Map reduce Use case – Titanic Data Analysis

There have been huge disasters in history of Map reduce, but the magnitude of the Titanic's disaster ranks as high as the depth it sank to. So much so that subsequent disasters have always been described as "titanic in proportion" – implying huge losses.

Anyone who has read about the Titanic, know that a perfect combination of natural events and human errors, led to the sinking of the Titanic on its fateful maiden journey from Southampton to New York on April 14, 1912.

There have been several questions put forward to understand the cause/s of the tragedy – foremost among them is: What made it sink and even more intriguing How can a 46,000 ton ship sink to the depth of 13,000 feet in a matter of 3 hours? This is a mind boggling question indeed!

There have been as many inquisitions as there have been questions raised and equally that many types of analysis methods applied to arrive at conclusions. But this blog is not about analyzing why or what made the Titanic sink – it is about analyzing the data that is present about the Titanic publicly. It actually uses Hadoop MapReduce to analyze and arrive at:

The average age of the people (both male and female) who died in the tragedy using Hadoop MapReduce.

How many persons survived – class wise.

This blog is about analyzing the data of Titanic. This total analysis is performed in Hadoop MapReduce.

This Titanic data is publically available and the titanic data set is described below under the heading Data Set Description.

Using that dataset we will perform some Analysis and will draw out some insights like finding the average age of male and females died in Titanic, Number of males and females died in each compartment.

DATA SET DESCRIPTION

Column 1: **PassengerId**
Column 2: **Survived** (survived=0 & died=1)
Column 3: **Pclass**
Column 4: **Name**
Column 5: **Sex**
Column 6: **Age**
Column 7: **SibSp**
Column 8: **Parch**
Column 9: **Ticket**
Column 10: **Fare**
Column 11: **Cabin**
Column 12: **Embarked**

Problem Statements:

Problem statement 1:

In this problem statement we will find the average age of males and females who died in the Titanic tragedy.

Source code:

Now from the mapper we will derive:

- the gender as **key**
- age as **values**

These values will be passed to the shuffle and sort phase and are further sent to the reducer phase where the aggregation of the values is performed.

```
1  public class Average_age {
2  public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
3  private Text gender = new Text();
4  private IntWritable age = new IntWritable();
5  public void map(LongWritable key, Text value, Context context ) throws IOException, InterruptedException {
6    String line = value.toString();
7    String str[]=line.split(",");
8         if(str.length>6){
9            gender.set(str[4]);
10        if((str[1].equals("0")) ){
11          if(str[5].matches("\\d+")){
12           int i=Integer.parseInt(str[5]);
13             age.set(i);
14    }
15   }
16  }
17    context.write(gender, age);
18   }
19 }
```

Mapper code:

The explanation for the code in each line of the above Mapper is:

- **line 1** we are taking a class by name Average_age,
- In **line 2** the Mapper default class is extended to have the arguments keyIn as LongWritable and ValueIn as Text and KeyOut as Text and ValueOut as IntWritable.
- In **line 3**, the variables declared are: a private Text variable gender which will store the gender of the person who died in the Titanic tragedy.
- In **line 4** the variables declared are: a private IntWritable variable age which will store the age of the MapReduce deals with Key and Value pairs. Here the key is set as gender and value as age.
- In **line 5** we are overriding the map method which will run one time for every line.
- In **line 6** we are storing the line in a string variable
- In **line 7** we are splitting the line by using comma “,” delimiter and storing the values in a String Array so that all the columns in a row are stored in the string array.
- In **line 8** we are taking a condition if we have the string array length greater than 6 which means if the line or row has at least 7 columns then the do the rest this helps in eliminating the ArrayIndexOutOfBoundsException in some cases if the data inside the data set is incorrect.

- In **line 9** we are storing the gender which is in 5th
- In **line 10** we are including a condition whether the person is alive or not if he is not alive then proceed.
- In **line 11** we are checking whether the data in that index is numeric data or not by using a regular expression which can be achieved by "matches function in java", if it is numeric data then it will proceed.
- In **line 12** we are converting that numeric data into an integer data by type casting.
- In **line 13** we are storing the age of the person in age
- In **line 17** we are writing the key and value into the context which will be the output of the map method.

