CS6240 Homework4

Name: Wenqing Xu

Week 1: PageRank in Spark

Pseudo-code

```
//store the initial ranks in list called ranks and linked edges in list called edges
val graphRDD = spark.sparkContext.parallelize(edges).cache()
var rankRDD = spark.sparkContext.parallelize(ranks)

val iters = 10

    for (i <- 1 to iters) {
      val newRDD = graphRDD.join(rankRDD)
       val tempRDD = newRDD.flatMap(joinPair =>
        if (joinPair._1.toInt % k == 1)
          List((joinPair._1, 0.0), joinPair._2) // the first one of each k nodes get 0.0
pagerank
        else
          List(joinPair._2))

      val temp2RDD = tempRDD.groupByKey().map(x => {
              (x._1, x._2.sum)})

      val delta = temp2RDD.lookup("0").head
          //add the mass loss to every node
      rankRDD = temp2RDD.map(v => {
        if (v._1.equals("0")) {
          (v._1, v._2)
        } else {
          (v._1, alpha*initialPR + (1-alpha) * (v._2 + delta * initialPR))
        }
```

```
        })
logger.warn("PageRank sum for iteration " + i + " : " + rankRDD.filter(_._1 !=
"0").map(_._2).sum())
                println("PageRank sum for iteration " + i + " : " + rankRDD.filter(_._1 !=
"0").map(_._2).sum())
        }
```

First, after pass the PageRank, the action lookup() triggered the above transformations like join() and flatMap()
Then, following transformations such as groupByKey and map are triggered by the action sum() to add up the total PageRank in the network.

The final PageRanks of the top k vertices are reported in the appendix.

The lineage for RDD PagRank in the first three loop:

first loop:

```
(4) MapPartitionsRDD[10] at map at rdd.scala:55 []
 |  MapPartitionsRDD[7] at map at rdd.scala:49 []
 |  ShuffledRDD[6] at groupByKey at rdd.scala:49 []
 +-(4) MapPartitionsRDD[5] at flatMap at rdd.scala:43 []
    |  MapPartitionsRDD[4] at join at rdd.scala:41 []
    |  MapPartitionsRDD[3] at join at rdd.scala:41 []
    |  CoGroupedRDD[2] at join at rdd.scala:41 []
    +-(4) ParallelCollectionRDD[0] at parallelize at rdd.scala:34 []
    |  |  CachedPartitions: 4; MemorySize: 1202.0 KB;
ExternalBlockStoreSize: 0.0 B; DiskSize: 0.0 B
    +-(4) ParallelCollectionRDD[1] at parallelize at rdd.scala:35 []
```

Second loop:

```
(4) MapPartitionsRDD[23] at map at rdd.scala:55 []
 |  MapPartitionsRDD[20] at map at rdd.scala:49 []
 |  ShuffledRDD[19] at groupByKey at rdd.scala:49 []
 +-(4) MapPartitionsRDD[18] at flatMap at rdd.scala:43 []
```

```
    |  MapPartitionsRDD[17] at join at rdd.scala:41 []
    |  MapPartitionsRDD[16] at join at rdd.scala:41 []
    |  CoGroupedRDD[15] at join at rdd.scala:41 []
    +-(4) ParallelCollectionRDD[0] at parallelize at rdd.scala:34 []
    |      |                CachedPartitions: 4; MemorySize: 1202.0 KB;
ExternalBlockStoreSize: 0.0 B; DiskSize: 0.0 B
    +-(4) MapPartitionsRDD[10] at map at rdd.scala:55 []
        |  MapPartitionsRDD[7] at map at rdd.scala:49 []
        |  ShuffledRDD[6] at groupByKey at rdd.scala:49 []
        +-(4) MapPartitionsRDD[5] at flatMap at rdd.scala:43 []
            |  MapPartitionsRDD[4] at join at rdd.scala:41 []
            |  MapPartitionsRDD[3] at join at rdd.scala:41 []
            |  CoGroupedRDD[2] at join at rdd.scala:41 []
            +-(4) ParallelCollectionRDD[0] at parallelize at rdd.scala:34 []
            |      |                CachedPartitions: 4; MemorySize: 1202.0 KB;
ExternalBlockStoreSize: 0.0 B; DiskSize: 0.0 B
            +-(4) ParallelCollectionRDD[1] at parallelize at rdd.scala:35 []
```

