ex43

August 12, 2022

[14]:
```
# Solution Ex. 43
```

[1]:
```
#inputPathReadings = "/data/students/bigdata-01QYD/ex_data/Ex43/data/readings.
 ↪txt"
#inputPathNeighbors = "/data/students/bigdata-01QYD/ex_data/Ex43/data/neighbors.
 ↪txt"
#outputPath = "res_out_Ex43/"
#outputPath2 = "res_out_Ex43_2/"
#outputPath3 = "res_out_Ex43_3/"
#thresholdFreeSlots = 3
#thresholdCriticalPercentage = 0.8

inputPathReadings = "data/Ex43/data/readings.txt"
inputPathNeighbors = "data/Ex43/data/neighbors.txt"
outputPath = "res_out_Ex43/"
outputPath2 = "res_out_Ex43_2/"
outputPath3 = "res_out_Ex43_3/"
thresholdFreeSlots = 3
thresholdCriticalPercentage = 0.8
```

[2]:
```
# Solution Ex. 43 - part I
# Selection of the stations with a percentage of critical situations
# greater than 80%
```

[3]:
```
# Read the content of the readings file
readingsRDD = sc.textFile(inputPathReadings).cache()
```

[4]:
```
def criticalSituation(line):
    fields = line.split(",")
    # fields[0] is the station id
    # fields[5] is the number of free slots
    stationId = fields[0]
    numFreeSlots = int(fields[5])

    if  numFreeSlots < thresholdFreeSlots:
        return (stationId, (1, 1))
    else:
```

```
        return (stationId, (1, 0))
```

[5]:
```
# Count the number of total and critical readings for each station
# Create an RDD of pairs with
# key: stationId
# value: (numReadings, numCriticalReadings)
# ------- numReadings: 1 for each input line
# --------numCriticalReadings: 0 if the situation is not critical. 1 if it is␣
 ↪critical
stationCountPairRDD = readingsRDD.map(criticalSituation)
```

[7]:
```
#stationCountPairRDD.collect()
```

[8]:
```
# Compute the number of total and critical readings for each station
stationTotalCountPairRDD = stationCountPairRDD\
.reduceByKey(lambda c1, c2: (c1[0]+c2[0], c1[1]+c2[1]) )
```

[9]:
```
#stationTotalCountPairRDD.collect()
```

[10]:
```
# Compute the percentage of critical situations for each station
stationPercentagePairRDD = stationTotalCountPairRDD\
.mapValues(lambda counters: counters[1]/counters[0])
```

[11]:
```
#stationPercentagePairRDD.collect()
```

[12]:
```
# Select stations with percentage > 80%
selectedStationsPairRDD = stationPercentagePairRDD\
.filter(lambda sensorPerc: sensorPerc[1]>thresholdCriticalPercentage)
```

[13]:
```
#selectedStationsPairRDD.collect()
```

[14]:
```
# Sort the stored stations by decreasing percentage of critical situations
selectedStationsSortedPairRDD = selectedStationsPairRDD\
.sortBy(lambda sensorPerc: sensorPerc[1], ascending=False)
```

[15]:
```
#selectedStationsSortedPairRDD.collect()
```

[17]:
```
selectedStationsSortedPairRDD.saveAsTextFile(outputPath)
```

[18]:
```
# Solution Ex. 43 - part II
# Selection of the pairs (timeslot, station) with a percentage of
# critical situations greater than 80%
```

[19]:
```
def criticalSituationTimeslots(line):

    fields = line.split(",")
```

```
    # fields[0] is the station id
    # fields[2] is the hour
    # fields[5] is the number of free slots

    stationId = fields[0]
    numFreeSlots = int(fields[5])

    minTimeslotHour = 4 * ( int(fields[2]) // int(4))
    maxTimeslotHour = minTimeslotHour + 3

    timestamp = "ts[" + str(minTimeslotHour) + "-" + str(maxTimeslotHour) + "]"

    key = (timestamp, stationId)

    if  numFreeSlots < thresholdFreeSlots:
        return (key, (1, 1))
    else:
        return (key, (1, 0))
```

[20]:
```
# The input data are already in readingsRDD

# Count the number of total and critical readings for each (timeslot,stationId)
# Create an RDD of pairs with
# key: (timeslot,stationId)
# value: (numReadings, numCriticalReadings)
# ------- numReadings: 1 for each input line
# --------numCriticalReadings: 0 if the situation is not critical. 1 if it is␣
 ↪critical

timestampStationCountPairRDD = readingsRDD.map(criticalSituationTimeslots)
```

[21]:
```
#timestampStationCountPairRDD.collect()
```

[22]:
```
# Compute the number of total and critical readings for each (timeslot,station)
timestampStationTotalCountPairRDD = timestampStationCountPairRDD \
.reduceByKey(lambda c1, c2: (c1[0]+c2[0], c1[1]+c2[1]) )
```

[23]:
```
#timestampStationTotalCountPairRDD.collect()
```

[24]:
```
# Compute the percentage of critical situations for each (timeslot,station)
timestampStationPercentagePairRDD = timestampStationTotalCountPairRDD\
.mapValues(lambda counters: counters[1]/counters[0])
```

