CS 6240- Large-scale Parallel Data Processing
Homework 3
Name: Srikanth Babu Mandru

Words used in document:
user1 – user at first position in edge
user2 - user at second position in edge
PATH2 – number of paths of length 2
TRIANGLES– number of TRIANGLES in twitter edges dataset

<u>Twitter Followers Count (Spark/Scala) Implementation:</u>

Pseudo Codes:

As the Spark/Scala code is very concise, I have explained the pseudo code in terms of Spark/Scala code (without "." chaining") with comments followed by for each statement. All the functions are applied in an order as mentioned in the pseudo-code.

RDD-G:

```
textFile  ← Read the input file

textFile.
     flatMap(line => line.split("\n"))     // split the input records by "\n"

     flatMap(line => line.split(",")(1))   //parse the records by using "," separation &emit user2

     map(word => (word, 1))                // assign "1" to each user

     groupByKey()                          // group the records using "user2" as key

     map(token => (token._1,token._2.sum)) //for each user, emit user id and count of followers
```

RDD-R:

```
textFile.flatMap (line => line.split("\n"))       // split the input records by "\n"

         map (line => (line.split(",")(1), 1 ))   // parse records by using "," separation
                                                  // & emit (user2, count of "1")
        reduceByKey((a, b) => a+b)                // For each user, sum the followers and emit the
                                                  // (user id, followers count)
```

RDD-F:

```
textFile.
        flatMap(line => line.split("\n"))        // split the input records by "\n"

        map (line => (line.split(",")(1), 1 ))   // parse records by using "," separation
                                                 // & emit ( user2, count of "1")

        foldByKey(0)((a, b) => a + b )           // For each user, sum the followers starting from "0"
                                                 // and emit the (user id, followers count)
```

RDD-A:

```
textFile.flatMap(line => line.split("\n"))       // split the input records by "\n"

        flatMap (line => line.split(",")(1))     //parse records by using "," separation &emit user2

        map (word => (word, 1))                  // assign "1" to each user

        aggregateByKey((k,v) => k+v , (k,v) => k+v) // For each user, sum the followers
                                                 // and emit the (user id, followers count)
```

DSET:

input_dataset ← read the input file into dataset with "user1", "user2" as columns

```
input_dataset.groupBy($"user2").count()          // group all the records for each user2 and count
                                                 //number of records which will be followers count
```

<u>Debug String Analysis</u>:

RDD-G:

```
(Debug info :,(10) CoalescedRDD[7] at coalesce at FollowerCount_RDDG.scala:29 []
 |  MapPartitionsRDD[6] at map at FollowerCount_RDDG.scala:29 []
 |  ShuffledRDD[5] at groupByKey at FollowerCount_RDDG.scala:28 []
 +-(40) MapPartitionsRDD[4] at map at FollowerCount_RDDG.scala:27 []
    |  MapPartitionsRDD[3] at flatMap at FollowerCount_RDDG.scala:26 []
    |  MapPartitionsRDD[2] at flatMap at FollowerCount_RDDG.scala:26 []
    |  input MapPartitionsRDD[1] at textFile at FollowerCount_RDDG.scala:25 []
    |  input HadoopRDD[0] at textFile at FollowerCount_RDDG.scala:25 [])
```

RDD-R:

```
(Debug info :,(40) ShuffledRDD[4] at reduceByKey at FollowerCount_RDDR.scala:23 []
 +-(40) MapPartitionsRDD[3] at map at FollowerCount_RDDR.scala:23 []
    |  MapPartitionsRDD[2] at flatMap at FollowerCount_RDDR.scala:22 []
    |  input MapPartitionsRDD[1] at textFile at FollowerCount_RDDR.scala:21 []
    |  input HadoopRDD[0] at textFile at FollowerCount_RDDR.scala:21 [])
```

RDD-F:

```
(Debug info :,(40) ShuffledRDD[4] at foldByKey at FollowerCount_RDDF.scala:23 []
 +-(40) MapPartitionsRDD[3] at map at FollowerCount_RDDF.scala:22 []
    |  MapPartitionsRDD[2] at flatMap at FollowerCount_RDDF.scala:22 []
    |  input MapPartitionsRDD[1] at textFile at FollowerCount_RDDF.scala:21 []
    |  input HadoopRDD[0] at textFile at FollowerCount_RDDF.scala:21 [])
```