Reducer Code:

```
1  public static class Reduce extends Reducer<Text,IntWritable, Text, IntWritable> {
2  public void reduce(Text key, Iterable<IntWritable> values, Context context)
3   throws IOException, InterruptedException {
4   int sum = 0;
5   int l=0;
6   for (IntWritable val : values) {
7   l+=1;
8   sum += val.get();
9   }
10  sum=sum/l;
11  context.write(key, new IntWritable(sum));
12    }
13  }
```

Explanation of the code in the Reducer code:

- **Line 1** extends the default Reducer class with arguments **KeyIn** as Text and **ValueIn** as IntWritable which are same as the outputs of the mapper class and KeyOut as Text and **ValueOut** as IntWritbale which will be final outputs of our MapReduce program.
- In **line 2** we overriding the Reduce method which will run each time for every key.
- In **line 4** declaring an integer **sum** which will store the sum of all the ages of people into it
- In **line 5** we are taking another variable as "l" which will be incremented every time as many values are there for that key.
- In **line 6** a foreach loop is taken which will run each time for the values inside the "Iterable values" which are coming from the shuffle and sort phase after the mapper phase.
- Value iterated.
- In **line 8** we are storing and calculating the sum of the values.
- Closing bracket.
- In **line 10** perform the average of the obtained sum and writes the respected key and the obtained sum as value to the context.

Configuration Code:

```
1  job.setMapOutputKeyClass(Text.class);
2  job.setMapOutputValueClass(IntWritable.class);
```

This two configuration classes are included in the main class whereas to clarify the Output key type of mapper and the output value type of the Mapper.

Way to to execute the Jar file to get the result of the first problem statement:

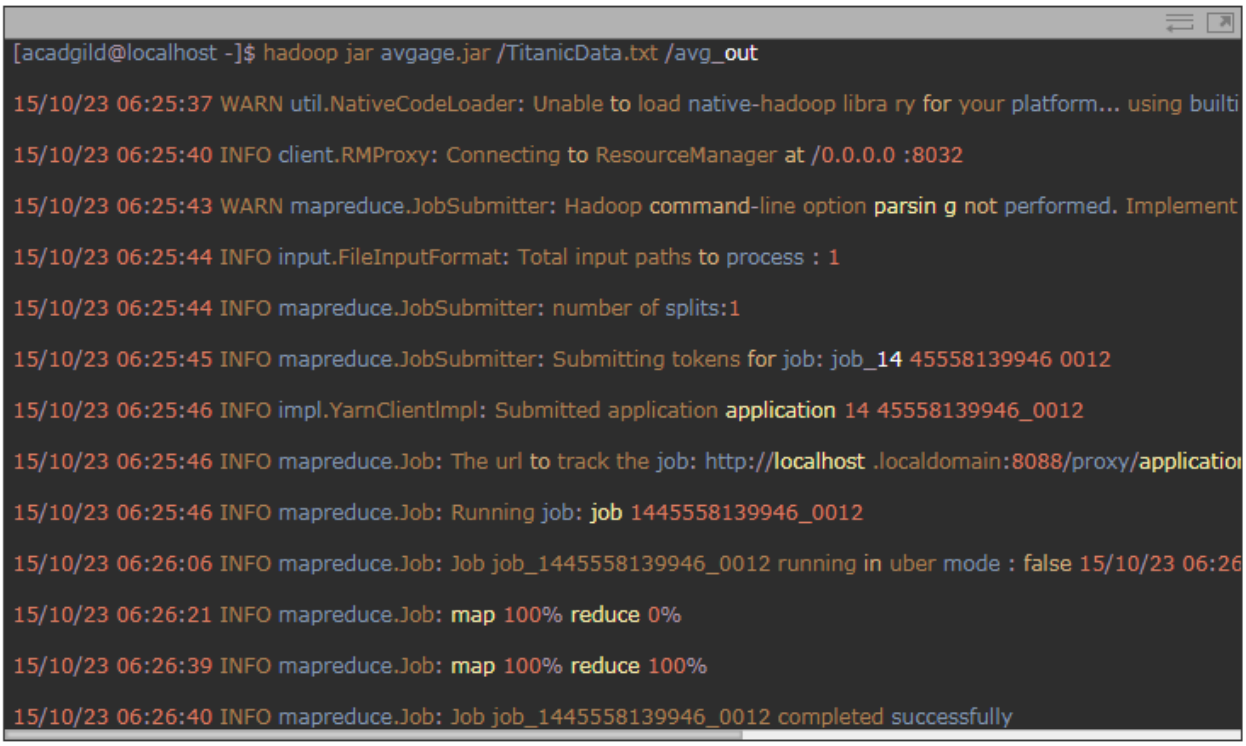
hadoop jar average.jar /TitanicData.txt /avg_out

Here 'hadoop' specifies we are running a Hadoop command and jar specifies which type of application we are running and average.jar is the jar file which we have created which consists the above source code and the path of the Input file name in our case it is TitanicData.txt and the output file where to store the output here we have given it as avg_out.

