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'2549132̂56802', u'254913̂256798'
- 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)

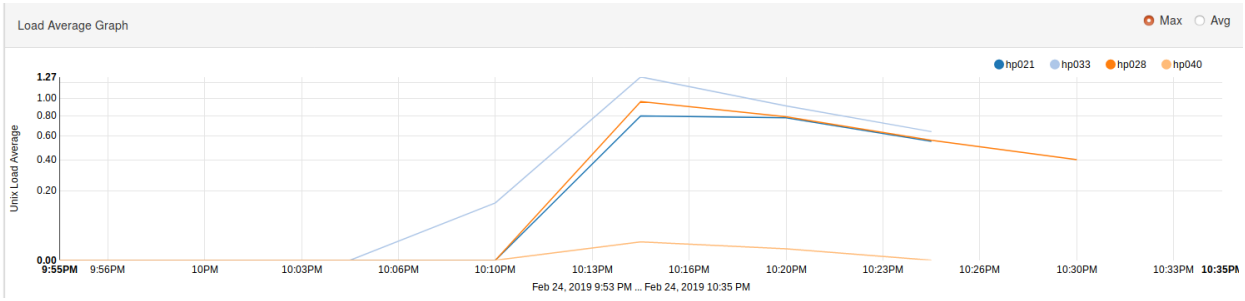
- This step is equivalent to step 2 explained in the overview. Where the output here is the sum of the neighbor rank value collection which was provided through a "transpose" of the original RDD.

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

  (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.

In the picture below is possible to see the plot for the MAX load of the nodes. The data set used was the one from Wikipedia and no custom partitioning or in-memory persistence have been used. Since the leader node behaves just as a Master and there are few Workers its load is pretty low. Probably having a Worker thread running also on the Master node would have increased the performance.

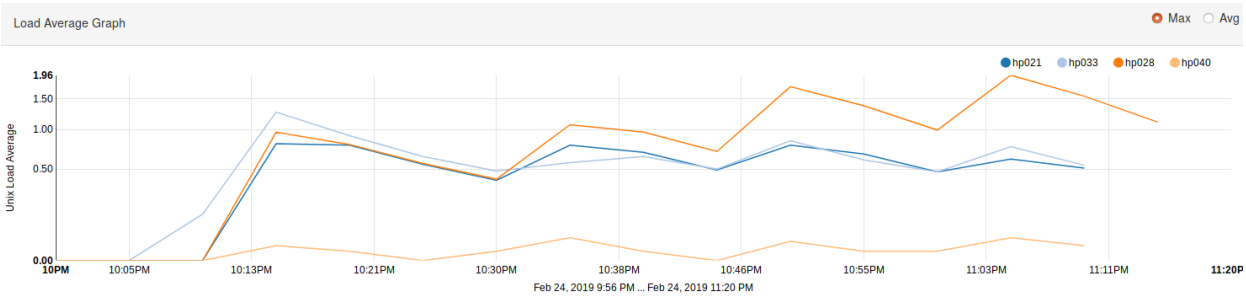


Custom Partitioning

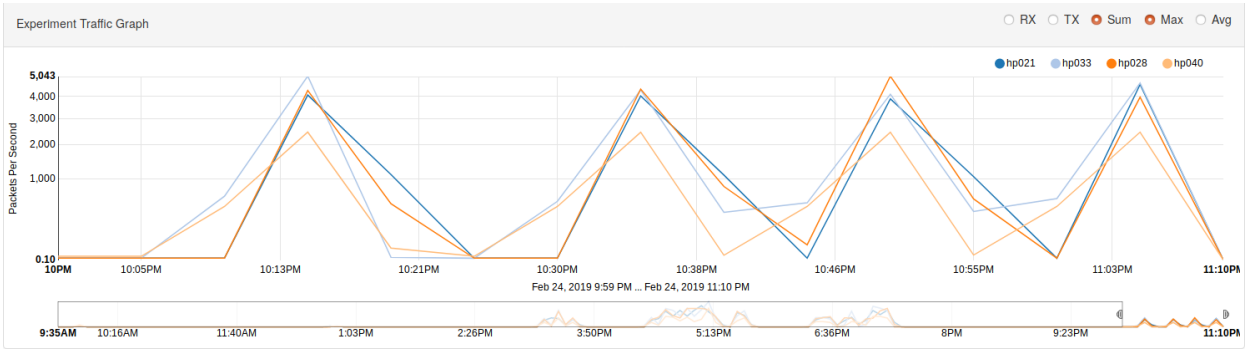
We tuned the page rank application for the Wikipedia data set according to our configuration by allocating static partitioning for the job. We evaluated performance in terms of execution time and varied the partitioning configuration for both the `links` and `ranks` RDDs.

