1. Pseudo-code for 5 Twitter-follower-count Programs

- RDD-G

```
FOLLOWER-COUNT-RDD-G


val textFile <- read input csv file
val counts =
  textFile
    .map(edge => edge.split(",")(1)) // split edge to get followedID
    .map(followedID => (followedID, 1)) // map each node to (node, 1)
    .groupByKey() // group by key (followedID)
    .map(pair => (pair._1, pair._2.sum)) // for each node, aggregate count
    .filter{case (userID, count) => count % 100 == 0} // get counts divisible by 100

output <- counts
```

- RDD-R

```
FOLLOWER-COUNT-RDD-R


val textFile <- read input csv file
val counts =
  textFile
    .map(edge => edge.split(",")(1)) // split edge to get followedID
    .map(followedID => (followedID, 1)) // map each node to (node, 1)
    .reduceByKey(_ + _) // reduce by key (followedID) which aggregates the count
    .filter{case (userID, count) => count % 100 == 0} // get counts divisible by 100

output <- counts
```

- RDD-F

```
FOLLOWER-COUNT-RDD-F


val textFile <- read input csv file
val counts =
  textFile
    .map(edge => edge.split(",")(1)) // split edge to get followedID
    .map(followedID => (followedID, 1)) // map each node to (node, 1)
    .foldByKey(0)(_ + _) // fold by key (followedID) which aggregates the count
    .filter{case (userID, count) => count % 100 == 0} // get counts divisible by 100

output <- counts
```

- RDD-A

```
FOLLOWER-COUNT-RDD-A


val textFile <- read input csv file

val initialCount = 0
val sumFollowerCounts = (c1: Int, c2: Int) => c1 + c2 // rule summing up int values
val sumPartitionCounts = (p1: Int, p2: Int) => p1 + p2 // rule summing up int values

val counts = textFile
             .map(edge => edge.split(",")(1)) // split edge to get followedID
             .map(followedID => (followedID, 1)) // map each node to (node, 1)
             .aggregateByKey(initialCount)(sumFollowerCounts, sumPartitionCounts)
               // sum up each of the follower counts & aggregate the partition counts
             .filter{case (userID, count) => count % 100 == 0}
               // get counts divisible by 100

output <- counts
```

- DSET

```
FOLLOWER-COUNT-DSET


val textFile <- read input csv file
val rdd = textFile
          .map(edge => edge.split(",")(1)) // split edge to get followedID
          .map(followedID => (followedID, 1)) // map each node to (node, 1)

val ds =  spark.createDataset(rdd) // convert RDD to DataSet

val counts =
  ds
  .groupBy(ds.col("_1")) // group by the first column (followedID)
  .agg(sum(ds.col("_2"))) // aggregate the values of second column (count)
  .filter("sum(_2) % 100 == 0") // get counts divisible by 100

output <- counts
```

2. Source code for 5 Twitter-follower-count Programs

- RDD-G: https://github.com/CS6240/hw-3-jill666666/blob/master/src/main/scala/fc/RDD-G.scala
- RDD-R: https://github.com/CS6240/hw-3-jill666666/blob/master/src/main/scala/fc/RDD-R.scala
- RDD-F: https://github.com/CS6240/hw-3-jill666666/blob/master/src/main/scala/fc/RDD-F.scala
- RDD-A: https://github.com/CS6240/hw-3-jill666666/blob/master/src/main/scala/fc/RDD-A.scala
- DSET: https://github.com/CS6240/hw-3-jill666666/blob/master/src/main/scala/fc/DSET.scala

3. Aggregation and Shuffle

* 'toDebugString' method logs the RDD lineage graph. Using this we can find out how the shuffle, partition, and parallelism work. Indentation indicates the shuffle boundary.

** Value members referenced from: https://spark.apache.org/docs/latest/api/scala/index.html

a. RDD-G

```
2021-10-22 22:14:08,247 INFO root: (40) MapPartitionsRDD[6] at filter at RDD-G.scala:31 []
 |   MapPartitionsRDD[5] at map at RDD-G.scala:30 []
 |   ShuffledRDD[4] at groupByKey at RDD-G.scala:28 []
 +-(40) MapPartitionsRDD[3] at map at RDD-G.scala:28 []
    |   MapPartitionsRDD[2] at map at RDD-G.scala:27 []
    |   /Users/sunho/Dropbox/Boston/CS6240/twitter-dataset/data/ MapPartitionsRDD[1] at textFile at RDD-G.scala:25 []
    |   /Users/sunho/Dropbox/Boston/CS6240/twitter-dataset/data/ HadoopRDD[0] at textFile at RDD-G.scala:25 []
```