RDD-A:

```
(Debug info :,(40) ShuffledRDD[5] at aggregateByKey at FollowerCount_RDDA.scala:24 []
 +-(40) MapPartitionsRDD[4] at map at FollowerCount_RDDA.scala:23 []
    |  MapPartitionsRDD[3] at flatMap at FollowerCount_RDDA.scala:22 []
    |  MapPartitionsRDD[2] at flatMap at FollowerCount_RDDA.scala:22 []
    |  input MapPartitionsRDD[1] at textFile at FollowerCount_RDDA.scala:21 []
    |  input HadoopRDD[0] at textFile at FollowerCount_RDDA.scala:21 [])
```

Explain() Analysis for DSET:

Logical and Physical plan:

```
== Parsed Logical Plan ==
'Aggregate ['users2], [unresolvedalias('users2, None), count(1) AS count#24L]
+- Project [_c0#10 AS users1#14, _c1#11 AS users2#15]
   +- Relation[_c0#10,_c1#11] csv

== Analyzed Logical Plan ==
users2: string, count: bigint
Aggregate [users2#15], [users2#15, count(1) AS count#24L]
+- Project [_c0#10 AS users1#14, _c1#11 AS users2#15]
   +- Relation[_c0#10,_c1#11] csv

== Optimized Logical Plan ==
Aggregate [users2#15], [users2#15, count(1) AS count#24L]
+- Project [_c1#11 AS users2#15]
   +- Relation[_c0#10,_c1#11] csv
```

== Physical Plan ==
*(2) HashAggregate(keys=[users2#15], functions=[count(1)], output=[users2#15, count#24L])
+- Exchange hashpartitioning(users2#15, 200)
   +- *(1) HashAggregate(keys=[users2#15], functions=[partial_count(1)], output=[users2#15, count#29L])
      +- *(1) Project [_c1#11 AS users2#15]
         +- *(1) FileScan csv [_c1#11] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/Users/srikanthmandru/my-system/semester 3/lsp/homeworks/HW3/Spark-traingl..., PartitionFilters: [], PushedFilters: [], ReadSchema: struct<_c1:string>

Important points to note from analysis:

From the above, it can be inferred that RDD-F(foldByKey), RDD-A (aggregatebykey), RDD-R (reducebykey) are similar to MapReduce's in-mapper combining.

(Twitter Social Amplifier)

Triangles Count through Joins (Spark/Scala) Implementation:

(1) Reduce-side Join RDD Implementation:

Pseudo-code:

input_data ← read the data from input file

traingle_counter ← 0

filterdata ← input_data.filter(user1 < max_filter and user2 < max_filter) // filter the input data to include
// only users with user ids less than max_filter

edges ← filterdata.map(user1 , user2) // emit tuples of (user1 , user2)

edges_reversed ← filterdata.map(user2 , user1) // emit tuples of (user2 , user1)

path2_edges ← edges_reversed.join(edges) // Join two data with one table's key as "user2" and other
// table's key as "user1". This forms PATH2 edges.

path3_edges ← path2_edges.join(edges_reversed) // Join path2 edges with edges table in reverse order.
// This forms Path3 edges

path3_edges.foreach(users => if (user1 == user2){ traingle_counter += 1}) // Check if users are same,
// if same increment the triangle count by "1"

return (traingle_counter /3) // divide by "3" to get exact triangles

(2) Reduce-side Join Dataset/ DataFrame Implementation:

Pseudo-code:

```
twitterDS1 ← load (input file)
            .where(user1 < max_filter  and user2 <  max_filter) // Read input file into dataset and remove
                                                              // edges with user ids greater than max_filter
twitterDS2 ← twitterDS1

PATH2DS ←
 twitterDS1.joinWith(twitterDS2, twitterDS1.user2 == twitterDS2.user1) // Join both the datasets with
                                                  // keys as "user2" and "user1" of different datasets

TRIANGLEDS ←
 PATH2DS
 .joinWith(twitterDS1, PATH2DS.user1 == twitterDS1.user2 && PATH2DS.user2 == twitterDS1.user1 )
                              // Join edges of length 2 with original edges and check if users are same
                              // If same, keep the records in dataset.
return (size (TRIANGLEDS) / 3)   // count the records in dataset and divide by "3" to get exact triangles
```

