

## EXPERIMENT - 5

### AIM:-

Write a Map Reduce Program that mines Weather Data

### DESCRIPTION:

Climate change has been seeking a lot of attention since long time. The antagonistic effect of this climate is being felt in every part of the earth. There are many examples for these, such as sea levels are rising, less rainfall, increase in humidity. The propose system overcomes the some issues that occurred by using other techniques. In this project we use the concept of Big data Hadoop. In the proposed architecture we are able to process offline data, which is stored in the National Climatic Data Centre (NCDC). Through this we are able to find out the maximum temperature and minimum temperature of year, and able to predict the future weather forecast. Finally, we plot the graph for the obtained MAX and MIN temperature for each moth of the particular year to visualize the temperature. Based on the previous year data weather data of coming year is predicted.

### ALGORITHM:-

#### MAPREDUCE PROGRAM

WordCount is a simple program which counts the number of occurrences of each word in a given text input data set. WordCount fits very well with the MapReduce programming model making it a great example to understand the Hadoop Map/Reduce programming style. Our implementation consists of three main parts:

1. Mapper
2. Reducer
3. Main program

#### Step-1. Write a Mapper

A Mapper overrides the `—map()` function from the Class

`"org.apache.hadoop.mapreduce.Mapper"` which provides `<key, value>` pairs as the input. A Mapper implementation may output `<key,value>` pairs using the provided Context .

Input value of the WordCount Map task will be a line of text from the input data file and the key would be the line number `<line_number, line_of_text>` . Map task outputs `<word, one>` for each word in the line of text.

#### Pseudo-code

```
void Map (key, value){
    for each max_temp x in value:
        output.collect(x, 1);
    }
void Map (key, value){
    for each min_temp x in value:
        output.collect(x, 1);
    }
```

## Step-2 Write a Reducer

A Reducer collects the intermediate <key,value> output from multiple map tasks and assemble a single result. Here, the WordCount program will sum up the occurrence of each word to pairs as <word, occurrence>.

### Pseudo-code

```
void Reduce (max_temp, <list of value>){
    for each x in <list of value>:
        sum+=x;
    final_output.collect(max_temp, sum);
}

void Reduce (min_temp, <list of value>){
    for each x in <list of value>:
        sum+=x;
    final_output.collect(min_temp, sum);
}
```

## 3. Write Driver

The Driver program configures and run the MapReduce job. We use the main program to perform basic configurations such as:

Job Name : name of this Job

Executable (Jar) Class: the main executable class. For here, WordCount.

Mapper Class: class which overrides the "map" function. For here, Map.

Reducer: class which override the "reduce" function. For here , Reduce.

Output Key: type of output key. For here, Text.

Output Value: type of output value. For here, IntWritable.

