O2 FEB 2023 – HPC House Power Consumption

HADOOP & MAPREDUCE

```
public class CityCount implements org.apache.hadoop.io.Writable {
    public String city;
    public int count;


    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public int getCount() {
        return count;
    }
    public void setCount(int count) {
        this.count = count;
    }

    public String toString() {
        return city + "," + count;
    }
    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(city);
        out.writeInt(count);
    }

    @Override
    public void readFields(DataInput in) throws IOException {
        city = in.readUTF();
        count = in.readInt();
    }
}


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

    protected void map(
```

```java
                LongWritable key,
                Text value,
                Context context) throws IOException, InterruptedException {

                String[] fields = value.toString().split(",");
                String city = fields[1];
                Double sqm = Double.parseDouble(fields[3]);

                if(sqm < 60) {
                    context.write(new Text(city), new IntWritable(1));
                }
        }
}


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

    private String maxCity;
    private int maxCount;

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

    @Override
    protected void reduce(
            Text key,
            Iterable<IntWritable> values,
            Context context) throws IOException, InterruptedException {
        int val = -1;
        int sum = 0;
        String city = key.toString();
        for (IntWritable v : values) {
            val = v.get();
            sum += val;
        }

        if (maxCount == -1 || sum > maxCount || (sum == maxCount &&
city.compareTo(maxCity) < 0)) {
            maxCount = sum;
            maxCity = city;
        }
    }

    @Override
    protected void cleanup(Context context) throws IOException,
InterruptedException {
        if (this.maxCity != null) {
            context.write(new Text(maxCity), new IntWritable(maxCount));
        }
    }
}

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

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

        CityCount localCityCount = new CityCount();
        localCityCount.city=key.toString();
        localCityCount.count=Integer.parseInt(value.toString());


        context.write(NullWritable.get(), localCityCount);
    }
}

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



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

```
            String maxCity=null;
            int maxCount=-1;


            for(CityCount v : values) {
                if(maxCount == -1 || v.count > maxCount ||
                (v.count == maxCount && v.city.compareTo(maxCity) < 0)) {
                    maxCount = v.count;
                    maxCity = v.city;
                }
            }

            context.write(new Text(maxCity), new IntWritable(maxCount));
    }

}
```

SPARK

```
import pyspark

from pyspark import SparkContext
from pyspark.sql import SparkSession

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

housePath = "data/Houses.txt"
consumptionPath = "data/DailyPowerConsumption.txt"

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

houseRDD = sc.textFile(housePath)
consumptionRDD = sc.textFile(consumptionPath)

# Part 1
# filter only the readings associated with year 2022
consumption2022 = consumptionRDD.filter(lambda s:
s.split(",")[1].startswith("2022"))

# compute the total amount of energy consumed in year 2022 for each house
# key = houseID
# value = kWh consumed in year 2022

def mapCons(s):
    fields = s.split(",")
    hid = fields[0]
    consumption = float(fields[2])
    return (hid, consumption)


totalCons2022 = consumption2022.map(mapCons)\
                                .reduceByKey(lambda v1, v2: v1 + v2)

# compute the avg power consumption per day
# key = houseID
# value = avg kWh consumed per day in year 2022
# and filter only those with high avg consumption
highAvgDailyCons = totalCons2022\
                    .mapValues(lambda v: v / 365)\
                    .filter(lambda i: i[1] > 30)
```

```
# compute the pairRDD house -> country
# key = houseID
# value = country

def mapHouseCountry(s):
    fields = s.split(",")
    hid = fields[0]
    country = fields[2]

    return (hid, country)


houseCountry = houseRDD.map(mapHouseCountry)

# keep an RDD containing countries with at least one house with high
# avg power consumption
countriesWithHighAvgPwrConsHouses = houseCountry.join(highAvgDailyCons)\
                    .map(lambda it: it[1][0]) # Country

# compute an RDD with all the countries
# and subtract the countries with at least one house with high avg power
consumption
res1 = houseCountry.map(lambda v: v[1])\
                    .distinct()\
                    .subtract(countriesWithHighAvgPwrConsHouses)

res1.saveAsTextFile(outputPath1)

# Part 2
# keep only the houses for which the total power consumption over 2021 is > 10000
kWh

def mapHidCons(s):
    fields = s.split(",")
    hid = fields[0]
    consumption = float(fields[2])

    return (hid, consumption)

highTotalPowerCons2021 = consumptionRDD\
            .filter(lambda s: s.split(",")[1].startswith("2021"))\
            .map(mapHidCons)\
            .reduceByKey(lambda v1, v2: v1 + v2)\
            .filter(lambda v: v[1]>10000)
```

```
# compute an RDD with
# key = houseID
# value = (country, city)

def mapHidCountryCity(s):
    fields = s.split(",")
    hid = fields[0]
    city = fields[1]
    country = fields[2]

    return (hid, (country, city))


citiesRDD = houseRDD.map(mapHidCountryCity)

# join the two RDDs and count for each city the number of houses with high annual
power consumption
# and filter only those cities with value > 500
# key = (country, city)
# value = #houses with high power consumption
highPwrConsHousesPerCity = highTotalPowerCons2021.join(citiesRDD)\
                    .map(lambda p: (p[1][1], 1))\
                    .reduceByKey(lambda v1, v2: v1 + v2)\
                    .filter(lambda p: p[1]>500)

# count for each country the number of cities with at least 500 houses with high
annual power consumption

# Map each input pair to a new pair
# key = country
# value = +1 (one more city for this country with at least 500 houses with high
annual power consumption)
countryOneMorehighPwrConsCity = highPwrConsHousesPerCity\
                                .map(lambda p: (p[0][0], 1))

# Map each country to pair
# key = country
# value = 0
# This is used to keep also countries without cities with at least
# 500 houses with high annual power consumption
countriesZero = houseCountry.map(lambda p: (p[1], 0))

# count for each country the number of cities with at least 500 houses with high
annual power consumption
```

```
# Union countryOneMorehighPwrConsCity with countriesZero and apply reduceByKey
# Output
# key = country
# value = number of cities with at least 500 houses with high power consumption
highPwrConsCitiesPerCountry = countryOneMorehighPwrConsCity.union(countriesZero)\
                                    .reduceByKey(lambda v1, v2: v1 + v2)

highPwrConsCitiesPerCountry.saveAsTextFile(outputPath2)
```