CS 494 - Cloud Data Center Systems

Homework 1

University of Illinois at Chicago

Montanari Claudio

Porter David

Shinde Komal

1 A simple Spark sorting application

In order to complete the part one of the homework we coded a quite simple program that takes advantage of the functionalities of the `pyspark.sql` library. Such library expose a familiar python object, a sort of RDD `dataframe`. Shortly, the application executes the following steps:

1. Set up a Spark Session using the appropriate config values:

```
spark = SparkSession.builder.master("spark://128.110.153.141:7077")
    .appName("homework 1 part 1")
    .config("spark.submit.deployMode", "cluster")
    .config("spark.driver.memory", "32g")
    .config("spark.executor.memory", "32g")
    .config("spark.executor.cores", "10")
    .config("spark.task.cpus", "1")
    .getOrCreate()
```

2. Load the data from HDFS into a dataframe:

```
data = spark.read.csv(input_path, header=True)
```

3. Sort the data using the "cca2" and "timestamp" columns:

```
ordered = logData.orderBy(["cca2", "timestamp"], ascending=[1,1])
```

4. Store back the data into HDFS:

```
ordered.write.csv(output_path, header=True)
```

The given dataset was quite small thus, no relevant performance evaluation from the Distributed System point of view can be done. In any case, it was a great starting point to start understanding the Spark programming model which is quite different with respect to the usual way of coding algorithms.

2 PageRank in Spark

Introduction:

PageRank is a popular link analysis algorithm that was developed by the founders of Google for determining the prestige of a website. Prestige of a website is defined by how many in-links (weighted by prestige of each in-link) point to that website. There are two primary functions of PageRank that we focused on for our implementation of a distributed PageRank:

1. Googles web crawler will crawl the internet and will create an adjacency matrix Aij of all the links on any given page. Since outlinks/page are much easier to find as a lonely web crawler, the matrix cells represent the 1/#outlinks for each outlink in Aj.
2. You take the transpose of that matrix which will give you the in-links for Ai represented as a fraction over the outlinks for each inlink source. Essentially, this is a probability distribution that will converge as the PageRank values get updated though each iteration, and the final rank is just the sum of these probabilities for Ai. Therefore our application will follow the same process.

This page rank algorithm requires a lot of memory when dealing with data corpus the size of the internet, so distributed computing techniques are required to successfully execute at scale. Map Reduce was presented as a way to execute data manipulation tasks in parallel by partitioning disk space across nodes, and furthermore, delegating execution of the tasks to nodes where the partitioned data resides. Spark is the leading framework for running these types of jobs, and to help Spark, Hadoop's file system is integrated so it will lend itself for reading and writing in a distributed environment.

Algorithm Steps:

- Read data into RDD
  $>>> lines = spark.read.text(input\_path).rdd.map(lambda\ r : r[0])$

  u'254913$\widehat{2}$56802', u'254913$\widehat{2}$56798'

- Map RDD to $->$ link, neighbor
- Group by $->$ since we need to reduce by a key resulting in an iterable with unique values, we need to do a shuffle $->$ requires groupBy which is a longer operation

  $>>> links = lines.map(lambda\ ids : parseNeighbors(ids)).distinct().groupByKey()$

  (u'378466',$<pyspark.resultiterable.ResultIterableobjectat0x10edb9310>$), (u'35540', $<pyspark.resultiterable.ResultIterableobjectat0x10edb90d0>$)

- This is where we set the initial ranks for all the keys
  $>>> ranks = links.map(lambda\ website : (website[0], 1.0))$
  $>>> ranks.take(3)$

(u'378466', 1.0), (u'35540', 1.0), (u'553863', 1.0)

- Pull together the entire RDD object required for updating respective rank values.
  $>>> contribs = links.join(ranks)$

  (u'378466',($< pyspark.resultiterable.ResultIterableobjectat0x10eddba90 >$, 1.0)), (u'35540', ($< pyspark.resultiterable.ResultIterableobjectat0x10eddba50 >$, 1.0))

- Computes the weighted rank for each neighbor the website points to as indicated in step 1 in the overview.

  $>>> contribs.flatMap(lambda\ id\_ids\_rank : computeContribs(id\_ids\_rank[1][0], id\_ids\_rank[1][1]))$

  (u'378760', 0.2), (u'380036', 0.2), (u'378494', 0.2), (u'378492', 0.2), (u'378761', 0.2), (u'184279', 0.1), (u'743', 0.1), (u'401873', 0.1), (u'927', 0.1), (u'768', 0.1), (u'184094', 0.1), (u'184332', 0.1), (u'438238', 0.1), (u'184142', 0.1), (u'33', 0.1), (u'265970', 0.3333333333333333)

- This step is equivalent to step 2 explained in the overview. Where the output here is the sum of the neighbor rank value collection which was provided through a "transpose" of the original RDD.

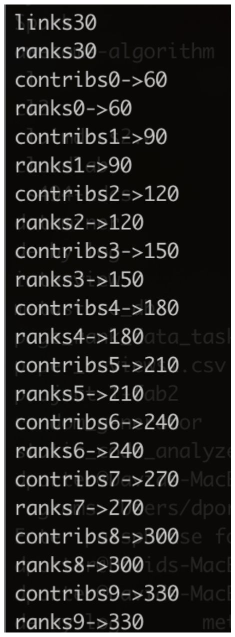
  $>>> ranks = contribs.reduceByKey(add).mapValues(lambda\ rank : rank * 0.85 + 0.15)$
  $>>> ranks.take(100)$

  (u'378466', 1.2379113142292488), (u'35540', 0.17428571428571427), (u'370255', 0.6964285714285714), (u'425876', 0.6924933862433862), (u'286891', 0.5345238095238095), (u'477259', 0.22727272727272727) (u'676957', 0.3625), (u'60375', 0.22727272727272727), (u'127473', 0.20672969966629587), (u'120128', 0.7409523809523809), (u'613260', 0.5854938271604938), (u'344692', 0.29166666666666663) (u'493657', 0.266812865497076), (u'300252', 0.6033333333333333), (u'123859', 0.373125), (u'235383', 5.302363408521304), (u'82991', 0.3113443548788376), (u'472331', 0.43333333333333335), (u'458339', 0.2950231481481481)...