Third loop:

```
(4) MapPartitionsRDD[36] at map at rdd.scala:55 []
 |  MapPartitionsRDD[33] at map at rdd.scala:49 []
 |  ShuffledRDD[32] at groupByKey at rdd.scala:49 []
 +-(4) MapPartitionsRDD[31] at flatMap at rdd.scala:43 []
    |  MapPartitionsRDD[30] at join at rdd.scala:41 []
    |  MapPartitionsRDD[29] at join at rdd.scala:41 []
    |  CoGroupedRDD[28] at join at rdd.scala:41 []
    +-(4) ParallelCollectionRDD[0] at parallelize at rdd.scala:34 []
    |  |          CachedPartitions: 4; MemorySize: 1202.0 KB; ExternalBlockStoreSize: 0.0 B;
DiskSize: 0.0 B
    +-(4) MapPartitionsRDD[23] at map at rdd.scala:55 []
        |  MapPartitionsRDD[20] at map at rdd.scala:49 []
        |  ShuffledRDD[19] at groupByKey at rdd.scala:49 []
        +-(4) MapPartitionsRDD[18] at flatMap at rdd.scala:43 []
```

| MapPartitionsRDD[17] at join at rdd.scala:41 []

| MapPartitionsRDD[16] at join at rdd.scala:41 []

| CoGroupedRDD[15] at join at rdd.scala:41 []

+-(4) ParallelCollectionRDD[0] at parallelize at rdd.scala:34 []

| | CachedPartitions: 4; MemorySize: 1202.0 KB; ExternalBlockStoreSize: 0.0 B; DiskSize: 0.0 B

+-(4) MapPartitionsRDD[10] at map at rdd.scala:55 []

| MapPartitionsRDD[7] at map at rdd.scala:49 []

| ShuffledRDD[6] at groupByKey at rdd.scala:49 []

+-(4) MapPartitionsRDD[5] at flatMap at rdd.scala:43 []

| MapPartitionsRDD[4] at join at rdd.scala:41 []

| MapPartitionsRDD[3] at join at rdd.scala:41 []

| CoGroupedRDD[2] at join at rdd.scala:41 []

+-(4) ParallelCollectionRDD[0] at parallelize at rdd.scala:34 []

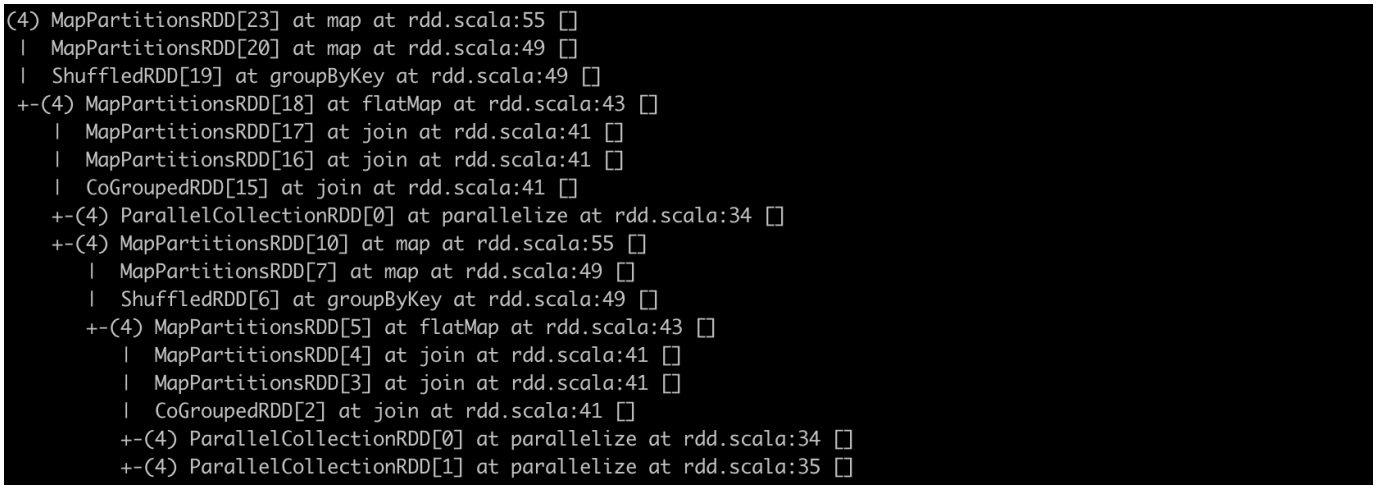
| | CachedPartitions: 4; MemorySize: 1202.0 KB; ExternalBlockStoreSize: 0.0 B; DiskSize: 0.0 B

