**CS-6240 HW-4 Report**

**Ajay Baban Kauthale(**ajayk@ccs.neu.edu**)**

## Program Discussion

Program consist of 3 main Stages

1. ### Parser Job

In this stage there is no shuffling necessary

```
public static void preprocessBZ2() {
        // create job by setting parser input, output, mapper class and another configuration
        Job job;
        // set number of reducers to single
        job.setNumReduceTasks(1);
        // run the job and wait for completion
boolean ok = job.waitForCompletion(true);
Counters counters = job.getCounters();
// update the page count with counter
pageCnt = counters.findCounter(PageRankCounter.PAGE_COUNTER).getValue();
}
```

Following class is used for parser mapping

```
class ParserMapper {
        // This reads records from BZ2 input files and emits records in following format
        // Z#A~B~C#PR_VALUE
        // where A,B,C are outlinks(adjacency list) of NODE Z
        // and, PR_VALUE is page rank value of the NODE Z
}
```

2. ### Page Rank Job

```
def pageRankJob() {
        // read the parser output
        For each line on parser output do
                // Split the line
                // put node <- outlinks in map
        // convert map to sequence
        val adjList: List[(String, List[String])] = map.toList
        // convert sequence to RDD
        val links = sc.parallelize(adjList).partitionBy(new HashPartitioner(1)).persist()
        // call getPageRank method 10 times
```

}

In pageRankJob method shuffling is not necessary because of narrow dependencies

```
[parent RDD partition] ---> [child RDD partition]
```

```
def getPageRanks() {
        // create flatmap(String: Node, Double: contributions) from joining links to pageranks
        // join links to page ranks
        val r = links.join(ranks)
        // calculate dangling node score and distribution bases on initial page ranks
        // calculate final page ranks using alpha and dangling node distribution
        var finalranks = contributions.reduceByKey((x, y) => x + y).mapValues(v => (0.15 / pageCnt) +
0.85 * (danglingScoreDistribution + v))
}
```

In getPageRanks both norrow and wide shuffling necessary
Join method widens the dependencies

```
                        ---> [child RDD partition 1]
[parent RDD partition] ---> [child RDD partition 2]
                        ---> [child RDD partition 3]
```

And reduceByKey narrows the dependencies

```
[parent RDD partition] ---> [child RDD partition]
```

### 3. TopK Job

```
def topKResults() {
        // sort the links in descending order by ranks
        var sortedRanks = ranks.sortBy(_._2, false, 1)
        // output the top 100 ranks
        sc.parallelize(sortedRanks.take(100)).partitionBy(new
HashPartitioner(1)).saveAsTextFile(args(1))
}
```

In TopK no shuffling is necessary

## Performance Comparison

| Cluster | pre-processing time | time to run ten iterations of PageRank | time to find the top-100 pages |
|---|---|---|---|
| 6 m4.large machines | 7 min | 20 min | 1 min |
| 11 m4.large machines | 7 min | 12 min 45 seconds | 1 min |

I was expecting same processing time since because it must be independent on number if machines since we do not use reducers which is right.

Also, time to run 10 iterations also makes sense since 11 machine cluster time will be always less than 6 machine cluster, but I was expecting time difference around 50% less for 11 machine cluster since cluster machines are doubled which is not the case.

Finally, for getting top 100 pages time for 11 machine cluster should be less than 6 machine cluster which is correct. Here the difference is almost 50% less for 11 machine cluster since cluster machines are doubled.