Infrastructure Big Data

Examen

Durée: 1h, documents autorisés

Understanding questions (6 points)

Instructions: In the MCQ, several answers may be possible on some questions.

Question 1

We consider the execution of the wordcount application on a Spark cluster composed of several servers. The execution should run faster than a centralized sequential version of wordcount :

- [ ] for all input files
- [ ] for input files with a minimal size
- [x] for input files up to a maximal size

Question 2

In a Spark/HDFS cluster on top of Linux, what are the strong requirements (meanning it cannot run without it)?

- [ ] ssh server installed on each node
- [ ] same type of processor on each node
- [ ] same number of processor on each node
- [x] Java installed on each node

Question 3

In a distributed Spark/HDFS deployment :

- [x] the Master daemon is running on the master node
- [ ] the NameNode daemon is running on all slave nodes
- [x] the NameNode daemon is running on the master node
- [x] the Worker daemon is running on all slave nodes
- [ ] the DataNode daemon is running on the master node

Question 4

In Spark, the `reduceByKey()` method:

- [x] results into a dataset where each key is unique
- [x] results into a dataset where the number of different keys is reduced
- [ ] gathers all the pairs on the master node
- [ ] distributes all the pairs over the slave nodes

Question 5

In Spark, the `reduceByKey (func)` method :

- [ ] can be applied to any JavaRDD
- [x] can only be applied to a JavaPairRDD
- [ ] func is a function which aggregates 2 pairs into one pair
- [x] func is a function which aggregates 2 values into one value

Question 6

In Spark, the `mapValues (func)` method:

- [ ] can be applied to any JavaRDD
- [x] can only be applied to a JavaPairRDD
- [x] func is a function which transforms a value into another value
- [ ] func is a function which aggregates 2 values into one value

Problem (14 points)

We consider a large file including meteorological data. Each line in the file provides an observation:

```
day month year temperature city
```

For instance : `12 07 1995 30 Paris` is an observation that on the 12th of july 1995, the temperaiure was 30 degrees in Paris.

You can extract the fields from a line L with `L.splitO[0]`, `L.splitO[1]`, etc.
Your algorithms start with the following `RDD` (initialized with a file available in HDFS) :

```
JavaRDD<String> data = sc.textFile(inputFile);
```

Question 1

We want to compute for each city the average temperature. What's wrong in the following algorithm ? justify (1 point)

```
JavaPairRDD<String, Integer> average = data.mapToPair(
    s -> new Tuple2<String, Integer> (s.split()[4],
                                      Integer.parseInt(s.split()[3])))
    .reduceByKey((t1,t2)->(t1 + t2)/2);
```

Answer :

The issue is with the `reduceByKey()` transformation. The current implementation uses an averaging function that computes the average of two temperatures, but it does not take into account the number of observations for each city. Therefore, the resulting average temperature may be incorrect.

$$avg \neq \frac{\frac{\frac{temp_1 + temp_2}{2} + temp_3}{2} + temp_4}{2} \ldots \quad (1)$$

Question 2

Propose a correct solution to the previous question (1 point)

Answer :

```
JavaPairRDD<String, Tuple2<Integer, Integer>> cityTemp = data.mapToPair(
    s -> new Tuple2<String, Tuple2<Integer, Integer>>(  // <city, <temp, count>>
        s.split()[4], new Tuple2<Integer, Integer>(     // <city:string, <temp:int, 1:int>>
            Integer.parseInt(s.split()[3]), 1)))
    .reduceByKey((t1, t2) -> new Tuple2<Integer, Integer>(
        t1._1() + t2._1(), t1._2() + t2._2()));         // <city, <sumTemp, sumCount>>
JavaPairRDD<String, Float> average = cityTemp.mapValues(
    v -> (float) v._1() / v._2());                      // <city, sumTemp/sumCount>

```

Explain:

- In this updated algorithm, we first create a tuple of temperature and count of observations for each city, and then use the `reduceByKey()` transformation to sum the temperatures and counts for each city. The resulting intermediate RDD contains tuples of the form:

```
<city, <sum_of_temperatures, count_of_observations>>
```

- We then use the `mapValues()` transformation to compute the average temperature for each city by dividing the sum of temperatures by the count of observations. The resulting RDD contains tuples of the form:

```
<city, average_temperature>
```

- Note that we also changed the value type of the intermediate RDD to `Tuple2<Integer, Integer>` to store the sum of temperatures and the count of observations. Finally, we changed the value type of the resulting RDD to `Double` to store the average temperature.

Question 3

We want to compute for each year the number of cities where temperature is measured (2 points).

Result should be a `JavaPairRDD<year, count>`

Answer :

```
JavaPairRDD<String, String> count = data.mapToPair(
    // 1. Map each line to a tuple of <year, city>
    s -> new Tuple2<Integer, String>(Integer.parseInt(s.split(" ")[2]), s.split(" ")[4]))
    // 2. Distinct the tuples to keep only unique <year, city> pairs
    .distinct()
    // 3. Map each tuple to <year, 1> to get a count of cities per year.
    .mapToPair(t -> new Tuple2<Integer, Integer>(t._1, 1))
    // 4. Reduce by key to get the total count of cities per year.
    .reduceByKey((c1, c2) -> c1 + c2);
```

Question 4

We want to extract the list of cites where 2022 was the warmest year (considering the maximal temperature) (3 points).

Result should be a `JavaRDD<city>`

Answer :

```
JavaPairRDD<String, Integer> cityTempIn2022 = data
    // 1. Filter the RDD to keep only the observations from 2022.
    .filter(s -> s.split(" ")[2].equals("2022"))
    // 2. Map each line to a tuple of <city, temperature>   .
    .mapToPair(s -> new Tuple2<String, Integer>(
        s.split(" ")[4], Integer.parseInt(s.split(" ")[3])));

// 3. Reduce by key to keep only the maximum temperature for each city
JavaPairRDD<String, Integer> maxTemp = cityTempIn2022.reduceByKey((a, b) -> Math.max(a, b));

JavaRDD<String> cities = maxTemp
    // 4. Keep only the cities where the maximum temperature is reached in 2022.
    .filter(t -> t._2 == cityTempIn2022
            .filter(s -> s._1.equals(t._1))
            .map(s -> s._2)
            .max())
    // 5. Map each tuple to the city name to get the final RDD of cities, JavaRDD<city>.
    .map(t -> t._1);
```

```
class GetMaxTempInYear implements PairFunction<T, T, T>{
    ...
}

JavaPairRDD<String, Tuple2<String, Integer>> rdd_cityYearTemp = data
    .mapToPair(s -> {new Tuple2<String, Tuple2<String, Integer>>(
        s.split(" ")[4], new Tuple2<String, Integer>(
            s.split(" ")[2], Integer.parseInt(s.split(" ")[3])))})  // <city, <year, temp>>
    .sortBy(data -> data._2._2, False, 1)   // sorted by temp, descending, global partition
    // .reduceByKey(new GetMaxTempInYear())
    .reduceByKey(data1, data2 -> {      // <city, <warmest_year, warmest_temp>>
        if (data1._2 > data2._2) {
```

```
            return data1
        }
        else{
            return data2
        }
    })
    .filter(data -> data._2._1.equals("2022"))  // <city, <2022, warmest_temp>>
    .map(data -> data._1)   // <city>
```