05 FEB 2024 – ASTROPOLI

MAPREDUCE & HADOOP

```java
/**
 * Exam20240205 - Mapper first job
 */

class MapperBigData1 extends Mapper<LongWritable, // Input key type
        Text, // Input value type
        Text, // Output key type
        IntWritable> { // Output value type

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

        // Extract the fields of the review
        // ObservatoryID,Name,Lat,Lon,Country,Continent,Amateur
        String[] fields = value.toString().split(",");
        String country = fields[4];
        String flag = fields[6];


        if(flag.equals("True"))
            context.write(new Text(country), new IntWritable(1));
    }
}



/**
 * Exam20240205 - Reducer first job
 */
class ReducerBigData1 extends Reducer<Text, // Input key type
        IntWritable, // Input value type
        Text, // Output key type
        IntWritable> { // Output value type

    private int maxCount;
    private String maxCountry;

    @Override
    protected void setup(Context context) throws IOException,
InterruptedException {
        maxCount = -1;
        maxCountry = null;
    }


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

        String country = key.toString();
        int count = 0;
        for(IntWritable v : values)
            count += v.get();

        if ((count > maxCount) || (count == maxCount &&
country.compareTo(maxCountry) < 0)) {
            maxCount = count;
            maxCountry = country;
        }
    }

    @Override
    protected void cleanup(Context context) throws IOException,
InterruptedException {
        // Emit the local maximum for each instance of the mapper class
        context.write(new Text(maxCountry), new IntWritable(maxCount));
    }

}


/**
 * Exam20240205 - Mapper second job
 */
class MapperBigData2 extends Mapper<
                    Text,  // Input key type
                    Text,         // Input value type
                    NullWritable,         // Output key type
                    Text> {       // Output value type

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

            String country = key.toString();
```

```java
            String count = value.toString();

            context.write(NullWritable.get(), new Text(country + "_" + count));

    }
}


/**
 *  Exam20240205 - Reducer second job
 */
class ReducerBigData2 extends Reducer<
                NullWritable,           // Input key type
                Text,    // Input value type
                Text,    // Output key type
                IntWritable> {  // Output value type

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

        int maxCount = -1;
        String maxCountry = null;

        for(Text v : values) {
            String data = v.toString();
            String[] fields = data.split("_");
            String country = fields[0];
            Integer count = Integer.parseInt(fields[1]);

            if(count > maxCount || (maxCount == count &&
country.compareTo(maxCountry) < 0)) {
                maxCount = count;
                maxCountry = country;
            }
        }

        // Emit the final maximum: Country and Number of amateur observatories in
the selected country
        context.write(new Text(maxCountry), new IntWritable(maxCount));

    }}
```

SPARK

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setAppName('Exam 05 feb 2024')
sc = SparkContext(conf = conf)

neObjectData = "exam_ex2_data/NEObject.txt"
observationData = "exam_ex2_data/Observations.txt"

outputPath1 = "outPart1/"
outputPath2 = "outPart2/"

# Create RDDs from files
neObjectRDD = sc.textFile(neObjectData).cache() # neo_id, dim, mat_strength,
already_fallen
observationsRDD = sc.textFile(observationData) # neo_id, obs_id, obs_date_time,
ecl_lat, ecl_lon, estimated_distance

######################
# PART 1
######################

# Find the average Dimension of all NEObjects.
# Map each line of NEObject.txt to dimension and then apply mean

def DimOne(line):
    fields = line.split(",")
    dimension = float(fields[1])

    return dimension



dimensionAvg = neObjectRDD.map(DimOne).mean()


# Analyze NEObject by filtering by dimension and alreadyFallen fields.
# The resulting RDD contains the Most Relevant NEOIDs. map to pairs:
# Key = NEOID
# Value = +1

def filterMostRel(line, dimensionAvg):
    # NEOID,Dimension,MaterialStrength,alreadyFallen
    fields = line.split(",")
    dimension = float(fields[1])
    isFallen = fields[3]

    if (dimension>dimensionAvg and isFallen=='False'):
        return True
    else:
        return False



mostRelevantNeoId = neObjectRDD.filter(lambda line: filterMostRel(line,
dimensionAvg))\
                            .map(lambda line: (line.split(",")[0], 1))\
                            .cache()