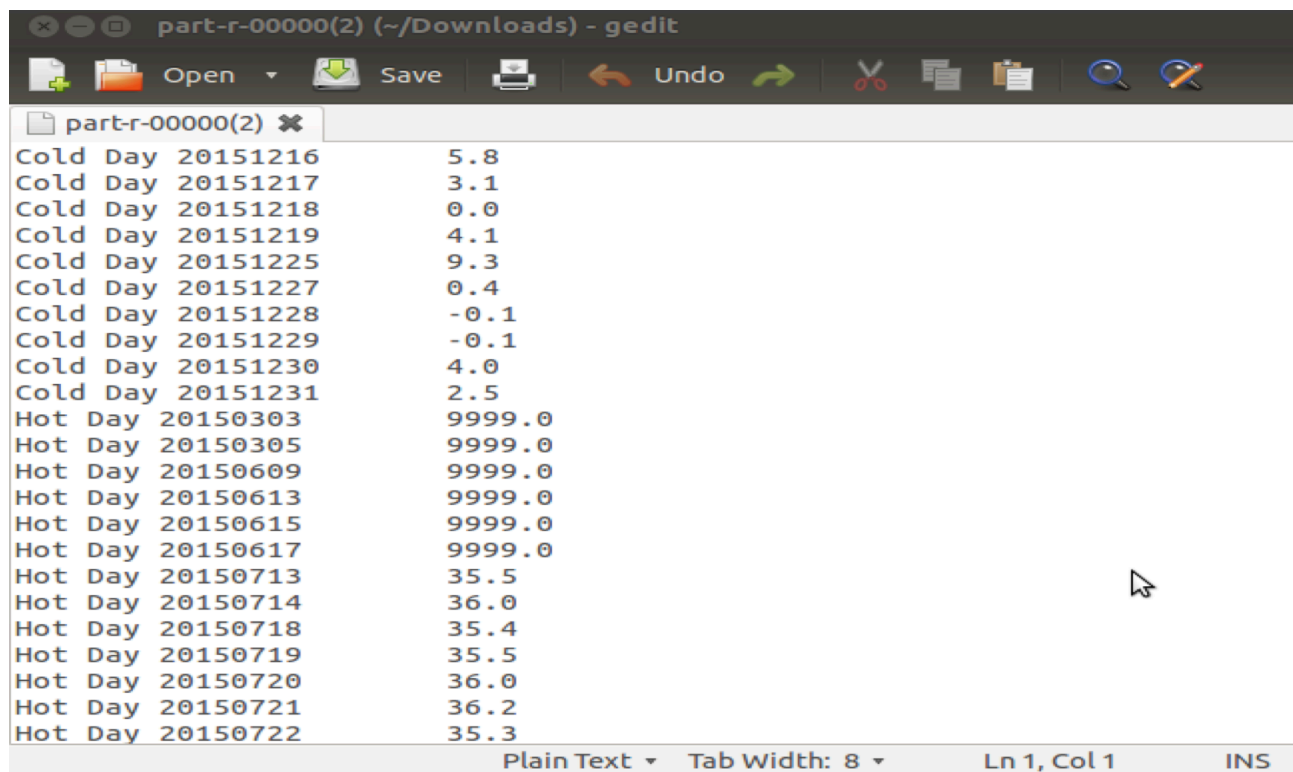
File Input Path

File Output Path

## INPUT:-

Set of Weather Data over the years

## OUTPUT:-



The screenshot shows a gedit text editor window titled "part-r-00000(2) (~/.Downloads) - gedit". The editor contains a list of temperature readings. The first 10 lines are for "Cold Day" with dates from 20151216 to 20151231 and values ranging from -0.1 to 5.8. The next 13 lines are for "Hot Day" with dates from 20150303 to 20150722 and values ranging from 35.3 to 9999.0. The status bar at the bottom indicates "Plain Text", "Tab Width: 8", "Ln 1, Col 1", and "INS".

Day Type	Date	Value
Cold Day	20151216	5.8
Cold Day	20151217	3.1
Cold Day	20151218	0.0
Cold Day	20151219	4.1
Cold Day	20151225	9.3
Cold Day	20151227	0.4
Cold Day	20151228	-0.1
Cold Day	20151229	-0.1
Cold Day	20151230	4.0
Cold Day	20151231	2.5
Hot Day	20150303	9999.0
Hot Day	20150305	9999.0
Hot Day	20150609	9999.0
Hot Day	20150613	9999.0
Hot Day	20150615	9999.0
Hot Day	20150617	9999.0
Hot Day	20150713	35.5
Hot Day	20150714	36.0
Hot Day	20150718	35.4
Hot Day	20150719	35.5
Hot Day	20150720	36.0
Hot Day	20150721	36.2
Hot Day	20150722	35.3

## EXPERIMENT - 6

### AIM:-

Write a Map Reduce Program that implements Matrix Multiplication.

### DESCRIPTION:

We can represent a matrix as a relation (table) in RDBMS where each cell in the matrix can be represented as a record (i,j,value). As an example let us consider the following matrix and its representation. It is important to understand that this relation is a very inefficient Relation if the matrix is dense. Let us say we have 5 Rows and 6 Columns, then we need to store only 30 values. But if you consider above relation we are storing 30 rowid, 30 col\_id and 30 values in other sense we are tripling the data. So a natural question arises why we need to store in this format? In practice most of the matrices are sparse matrices. In sparse matrices not all cells used to have any values, so we don't have to store those cells in DB. So this turns out to be very efficient in storing such matrices.

