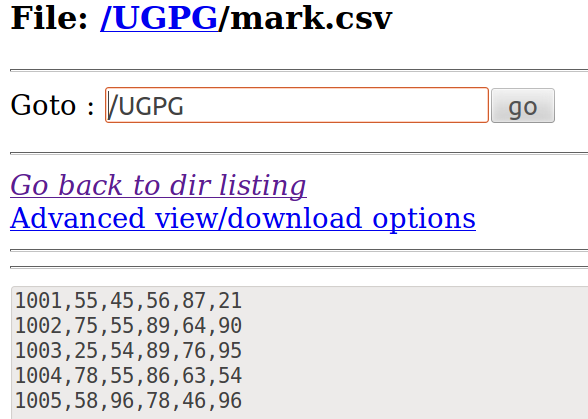
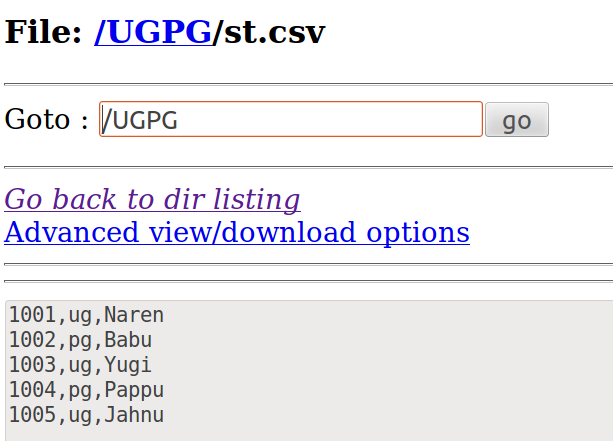
job.setOutputFormatClass(TextOutputFormat.class);

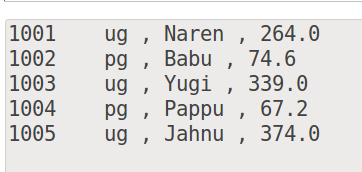
FileInputFormat.addInputPath(job, new Path(args[1]));

FileOutputFormat.setOutputPath(job, new Path(args[2]));

job.waitForCompletion(true);

}

}

****

**Prar:** An organization maintains the salary details that have employee-id, employee-name, age, city, country, and salary as attributes. Create 3 salary input files. **Partition** the files based on the **country**. Write the **country name and average salary** of each country in HDFS. Use map reduce programming.

**import** java.io.IOException;

**import** org.apache.hadoop.conf.Configuration;

**import** org.apache.hadoop.fs.Path;

**import** org.apache.hadoop.io.FloatWritable;

**import** org.apache.hadoop.io.IntWritable;

**import** org.apache.hadoop.io.LongWritable;

**import** org.apache.hadoop.io.Text;

**import** org.apache.hadoop.mapreduce.Job;

**import** org.apache.hadoop.mapreduce.Mapper;

**import** org.apache.hadoop.mapreduce.Partitioner;

**import** org.apache.hadoop.mapreduce.Reducer;

**import** org.apache.hadoop.mapreduce.Reducer.Context;

**import** org.apache.hadoop.mapreduce.lib.input.MultipleInputs;

**import** org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

**import** org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

**import** org.apache.hadoop.mapreduce.lib.output.MultipleOutputs;

**import** org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

**public** **class** prar\_country{

**public** **static** **class** mapper1 **extends** Mapper<LongWritable,Text,Text,IntWritable>{

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException{

String[] line = value.toString().split(",");

context.write(**new** Text(line[4]), **new** IntWritable(Integer.*parseInt*(line[5])));

}

}

**public** **static** **class** jreducer **extends** Reducer<Text,IntWritable,Text,FloatWritable>

{

**public** FloatWritable res1 = **new** FloatWritable();

**public** Text wrd1 = **new** Text();

**public** MultipleOutputs multipleOutputs;

**public** **void** setup(Context context) **throws** IOException, InterruptedException

{

multipleOutputs = **new** MultipleOutputs(context);

}

**public** **void** reduce(Text key, Iterable<IntWritable> values, Context context) **throws** IOException, InterruptedException {

**int** sum = 0;

**int** count=0;

**float** avg;

**for** (IntWritable val : values) {

multipleOutputs.write(key, **new** IntWritable(val.get()),key.toString());

count+=1;

sum+=val.get();

}

avg=sum/count;

multipleOutputs.write(**new** Text("Average salary of "+key+":"),**new** FloatWritable(avg),key.toString());

}

**public** **void** cleanup(Context context) **throws** IOException, InterruptedException

{

multipleOutputs.close();

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "wordcount");

job.setJarByClass(prar\_country.**class**);

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(IntWritable.**class**);

job.setReducerClass(jreducer.**class**);

job.setInputFormatClass(TextInputFormat.**class**);

job.setOutputFormatClass(TextOutputFormat.**class**);

MultipleInputs.*addInputPath*(job,**new** Path(args[0]),TextInputFormat.**class**,mapper1.**class**);

MultipleInputs.*addInputPath*(job,**new** Path(args[1]),TextInputFormat.**class**,mapper1.**class**);

MultipleInputs.*addInputPath*(job,**new** Path(args[2]),TextInputFormat.**class**,mapper1.**class**);

FileOutputFormat.*setOutputPath*(job, **new** Path(args[3]));

job.waitForCompletion(**true**);

}

}

**Prar:** The Government agency is maintaining the citizen's details. The citizen’s details (i.e name, DOB (date, month, year), location) are maintained in the text file. Use a suitable concept to identify the total number of citizens whose age is above 30. Write the **citizen name** and the **age** satisfying the above condition in HDFS. Develop a MapReduce program for the same. Use **year in DOB** attribute for age calculation.