[25]:
```
#timestampStationPercentagePairRDD.collect()
```

[26]:
```
# Select (timeslot,station) pairs with percentage > 80%
selectedTimestampStationsPairRDD = timestampStationPercentagePairRDD\
.filter(lambda sensorPerc: sensorPerc[1]>thresholdCriticalPercentage)
```

[27]:
```
#selectedTimestampStationsPairRDD.collect()
```

[28]:
```
# Sort the stored pairs by decreasing percentage of critical situations
percentageTimestampStationsSortedPairRDD = selectedTimestampStationsPairRDD\
.sortBy(lambda sensorPerc: sensorPerc[1], ascending=False)
```

[30]:
```
#percentageTimestampStationsSortedPairRDD.collect()
```

[31]:
```
percentageTimestampStationsSortedPairRDD.saveAsTextFile(outputPath2)
```

[32]:
```
# Solution Ex. 43 - part III
# Select a reading (i.e., a line) of the first input file if and only if the␣
 ↪following constraints are true
# - The line is associated with a full station situation
# - All the neighbor stations of the station Si are full in the time stamp␣
 ↪associated with the current line
```

[33]:
```
# Read the file containing the list of neighbors for each station
neighborsRDD = sc.textFile(inputPathNeighbors)
```

[34]:
```
# Map each line of the input file to a pair stationid, list of neighbor stations
nPairRDD = neighborsRDD.map(lambda line: (line.split(",")[0], line.
 ↪split(",")[1].split(" ")) )
```

[35]:
```
# nPairRDD.collect()
```

[36]:
```
# Create a local dictionary in the main memory of the driver that will be used␣
 ↪to store the mapping
# stationid -> list of neighbors
# There are only 100 stations. Hence, you can suppose that data about neighbors␣
 ↪can be stored in the main memory
neighbors=nPairRDD.collectAsMap()
```

[37]:
```
# The input data are already in readingsRDD
```

[38]:
```
# Select the lines/readings associated with a full status (number of free slots␣
 ↪equal to 0)
fullStatusLines = readingsRDD.filter(lambda line: int(line.split(",")[5])==0)
```

[39]:
```
def extractTimestamp(reading):
    fields = reading.split(",")
    timestamp = fields[1] + fields[2] + fields[3]
```

```
    return timestamp
```

[40]:
```
# Create an RDD of pairs with key = timestamp and value=reading associated with␣
 ↪that timestamp
# The concatenation of fields[1], fields[2], fields[3] is the timestamp of the␣
 ↪reading
fullLinesPRDD = fullStatusLines.map(lambda reading: (extractTimestamp(reading),␣
 ↪reading))
```

[42]:
```
#fullLinesPRDD.collect()
```

[43]:
```
#  Collapse all the values with the same key in one single pair (timestamp,␣
 ↪reading associated with that timestamp)
fullReadingsPerTimestamp = fullLinesPRDD.groupByKey()
```

[45]:
```
#fullReadingsPerTimestamp.mapValues(lambda v: list(v)).collect()
```

[46]:
```
def selectReadingssFunc(pairTimeStampListReadings):
    # Extract the list of stations that appear in the readings
    # associated with the current key
    # (i.e., the list of stations that are full in this timestamp)
    # The list of readings is in the value part of the inpput key-value pair
    stations = []
    for reading in pairTimeStampListReadings[1]:
        # Extract the stationid from each reading
        fields = reading.split(",")
        stationId = fields[0]
        stations.append(stationId)


    # Iterate again over the list of readings to select the readings satistying␣
 ↪the constraint on the
    # full status situation of all neighboors
    selectedReading = []

    for reading in pairTimeStampListReadings[1]:
        # This reading must be selected if all the neighbors of
        # the station of this reading are also in the value of
        # the current key-value pair (i.e., if they are in list stations)
        # Extract the stationid of this reading
        fields = reading.split(",")
        stationId = fields[0]

        # Select the list of neighbors of the current station
        nCurrentStation = neighbors[stationId]
```

```
        # Check if all the neighbors of the current station are in value
        # (i.e., the local list stations) of the current key-value pair
        allNeighborsFull = True

        for neighborStation in nCurrentStation:
            if neighborStation not in stations:
                # There is at least one neighbor of th current station
                # that is not in the full status in this timestamp
                allNeighborsFull = False

        if allNeighborsFull == True:
            selectedReading.append(reading)

    return selectedReading
```

[47]:
```
# Each pair contains a timestamp and the list of readings (with number of free␣
 ↪slots equal to 0)
# associated with that timestamp.
# Check, for each reading in the list, if all the neighbors of the station of␣
 ↪that reading are
# also present in this list of readings
# Emit one "string" for each reading associated with a completely full status
selectedReadingsRDD = fullReadingsPerTimestamp.flatMap(selectReadingssFunc)
```

[49]:
```
#selectedReadingsRDD.collect()
```

[130]:
```
# Store the result in HDFS
selectedReadingsRDD.saveAsTextFile(outputPath3)
```

[ ]:
```

```