12 SEPT 2024 – POLISALES

MAPREDUCE & HADOOP

```java
public class MapperBigData extends Mapper<LongWritable, Text, Text, IntWritable>
{

    @Override
    protected void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {

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

        String category = fields[2];

        // Emit (category, +1)
        context.write(new Text(category), new IntWritable(1));
    }

}


/**
 * Basic MapReduce Project - Reducer
 */
class ReducerBigData extends Reducer<Text, // Input key type
        IntWritable, // Input value type
        Text, // Output key type
        IntWritable> { // Output value type

    @Override
    protected void reduce(
            Text key, // Input key type
            Iterable<IntWritable> values, // Input value type
            Context context) throws IOException, InterruptedException {

        int count = 0;
        for (IntWritable v : values) {
            count += v.get();
        }

        // Emit (category, number of products)
        context.write(key, new IntWritable(count));
    }
}
```

```
/**
 * Basic MapReduce Project - Mapper
 */
class MapperBigData2 extends Mapper<Text, // Input key type
        Text, // Input value type
        NullWritable, // Output key type
        Text> {// Output value type

    ArrayList<String> locaTop;
    int localMax;

    protected void setup(Context context) {
        locaTop = new ArrayList<>();
        localMax = -1;
    }

    protected void map(
            Text key, // Input key type
            Text value, // Input value type
            Context context) throws IOException, InterruptedException {

        String category = key.toString();
        int count = Integer.parseInt(value.toString());

        if (count > localMax) {
            locaTop.clear();
            locaTop.add(category);
            localMax = count;
        } else if (count == localMax) {
            locaTop.add(category);
        }

    }

    protected void cleanup(Context context) throws IOException,
InterruptedException {
        // Emit the local top category(ies)
        for (String s : locaTop) {
            context.write(NullWritable.get(), new Text(s + "_" + localMax));
        }
    }

}
```

```
public class ReducerBigData2 extends Reducer<NullWritable, Text, Text,
NullWritable>{

    @Override
    protected void reduce(NullWritable key, Iterable<Text> values, Context
context)
            throws IOException, InterruptedException {
        ArrayList<String> acc = new ArrayList<>();
        int maxCount = -1;

        String category, count;
        int value_count;

        for(Text v : values) {
            String tmp = v.toString();
            category = tmp.split("_")[0];
            count = tmp.split("_")[1];
            value_count = Integer.parseInt(count);

            if (value_count > maxCount) {
                acc.clear();
                acc.add(category);
                maxCount = value_count;
            }
            else if (value_count == maxCount)
                acc.add(category);
        }

        for(String s : acc)
            context.write(new Text(s), NullWritable.get());
    }

}
```

SPARK

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setAppName('Exam 12 Sept 2024')
sc = SparkContext(conf = conf)

products_path = 'data/Products.txt'
prices_path = 'data/Prices.txt'
sales_path = 'data/Sales.txt'

# Define the input rdds
# Products: product_id, name, category
productsRdd = sc.textFile(products_path)
# Prices: product_id, starting_date, ending_date, price
pricesRdd = sc.textFile(prices_path)
# Sales: product_id, date, number_of_products_sold
salesRdd = sc.textFile(sales_path)

# Part 1
# products that decreased their total sales in 2021 with respect to sales in 2019
# start with salesRDD and filter only year == 2019 or year == 2021
# then, compute the following RDD
# key = prodID
# value = #sales in 2019, #sales in 2021

def filterYears(line):
    fields = line.split(',')
    date = fields[1]

    return date.startswith('2019') or date.startswith('2021')

def mapProductSales(line):
    fields = line.split(',')
    pid = fields[0]  # product_id
    date = fields[1]
    numSales = int(fields[2])

    if date.startswith('2019'):
        return (pid, (numSales, 0))
    else:
        return (pid, (0, numSales))

salesPerYearRdd = salesRdd.filter(filterYears)\
                    .map(mapProductSales)\
```

```
                    .reduceByKey(lambda v1, v2: (v1[0]+v2[0], v1[1]+v2[1]))

# filter and keep only the entries associated with #sales19 > #sales 21
# retrieve the resulting productids
res1 = salesPerYearRdd.filter(lambda p: p[1][0]>p[1][1])\
                        .keys()

res1.saveAsTextFile("out1/")

################################################################################

# Part 2
# compute the most sold products for each year

# compute the following pairRDD
# key = productID, year
# value = #sales

def mapPidYearSales(line):
    fields = line.split(",")
    pid = fields[0]
    year = fields[1].split("/")[0]
    numSales = int(fields[2])

    return ((pid, year), numSales)


salesPerYear = salesRdd.map(mapPidYearSales)

# use a reduceByKey to sum all sales within that year and cache the RDD
totalSalesPerYear = salesPerYear\
                .reduceByKey(lambda v1, v2: v1+v2).cache()

# determine, for each year, the maximum value
# by first doing a map to pairs
# key = year
# value = count
# and use a reduceByKey to compute the max for each year
maxPerYear = totalSalesPerYear.map(lambda p: (p[0][1], p[1]))\
                                .reduceByKey(lambda v1, v2: max(v1, v2))

# map maxPerYear to
# key = (year, max count per year)
# value = None
yearMaxNone = maxPerYear.map(lambda p: (p, None))
```

```
# for each product, keep only the pairs (product, year) associated with the max
of the year
# first, we transform totalSalesPerYear in
# key = (year, count)
# value = pid
#
# and then use a join to keep only the pairs (product, year) associated with the
maximum valur of the year
maxPidPerYear = totalSalesPerYear.map(lambda p: ( (p[0][1], p[1]), p[0][0])).\
                    join(yearMaxNone)

# Extract (year, pid)
res2 = maxPidPerYear.map(lambda p: (p[0][0], p[1][0]))

res2.saveAsTextFile("out2/")
```