ex58

August 20, 2022

[ ]:
```
from pyspark import SparkConf, SparkContext
from pyspark.streaming import StreamingContext

batch_size = 2

conf = SparkConf().setAppName("ex58")
sc = SparkContext(conf=conf)
ssc = StreamingContext(sc, batch_size)
```

[2]:
```
inputPath = "data/Ex58/data/"
outputPath = "out58/"
```

[3]:
```
lines = ssc.textFileStream(inputPath)
```

[4]:
```
fullStations = lines\
    .filter(lambda station: int(station.split(",")[1])==0)\
        .map(lambda station: (fullStations.split(",")[3], fullStations.
 ↪split(",")[0]))
```

[5]:
```
fullStations.pprint()
fullStations.saveAsTextFiles(outputPath)
```

[6]:
```
ssc.start()
```

[ ]:
```
ssc.awaitTerminationOrTimeout(10)
ssc.stop(stopSparkContext=False)
```

ex59

August 20, 2022

[ ]:
```
from pyspark import SparkConf, SparkContext
from pyspark.streaming import StreamingContext

batch_size = 2

conf = SparkConf().setAppName("ex59")
sc = SparkContext(conf=conf)
ssc = StreamingContext(sc, batch_size)
```

[2]:
```
inputPath = "data/Ex59/data/"
outputPath = "out59/"
```

[3]:
```
lines = ssc.textFileStream(inputPath)
```

[4]:
```
fullStations = lines.filter(lambda line : int(line.split(",")[1])==0)
```

[5]:
```
numFullStations = fullStations.count()
```

[6]:
```
numFullStations.pprint()
numFullStations.saveAsTextFiles(outputPath)
```

[7]:
```
ssc.start()
```

[ ]:
```
ssc.awaitTerminationOrTimeout(20)
ssc.stop(stopSparkContext=False)
```

ex60

August 20, 2022

[ ]:
```
from pyspark import SparkContext, SparkConf
from pyspark.streaming import StreamingContext

batch_size = 2

conf = SparkConf().setAppName("ex60")
sc = SparkContext(conf=conf)
ssc = StreamingContext(sc, batch_size)
```

[2]:
```
inputPath = "data/Ex60/data/"
```

[3]:
```
lines = ssc.textFileStream(inputPath)
```

[5]:
```
fullStations = lines\
    .filter(lambda station : int(station.split(",")[1])==0)\
        .map(lambda station : station.split(",")[0])\
            .transform(lambda batchRDD : batchRDD.distinct())\
                .pprint()
```

[ ]:
```
ssc.start()
```

[ ]:
```
ssc.awaitTerminationOrTimeout(20)
ssc.stop(stopSparkContext=False)
```

ex61

August 20, 2022

[ ]:
```
from pyspark import SparkConf, SparkContext
from pyspark.streaming import StreamingContext

batch_size = 2

conf = SparkConf().setAppName("ex61")
sc = SparkContext(conf=conf)
ssc = StreamingContext(sc, batch_size)
```

[2]:
```
inputPath = "data/Ex61/data/"
```

[3]:
```
lines = ssc.textFileStream(inputPath)
```

[4]:
```
stationBatch = lines.map(lambda station : station.split(",")[1])\
    .reduce(lambda v1, v2 : max(v1, v2))
```

[ ]:
```
stationBatch.pprint()
```

[ ]:
```
ssc.start()
```

[ ]:
```
ssc.awaitTerminationOrTimeout(20)
ssc.stop(stopSparkContext=False)
```

ex62Bis

August 20, 2022

[ ]:
```
from pyspark.streaming import StreamingContext
```

[ ]:
```
# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 30)
```

[ ]:
```
# Create a (Receiver) DStream that will connect to localhost:9999
linesDStream = ssc.socketTextStream("localhost", 9999)
```

[ ]:
```
# Computer for each stockID the price variation (compute it for each batch).
# Select only the stocks with a price variation (%) greater than 0.5%
```

