21 JUN 2023 – POLIMEETINGS

MAPREDUCE & HADOOP

```
class Mapper1BigData extends Mapper<
                    LongWritable,
                    Text,
                    Text,
                    IntWritable> {

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

        String[] fields = value.toString().split(",");
        String mid = fields[0];
        String accepted = fields[2];

        if(accepted.equals("Yes")) {
            context.write(new Text(mid), new IntWritable(1));
        }
    }
}


class Reducer1BigData extends Reducer<Text, IntWritable, Text, IntWritable> {

    private int max;
    private String midMax;

    @Override
    protected void setup(Context context) throws IOException,
InterruptedException {
        this.max = -1;
        this.midMax = "";
    }

    @Override
    protected void reduce(
            Text key,
            Iterable<IntWritable> values,
            Context context) throws IOException, InterruptedException {
        int sum = 0;
        String mid = key.toString();
        for (IntWritable val : values) {
            sum += val.get();
        }
        if (sum > this.max || (sum == this.max && mid.compareTo(this.midMax) >
0)) {
            this.max = sum;
            this.midMax = mid;
        }
    }

    @Override
    protected void cleanup(Context context) throws IOException,
InterruptedException {
        if (this.max != -1) {
            context.write(new Text(this.midMax), new IntWritable(this.max));
        }
    }
}


class Mapper2BigData extends Mapper<
        Text,
        Text,
        NullWritable,
        Text> {

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

        int v = Integer.parseInt(value.toString());
        context.write(NullWritable.get(), new Text(key+":"+v));
    }
}


class Reducer2BigData extends Reducer<NullWritable, Text, Text, IntWritable> {

    @Override
    protected void reduce(
            NullWritable key,
            Iterable<Text> values,
            Context context) throws IOException, InterruptedException {

        int max;
        String midMax;
        max = -1;
        midMax = "";

        for (Text value : values) {
            String[] fields = value.toString().split(":");
            String mid = fields[0];
            int expectedParticipants = Integer.parseInt(fields[1]);

            if (expectedParticipants > max || (expectedParticipants == max &&
mid.compareTo(midMax) > 0)) {
                max = expectedParticipants;
                midMax = mid;
            }

        }

        context.write(new Text(midMax), new IntWritable(max));
    }
}
```

SPARK

```
import pyspark

from pyspark import SparkContext
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
sc = spark.sparkContext

# mid, title, startTime, duration, orgUID
meetingsPath = "data/meetings.txt"
# uid, name, surname, dateOfBirth, pricingPlan
usersPath = "data/users.txt"
# mid, uid, accepted
invitationsPath = "data/invitations.txt"

# retrieve RDDs from input paths
meetingsRDD = sc.textFile(meetingsPath)
usersRDD = sc.textFile(usersPath)
invitationsRDD = sc.textFile(invitationsPath)

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

# Part 1 -- statistics  on the duration of the meetings considering only the
meetings organized
# by users with business plan
def businessFilter(line):
    # uid, name, surname, dateOfBirth, pricingPlan
    fields = line.split(',')
    pricingPlan = fields[4]
    return pricingPlan == "Business"

businessUsersRDD = usersRDD.filter(businessFilter)\
      .map(lambda line: (line.split(',')[0], None))  # business users (UID, None)

# let's retrieve from meetingsRDD pairs (orgUid, duration)
# then join with businessUsersRDD to keep only business users pairs (orgUID,
(mid, duration))
def orgUIDDurationMap(line):
    # mid, title, startTime, duration, orgUID
    fields = line.split(',')