- First of all, 'groupByKey()' reshuffles the data. Afterward, we use 'RDD.sum()' equivalent in order to sum up the data. Shuffling happens before the aggregati on. The operation is very expensive due to the shuffling.
- Ref: Apache Spark ScalaDoc groupByKey()

b. RDD-R

```
2021-10-22 22:22:26,061 INFO root: (40) MapPartitionsRDD[5] at filter at RDD-R.scala:30 []
 |   ShuffledRDD[4] at reduceByKey at RDD-R.scala:29 []
 +-(40) MapPartitionsRDD[3] at map at RDD-R.scala:28 []
    |   MapPartitionsRDD[2] at map at RDD-R.scala:27 []
    |   /Users/sunho/Dropbox/Boston/CS6240/twitter-dataset/data/ MapPartitionsRDD[1] at textFile at RDD-R.scala:25 []
    |   /Users/sunho/Dropbox/Boston/CS6240/twitter-dataset/data/ HadoopRDD[0] at textFile at RDD-R.scala:25 []
```

- Here, unlike 'groupByKey()', the 'reduceByKey()' work as a combiner. Before shuffling the data, it can combine values for each key on each partition. Basic ally, it works similarly to a 'combiner' in MapReduce.

- Ref: Apache Spark ScalaDoc reduceByKey()

c. RDD-F

```
2021-10-22 22:24:39,189 INFO root: (40) MapPartitionsRDD[5] at filter at RDD-F.scala:30 []
 |   ShuffledRDD[4] at foldByKey at RDD-F.scala:29 []
 +-(40) MapPartitionsRDD[3] at map at RDD-F.scala:28 []
    |   MapPartitionsRDD[2] at map at RDD-F.scala:27 []
    |   /Users/sunho/Dropbox/Boston/CS6240/twitter-dataset/data/ MapPartitionsRDD[1] at textFile at RDD-F.scala:25 []
    |   /Users/sunho/Dropbox/Boston/CS6240/twitter-dataset/data/ HadoopRDD[0] at textFile at RDD-F.scala:25 []
```

- 'foldByKey()' merges the values for each key using the combine functions. But this does not cause shuffle.
- Ref: Apache Spark ScalaDoc foldByKey()

d. RDD-A

```
2021-10-22 22:32:33,049 INFO root: (40) MapPartitionsRDD[5] at filter at RDD-A.scala:36 []
 |   ShuffledRDD[4] at aggregateByKey at RDD-A.scala:34 []
 +-(40) MapPartitionsRDD[3] at map at RDD-A.scala:33 []
    |   MapPartitionsRDD[2] at map at RDD-A.scala:32 []
    |   /Users/sunho/Dropbox/Boston/CS6240/twitter-dataset/data/ MapPartitionsRDD[1] at textFile at RDD-A.scala:25 []
    |   /Users/sunho/Dropbox/Boston/CS6240/twitter-dataset/data/ HadoopRDD[0] at textFile at RDD-A.scala:25 []
```

```
def aggregateByKey[U](zeroValue: U, numPartitions: Int)(seqOp: (U, V) ⇒ U, combOp: (U,
    U) ⇒ U)(implicit arg0: ClassTag[U]): RDD[(K, U)]
```

Aggregate the values of each key, using given combine functions and a neutral "zero value". This function can return a different result type, U, than the type of the values in this RDD, V. Thus, we need one operation for merging a V into a U and one operation for merging two U's, as in scala.TraversableOnce. The former operation is used for merging values within a partition, and the latter is used for merging values between partitions. To avoid memory allocation, both of these functions are allowed to modify and return their first argument instead of creating a new U.

- 'aggregateByKey()' use the given combine functions to aggregate both partition and the value pairs, but it will not cause shuffle.
- Ref: Apache Spark ScalaDoc aggregateByKey()

e. DSET

```
== Physical Plan ==
*(2) Project [_1#3, sum(_2)#11L]
+- *(2) Filter (isnotnull(sum(cast(_2#4 as bigint))#16L) AND ((sum(cast(_2#4 as bigint))#16L % 100) = 0))
   +- *(2) HashAggregate(keys=[_1#3], functions=[sum(cast(_2#4 as bigint))])
      +- Exchange hashpartitioning(_1#3, 200), ENSURE_REQUIREMENTS, [id=#72]
         +- *(1) HashAggregate(keys=[_1#3], functions=[partial_sum(cast(_2#4 as bigint))])
            +- *(1) SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, kn
ownnotnull(assertnotnull(input[0, scala.Tuple2, true])._1, true, false) AS _1#3, knownnotnull(assertnotnull(input[0, scala.Tupl
e2, true]))._2 AS _2#4]
               +- Scan[obj#2]
```