# Keep only Observations made in 2023 or later
# and map data into an RDD of pairs with
# key = NEOID
# value = ObservatoryID

def NeoObsID(line):
    fields = line.split(",")
    neoid = fields[0]
    observatoryId = fields[1]

    return (neoid, observatoryId)



# input:
NEOID,ObservatoryID,ObsDateTime,EclipticLat,EclipticLon,EstimatedDistance

observationsFiltered = observationsRDD\
            .filter(lambda line: int(line.split(",")[2].split("-")[0]) >= 2023)\
            .map(NeoObsID)\
            .cache()

# Join the two RDDs, obtaining all the observations in 2023 or later
# associated with the Most Relevant NEOIDs
# key = NEOID
# value = (ObservatoryID, +1)
# and map it to
# key = NEOID
# value = +1
# and use a reduceByKey to count the number of observations per
# Most RelevantNEOID
```

```
observationsPerNeoId = observationsFiltered.join(mostRelevantNeoId)\
                            .mapValues(lambda v: 1)\
                            .reduceByKey(lambda v1, v2: v1 + v2)

# sort it in descending order
res1 = observationsPerNeoId.sortBy(lambda p: p[1], ascending=False)

res1.saveAsTextFile(outputPath1)


######################
# PART 2
######################

# To compute the most relevant NEOIDs observed by less than 10 distinct
observatories,
# we can reuse the previously computed RDDs
# observationsFiltered: contains the observatories per each NEOID
# mostRelevantNeoId: contains the Most Relevant NEOIDs

# First, compute the list of NEOIDs with >= 10 distinct observatories starting
from 2023
# by performing a distinct over observationsFiltered RDD to compute the distinct
observatories per NEOID
# and by counting those
# key = NEOID
# value = count
# Then, we keep only the NEOIDs with count >= 10, which are the ones that should
be discarded

# distinct observations
distinctObservatoriesPerNeoId = observationsFiltered.distinct().cache()


# compute the list of NEOIDs with >= 10 distinct observatories starting from 2023
neoIdsWithManyObservatories = distinctObservatoriesPerNeoId\
                                .mapValues(lambda v: 1)\
                                .reduceByKey(lambda v1, v2: v1 + v2)\
                                .filter(lambda v: v[1] >= 10)


# From the complete list of Most Relevant NEOIDs, remove NEOIDs in the previously
computed RDD.
# We obtain those with less than 10 distinct observatories.
# key = NEOID
# value = +1
neoIdsOfInterest =
mostRelevantNeoId.subtractByKey(neoIdsWithManyObservatories).cache()

# Join the list of selected Most Relevant NEOIDs with
distinctObservatoriesPerNeoId
# to retrieve the ids of the observatories that observed the selected Most
Relevant NEOIDs
# starting from 2023.
# Keep only NEOID and OrbservatoryID
neoIdsOfInterestObservatories =
distinctObservatoriesPerNeoId.join(neoIdsOfInterest)\
                                .mapValues(lambda v: v[0])

# Identify the subset of Most Relevant NEOIDs of interest never observed starting
from 2023.
# They have been discarded by the previous join
# For those Most Relevant NEOIDs, return pairs (NEOID,"None")
neoIdsOfInterestnonObserved = neoIdsOfInterest\
                                .subtractByKey(distinctObservatoriesPerNeoId)\
                                .mapValues(lambda v: "None")

# The final result is the union of neoIdsOfInterestObservatories and
neoIdsOfInterestnonObserved
res2 = neoIdsOfInterestObservatories.union(neoIdsOfInterestnonObserved)

res2.saveAsTextFile(outputPath2)
```