(3) Replicated (Map-side) Join RDD Implementation:

Pseudo-code:

```
input_data ← read the data from input file

traingle_counter ← 0

filterdata ← input_data.filter( user1 < max_filter and user2 < max_filter )  // filter the input data
to include
                                                  // only users with user ids less than max_filter
edges  ←  filterdata.map(user1 , user2)                          // emit tuples of (user1 , user2)

edges_map ← broadcast(edges.groupby(user1))         //create Hashmap for each user1 and
                                                    // broadcast Hashmap to each partition

// Now, Join the edges with HashMap "edges_map" two times. While joining recursively, check if first
// and last users are same and if same, increment the number of "TRIANGLES"
edges.map{ (user1 , user2) =>
        user2_set ← edges_map.get(user2)
        for each user3 ∈ user2_set:
                Increment "PATH2_counter" by 1
                user3_set ← edges_map.get(user3)
                for each user4 ∈ user3_set:
                        if (user4 == user1)
                                Increment "traingle_counter" by 1
}
return ( traingle_counter /3)     // divide by "3" to get exact triangles
```

(4) Replicated (Map-side) Join Dataset/DataFrame Implementation:

Pseudo-code:

```
twitterDS1 ← load (input file)
             .where(user1 < max_filter  and user2 <  max_filter) // Read input file into dataset and remove
                                                      // edges with user ids greater than max_filter

twitterDS2 ← broadcast(twitterDS1)                    // broadcast the twitter edges data over partitions

PATH2DS ←
  twitterDS1.join(twitterDS2, twitterDS1.user2 == twitterDS2.user1) // Join both the datasets with
                                                      // keys as "user2" and "user1" of different datasets


TRIANGLEDS ←
  PATH2DS
 .joinWith(twitterDS1, PATH2DS.user1 == twitterDS1.user2 && PATH2DS.user2 == twitterDS1.user1 )
                              // Join edges of length 2 with original edges and check if users are same
                              // If same, keep the records in dataset.
return (size (TRIANGLEDS) / 3)   // count the records in dataset and divide by "3" to get exact triangles
```

**** Note:** 'broadcast(twitterDS1)' makes spark to automatically choose the "partition + broadcast" join type while using "join".

Output:

Output can be inferred from the both syslog files provided in the "log files path" directory mentioned below and output files from the spark/scala program:

Configuration	Small Cluster Result (1 master, 5 Workers)	Large Cluster Result (1 master, 8 Workers)
RS-R, MAX = 10000	Running time: 4.87 minutes, Triangle count: 520296	Running time: 3.9 minutes, Triangle count: 520296
RS-D, MAX = 10000	Running time: 10.4 minutes, Triangle count: 520296	Running time: 9.13 minutes, Triangle count: 520296
Rep-R, MAX = 10000	Running time: 2.2 minutes, Triangle count: 520296	Running time: 2.17 minutes, Triangle count: 520296
Rep-D, MAX = 10000	Running time: 10.9 minutes, Triangle count: 520296	Running time: 8.73 minutes, Triangle count: 520296

Note: Here, for large cluster analysis, I couldn't able to get more than 8 workers. All programs are executed with similar setup as HW2 like machine instance type, max_filter value, etc.

Note: For RDD-G, I got Java Heap error even after many attempts to solve the error. I have included the syslog for the run. Except that, all other programs ran successfully.

Log files path:
I have provided log files in the following directory separately for "Reduce side Join" and "Replicated join". Also, separated each with respect to "RDD" and "Dataset".

Special mention: Along with syslog from executions "steps" of aws, included syslog from aws containers as "* from container" beside respective syslog, where you can see the results of number of triangles and PATH2 by searching for "root" in the file.

HW3/logs/

Output files path:

HW3/output/ ; where '#' represents the run number

Readme path: (for execution steps or procedure)
HW3/Readme.txt

Report path:
HW3/Srikanth_Mandru_HW3.pdf