- Above is the physical plan we get by running 'counts.explain()' where variable count can be attained by following the steps on the DSET pseudo-code. The ' groupBy(col)' doesn't combine, but instead reshuffle the data. Afterward, 'agg(col)' and 'sum(col)' aggregate the data.
- Ref: Apache Spark ScalaDoc groupBy()

4. Pseudo-code for Triangle-counting Programs

a. RS-R

```
TRIANGLE-COUNT-RS-RDD


val textFile <- read input csv file

// create (X -> Y) RDD
val XtoYRDD =
  textFile
  .map(edge => edge.split(","))
    // split edge to get nodes
  .map(nodes => (nodes(0).toInt, nodes(1).toInt))
    // map nodes to (followerID, followedID)
  .filter{case (from, to) => from < MAX_VALUE && to < MAX_VALUE
    // only get node IDs less than user-defined max value

// create (Y -> Z) RDD
val YtoZRDD =
  XtoYRDD
  .map{case (from, to) => (to, from)} // swap each node's position

// create (Z -> X) RDD
// this represents candidates for valid closing edges to form triangle
val ZtoXRDD = XtoYRDD
            .join(YtoZRDD) // by joining we acquire (X, Z) for each key Y
            .map{case (_, (from, to)) => (from, to)} // (X, Z) -> (Z, X)

// valid closing edges which form triangle with given nodes
// check equality between the edges and possible closing edges
val closingEdges =
  XtoYRDD // contains all edges (can also use YtoZRDD or any edges RDD)
  .join(ZtoXRDD) // ensure followerID equality
  .filter{case (_, (to1, to2)) => to1 == to2} // ensure followedID equality
  .map{case (from, (to1, _)) => (from, to1)} // get closing edge

val triangleCount = closingEdges.count() / 3 // remove duplicate counts

output <- triangleCount
```

b. RS-D

```
TRIANGLE-COUNT-RS-DATAFRAME


val textFile <- read input csv file

// create edge RDD
val edgeRDD =
  textFile
  .map(edge => edge.split(",")) // split edge to get nodes
  .map(nodes => (nodes(0).toInt, nodes(1).toInt))
    // map nodes to (followerID, followedID)
  .filter{case (from, to) => from < MAX_VALUE && to < MAX_VALUE}
    // only get node IDs less than user-defined max value

// convert edge RDD to (X->Y) DataFrame
val XtoYDF = edgeRDD.toDF("X", "Y")

// construct (Y -> Z) DataFrame
val YtoZDF = XtoYDF.toDF("Y", "X").withColumnRenamed("X", "Z")

// construct (Z -> X) DataFrame
// this represents candidates for valid closing edges to form triangle
val ZtoXDF =
  XtoYDF
  .join(YtoZDF, XtoYDF("Y") === YtoZDF("Y") && XtoYDF("X") =!= YtoZDF("Z"))
  .select("Z", "X")

// valid closing edges which form triangle with given nodes
val closingEdgeDF =
  XtoYDF
  .withColumnRenamed("Y", "Z")
  .as("XtoZ")
  .join(ZtoXDF.as("ZtoX"), $"XtoZ.X" === $"ZtoX.Z" && $"XtoZ.Z" === $"ZtoX.X")
    // ensure followerID & followedID equality

val closingEdgeCount = closingEdgeDF.count()

val triangleCount = closingEdgeCount / 3 // remove duplicate counts

output <- triangleCount
```

c. Rep-R

```
TRIANGLE-COUNT-REP-RDD


val textFile <- read input csv file

// create edge RDD
val edges =
  textFile
    .map(edge => edge.split(",")) // split edge to get nodes
    .map(nodes => (nodes(0).toInt, nodes(1).toInt))
     // map nodes to (followerID, followedID)
    .filter{case (from, to) => from < MAX_VALUE && to < MAX_VALUE}
     // only get node IDs less than custom set max value

// create edge map which has set of followedIDs for each followerID
val edgeMap =
  edges
  .map{case (from, to) => (from, Set(to))}
  .reduceByKey(_ ++ _) // reduce by followerID to get set of followedIDs
  .collectAsMap()

// create broadcast variable map
val broadcastMap = sc.broadcast( edgeMap )

// calculate triangle count
val triangleCount =
  edges.mapPartitions(iter => {
        iter.flatMap {
          case (nodeX, nodeY) => broadcastMap.value.get(nodeY).map(
            // for each Y followed by X, get set of IDs Y is following
            setY => setY.foreach(nodeZ => broadcastMap.value.get(nodeZ).foreach(
              // for each Z followed by Y, get set of IDs Z is following
              setZ => if (setZ.contains(nodeX)) {
                // if Z follows X, triangle has been formed
                accumulator.add(1) // increment the accumulator
              }
            )
          )
        )
      }
    }
  ).collect()

output <- accumulator.value / 3 // remove duplicate counts
```