+-(4) ParallelCollectionRDD[1] at parallelize at rdd.scala:35 []

From the three lineage strings, we can see that the edge RDD joins with the RankRDD that is created in the last iteration. The edge RDD is saved in the cache so it do not need to recomputed the RDD. And the flatMap() create new RDD to store the passed Pagerank to the next link (0 for the first page in each k paged). From the lineage we can clearly see that the operation contains partitions and shuffled the RDD. The sum() operation triggered the map and groupByKey transformations to work and print the total PageRank.

Is Spark smart enough to figure out that it can re-use RDDs computed for an earlier action?

The answer is Yes. If I delete the cache() in my program, the lineage will be like this in the second loop:



If we compare it with the original one, we will find out that there is few differences except there is a line "CachedPartitions: 4; MemorySize: 1202.0 KB; ExternalBlockStoreSize: 0.0 B; DiskSize: 0.0 B" in the original one with "cache". However, the ParallelCollectionRDD[0] shows up multiple times in this lineage. It means that Spark can figure out that it can be re-used.

Actually, I read from the online material that *"Spark automatically persists some intermediate data in shuffle operations (reduceByKey), even without users calling persist. This is done to avoid recomputing the entire input if a node fails during the shuffle. But best practice recommends users call persist on the resulting RDD if reuse is required".* The next operation that uses the edgeRDD in my program is join() and it is a shuffle operation. So the Spark automatically persists the RDD in order to reuse it.

How do persist() and cache() change this behavior?

The cache() and persist() are called when the Spark can not automatically do such things for you. By calling them, you can store an RDD in cache when its size is under the memory limitation. Spark's cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it. In addition, you can change the storage level by calling persist().

week 2: PageRank in MapReduce

pseudo-code

```
class Mapper
    nodemapping = new HashMap<Stirng, Set<String>>
    method setup(Context context)
        read File rdr from distributed cache
        while ( (line = rdr. readline() ) != null) do
            String[] node = split(value, “,”);
                x = node[0]
                y = node[1]
                if x in nodemapping do
                    nodemapping[x] = y
                else do
                    nodemapping[y] = new Hashset<>.add(x)
// I use Repjoin to join the edge and PageRanks so the first step is to store the Pagerank in cache
    method map(Object Key, Text value)
        String node[] = split(value,”,”)
        x = node[0]
        y = node[1]
        if ( x mod k == 1) do
            EMIT( x , “0.0”)
 // just like the spark program, the first one of each k pages will have 0.0 //pagerank after passing the pagerank to the next page
        Set<String> pageranks = nodeMapping.get<x>
        for each pagerank in pageranks do
            EMIT(y, pagerank)
//The Mapper Class actually does a replicated join of the pageranks and edges, the calculation of pagerank will be performed in the reducer class.
Class Reducer
    map = new HashMap<String, Double>
```

```
        method reduce(Text key, Iterable<Text> values)
            double sum = 0.0
            for each Text value in values do
                sum += value.todouble()
            map[key] = sum
        method cleanup(Context context)
            double delta = map["0"]
            for each entry in map do
                String newkey = entry.getkey()
                Double newval = entry.getvalue()
                Double newpagerank = alpha * initialPR + (1 - alpha) * (newval +
delta * initialPR)
//adding back the PageRank mass loss to each page, alpha is set to 0 for
simplicity.
                EMIT(newkey, newpagerank)

void main
    cache_path = initial_pagerank
    for i= 1 to 10 do
        Set the Mapper number to 20
        execute Mapper(cache_path, input_path: edges.txt)
        execute Reducer(output_path: pagerank_round_i)
        cache_path = pagerank_round_i
```

My program uses an iteration to execute the Mapper and Reducer class for 10 times. Store the pagerank file generated by each loop in cache and use it in the replicated join.

As for the dangling-page, I link them to a page 0 and pass the PageRank to it. Just like the spark program, the pagerank mass loss in page 0 is calculated in reducer and is given back to each page.