**import** java.io.IOException;

**import** java.util.\*;

**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.Reducer.Context;

**import** org.apache.hadoop.mapreduce.lib.input.\*;

**import** org.apache.hadoop.mapreduce.lib.output.\*;

**public** **class** prar\_date {

**public** **static** **class** Map **extends** Mapper<LongWritable, Text, Text, Text>

{

IntWritable one = **new** IntWritable(1);

**public** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

String[] line = value.toString().split(",");

context.write(**new** Text(line[0]),**new** Text(line[1]+','+line[2]));

}

}

//input key is name,value is dob and location

**public** **static** **class** Reduce **extends** Reducer<Text, Text, Text, Text> {

**public** **void** reduce(Text key, Iterable<Text> values, Context context) **throws** IOException, InterruptedException {

**for** (Text val : values) {

String[] line = val.toString().split(","); //splitting the dob and location

String[] Date = line[0].split("/"); // splitting the date into dd mm yy and storing them in an array date[]

**int** day=(3-(Integer.*parseInt*(Date[0]))+30)%30; //calculating the diff between the days

**int** month=(6-(Integer.*parseInt*(Date[1]))+12)%12; //calculating the diff between months

**int** year=2021-(Integer.*parseInt*(Date[2])); //calculating the diff between years

**double** no\_years= (day+(month\*30)+(year\*365.25))/365.25;

**if**(no\_years>30.0){

context.write(key,val);

}

}

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration conf = **new** Configuration();

Job job = **new** Job(conf, "prar\_date");

job.setJarByClass(prar\_date.**class**);

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(Text.**class**);

job.setMapperClass(Map.**class**);

job.setReducerClass(Reduce.**class**);

job.setInputFormatClass(TextInputFormat.**class**); job.setOutputFormatClass(TextOutputFormat.**class**);

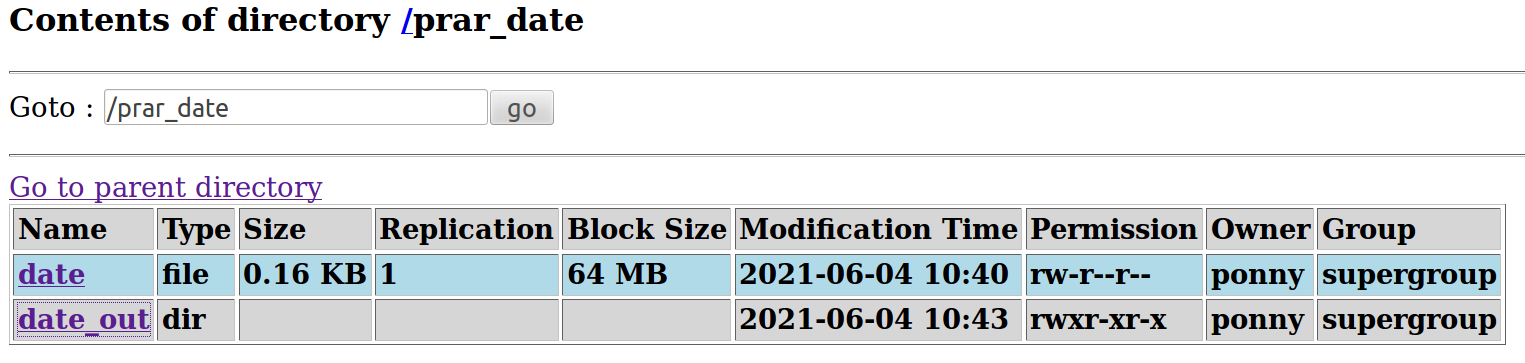
FileInputFormat.*addInputPath*(job, **new** Path(args[0]));

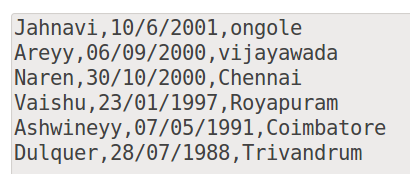
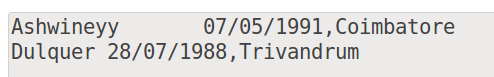
FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));

job.waitForCompletion(**true**);

}

}





**FAT LAB**

I. The electricity department maintains the power consumption details of the customers. Customer name,city and the units consumed are maintained in the text file. The unit values are ranged from 500 to 20,000. Department wishes to store the customer details in 4 different files based on the units consumed as:  
1. Customer details with the units <1000  
2. Customer details with the units <3000  
3. Customer details with the units >5000  
4. Customer details with the units other than the above three conditions.  
Write the unique city names and the count of the cities in HDFS.  
Use map reduce programming to perform the above job.  
Note: Create a text file with minimum of 10 records.

import java.io.IOException;

import java.util.\*;

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.Reducer.Context;

import org.apache.hadoop.mapreduce.lib.input.\*;

import org.apache.hadoop.mapreduce.lib.output.\*;

public class current {

public static class Map extends Mapper<LongWritable, Text, Text, IntWritable>

{

public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

String[] line = value.toString().split(",");

//Key: country, Value: Salary

context.write(new Text(line[1]),new IntWritable(Integer.parseInt(line[2])));

}

}

public static class dpart extends Partitioner<Text,IntWritable>

{

//"nr" receives the value from setNumReduceTasks( \_ )

public int getPartition(Text key,IntWritable value,int nr)

{

if(value.get()<1000)

return 0;

if(value.get() < 3000)

return 1;

if(value.get() > 5000)

return 2;

else

return 3;

}

}

public static class Reduce extends Reducer<Text, IntWritable, Text, Text> {

int count=0;

Text store;

public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {

for (IntWritable val : values) {

context.write(key,new Text("- Units: " + val.toString()));

store=key;

count++;

}

context.write(key, new Text("- Count: " +count+"\n"));

}

}

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

Job job = new Job(conf, "current");

job.setJarByClass(current.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

job.setMapperClass(Map.class);

job.setReducerClass(Reduce.class);

// give the no. of reducer

job.setPartitionerClass(dpart.class);

job.setNumReduceTasks(4);

job.setInputFormatClass(TextInputFormat.class);

job.setOutputFormatClass(TextOutputFormat.class);