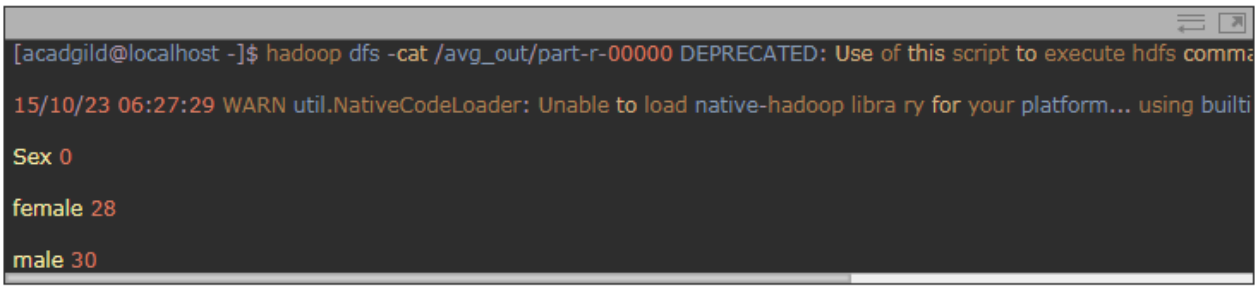
Way to view the output:

hadoop dfs –cat /avg_out/part-r-00000

Here 'hadoop' specifies that we are running a Hadoop command and dfs specifies that we are performing an operation related to Hadoop Distributed File System and '- cat' is used to view the contents of a file and avg_out/part-r-00000 is the file where output is stored. Part file is created by default by the TextInputFormat class of Hadoop.



Output:



Problem statement 2:

In this problem statement we will find the number of people died or survived in each class with their genders and ages.

Source code:

Now from the mapper we want to get the composite key which is the combination of Passenger class and gender and their age and final int value '1' as values which will be passed to the shuffle and sort phase and are further sent to the reducer phase where the aggregation of the values is performed.

Mapper code:

```
1  public class Female_survived {
2  public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {`
3  private Text people = new Text();
4  private final static IntWritable one = new IntWritable(1);
5  public void map(LongWritable key, Text value, Context context ) throws IOException, InterruptedException {
6    String line = value.toString();
7    String str[]=line.split(",");
8    if(str.length>6){
9    String survived=str[1]+" "+str[4]+" "+str[5];
10   people.set(survived);
11   context.write(people, one);
12   }
13  }
14 }
```

- In **line 1** we are taking a class by name survived
- In **line 2** extending the Mapper default class having the arguments **keyIn** as LongWritable and **ValueIn** as Text and **KeyOut** as Text and **ValueOut** as IntWritable.
- In **line 3** declaring a private Text variable 'people' which will store the Text value as
- In **line 4** declaring a private static final IntWritable variable **'one'** which will be one and can not be modified.MapReduce deals with Key and Value pairs.Here we can set the key as **pclass** and value as one.
- In **line 5** we are overriding the map method which will run one time for every line.
- In **line 6** we are storing the line in a string variable **'line'** .
- In **line 7** we are splitting the line by using comma "," delimiter and storing the values in a String Array so that all the columns in a row are stored in the string array.
- In **line 8** we are taking a condition if we have the string array length greater than 6 which means if the line or row has at least 7 columns then do the rest this helps in eliminating the **ArrayIndexOutOfBoundsException** in some cases if the data inside the data set is incorrect.
- In **line 9** we are making a composite by combining the necessary columns 2nd column which specifies whether the person is alive or not and the 5th column to specify the persons gender and 6th column to specify their age.
- This is the easiest way of forming a composite key in MapReduce and the functionality is very similar to the original way of making a composite key.
- In **line 10** we are storing the prepare composite key in people.
- In **line 11** we are writing the key and value into the context which will be the output of the map method.