d. Rep-D

```
TRIANGLE-COUNT-REP-DATAFRAME


val textFile <- read input csv file

// create edge RDD
val edgeRDD = textFile
            .map(edge => edge.split(",")) // split edge to get nodes
            .map(nodes => (nodes(0).toInt, nodes(1).toInt))
              // map nodes to (followerID, followedID)
            .filter{case (from, to) => from < MAX_VALUE && to < MAX_VALUE}
              // only get node IDs less than custom set max value

// convert edge RDD to (X->Y) DataFrame
val XtoYDF = edgeRDD.toDF("X", "Y")

// construct (Y -> Z) DataFrame
val YtoZDF = XtoYDF.toDF("Y", "X").withColumnRenamed("X", "Z")

// use broadcast join to construct (Z -> X) DataFrame
val ZtoXDF =
  XtoYDF
  .join(broadcast(YtoZDF),
    XtoYDF("Y") <=> YtoZDF("Y") && XtoYDF("X") =!= YtoZDF("Z"))
    // join on node Y where X is not equal to Z (filter out unncessary Path2)
  .select("Z", "X")

// use broadcast join to get all valid closing edges
val closingEdgeDF =
  XtoYDF
  .withColumnRenamed("Y", "Z")
  .as("XtoZ")
  .join(broadcast(ZtoXDF.as("ZtoX")),
    $"XtoZ.X" <=> $"ZtoX.Z" && $"XtoZ.Z" === $"ZtoX.X")
      // ensure followerID & followedID equality

val closingEdgeCount = closingEdgeDF.count()

val triangleCount = closingEdgeCount / 3 // remove duplicate counts

output <- triangleCount
```

5. Source code for Triangle-counting Programs

- RS-R: https://github.com/CS6240/hw-3-jill666666/blob/master/src/main/scala/tc/RS-R.scala
- RS-D: https://github.com/CS6240/hw-3-jill666666/blob/master/src/main/scala/tc/RS-D.scala
- Rep-R: https://github.com/CS6240/hw-3-jill666666/blob/master/src/main/scala/tc/Rep-R.scala
- Rep-D: https://github.com/CS6240/hw-3-jill666666/blob/master/src/main/scala/tc/Rep-D.scala

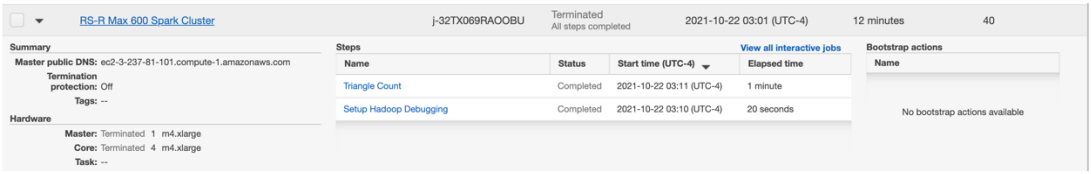
6. Run Triangle-counting Programs on EMR (1 Master & 4 Workers)

* The MAX_VALUE values from HW2 were 600, 2,400, and 6,000, but after receiving a feedback that the running time is too short, the values for HW3 have been updated to 600, 2,400, and 42,000. We also use MAX_VALUE of 12,000 to compare between the Spark and MapReduce implementations, which will be discussed in Question 8.