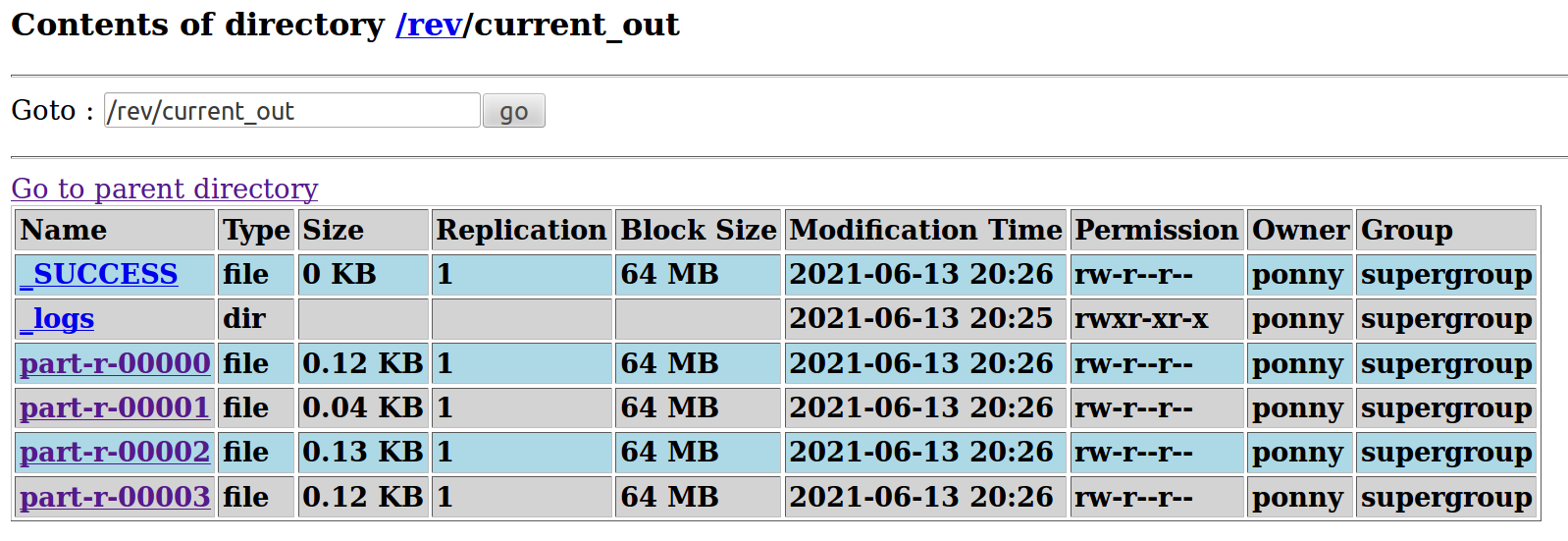
FileInputFormat.addInputPath(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[1]));

job.waitForCompletion(true);

}

}



**Missing Count / Missing column**

Identify missing data , count

If a particular value is empty , count++

prod , country name

under country find the missing

import java.io.IOException;

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.\*;

import org.apache.hadoop.mapreduce.lib.output.\*;

public class missingcount {

public enum ct

{

cnt //cnt => Condition 1

};

public static class Map extends Mapper<LongWritable, Text, Text, Text>

{

public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

String[] line = value.toString().split(",");

if(line[7].equals(""))

{

context.getCounter(ct.cnt).increment(1);

line[7]="Empty";

}

context.write(new Text(line[1]),new Text(line[7]));

}

}

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

Job job = new Job(conf, "missingcount");

job.setJarByClass(missingcount.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

job.setMapperClass(Map.class);

//job.setReducerClass(Reduce.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);

Counters cn = job.getCounters();

cn.findCounter(ct.cnt).getValue();

}

}

**A social media maintains the Friendship Network in the text file. Text file maintains the network as person name and friend name. Identify the occurrences of the friends names using map reduce programming.**

Note: Create a text file with minimum of 15 records.

Sample Text file

Ram,raj

Ram,tej

Tej,raj

Raj,kamal

Tej,ravi

Karthik,anand

Sample Output

Raj - 2

Tej-1

Kamal-1

Ravi-1

Anand-1

**HIVE Query**

cd hadoop/ cd hive-0.9.0-bin/

cd bin

./hive

**Enter into hive:**  
cd /hadoop/hive-0.9.0-bin/bin$ ./hive  
  
**Display the Database:**  
show databases;

**Display the Tables:**

show tables;  
  
**Create a database:**  
create database db1;  
  
**To find the location of the DB:**  
describe database db1;  
  
**To use DB:**  
use db1;  
  
**Create table:**  
create table dt(a int,b string,c int,d int,e int) **row** format delimited fields terminated by ',' stored as **textfile**;  
By default => Textfile Format

**Load file content into the table:**  
load data inpath '/mar' into table dt;  
load data inpath '/mar' overwrite into table dt;

**Load file from local directory**load data local inpath ‘/home/ponny/file.txt into table dt;   
  
**Display the Table contents:**  
select \* from dt;  
  
**Create a table like the old table:**  
create table dt1 **like** dt;  
load data local inpath '/mar' into table dt1;  
  
**Create a table like the old table along with the contents:**  
create table dt2 as select\*from dt1;  
  
**Drop table**  
drop table dt2;  
  
**Drop Databases along with Tables inside**  
drop database if exists dk **cascade**;  
  
**Add columns to existing table:**  
alter table dt add columns(Class\_section STRING);  
  
**Rename columns:**  
alter table dt change column Class\_section Class\_\_section STRING;  
  
**Alter table and change position of column**  
alter table dt change column Class\_\_section Class\_section STRING after avg;  
  
**Replace Columns**  
alter table dt replace columns(a string,b int, c int);  
  
**External files:**  
create external table et(a int, b string) row format delimited fields terminated by ',' location '/user/ponny/newfile/';  
  
**Describe Extended table:**  
describe extended et;

**DROP DATABASE IF EXISTS db\_name CASCADE;**

**Add Columns to existing table**

ALTER TABLE log\_messages ADD COLUMNS ( app\_name STRING, session\_id LONG);