### MapReduceLogic:

Logic is to send the calculation part of each output cell of the result matrix to a reducer. So in matrix multiplication the first cell of output (0,0) has multiplication and summation of elements from row 0 of the matrix A and elements from col 0 of matrix B. To do the computation of value in the output cell (0,0) of resultant matrix in a separate reducer we need to use (0,0) as output key of map phase and value should have array of values from row 0 of matrix A and column 0 of matrix B. Hopefully this picture will explain the point. So in this algorithm output from map phase should be having a <key,value>, where key represents the output cell location (0,0), (0,1) etc.. and value will be list of all values required for reducer to do computation. Let us take an example for calculating value at output cell (0,0). Here we need to collect values from row 0 of matrix A and col 0 of matrix B in the map phase and pass (0,0) as key. So a single reducer can do the calculation.

### ALGORITHM:-

We assume that the input files for A and B are streams of (key,value) pairs in sparse matrix format, where each key is a pair of indices (i,j) and each value is the corresponding matrix element value. The output files for matrix  $C=A*B$  are in the same format.

We have the following input parameters:

- The path of the input file or directory for matrix A.

- The path of the input file or directory for matrix B.

- The path of the directory for the output files for matrix C. Strategy = 1, 2, 3

- R = the number of reducers.

- I = the number of rows in A and C.

$K$  = the number of columns in A and rows in B.  
 $J$  = the number of columns in B and C.  
 $IB$  = the number of rows per A block and C block.  
 $KB$  = the number of columns per A block and rows per B block.  
 $JB$  = the number of columns per B block and C block.

In the pseudo-code for the individual strategies below, we have intentionally avoided factoring common code for the purposes of clarity.

Note that in all the strategies the memory footprint of both the mappers and the reducers is flat at scale.

Note that the strategies all work reasonably well with both dense and sparse matrices. For sparse matrices we do not emit zero elements. That said, the simple pseudo-code for multiplying the individual blocks shown here is certainly not optimal for sparse matrices. As a learning exercise, our focus here is on mastering the MapReduce complexities, not on optimizing the sequential matrix multiplication algorithm for the individual blocks.

## Steps

1. setup ()
2. var  $NIB = (I-1)/IB+1$
3. var  $NKB = (K-1)/KB+1$
4. var  $NJB = (J-1)/JB+1$
5. map (key, value)
6. if from matrix A with key=(i,k) and value=a(i,k)
7. for  $0 \leq jb < NJB$
8. emit (i/IB, k/KB, jb, 0), (i mod IB, k mod KB, a(i,k))
9. if from matrix B with key=(k,j) and value=b(k,j)
10. for  $0 \leq ib < NIB$ 
  - emit (ib, k/KB, j/JB, 1), (k mod KB, j mod JB, b(k,j))

Intermediate keys (ib, kb, jb, m) sort in increasing order first by ib, then by kb, then by jb, then by m. Note that  $m = 0$  for A data and  $m = 1$  for B data.

**The partitioner maps intermediate key (ib, kb, jb, m) to a reducer r as follows:**

11.  $r = ((ib*JB + jb)*KB + kb) \bmod R$
12. These definitions for the sorting order and partitioner guarantee that each reducer  $R[ib,kb,jb]$  receives the data it needs for blocks  $A[ib,kb]$  and  $B[kb,jb]$ , with the data for the A block immediately preceding the data for the B block.
13. var A = new matrix of dimension  $IB \times KB$  14. var B = new matrix of dimension  $KB \times JB$
15. var sib = -1
16. var skb = -1