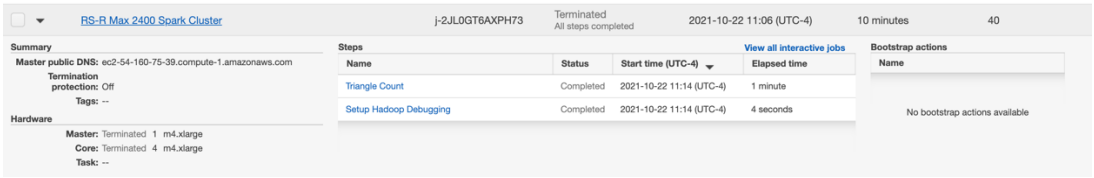
** Due to out-of-memory error, for Rep-D, the greatest MAX_VALUE has been set to 6,000, instead of 42,000 (Also tested with values 28,000 and 12,000, but same errors have been occurred).

a. RS-R

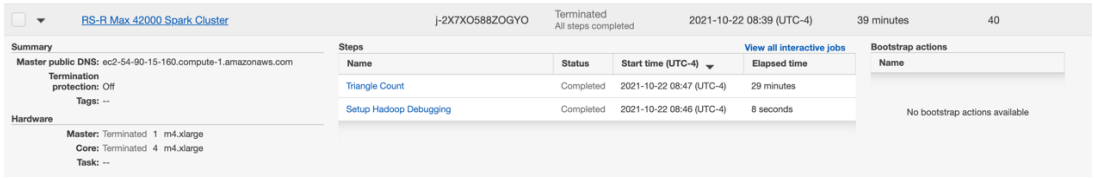
- MAX: 600 / Running Time: 1 min 20 seconds / Triangle Count: 572 / Machine Type: 1 Master (m4.xlarge) & 4 Workers (m4.xlarge)



- MAX: 2,400 / Running Time: 1 min 4 seconds / Triangle Count: 8,798 / Machine Type: 1 Master (m4.xlarge) & 4 Workers (m4.xlarge)

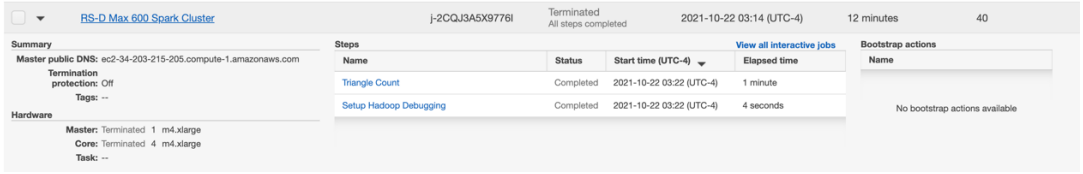


- MAX: 42,000 / Running Time: 29 mins 8 seconds / Triangle Count: 4,912,313 / Machine Type: 1 Master (m4.xlarge) & 4 Workers (m4.xlarge)

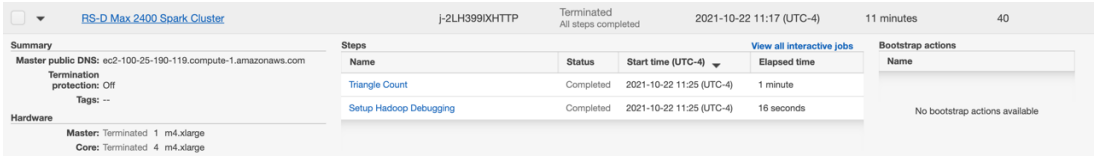


b. RS-D

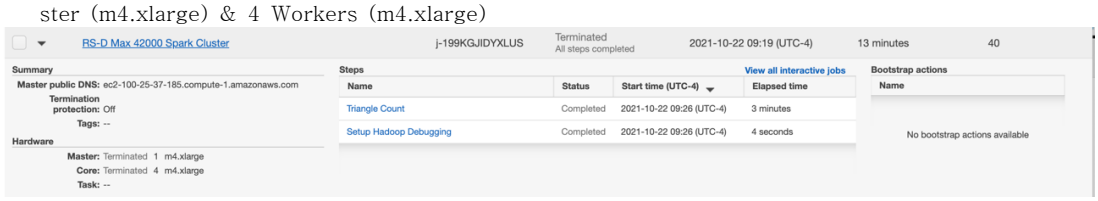
- MAX: 600 / Running Time: 1 min 4 seconds / Triangle Count: 572 / Machine Type: 1 Master (m4.xlarge) & 4 Workers (m4.xlarge)



- MAX: 2,400 / Running Time: 1 min 16 seconds / Triangle Count: 8,798 / Machine Type: 1 Master (m4.xlarge) & 4 Workers (m4.xlarge)