**Rename Table**

ALTER TABLE before\_tname RENAME TO after\_tname;

**Alter Column ( Name / Datatype )**

alter table t4 change column oldcolname newcolname datatype\_newcol;

alter table d1 change column c y int ;

**Rename and change the position of column after some col (k)**

ALTER TABLE Tname change column oldcolname newcolname datatype\_newcol after k;

alter table d1 change column c y int after x;

**Replace columns**

Remove the existing columns and replace the positions with only selected columns;

a , b , d => x , y , d

ALTER TABLE d1 REPLACE COLUMNS(x int, y string, d int) ;

describe d1;

OK

x int

y string

d int

**Load File to Table**

load data inpath '/hive/lab1' into table **t1**; // From HDFS

load data **local** inpath ‘/home/ponny/ “ into table t1; // From OS

**local ⇒ from OS**

**in hdfs ⇒ remove local**

**Import file from external directory not from HIVE warehouse**

Hive won't take the ownership of this table

while dropping metadata will be deleted and not the contents

create **external table** et(a int, b string) row format delimited fields terminated by ',' location '/home/ponny/Desktop/Hive\_tt/ext\_table';

hive> describe et;

OK

a int

b string

### check table path;

*describe extended tname;*

External table location is in some other location

location:hdfs://localhost:54310/home/ponny/Desktop/Hive\_tt/ext\_table

Internal table location is in Hive warehouse

location:hdfs://localhost:54310/user/hive/warehouse/c2.db/d1

### Table Creation - stud table

**cols : name , id , marks , grade ,address, contact\_num**

create table stud( name string, id int, marks int, grade string, address string, contact\_num bigint) row format delimited fields terminated by ',' stored as textfile;

**describe stud;**

OK

name string

id int

marks int

grade string

address string

contact\_num bigint

Write a query...

1. **Using concat() , join address and contact\_num**

SELECT name, concat(column1,column2) AS x FROM table;

SELECT name , CONCAT(address,contact\_num) AS concated FROM stud;

Naren Tambaram9840999509

Babu Chitlapakkam9988776655

Kavya Arthi Tower8877665544

Pappu Kunjappan7766554433

Jahnu Ongole6655443322

Vaishu Beach9977553311

**add “ - “ inbetween**

SELECT name , CONCAT(address,’ - ‘, contact\_num) AS concated FROM stud;

OK

Naren Tambaram - 9840999509

Babu Chitlapakkam - 9988776655

Kavya Arthi Tower - 8877665544

Pappu Kunjappan - 7766554433

Jahnu Ongole - 6655443322

Vaishu Beach – 9977553311

**2. Max(Marks) , Min(Marks) , Avg(Marks) of the class from the Table**

SELECT concat(' Max : ',max(marks), ' - Min : ',min(marks),' - Avg : ',avg(marks)) FROM stud;

Max : 100 - Min : 93 - Avg : 96.5

**3.Stud\_name to Uppercase**

select id, ucase(name),contact\_num from stud;

1001 NAREN 9840999509

1002 BABU 9988776655

1003 KAVYA 8877665544

1004 PAPPU 7766554433

1005 JAHNU 6655443322

1006 VAISHU 9977553311

**LAB**

**1. Create a table with column names as**

**1.ID**

**2.NAME**

**3.DOB**

**4.GENDER**

create table t1(id int,name string,dob string,gender string) row format delimited fields terminated by ',' stored as textfile;

describe t1;

**Textfile ⇒ HDFS**

hadoop fs -copyFromLocal /home/ponny/Desktop/Hive\_tt/lab1 /hive/

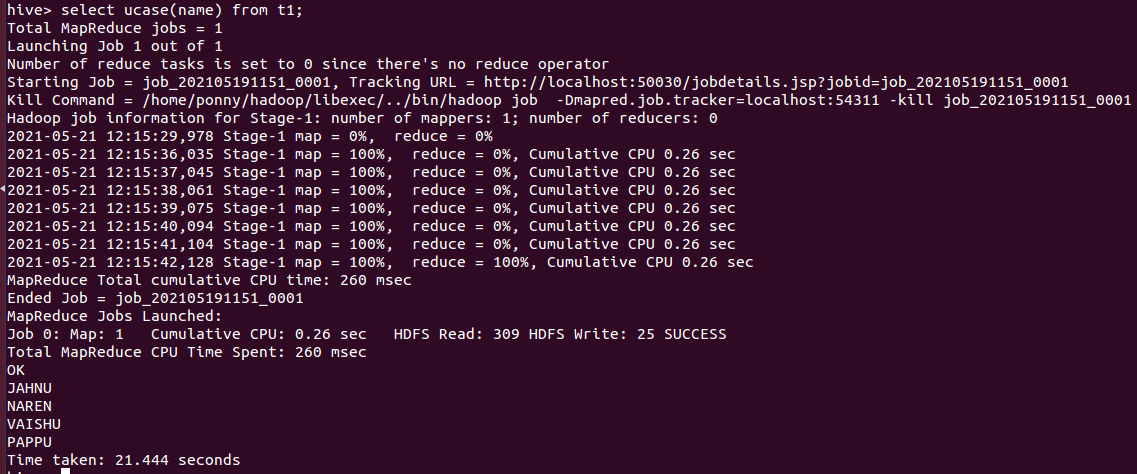
**Load File to Table**

load data inpath '/hive/lab1' into table **t1**;