[ ]:
```
# Return one pair (stockId, (price, price) )  for each input record

def extractStockIdPricePrice(line):
    fields = line.split(",")

    stockId = fields[1]
    price = fields[2]

    return (stockId, (float(price), float(price)) )


stockIdPriceDStream = linesDStream.map(extractStockIdPricePrice)
```

[ ]:
```
# Compute max and min for each stockId
# Set the windows zise to 60 seconds
# The sliding interval is equal to 30 seconds, i.e., 1 batch
stockIdMaxMinPriceDStream = stockIdPriceDStream\
.reduceByKeyAndWindow(lambda v1, v2: ( max(v1[0],v2[0]), min(v1[1],v2[1]) ),␣
 ↪None, 60)
```

[ ]:
```
# Compute variation for each stock
stockIdVariationDStream = stockIdMaxMinPriceDStream\
.mapValues(lambda MaxMinValue: 100.0*(MaxMinValue[0]-MaxMinValue[1])/
 ↪MaxMinValue[0] )
```

[ ]:
```
# Select only the stocks with variation greater than 0.5%
selectedStockIdsVariationsDStream = stockIdVariationDStream.filter(lambda pair:␣
↪pair[1]>0.5)
```

[ ]:
```
selectedStockIdsVariationsDStream.pprint()
```

[ ]:
```
#Start the computation
ssc.start()
```

[ ]:
```
# Run this application for 200 seconds
ssc.awaitTerminationOrTimeout(200)
ssc.stop(stopSparkContext=False)
```

[ ]:
```

```

ex62

August 20, 2022

[25]:
```
from pyspark.streaming import StreamingContext
```

[26]:
```
# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 30)
```

[27]:
```
# Create a (Receiver) DStream that will connect to localhost:9999
linesDStream = ssc.socketTextStream("localhost", 9999)
```

[28]:
```
# Computer for each stockID the price variation (compute it for each batch).
# Select only the stocks with a price variation (%) greater than 0.5%
```

[29]:
```
# Return one pair (stockId, (price, price) )  for each input record

def extractStockIdPricePrice(line):
    fields = line.split(",")

    stockId = fields[1]
    price = fields[2]

    return (stockId, (float(price), float(price)) )


stockIdPriceDStream = linesDStream.map(extractStockIdPricePrice)
```

[30]:
```
# Compute max and min for each stockId
stockIdMaxMinPriceDStream = stockIdPriceDStream\
.reduceByKey(lambda v1, v2: ( max(v1[0],v2[0]), min(v1[1],v2[1]) ) )
```

[31]:
```
# Compute variation for each stock
stockIdVariationDStream = stockIdMaxMinPriceDStream\
.mapValues(lambda MaxMinValue: 100.0*(MaxMinValue[0]-MaxMinValue[1])/
 ↪MaxMinValue[0] )
```

[32]:
```
# Select only the stocks with variation greater than 0.5%
selectedStockIdsVariationsDStream = stockIdVariationDStream.filter(lambda pair:␣
 ↪pair[1]>0.5)
```

[33]:
```
selectedStockIdsVariationsDStream.pprint()
```

[38]:
```
#Start the computation
ssc.start()
```

[ ]:
```
# Run this application for 90 seconds
ssc.awaitTerminationOrTimeout(90)
ssc.stop(stopSparkContext=False)
```

[ ]:
```

```

ex63

August 20, 2022

[1]:
```
from pyspark.streaming import StreamingContext
```

[4]:
```
# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 2)
```

[5]:
```
inputFileStations = "data/Ex63/data/stations.csv"
```

[6]:
```
# "Standard" RDD associated with the characteristics of the stations
# Extract (stationId, name)
stationNameRDD = sc.textFile(inputFileStations)\
.map(lambda line: (line.split("\t")[0], line.split("\t")[3]) ).cache()
```