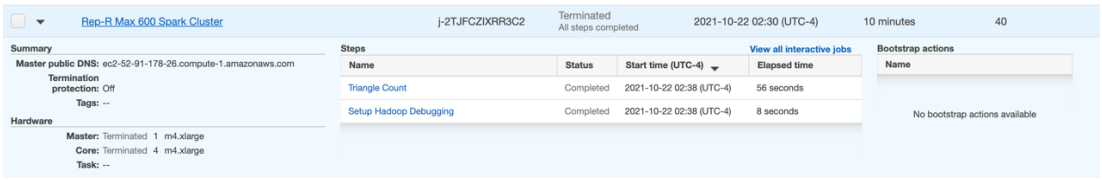


- MAX: 42,000 / Running Time: 3 min 4 seconds / Triangle Count: 4,912,313 / Machine Type: 1 Master (m4.xlarge) & 4 Workers (m4.xlarge)

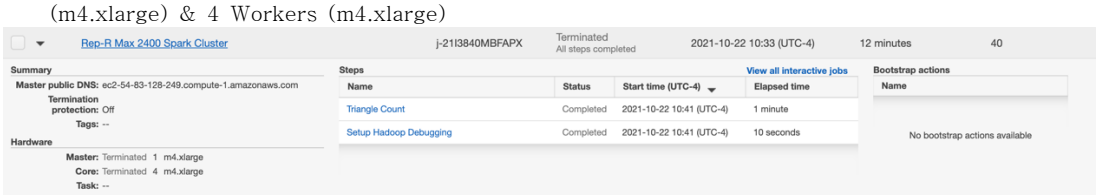


c. Rep-R

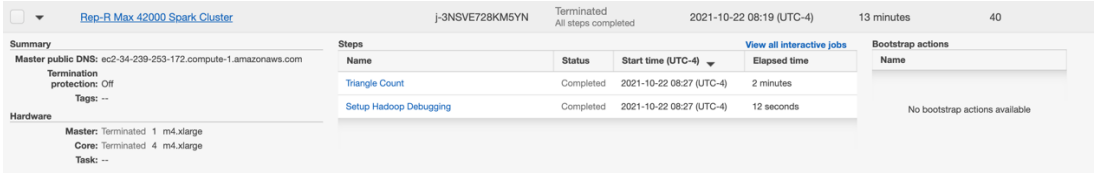
- MAX: 600 / Running Time: 1 min 4 seconds / Triangle Count: 572 / Machine Type: 1 Master (m4.xlarge) & 4 Workers (m4.xlarge)



- MAX: 2,400 / Running Time: 1 min 10 seconds / Triangle Count: 8,798 / Machine Type: 1 Master (m4.xlarge) & 4 Workers (m4.xlarge)

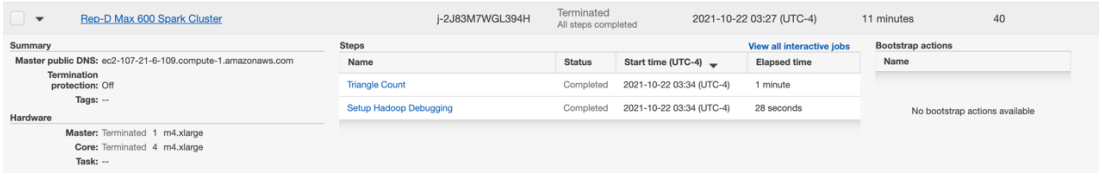


- MAX: 42,000 / Running Time: 2 mins 12 seconds / Triangle Count: 4,912,313 / Machine Type: 1 Master (m4.xlarge) & 4 Workers (m4.xlarge)

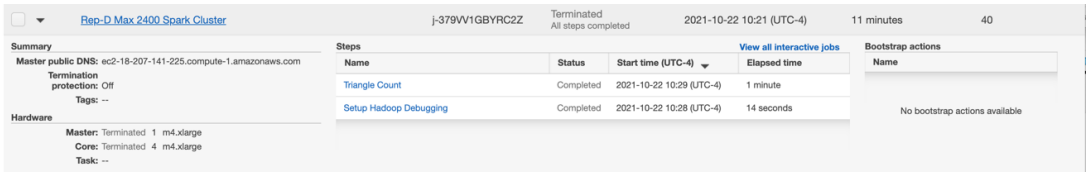


d. Rep-D

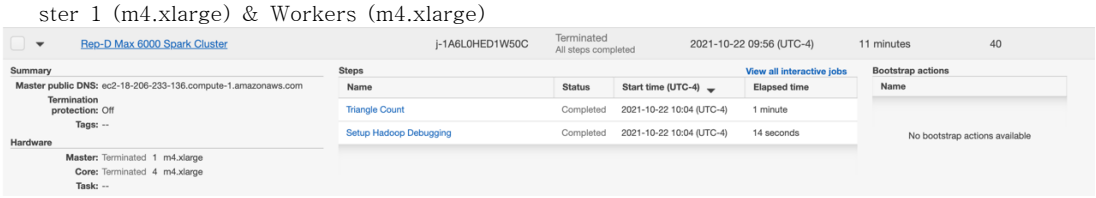
- MAX: 600 / Running Time: 1 min 28 seconds / Triangle Count: 572 / Machine Type: 1 Master (m4.xlarge) & 4 Workers (m4.xlarge)



