Networked Software for Distributed Systems – Project 4 Documentation

Group 7 – De Novellis Torralba Venkataraman

Introduction

The goal of the project was to create a program to analyze a dataset with the number of new daily COVID infections per country. Three queries, to be computed starting from the raw data given, were requested:

1. **Seven days moving average of new reported cases, for each county and for each day.**
   The moving average is defined as the mean of the previous k data points. In our case, we have to restrict it to each single country, so the rows have to be partitioned into one partition per country and for each day the value is computed using that day's cases and the 6 previous days.

2. **Percentage increase (with respect to the day before) of the seven days moving average, for each country and for each day.**
   The percentage increase of the seven-day moving average is computed starting from the result of query 1. For each day, the current and the previous moving average are used to calculate their difference, and this value is divided by the previous moving average. As in the first case, the rows are partitioned by country.

3. **Top 10 countries with the highest percentage increase of the seven days moving average, for each day.**
   The resulting dataset of query 2 is partitioned by days and sorted by decreasing percentage increase. This way, for each day the top countries with respect to the percentage increase of moving average are shown.

Dataset

The dataset is given in the form of a csv file, composed of rows reporting the date, number of cases, nation name and other columns not relevant for this application.

Technologies and implementation

The program has been implemented using Apache Spark. In particular, this technology offers the spark SQL library, well suited for structured data, as in our case. Spark SQL gives the possibility to infer a table-like data structure from the input file and then has an high level API that permits to compute different operations on the data and to create new tables manipulating the original one. We will now go through the code of the different phases of the data elaboration and explain them.

1. **Reading the data**

   First of all an array of fields is created. Each field is an object that describes the name of the column, the type of data contained and if the value can be nullable. As it is shown in the code, the dataset actually contains many more column other than the ones needed for this analysis,

so this program could be expanded with deeper queries, for example involving the continent or the actualy country population.

```
final List<StructField> mySchemaFields = new ArrayList<>();
mySchemaFields.add(DataTypes.createStructField( name: "Date", DataTypes.DateType,  nullable: true));
mySchemaFields.add(DataTypes.createStructField( name: "Day", DataTypes.IntegerType,  nullable: true));
mySchemaFields.add(DataTypes.createStructField( name: "Month", DataTypes.IntegerType,  nullable: true));
mySchemaFields.add(DataTypes.createStructField( name: "Year", DataTypes.IntegerType,  nullable: true));
mySchemaFields.add(DataTypes.createStructField( name: "Cases", DataTypes.IntegerType,  nullable: false));
mySchemaFields.add(DataTypes.createStructField( name: "Deaths", DataTypes.IntegerType,  nullable: false));
mySchemaFields.add(DataTypes.createStructField( name: "Country", DataTypes.StringType,  nullable: true));
mySchemaFields.add(DataTypes.createStructField( name: "GeoID", DataTypes.StringType,  nullable: true));
mySchemaFields.add(DataTypes.createStructField( name: "CountryID", DataTypes.StringType,  nullable: true));
mySchemaFields.add(DataTypes.createStructField( name: "Population", DataTypes.IntegerType,  nullable: true));
mySchemaFields.add(DataTypes.createStructField( name: "Continent", DataTypes.StringType,  nullable: true));
final StructType mySchema = DataTypes.createStructType(mySchemaFields);

final Dataset<Row> rawData = spark
        .read()
        .option("delimiter", ",")
        .option("dateFormat", "dd/MM/yyyy")
        .schema(mySchema)
//        .csv(filePath + "spark-project.4/data/covid19data.csv"); USE THIS TO RUN IN INTELLIJ
        .csv(filePath); // USE THIS TO COMPILE AND THEN CREATE THE JAR
```

2. Data Cleaning

The dataset contains some rows where the number of cases is either missing or negative. Because of this those rows are not relevant for the analysis and simply discarded using the filter method and some conditions. A new dataset, the result of these operations on the original one, is returned.

```
//Removing rows where there is a negative or not reported number of cases
final Dataset<Row> cleaned = rawData.filter(rawData.col( colName: "Cases").$greater$eq( other: 0).
       and(rawData.col( colName: "Cases").isNotNull()));
```

3. Query 1

First, a Window object is instantiated, specifying that we want to partition the dataset by country and to compute each result using the current and the previous 6 rows. Then, a new dataset consisting of the old one plus a new column with the resulting moving average of each row is created.

```
//Splitting the rows into groups where there is a single day and the 6 previous days of the same country
WindowSpec window = Window.partitionBy( colName: "Country").rowsBetween(-6,0);
//The moving average is the average of the current day's cases and the six previous values
final Dataset<Row> mAvg = sortedData.
           withColumn( colName: "movingAverage", avg( columnName: "Cases").over(window));
```