[3]:
```
# Create a (Receiver) DStream that will connect to localhost:9999
readingsDStream = ssc.socketTextStream("localhost", 9999)
```

[7]:
```
# Each readings has the format:
# stationId,#free slots,#used slots,timestamp
# Select readings with num. free slots = 0
fullReadingsDStream = readingsDStream.filter(lambda line: int(line.
 ↪split(",")[1])==0)
```

[8]:
```
# Extract pairs (stationId, timestamp)
stationIdTimestampDStream = fullReadingsDStream.map(lambda line: (line.
 ↪split(",")[0],line.split(",")[3]))
```

[9]:
```
# Join the content of the DStream with the "standard" RDD to retrieve
# the name of each station.
# To perform this join between streaming and
# non-streaming RDDs the transform transformation must be used
joinDStream = stationIdTimestampDStream.transform(lambda batchRDD: batchRDD.
 ↪join(stationNameRDD))
```

[10]:
```
# Extract (name of the station, timestamp)
# It is the value part of the returned pairs
stationNameTimestampDStream = joinDStream.map(lambda pair: pair[1])
```

[11]:
```
stationNameTimestampDStream.pprint()
```

[14]:
```
#Start the computation
ssc.start()
```

[ ]:
```
# Run this application for 90 seconds
ssc.awaitTerminationOrTimeout(90)
ssc.stop(stopSparkContext=False)
```

[ ]:
```

```

ex64

August 20, 2022

[ ]:
```
from pyspark.streaming import StreamingContext
```

[ ]:
```
historicalInputFile = "data/Ex64/data/historicalData.txt"
```

[ ]:
```
# Read the historical data and compute the maximum and minimum price for each␣
 ↪stock
# Non-streaming RDD
historicalDataRDD = sc.textFile(historicalInputFile)
```

[ ]:
```
# Return one pair (stockId, (price, price) )  for each input record
def extractStockIdPricePrice(line):
    fields = line.split(",")

    stockId = fields[1]
    price = fields[2]

    return (stockId, (float(price), float(price)) )



stockIdPriceHistoricalRDD = historicalDataRDD.map(extractStockIdPricePrice)
```

[ ]:
```
# Compute max and min for each stockId based on the historical data
stockIdPriceHistoricalMaxMinRDD = stockIdPriceHistoricalRDD\
.reduceByKey(lambda v1, v2: ( max(v1[0],v2[0]), min(v1[1],v2[1]) ) ).cache()
```

[ ]:
```
# Create a Spark Streaming Context object
#ssc = StreamingContext(sc, 60)
ssc = StreamingContext(sc, 10)
```

[ ]:
```
# Create a (Receiver) DStream that will connect to localhost:9999
pricesDStream = ssc.socketTextStream("localhost", 9999)
```

[ ]:
```
# Join on the stockid each input record of the input stream with the
# content of stockIdPriceHistoricalMaxMinRDD to retrieve
# the historical maximum-minimum range of the stock
```

[ ]:
```
# Return one pair (stockId,price) for each input record
stockIdPriceDStream = pricesDStream.map(lambda record: ( record.split(",")[1] ,␣
 ↪float(record.split(",")[2])) )
```

[ ]:
```
# Join the RDD associated with the content of the current batch and
# the non-streaming RDD stockIdPriceHistoricalMaxMinRDD
stockIdPriceMaxMinDStream = stockIdPriceDStream\
.transform(lambda batchRDD: batchRDD.join(stockIdPriceHistoricalMaxMinRDD))
```

[ ]:
```
# Select only lines with price > maximum historical price
# or price < minimum historical price
def anomalyValue(pair):
    currentPrice = pair[1][0]
    stockHistoricalMaxPrice = pair[1][1][0]
    stockHistoricalMinPrice = pair[1][1][1]

    if currentPrice>stockHistoricalMaxPrice or␣
 ↪currentPrice<stockHistoricalMinPrice:
        return True
    else:
        return False



selectedStockPricesDStream = stockIdPriceMaxMinDStream.filter(anomalyValue)
```