    orgUID = fields[4]
    mid = fields[0]
    duration = int(fields[3])
    return (orgUID, (mid, duration))

orgIdDurationRDD = meetingsRDD\
    .map(orgUIDDurationMap)\
    .join(businessUsersRDD)\
    .map(lambda p: (p[0], p[1][0])).cache()  # (orgUID, (mid, duration)) only for
business users

# let's find the avg:
# retrieve pairs (orgUID, (duration, 1))
# compute (orgUID, (totDuration, nMeetings))
# map to (orgUID, totDuration/nMeetings)
avgResRDD = orgIdDurationRDD\
    .map(lambda p: (p[0], (p[1][1], 1)))\
    .reduceByKey(lambda v1, v2: (v1[0] + v2[0], v1[1] + v2[1]))\
    .mapValues(lambda v: v[0] / v[1])  # (orgUID, avgMeetingDuration)

# let's find min and max:
# retrieve pairs (orgUID, (duration, duration))
# reduceByKey -> for each orgUID -> (orgUID, (minDuration, maxDuration))
minMaxResRDD = orgIdDurationRDD\
    .map(lambda p: (p[0], (p[1][1], p[1][1])))\
    .reduceByKey(lambda v1, v2: (min(v1[0], v2[0]), max(v1[1], v2[1])))

# join and obtain final result:
def res1Map(p):
    # p = (orgUID, (avgMeetingDuration, (minDuration, maxDuration)))
    orgUID = p[0]
    avgDuration = str(p[1][0])
    minDuration = str(p[1][1][0])
    maxDuration = str(p[1][1][1])
    return (orgUID, avgDuration + ", " + minDuration + ", " + maxDuration)

res1RDD = avgResRDD\
    .join(minMaxResRDD)\
    .map(res1Map)

# store result in hdfs:
res1RDD.saveAsTextFile(outputPath1)


# Part 2

# from invitationsRDD i retrieve these pairs: ((uid, mid), 1)
# and then proceed to compute ((uid, mid), nInvitations)
def uidMidOnesMap(line):
    # mid, uid, accepted
    fields = line.split(',')
    uid = fields[1]
    mid = fields[0]
    return ((uid, mid), 1)

invitationsPerMeetingForUIDRDD = invitationsRDD\
    .map(uidMidOnesMap)\
    .reduceByKey(lambda v1, v2: v1 + v2)  # ((uid, mid), nInvitations) for all
kind of pricingPlan users

# from orgIdDurationRDD I retrieve pairs ((orgUID, mid), 0)
def orgIdMidMap(p):
    # p = (orgUID, (mid, duration)) [only for business users]
    orgUid = p[0]
    mid = p[1][0]
    return ((orgUid, mid), 0)

businessUsersMidsRDD = orgIdDurationRDD\
    .map(orgIdMidMap).cache()  # ((orgUID, mid), 0) for all business users mids
(also those with no invitations)

### QUA C'E' UN ERRORE PERCHE' FACCIO UN join TRA (uid,mid) DOVE NEL SECONDO CASO
CI SONO
### GLI UID DEGLI INVITATI E NON DEGLI ORGANIZZATORI :(

# I compute only business organizers ((uid, mid), invitation)
# by joining invitationsPerMeetingForUIDRDD with businessUsersMidsRDD
# res join: ((uid, mid), (nInvitations, 0))
# keep only ((uid, mid), nInvitations)
businessUsersInvitationsPerMidRDD = invitationsPerMeetingForUIDRDD\
    .join(businessUsersMidsRDD)\
    .map(lambda p: (p[0], p[1][0]))\
    .cache() # i'm using this in the join and in the union

# I calculate the business users who organized meetings without any invitations:
# in order to do so --> businessUsersMidsRDD -> subtractByKey busines users that
actually have invitations
zeroInvitationsRDD = businessUsersMidsRDD\
    .subtractByKey(businessUsersInvitationsPerMidRDD)

# I do the union between the ones with zero and positive number of invitations
# from these I compute (uid, (smallMeeting, mediumMeedings, largeMeetings))
```

```python
# I then format to string values
def uidCountersMap(p):
    # p = ((uid, mid), nInvitations)
    # I want (uid, (0/1, 0/1, 0/1))
    uid = p[0][0]
    nInvitations = p[1]
    if nInvitations > 20:
        return (uid, (0, 0, 1))
    elif nInvitations <= 20 and nInvitations >= 5:
        return (uid, (0, 1, 0))
    else:
        return (uid, (1, 0, 0))

res2RDD = businessUsersInvitationsPerMidRDD\
    .union(zeroInvitationsRDD)\
    .map(uidCountersMap)\
    .reduceByKey(lambda v1, v2: (v1[0] +v2[0], v1[1] +v2[1], v1[2] +v2[2]))\
    .mapValues(lambda v: f"{v[0], v[1], v[2]}")

# save results in hdfs
res2RDD.saveAsTextFile(outputPath2)
```