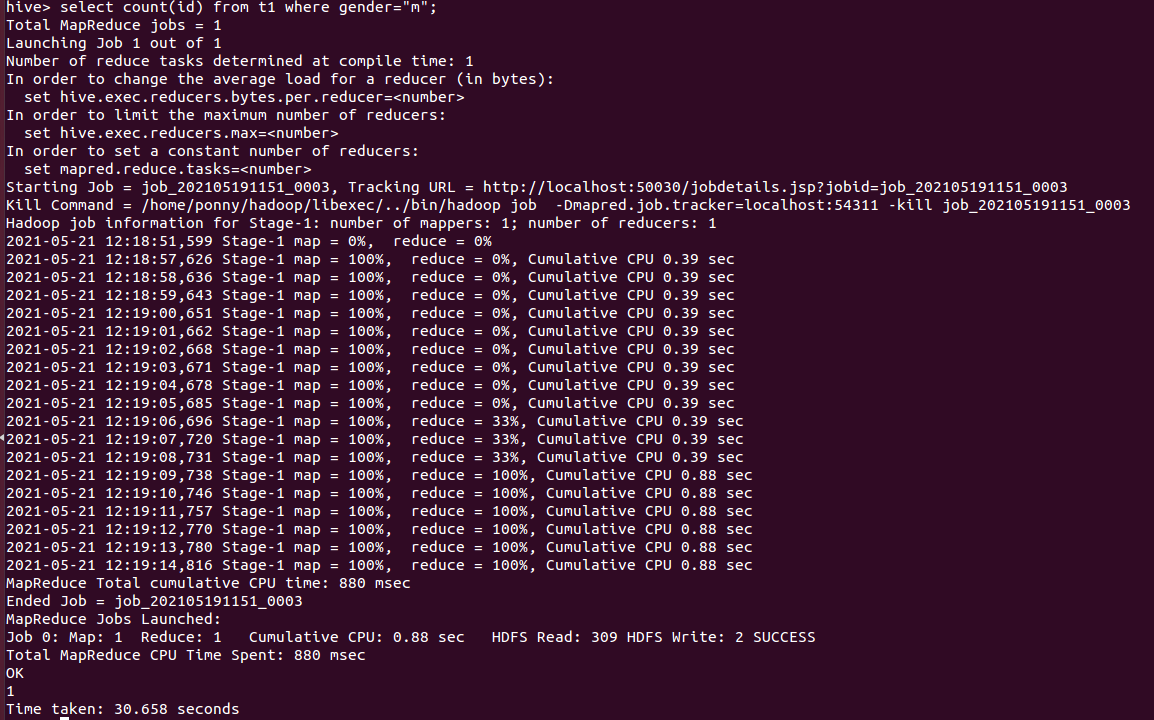
**Q1.Write the query to display name in uppercase letters**

select ucase(name) from t1;

****

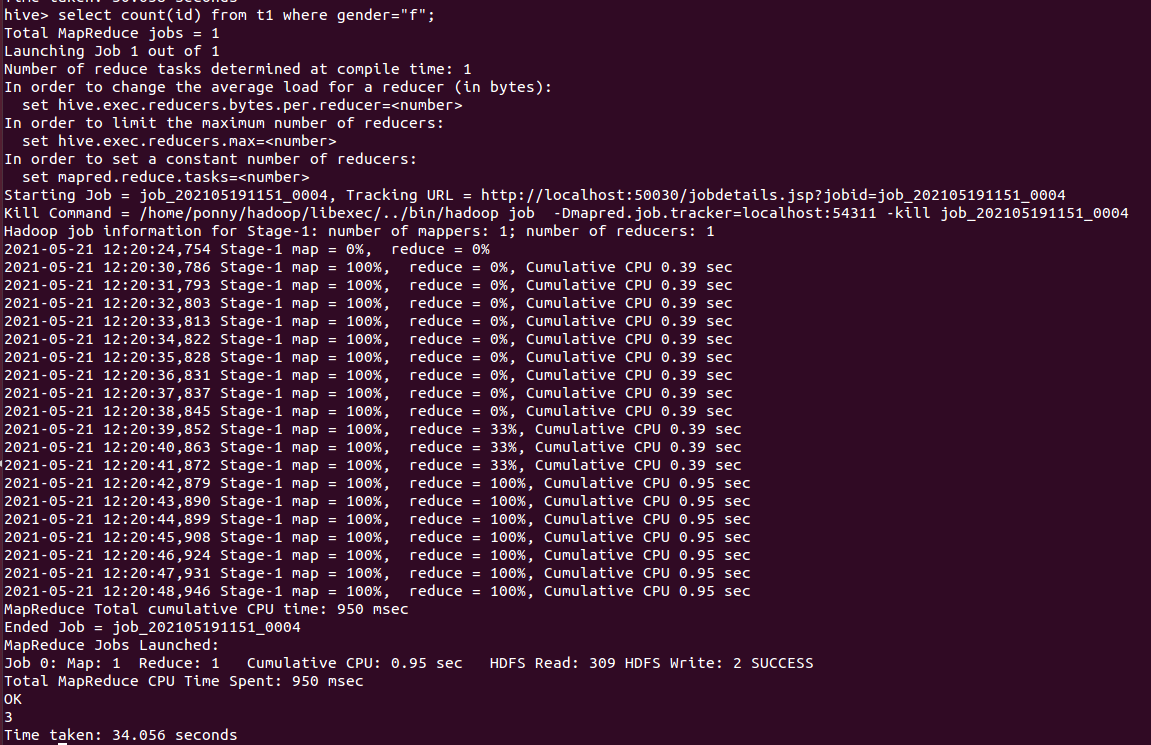
**Q2.write the query to count number of male and female.**

select count(id) from t1 where gender="m";

1

Select count(id) from t1 where gender="f";

3

****

**2. Create a table with column as**

**1.FID**

**2.FNAME**

**3.GENDER**

**4.DOB**

**5.QUALIFICATION**

**6.JOB**

**7.CONTACT NUMBER**

**8.ADDRESS**

**9.FRIENDS**

**10.VIDEOS SHARED**

**11.MSG(POSTED/SHARED)**

create table t2( fid int,fname string,gender string,dob string,qual string, job string, cnum int, addr string, friends int, vids string, msg string) row format delimited fields terminated by ',' stored as textfile;