[ ]:
```
# Retrieve only the stockIDs and apply distinct to remove duplicates
# keys and distinct are not available for DStreams.
# transform must be used
selectStockIdsDStream = selectedStockPricesDStream\
.transform(lambda batchRDD: batchRDD.keys().distinct())
```

[ ]:
```
selectStockIdsDStream.pprint()
```

[ ]:
```
#Start the computation
ssc.start()
```

[ ]:
```
# Run this application for 90 seconds
ssc.awaitTerminationOrTimeout(90)
ssc.stop(stopSparkContext=False)
```

[ ]:
```

```

ex64v2

August 20, 2022

[ ]:
```
# Second version. This version is more efficient than the previous one
# because the amount of joined data is reduced.
from pyspark.streaming import StreamingContext
```

[ ]:
```
historicalInputFile = "data/Ex64/data/historicalData.txt"
```

[ ]:
```
# Read the historical data and compute the maximum and minimum price for each␣
 ↪stock
# Non-streaming RDD
historicalDataRDD = sc.textFile(historicalInputFile)
```

[ ]:
```
# Return one pair (stockId, (price, price) )  for each input record
def extractStockIdPricePrice(line):
    fields = line.split(",")

    stockId = fields[1]
    price = fields[2]

    return (stockId, (float(price), float(price)) )


stockIdPriceHistoricalRDD = historicalDataRDD.map(extractStockIdPricePrice)
```

[ ]:
```
# Compute max and min for each stockId based on the historical data
stockIdPriceHistoricalMaxMinRDD = stockIdPriceHistoricalRDD\
.reduceByKey(lambda v1, v2: ( max(v1[0],v2[0]), min(v1[1],v2[1]) ) ).cache()
```

[ ]:
```
# Create a Spark Streaming Context object
#ssc = StreamingContext(sc, 60)
ssc = StreamingContext(sc, 10)
```

[ ]:
```
# Create a (Receiver) DStream that will connect to localhost:9999
pricesDStream = ssc.socketTextStream("localhost", 9999)
```

[ ]:
```
# Compute max and min for each stockId of each input batch
stockIdPriceDStream = pricesDStream.map(extractStockIdPricePrice)\
.reduceByKey(lambda v1, v2: ( max(v1[0],v2[0]), min(v1[1],v2[1]) ) )
```

[ ]:
```
# Join stockIdPriceDStream with stockIdPriceHistoricalMaxMinRDD
# Join the RDD associated with the content of the current batch and
# the non-streaming RDD stockIdPriceHistoricalMaxMinRDD
stockIdPriceMaxMinDStream = stockIdPriceDStream\
.transform(lambda batchRDD: batchRDD.join(stockIdPriceHistoricalMaxMinRDD))
```

[ ]:
```
# Select only stocks with stream max price > maximum historical price
# or stream min price < minimum historical price
def anomalyValue(pair):
    stockBatchMaxPrice = pair[1][0][0]
    stockBatchMinPrice = pair[1][0][1]

    stockHistoricalMaxPrice = pair[1][1][0]
    stockHistoricalMinPrice = pair[1][1][1]

    if stockBatchMaxPrice>stockHistoricalMaxPrice or␣
 ↪stockBatchMinPrice<stockHistoricalMinPrice:
        return True
    else:
        return False



selectedStockPricesDStream = stockIdPriceMaxMinDStream\
.filter(anomalyValue)
```

[ ]:
```
# Retrieve only the stockIDs of the selected stocks
# keys is not available for DStreams.
# transform must be used or map
selectStockIdsDStream = selectedStockPricesDStream\
.transform(lambda batchRDD: batchRDD.keys())
```