- MAX: 2,400 / Running Time: 1 min 14 seconds / Triangle Count: 8,798 / Machine Type: 1 Master 1 (m4.xlarge) & 4 Workers (m4.xlarge)



- MAX: 6,000** / Running Time: 1 min 14 seconds / Triangle Count: 131,654** / Machine Type: Master 1 (m4.xlarge) & Workers (m4.xlarge)



7. EMR Output File std-err (1 Master & 4 Workers)

- RS-R: https://github.com/CS6240/hw-3-jill666666/tree/master/aws-outputs/RS-R/1M4W
- RS-D: https://github.com/CS6240/hw-3-jill666666/tree/master/aws-outputs/RS-D/1M4W
- Rep-R: https://github.com/CS6240/hw-3-jill666666/tree/master/aws-outputs/Rep-R/1M4W
- Rep-D: https://github.com/CS6240/hw-3-jill666666/tree/master/aws-outputs/Rep-D/1M4W

8. Run Triangle-counting Programs on EMR (1 Master & 8 Workers)

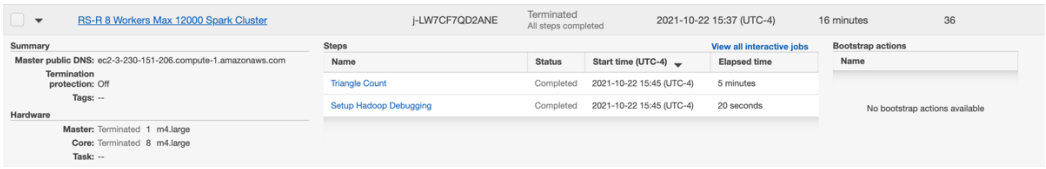
* The MAX_VALUE values from HW2 were 600, 2,400, and 6,000, but after receiving a feedback that the running time is too short, the values for HW3 have been updated to 600, 2,400, and 42,000. We also use MAX_VALUE of 12,000 to compare between the Spark and MapReduce implementations.

** Every implementations **except** Spark Rep-D has the same setup, MAX_VALUE and machine type, as well as triangle count result. As mentioned in the Question 6, due to out-of-memory error, Rep-D program has been ran using the MAX_VALUE of 6,000. The details including the comparison with the MapReduce are shown below.

1) Spark (HW3)

a. RS-R

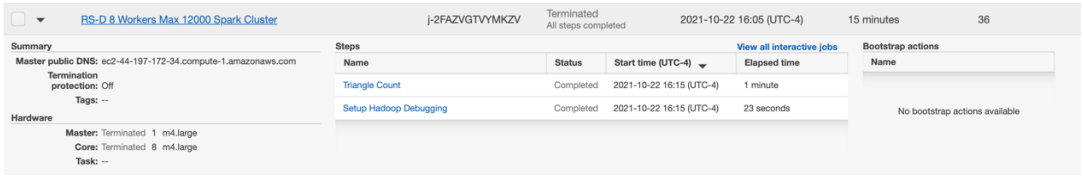
- MAX: 12,000
- Running Time: 5 mins 20 seconds
- Triangle Count: 856,482
- Machine Type: 1 Master (m4.large) & 8 Workers (m4.large)



b. RS-D

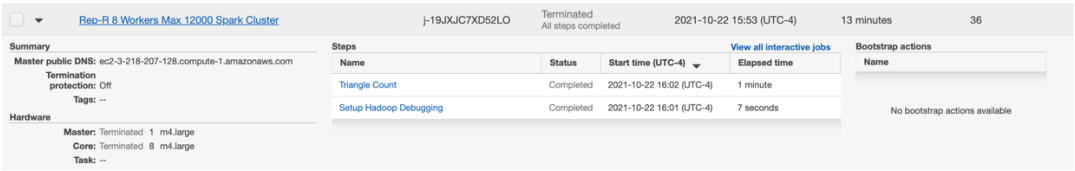
- MAX: 12,000
- Running Time: 1 min 23 seconds
- Triangle Count: 856,482