Time for each cluster (in controller log)

Small cluster: 946 sec

Large cluster: 720 sec

Special clarification for output files and logs

1. The spark program doesn't generate output files; the result will be printed in console. The log is in the logs folder.

2. My MapReduce program actually has 10 output files, one for each loop. However, I only add head of the final one in the tar ball because it's the final outcome. The syslog and controller logs are in the log folder.

Appendix

(0,0.010448538066009918) (33,1.0470069929669521E-4) (66,1.0470069929669521E-4)

(1,1.5888125735610845E-5) (34,1.0470069929669521E-4) (67,1.0470069929669521E-4)

(2,2.9391220523344354E-5) (35,1.0470069929669521E-4) (68,1.0470069929669521E-4)

(3,4.086705694684288E-5) (36,1.0470069929669521E-4) (69,1.0470069929669521E-4)

(4,5.061974152456418E-5) (37,1.0470069929669521E-4) (70,1.0470069929669521E-4)

(5,5.890776462131795E-5) (38,1.0470069929669521E-4) (71,1.0470069929669521E-4)

(6,6.595084287305437E-5) (39,1.0470069929669521E-4) (72,1.0470069929669521E-4)

(7,7.193573524791715E-5) (40,1.0470069929669521E-4) (73,1.0470069929669521E-4)

(8,7.702118669812164E-5) (41,1.0470069929669521E-4) (74,1.0470069929669521E-4)

(9,8.134213026403417E-5) (42,1.0470069929669521E-4) (75,1.0470069929669521E-4)

(10,8.501325886262295E-5) (43,1.0470069929669521E-4) (76,1.0470069929669521E-4)

(11,1.0470069929669521E-4) (44,1.0470069929669521E-4) (77,1.0470069929669521E-4)

(12,1.0470069929669521E-4) (45,1.0470069929669521E-4) (78,1.0470069929669521E-4)

(13,1.0470069929669521E-4) (46,1.0470069929669521E-4) (79,1.0470069929669521E-4)

(14,1.0470069929669521E-4) (47,1.0470069929669521E-4) (80,1.0470069929669521E-4)

(15,1.0470069929669521E-4) (48,1.0470069929669521E-4) (81,1.0470069929669521E-4)

(16,1.0470069929669521E-4) (49,1.0470069929669521E-4) (82,1.0470069929669521E-4)

(17,1.0470069929669521E-4) (50,1.0470069929669521E-4) (83,1.0470069929669521E-4)

(18,1.0470069929669521E-4) (51,1.0470069929669521E-4) (84,1.0470069929669521E-4)

(19,1.0470069929669521E-4) (52,1.0470069929669521E-4) (85,1.0470069929669521E-4)

(20,1.0470069929669521E-4) (53,1.0470069929669521E-4) (86,1.0470069929669521E-4)

(21,1.0470069929669521E-4) (54,1.0470069929669521E-4) (87,1.0470069929669521E-4)

(22,1.0470069929669521E-4) (55,1.0470069929669521E-4) (88,1.0470069929669521E-4)

(23,1.0470069929669521E-4) (56,1.0470069929669521E-4) (89,1.0470069929669521E-4)

(24,1.0470069929669521E-4) (57,1.0470069929669521E-4) (90,1.0470069929669521E-4)

(25,1.0470069929669521E-4) (58,1.0470069929669521E-4) (91,1.0470069929669521E-4)

(26,1.0470069929669521E-4) (59,1.0470069929669521E-4) (92,1.0470069929669521E-4)

(27,1.0470069929669521E-4) (60,1.0470069929669521E-4) (93,1.0470069929669521E-4)

(28,1.0470069929669521E-4) (61,1.0470069929669521E-4) (94,1.0470069929669521E-4)

(29,1.0470069929669521E-4) (62,1.0470069929669521E-4) (95,1.0470069929669521E-4)

(30,1.0470069929669521E-4) (63,1.0470069929669521E-4) (96,1.0470069929669521E-4)

(31,1.0470069929669521E-4) (64,1.0470069929669521E-4) (97,1.0470069929669521E-4)

(32,1.0470069929669521E-4) (65,1.0470069929669521E-4) (98,1.0470069929669521E-4)

(99,1.0470069929669521E-4)

(100,1.0470069929669521E-4)