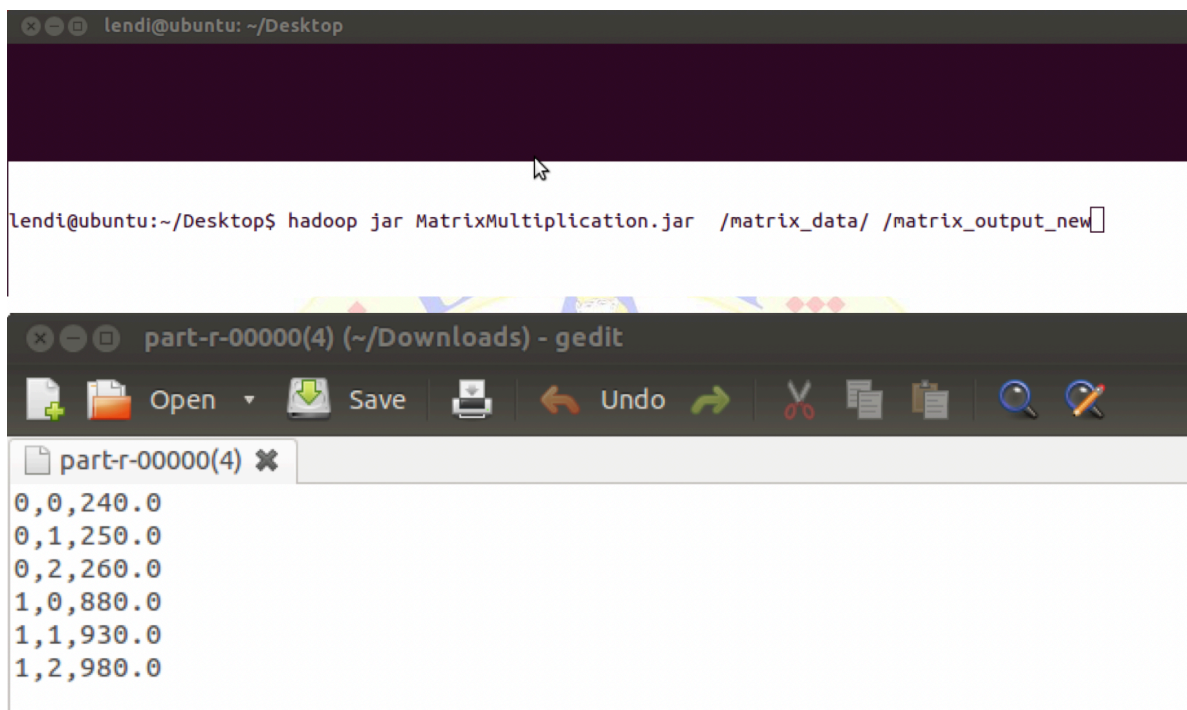
### Reduce (key, valueList)

```
17. if key is (ib, kb, jb, 0)
18. // Save the A block.
19. sib = ib
20. skb = kb
21. Zero matrix A
22. for each value = (i, k, v) in valueList A(i,k) = v
23. if key is (ib, kb, jb, 1)
24. if ib != sib or kb != skb return // A[ib,kb] must be zero!
25. // Build the B block.
26. Zero matrix B
27. for each value = (k, j, v) in valueList B(k,j) = v
28. // Multiply the blocks and emit the result.
29. ibase = ib*IB
30. jbase = jb*JB
31. for 0 <= i < row dimension of A
32. for 0 <= j < column dimension of B
33. sum = 0
34. for 0 <= k < column dimension of A = row dimension of B
    a. sum += A(i,k)*B(k,j)
35. if sum != 0 emit (ibase+i, jbase+j), sum
```

### INPUT:-

Set of Data sets over different Clusters are taken as Rows and Columns

### OUTPUT:-



The screenshot shows two windows. The top window is a terminal with the title 'lendi@ubuntu: ~/Desktop'. It displays the command: `lendi@ubuntu:~/Desktop$ hadoop jar MatrixMultiplication.jar /matrix_data/ /matrix_output_new`. The bottom window is a gedit editor with the title 'part-r-00000(4) (~/Downloads) - gedit'. It shows a list of output values: `0,0,240.0`, `0,1,250.0`, `0,2,260.0`, `1,0,880.0`, `1,1,930.0`, and `1,2,980.0`.