4. **Query 2**

First, a Dataset called previousAverage with an additional column containing the moving average value of the previous day is created. Then, the difference is calculated, stored in a column, and a Dataset called difference is returned. Finally, the resulting dataset percentageChange is created calculating the percentage increase as explained in the introduction, dividing the value of the difference by the old movingAverage value and multiplying by 100. As it can be noted, here we are following a "pipeline" approach, where basically at each stage of the computation a new dataset is created, containing the intermediate result.

```
WindowSpec window2 = Window.partitionBy( colName: "Country").rowsBetween(-1,0);
final Dataset<Row> previousAverage = mAvg.withColumn( colName: "previousAvg",
    first( columnName: "movingAverage").over(window2));

final Dataset<Row> difference = previousAverage.withColumn( colName: "difference" , previousAverage.col( colName: "movingAverage")
        .$minus(previousAverage.col( colName: "previousAvg")));
final Dataset<Row> percentageChange = difference.withColumn( colName: "percentageChange",
        difference.col( colName: "difference").$div(difference.col( colName: "previousAvg")).multiply( other: 100));
```

5. **Query 3**

A window object is created, to specify that the data need to be partitioned by day and sorted in decreasing order of percentage increase, since we are computing a ranking of the nations on the same day. The final dataset contains a new column with the ranking of each row in the considered day. Rows ranked over 10 are not counted, since the request is to show the top 10 countries for each day.

```
//Partitioning by date because we want to rank for each day and ordering by decreasing percentage change
WindowSpec window3 = Window.partitionBy( colName: "Date").orderBy(desc( columnName: "percentageChange"));
//Adding a column with the rank in that day of each row
final Dataset<Row> percentageRanking = percentageChange.withColumn( colName: "row", row_number().over(window3))
        .filter(col( colName: "row").$less$eq( other: 10));
```

Testing Runs

To test the program, a cluster of virtual machines composed of one master node and two workers node has been setup. To observe the different behaviors, both the full dataset and a reduced version with only two countries and a reduced number of data points have been used. Also, trials using only one worker have been done:

- Run with 2 workers and full dataset:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
0	192.168.56.102:36281	Active	0	0.0 B / 434.4 MiB	0.0 B	4	0	0	8	8	4 s (0.1 s)	4.9 MiB	3 MiB	3 MiB
driver	192.168.56.101:46465	Active	0	0.0 B / 434.4 MiB	0.0 B	0	0	0	0	0	17 s (0.0 ms)	0.0 B	0.0 B	0.0 B
1	192.168.56.103:40563	Active	0	0.0 B / 434.4 MiB	0.0 B	4	0	0	13	13	4 s (0.1 s)	11.3 MiB	3.3 MiB	3.3 MiB

- Run with 2 workers and reduced dataset:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
0	192.168.56.102:41919	Active	0	0.0 B / 434.4 MiB	0.0 B	4	0	0	15	15	3 s (43.0 ms)	19.8 KiB	29.1 KiB	29.1 KiB
driver	192.168.56.101:45113	Active	0	0.0 B / 434.4 MiB	0.0 B	0	0	0	0	0	14 s (0.0 ms)	0.0 B	0.0 B	0.0 B
1	192.168.56.103:36151	Active	0	0.0 B / 434.4 MiB	0.0 B	4	0	0	6	6	2 s (30.0 ms)	13.2 KiB	5.5 KiB	5.5 KiB

- Run with 1 worker and full dataset:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
0	192.168.56.102:41695	Active	0	0.0 B / 434.4 MiB	0.0 B	4	0	0	21	21	6 s (0.2 s)	16.2 MiB	6.3 MiB	6.3 MiB
driver	192.168.56.101:40839	Active	0	0.0 B / 434.4 MiB	0.0 B	0	0	0	0	0	14 s (0.0 ms)	0.0 B	0.0 B	0.0 B

- Run with 1 worker and reduced dataset:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
0	192.168.56.102:41113	Active	0	0.0 B / 434.4 MiB	0.0 B	4	0	0	21	21	3 s (43.0 ms)	32.9 KiB	34.5 KiB	34.5 KiB
driver	192.168.56.101:44379	Active	0	0.0 B / 434.4 MiB	0.0 B	0	0	0	0	0	12 s (0.0 ms)	0.0 B	0.0 B	0.0 B

Results

As we can see from the performance measured using the history server, the bottleneck of the program is not the actual computation of the results from the dataset, since the task time of the workers in all the cases is much smaller than the time used by the driver node, both with the full or the reduced dataset. We can therefore conclude that the communication and synchronization activities performed by the driver, as well as the printing of the results in the console, are more demanding than what is needed to compute in this specific application.

Useful links

Dataset: https://www.ecdc.europa.eu/en/publications-data/data-daily-new-cases-covid-19-eueea-country

Cluster setup: https://www.innovationmerge.com/2021/06/26/Setting-up-a-multi-node-Apache-Spark-Cluster-on-a-Laptop/