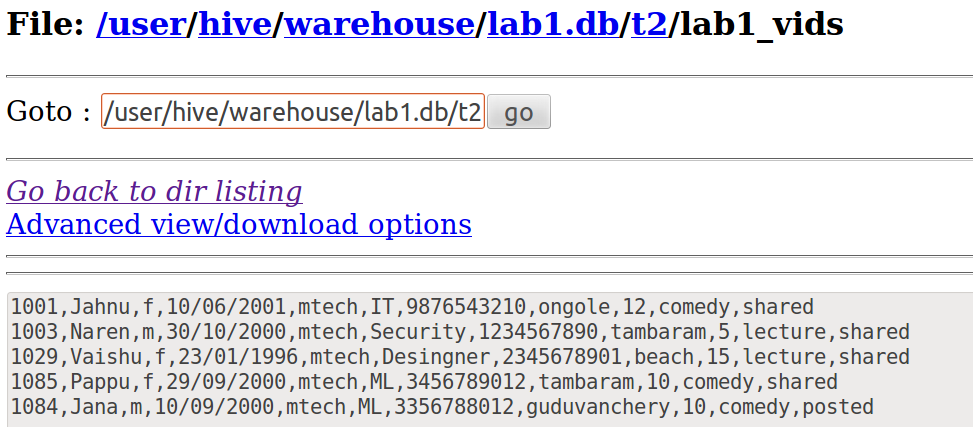
describe t2;

**Textfile ⇒ HDFS**

hadoop fs -copyFromLocal /home/ponny/Desktop/Hive\_tt/lab1\_vids /hive/

**Load File to Table**

load data inpath '**/hive/lab1\_vids**' into table **t2**;



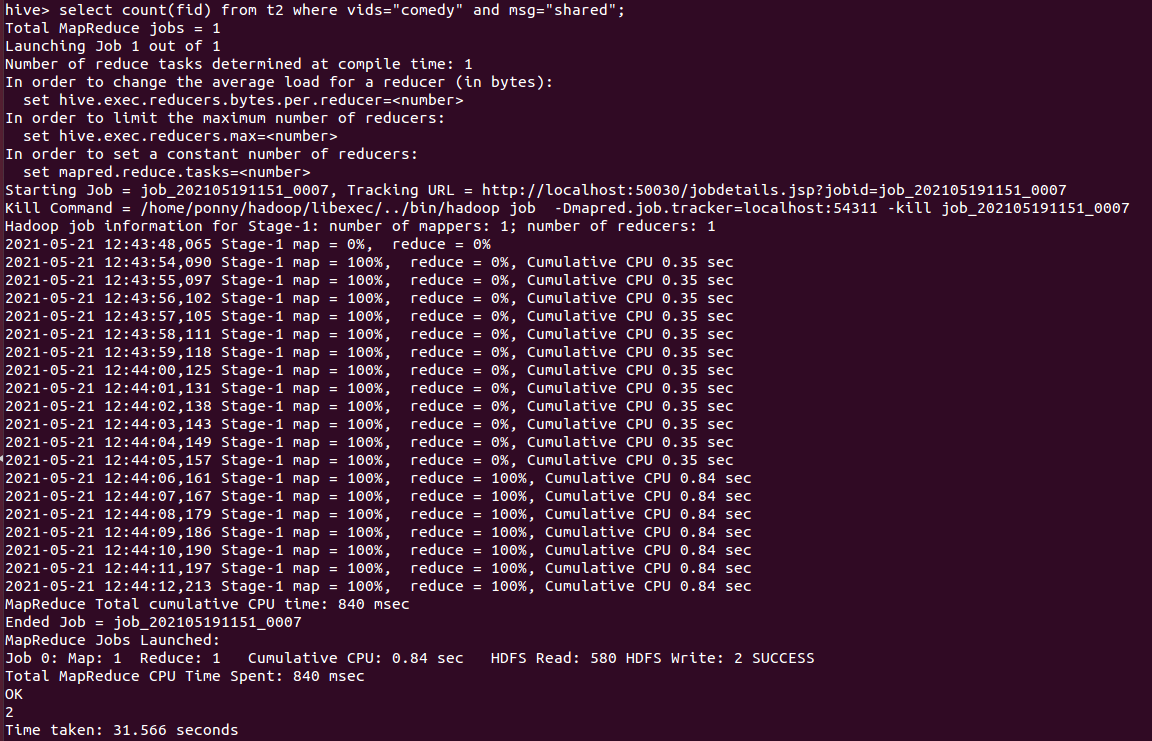
**Q1.Write the Query to get the count of lecture videos shared from the database**

select count(fid) from t2 where vids="lecture" and msg="shared";

****

**Q2.Write the Query to get th count of comedy videos shared from the database**

select count(fid) from t2 where vids="comedy" and msg="shared";



**Partitioning and Bucketing**

* Helps to improve hive performance
* Splits (table) ⇒ creates subdirectories / sampling of the table
* Horizontally the load gets balanced

**1. Dynamic Partition**

**non-strict⇒ dynamic**

by default => strict

Partition based on a column

set hive.exec.dynamic.partition = true;

set hive.exec.dynamic.partition.mode = nonstrict;

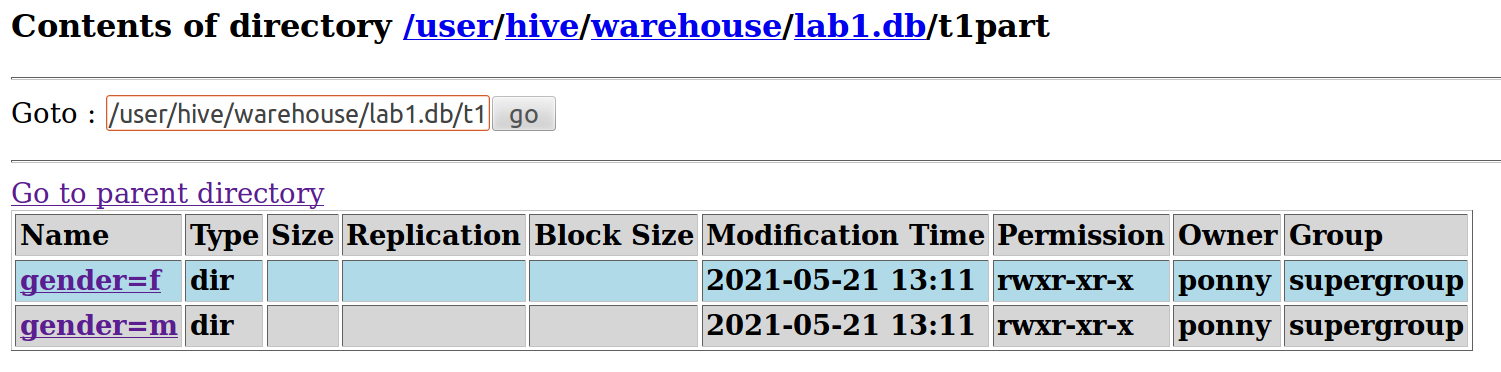
create table t1part(id int,name string,dob string) partitioned by (gender string) row format delimited fields terminated by ',' stored as textfile;