Reducer code:

```
1  public static class Reduce extends Reducer<Text,IntWritable, Text, IntWritable> {
2  public void reduce(Text key, Iterable<IntWritable> values, Context context)
3     throws IOException, InterruptedException {
4     int sum = 0;
5     for (IntWritable val : values) {
6     sum += val.get();
7     }
8     context.write(key, new IntWritable(sum));
9   }
10 }
```

While coming to the Reducer code

- In **line 1** extends the default Reducer class with arguments **KeyIn** as Text and **ValueIn** as IntWritable which are same as the outputs of the mapper class and **KeyOut** as Text and **ValueOut** as IntWritbale which will be final outputs of our MapReduce program.
- In **line 2** we overriding the Reduce method which will run each time for every key.
- In **line 4** declaring an integer variable sum which will store the sum of all the values for each key.
- In **line 5** a foreach loop is taken which will run each time for the values inside the "Iterable values" which are coming from the shuffle and sort phase after the mapper phase.
- In **line 6** we are storing and calculating the sum of the values.
- In **line 8** writes the respected key and the obtained sum as value to the context.

In the above example use of composite key made the complicated program simple.

Conf code

```
1  job.setMapOutputKeyClass(Text.class);
2  job.setMapOutputValueClass(IntWritable.class);
```

This two configuration classes are included in the main class whereas to clarify the Output key type of mapper and the output value type of the Mapper.

Way to to execute the Jar file to get the result of the fourth problem statement:

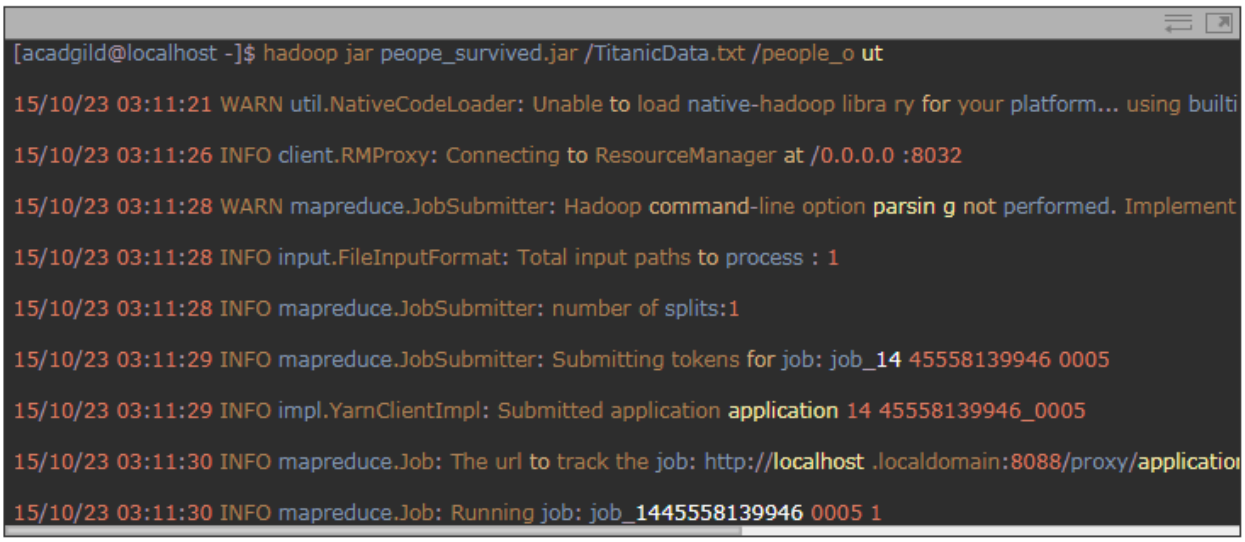
hadoop jar peope_survived.jar /TitanicData.txt /people_out

Here 'hadoop' specifies we are running a Hadoop command and jar specifies which type of application we are running and people_survived.jar is the jar file which we have created which consists the above source code and the path of the Input file name in our case it is TitanicData.txt and the output file where to store the output here we have given it as people_out.

