Class – CS6240 Fall-2018 Sec 2
HW – 1
Name- Mustafa Kapadia
Github - https://github.ccs.neu.edu/mustafa8895/CS6240/HW1

Map-Reduce Implementation (20 points)

```
Map(Line of Text L)
        If Line contains ","
                Node = Line.substring ( indexOf (",") + 1)
                Emit(Node, 1)
        Else
                Emit(Line, 0)
```

The mapper checks for the presence of a "," to determine whether the input comes from the nodes.csv file or the edges.csv file. Based on this it emits either 0 or 1 for the key.

```
Reduce(Key K, Array of values V)
        For all val v in Array V
                Count += v
        Emit(K, Count)
```

The reducer simply adds the values for every key.

Spark Scala Implementation (30 points total)

```
Input = Text files ( nodes.csv, edges.csv)
val counts = map ( Line L from input => mapBasedOnFile ( L ) )
                .reduceByKey ( (x, y) = x+y)

mapBasedOnFile(args: String) : (String, Int)
        if  args contains ","
        emit (args.substring(args.indexOf(",")+1), 1)
        else
        emit (args, 0)
```

- The Spark implementation uses a helper function for the map task(mapBasedOnFile), which checks for the occurrence of "," to determine whether to emit a 1 or a 0.

- The reduceByKey adds all the values emitted the map task for each key.

Scala function used to report execution details – toDebugString

(43) ShuffledRDD[3] at reduceByKey at twitterFollowers.scala:27 []
+-(43) MapPartitionsRDD[2] at map at twitterFollowers.scala:26 []
| input MapPartitionsRDD[1] at textFile at twitterFollowers.scala:25 []
| input HadoopRDD[0] at textFile at twitterFollowers.scala:25 []

The toDebugString command reports the order of execution, the RDDs formed and the input RDD for each stage.

The value within round brackets states the level of parallelism, which in this case is 43.
The indentation signifies a shuffling phase, which occurs between the mapPartitions and reduceByKey calls. The sequence of execution is as follows:

1. The Input file is loaded into a HadoopRDD[0]
2. The data is partitioned into a MapPartitionsRDD[1] which serves as inputs for the mappers
3. The map task transforms the MapPartitionsRDD into a MapRDD[2] which contains key value pair.
4. The Shuffle phase groups the data by key across partitions and creates the ShuffleRDD[3] which serves as the input for the reduce phase.
5. The Reduce call aggregates the values for each key and emits the output

There is no aggregation before shuffling. The aggregation takes place in the reduce phase.

Time Measurements

- MR 6 machines:
  - Run 1 - 120 sec
  - Run 2 – 129 sec

- MR 11 machines:
  - Run 1 – 77 sec
  - Run 2 – 91 sec

$$\text{Speedup} = \frac{Run\ time\ for\ 6\ machines}{Run\ time\ for\ 11\ machines} = \frac{120}{77} = 1.558$$

- Spark 6 machines:
  - Run 1 - 113 sec
  - Run 2 – 115 sec

- Spark 11 machines:
  - Run 1 – 94 sec

- Run 2 - 94 sec

$$\text{Speedup} = \frac{Run\ time\ for\ 6\ machines}{Run\ time\ for\ 11\ machines} = \frac{113}{94} = 1.20$$

Data Transferred:

For Cluster of size 6 (MR):

- FILE: Number of bytes read=154718884
- Map input records=96648656
- Map output records=96648656
- Combine input records=96648656
- Combine output records=26679393
- Reduce input records=26679393
- Reduce output records=11316811

For Cluster of size 11 (MR):

- FILE: Number of bytes read=160820380
- Map input records=96648656
- Map output records=96648656
- Combine input records=96648656
- Combine output records=26679393
- Reduce input records=26679393
- Reduce output records=11316811

Optimal speedup for almost doubling the machines should be close to 2. The calculated speedups however are 1.558 for MR and 1.20 for Spark which are not close to optimal. The following reasons come to mind:

- An important point to note is that for MR even though the number of Map tasks are close to the same (around 20), the number of reduce tasks double (9 to 19) when the machines increase from 6 to 11.
- Whereas for Spark, the number of reduce tasks for both, 6 machines and 11 machines remain the same (21).
- MR seems to be using the additional machines more efficiently than Spark which is supported by the fact that it has a higher speedup.
- The map and reduce tasks respectively are executed in parallel, the reduce tasks cannot be executed before all the map tasks are completed. This is a part of the code that is inherently sequential and which is why the speedup, according to Amdahl's law is not close to optimal.

- It is also important to remember that, as the number of machines increase, the control messages between them increase too. And so does the time it takes for shuffling and fetching data for the reducer. This seems the reason for the speedup not being near optimal.