**insert ⇒** insert the table values from one table **(not partitioned)** to other **(newly created partitioned table)**

insert into table t1part partition(gender) select id ,name, dob, gender from t1;

**NOTE :** partitioned col is given at the end , bcde"q"

**LOCATION :** /user/hive/warehouse/lab1.db/t1part



**2.Static Partition**

**strict ⇒ static**

**NOTE :** c=90, only for that condition a partition will be created

set hive.exec.dynamic.partition.mode = strict;

create table stpq(a int,b string,d int,e int) partitioned by (c int) row format delimited fields terminated by ',' stored as textfile;

insert into table stpq partition(c = 90) select a,b,d,e from dtbl where c = 90;

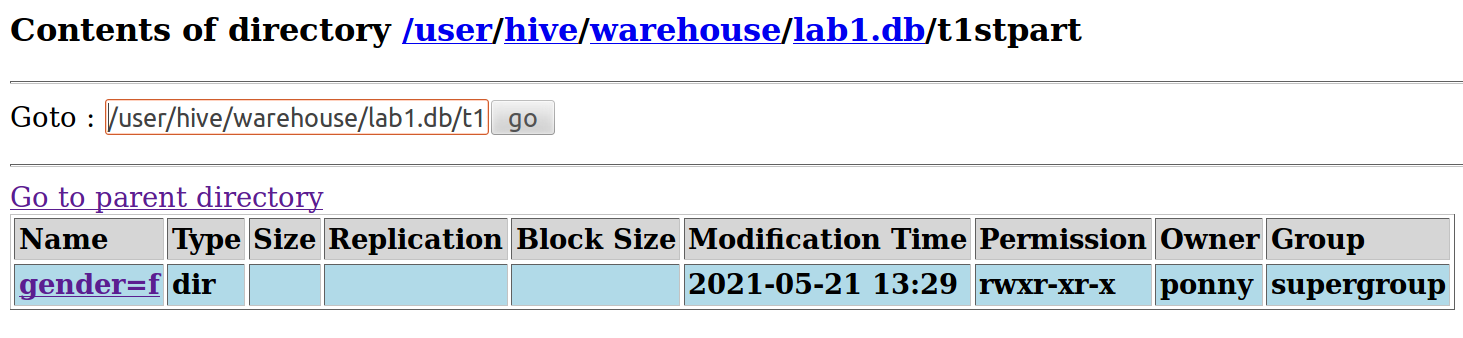
**Partition the table using the column “gender”**

**NOTE :** Based on every value of the col “gender” a directory will be created

set hive.exec.dynamic.partition.mode = strict;

create table t1stpart(id int,name string,dob string) partitioned by (gender string) row format delimited fields terminated by ',' stored as textfile;

insert into table t1stpart partition(gender='f') select id ,name, dob from t1 where gender='f';

**LOC****ATION :** /user/hive/warehouse/lab1.db/t1stpart

**NOTE :** don’t mention part\_col in "select"

else

FAILED: Error in semantic analysis: Line 1:18 Cannot insert into target table because column number/types are different ''f'': Table insclause-0 has 3 columns, but query has 4 columns.

**Bucketing**

Creates multiple sub-samples of original table

**Partitions** are created as direectories

**Buckets** will be created as files in a directory

We can specify the no. of buckets , thus we can restrict the no.of partitions.

set hive.enforce.bucketing = true;

create table bt(a int,b string, c int, d int, e int) clustered by (d) into 2 buckets row format delimited fields terminated by ',' stored as textfile;

**Transfer column from the non\_bucketing table to the newly created table**

insert into table\_name select a,b,c,d,e from dtbl;

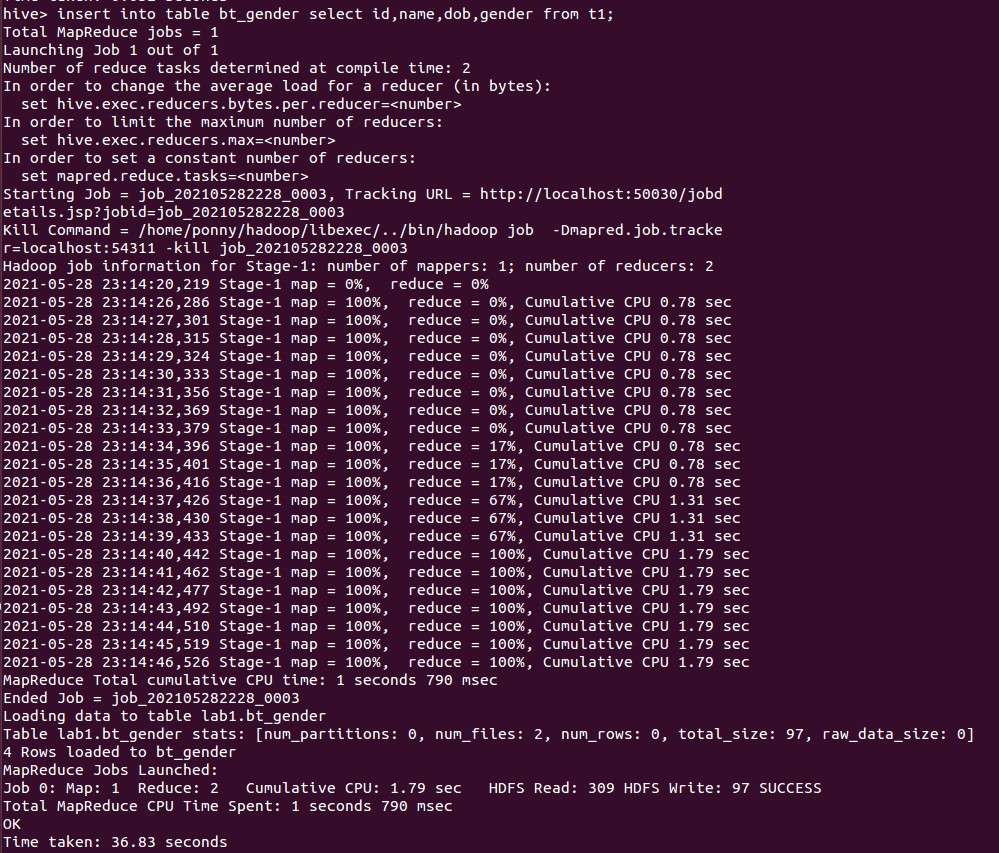
**Creating 2 Buckets on the column “gender”**