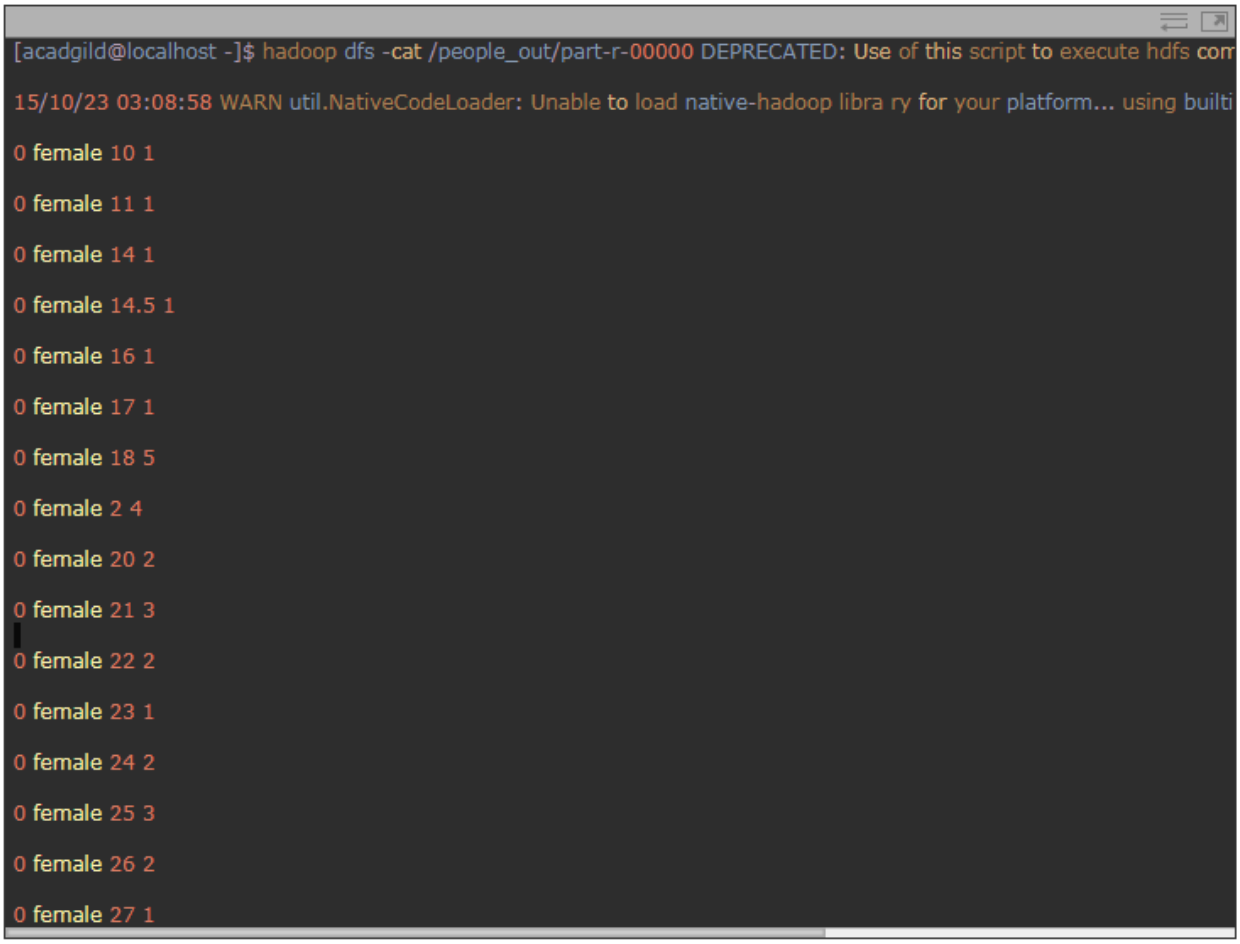
Way to view the output

hadoop dfs –cat /people_out/part-r-00000

Here 'hadoop' specifies that we are running a Hadoop command and dfs specifies that we are performing an operation related to Hadoop Distributed File Sysytem and '- cat' is used to view the contents of a file and people_out/part-r-00000 is the file where output is stored.Part file is created by default by the TextInputFormat class of Hadoop.



Output:


[acadgild@localhost ~]$ hadoop dfs -cat /people_out/part-r-00000 DEPRECATED: Use of this script to execute hdfs com
15/10/23 03:08:58 WARN util.NativeCodeLoader: Unable to load native-hadoop libra ry for your platform... using builti
0 female 10 1
0 female 11 1
0 female 14 1
0 female 14.5 1
0 female 16 1
0 female 17 1
0 female 18 5
0 female 2 4
0 female 20 2
0 female 21 3
0 female 22 2
0 female 23 1
0 female 24 2
0 female 25 3
0 female 26 2
0 female 27 1