This final step continued for 10 iterations until the ranks begin to converge.

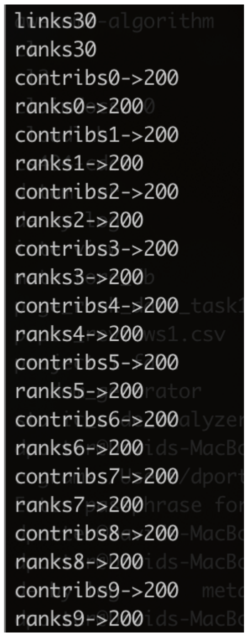
We tuned the page rank application according to our configuration by allocating static partitioning for the job. The cluster is configured to use 30 cores and 1 task per core = 30 partitions. With this in mind, we noticed setting first transformations to 30 partitions seemed to give the best performance. However, spark automatically will start incrementing partitions, and if we kept all partitions at 30, the app would run slower. If we demanded 30 partitions and somewhere in the middle of what spark automatically allocated 200, our results seemed to improve performance by 25% on the smaller data set. Running the larger data set over our tuned application demonstrated inconsistent performance. We noticed the larger data set was a sample of an internet graph, and the graph was loosely connected making the tuning steps difficult to assess. The spark application did not handle the killed worker very well since the job did not successfully complete.

Results for larger data set: These terminal images shows pagerank application's number of partitions for each shuffle as a result of our tuning. The rows below show the performance of these three along side of the app tuned with cache() and the app mimicking a worker failure.



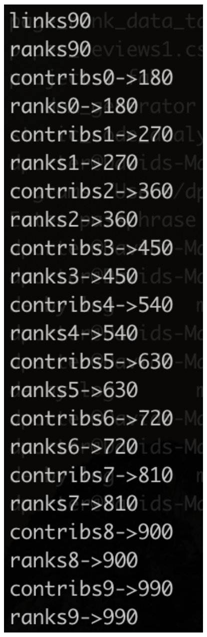
```
links30
ranks30
contribs0->60
ranks0->60
contribs1->90
ranks1->90
contribs2->120
ranks2->120
contribs3->150
ranks3->150
contribs4->180
ranks4->180
contribs5->210
ranks5->210
contribs6->240
ranks6->240
contribs7->270
ranks7->270
contribs8->300
ranks8->300
contribs9->330
ranks9->330
```

30par.png



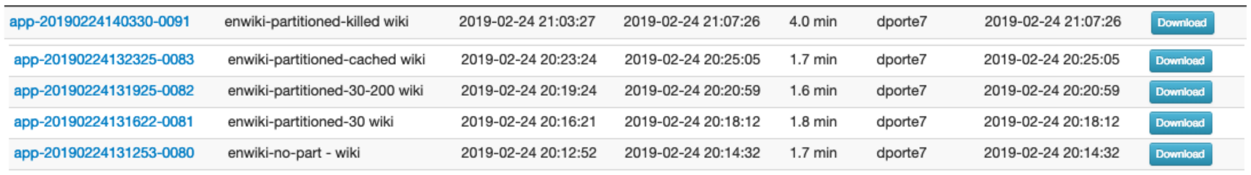
```
links30
ranks30
contribs0->200
ranks0->200
contribs1->200
ranks1->200
contribs2->200
ranks2->200
contribs3->200
ranks3->200
contribs4->200
ranks4->200
contribs5->200
ranks5->200
contribs6->200
ranks6->200
contribs7->200
ranks7->200
contribs8->200
ranks8->200
contribs9->200
ranks9->200
```

30-200par.png



```
links90
ranks90
contribs0->180
ranks0->180
contribs1->270
ranks1->270
contribs2->360
ranks2->360
contribs3->450
ranks3->450
contribs4->540
ranks4->540
contribs5->630
ranks5->630
contribs6->720
ranks6->720
contribs7->810
ranks7->810
contribs8->900
ranks8->900
contribs9->990
ranks9->990
```

sparkDefault.png



app-20190224140330-0091	enwiki-partitioned-killed wiki	2019-02-24 21:03:27	2019-02-24 21:07:26	4.0 min	dporte7	2019-02-24 21:07:26	Download
app-20190224132325-0083	enwiki-partitioned-cached wiki	2019-02-24 20:23:24	2019-02-24 20:25:05	1.7 min	dporte7	2019-02-24 20:25:05	Download
app-20190224131925-0082	enwiki-partitioned-30-200 wiki	2019-02-24 20:19:24	2019-02-24 20:20:59	1.6 min	dporte7	2019-02-24 20:20:59	Download
app-20190224131622-0081	enwiki-partitioned-30 wiki	2019-02-24 20:16:21	2019-02-24 20:18:12	1.8 min	dporte7	2019-02-24 20:18:12	Download
app-20190224131253-0080	enwiki-no-part - wiki	2019-02-24 20:12:52	2019-02-24 20:14:32	1.7 min	dporte7	2019-02-24 20:14:32	Download