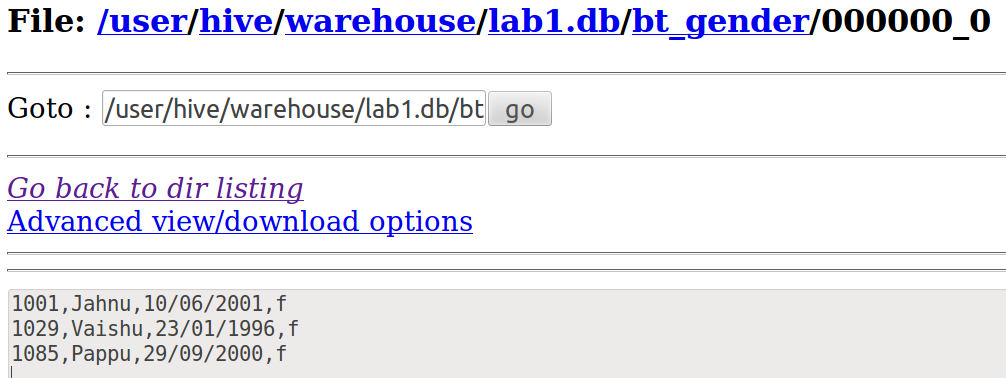
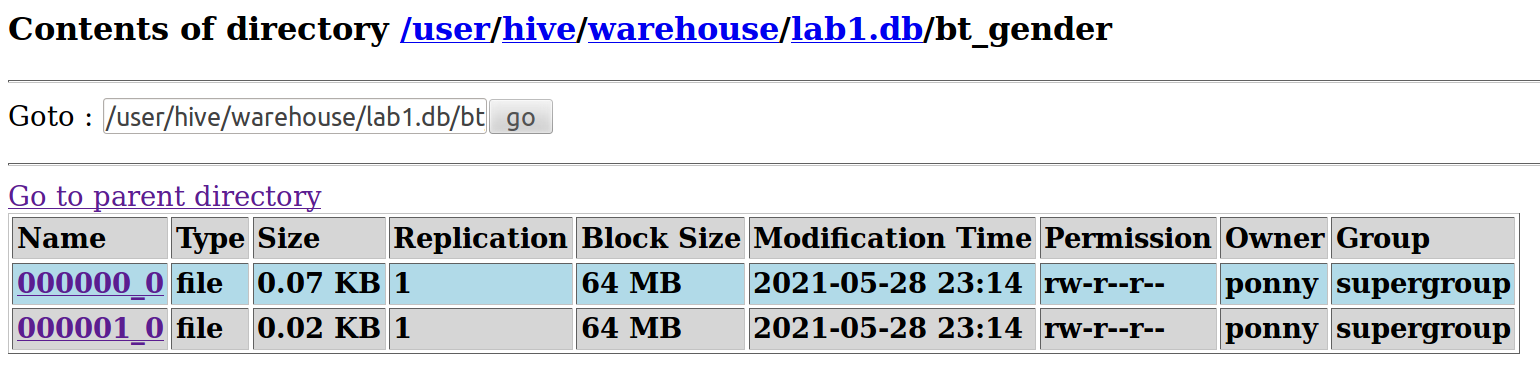
set hive.enforce.bucketing = true;

create table bt\_gender(id int,name string,dob string,gender string) clustered by (gender) into **2** buckets row format delimited fields terminated by ',' stored as textfile;

insert into table bt\_gender select id,name,dob,gender from t1;

insert overwrite table bt\_gender select id,name,dob,gender from t1;

**LOCATION:** /user/hive/warehouse/lab1.db/bt\_gender



### Group By Class

Table Name Stocks

Fields

Exchange String

Year String ( repeat same year 5 times) Give 2 Years

Open Price FLoat

Close Price

hadoop fs -copyFromLocal ~/Desktop/stocks /hive/

create table stock(year string, exchange string, op float ,cp float) row format delimited fields terminated by ',' stored as textfile;

load data inpath '/hive/stocks' into table stock;

select \* from stock;

OK

2000 MARUTI 164.9 164.3

2000 TITAN 146.0 155.7

2000 RELIANCE 237.5 251.7

2000 WIPRO 2724.0 2724.2

2000 BRITANNIA 705.0 756.9

2001 MARUTI 167.75 173.35

2001 TITAN 144.0 149.5

2001 RELIANCE 256.65 282.5

2001 WIPRO 2942.15 2990.1

2001 BRITANNIA 820.0 798.75

select year, count(\*), avg(op) from stock group by year; // 5 exchanges ⇒ count = 5

OK

2000 5 795.4799987792969

2001 5 866.1099792480469

select exchange,year,avg(op) from stock group by exchange,year having avg(op)>800;

BRITANNIA 2001 820.0   
WIPRO 2000 2724.0   
WIPRO 2001 2942.14990234375