[ ]:
```
selectStockIdsDStream.pprint()
```

[ ]:
```
#Start the computation
ssc.start()
```

[ ]:
```
# Run this application for 90 seconds
ssc.awaitTerminationOrTimeout(90)
ssc.stop(stopSparkContext=False)
```

[ ]:
```
ssc.stop(stopSparkContext=False)
```

[ ]:
```

```

ex65v2

August 20, 2022

[ ]:
```
# Second version. This version is more efficient than the previous one
# because the amount of joined data is reduced.
from pyspark.streaming import StreamingContext
```

[ ]:
```
historicalInputFile = "data/Ex65/data/historicalData.txt"
```

[ ]:
```
# Read the historical data and compute the maximum and minimum price for each␣
 ↪stock
# Non-streaming RDD
historicalDataRDD = sc.textFile(historicalInputFile)
```

[ ]:
```
# Return one pair (stockId, (price, price) )  for each input record
def extractStockIdPricePrice(line):
    fields = line.split(",")

    stockId = fields[1]
    price = fields[2]

    return (stockId, (float(price), float(price)) )



stockIdPriceHistoricalRDD = historicalDataRDD.map(extractStockIdPricePrice)
```

[ ]:
```
# Compute max and min for each stockId based on the historical data
stockIdPriceHistoricalMaxMinRDD = stockIdPriceHistoricalRDD\
.reduceByKey(lambda v1, v2: ( max(v1[0],v2[0]), min(v1[1],v2[1]) ) ).cache()
```

[ ]:
```
# Create a Spark Streaming Context object
#ssc = StreamingContext(sc, 60)
ssc = StreamingContext(sc, 5)
```

[ ]:
```
# Create a (Receiver) DStream that will connect to localhost:9999
pricesDStream = ssc.socketTextStream("localhost", 9999)
```

[ ]:
```
# Compute max and min for each stockId of each input window
# - windowDuration = 60 seconds
# - slideDuration = 30 seconds
stockIdPriceDStream = pricesDStream.map(extractStockIdPricePrice)\
.reduceByKeyAndWindow(lambda v1, v2: ( max(v1[0],v2[0]), min(v1[1],v2[1])␣
 ↪),None\
                      ,10,5)
#                      ,60, 30)
```

[ ]:
```
# Join stockIdPriceDStream with stockIdPriceHistoricalMaxMinRDD
# Join the RDD associated with the content of the current batch and
# the non-streaming RDD stockIdPriceHistoricalMaxMinRDD
stockIdPriceMaxMinDStream = stockIdPriceDStream\
.transform(lambda batchRDD: batchRDD.join(stockIdPriceHistoricalMaxMinRDD))
```

[ ]:
```
# Select only stocks with stream max price > maximum historical price
# or stream min price < minimum historical price
def anomalyValue(pair):
    stockBatchMaxPrice = pair[1][0][0]
    stockBatchMinPrice = pair[1][0][1]

    stockHistoricalMaxPrice = pair[1][1][0]
    stockHistoricalMinPrice = pair[1][1][1]

    if stockBatchMaxPrice>stockHistoricalMaxPrice or␣
 ↪stockBatchMinPrice<stockHistoricalMinPrice:
        return True
    else:
        return False



selectedStockPricesDStream = stockIdPriceMaxMinDStream\
.filter(anomalyValue)
```

[ ]:
```
# Retrieve only the stockIDs of the selected stocks
# keys is not available for DStreams.
# transform must be used or map
selectStockIdsDStream = selectedStockPricesDStream\
.transform(lambda batchRDD: batchRDD.keys())
```

[ ]:
```
selectStockIdsDStream.pprint()
```

[ ]:
```
#Start the computation
ssc.start()
```

[ ]:
```
# Run this application for 90 seconds
ssc.awaitTerminationOrTimeout(90)
ssc.stop(stopSparkContext=False)
```

[ ]:
```
ssc.stop(stopSparkContext=False)
```

[ ]:
```

```