- Machine Type: 1 Master (m4.large) & 8 Workers (m4.large)



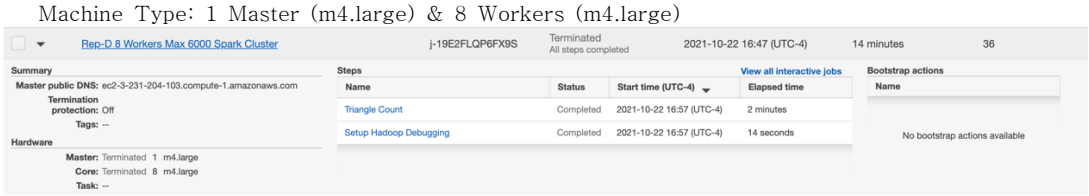
c. Rep-R

- MAX: 12,000
- Running Time: 1 min 7 seconds
- Triangle Count: 856,482
- Machine Type: 1 Master (m4.large) & 8 Workers (m4.large)



d. Rep-D*

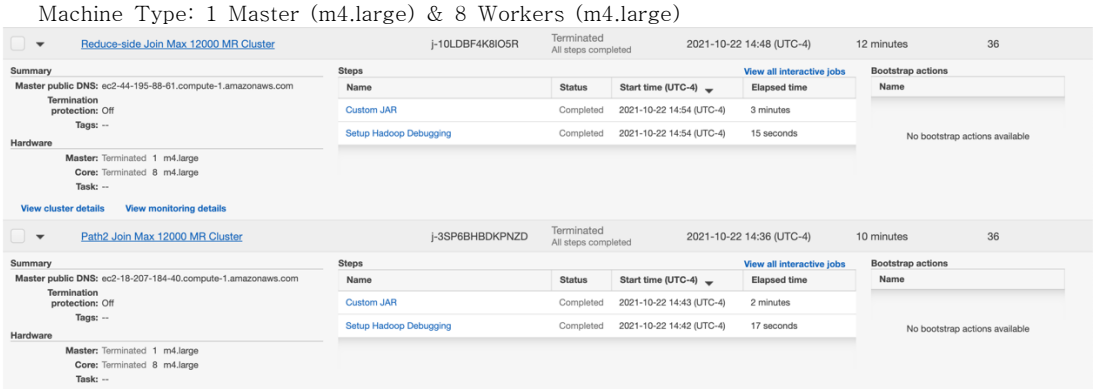
- MAX: 6,000*
- Running Time: 2 mins 14 seconds
- Triangle Count: 131,654*
- Machine Type: 1 Master (m4.large) & 8 Workers (m4.large)



2) MapReduce (HW2)

a. Reduce-side Join

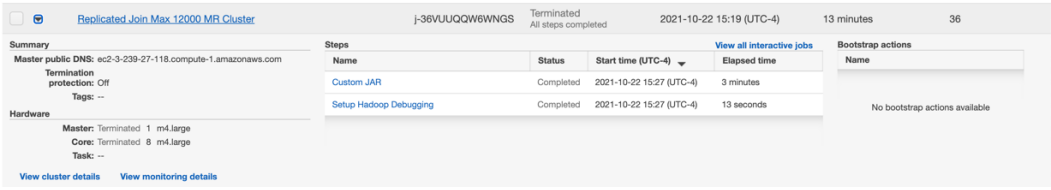
- MAX: 12,000
- Total Running Time: Approximately 5 mins 30 seconds
- Triangle Count: 856,482
- Machine Type: 1 Master (m4.large) & 8 Workers (m4.large)



- 'Path2 Join Max 12000 MR Cluster' stores output Path2 in the S3 bucket, and 'Reduce-side Join Max 12000 MR Cluster' take it as an input and calculates tr iangle count.

b. Replicated Join

- MAX: 12,000
- Running Time: 3 mins 13 seconds
- Triangle Count: 856,482
- Machine Type: 1 Master (m4.large) & 8 Workers (m4.large)



9. EMR Output File std-err (1 Master & 8 Workers)

- RS-R: https://github.com/CS6240/hw-3-jill666666/tree/master/aws-outputs/RS-R/1M8W
- RS-D: https://github.com/CS6240/hw-3-jill666666/tree/master/aws-outputs/RS-D/1M8W
- Rep-R: https://github.com/CS6240/hw-3-jill666666/tree/master/aws-outputs/Rep-R/1M8W
- Rep-D: https://github.com/CS6240/hw-3-jill666666/tree/master/aws-outputs/Rep-D/1M8W