What we discovered is that with a low or high partitioning (below 10 and more than 30) performance starts to deteriorate. The best results are found within 20 and 30 partitions which makes sense since the cluster is configured to use 30 cores in total so we have 1 partition per core. However, Spark dynamically handles partitioning of the RDDs, and if we kept all partitions at 30, the app would run slower with respect to not imposing a partitioning. Which is the reason why the load for the Workers is above 1 when using a custom partitioning (the Workers are overloaded of work).

Below are shown the plots for the MAX load for each node while running page-rank with different custom partitioning. Even in this case the load on the leader node is quite small while the Workers are working with load around 1 or above. It's interesting to notice how packet traffic have slight variations when different levels of partitioning are applied: less traffic is generated with 20 partitions with respect to 30 or 40 partitions.



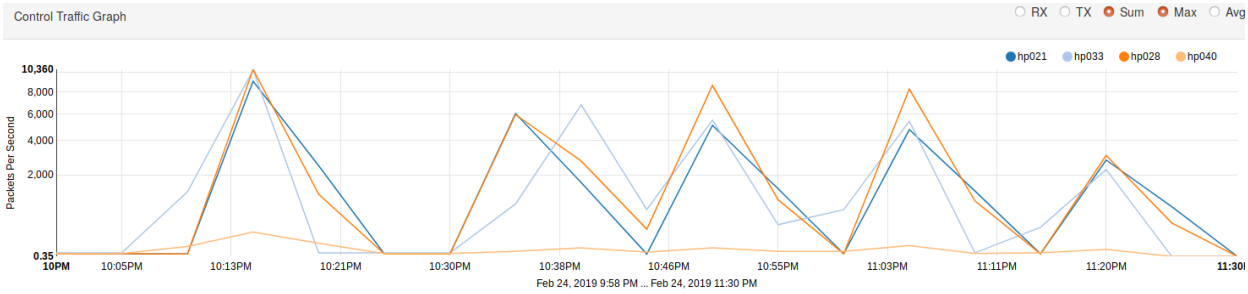
In order from left to right it's possible to compare the peaks load for: the basic pagerank implementation, the implementation with 20 partitions, with 30 and finally with 40. The completion times were respectively: 1.8 min, 1.9 min, 2.3 min and 2.4 min



Network Traffic expressed in terms of Packets per second. In order from left to right the peaks for: basic pagerak implementation, the implementation with 20 partitions, with 30 and finally with 40.

In-memory RDDs

When we forced the caching of both the RDDs (in addition to a partitioning of 20) the performance improved as the completion time dropped to 1.5 min. At the same time, the control traffic slightly decreased with respect to no in-memory persistence as it's possible to see from the picture below.



Network Control Traffic expressed in terms of Packets per second. In order from left to right the peaks for: basic pagerak implementation, the implementation with 20, 30, 40 partitions and with in-memory persistence and 20 partitions.

Killing one worker

We then tried to kill a Worker process after cleaning the cache. What happened was the following: the spark task scheduler would notice that one Worker is no more responsive and consequently re-schedule the last task and proceed from that point with just 2 Workers (which of course makes to double the completion time).

Also it's possible to notice how the communication traffic and the nodes load were affected by this in the pictures below (the order is the same as before with the last peak representing the experiment were a Worker was killed). In particular the load and traffic of the node killed decreased rapidly while the opposite happens for the other nodes in the cluster.

Workers

Worker Id	Address	State	Cores	Memory
worker-20190224162648-128.110.154.102-45907	128.110.154.102:45907	DEAD	20 (10 Used)	61.8 GB (32.0 GB Used)
worker-20190224162648-128.110.154.109-45951	128.110.154.109:45951	ALIVE	20 (0 Used)	61.8 GB (0.0 B Used)
worker-20190224162648-128.110.154.114-36509	128.110.154.114:36509	ALIVE	20 (0 Used)	61.8 GB (0.0 B Used)

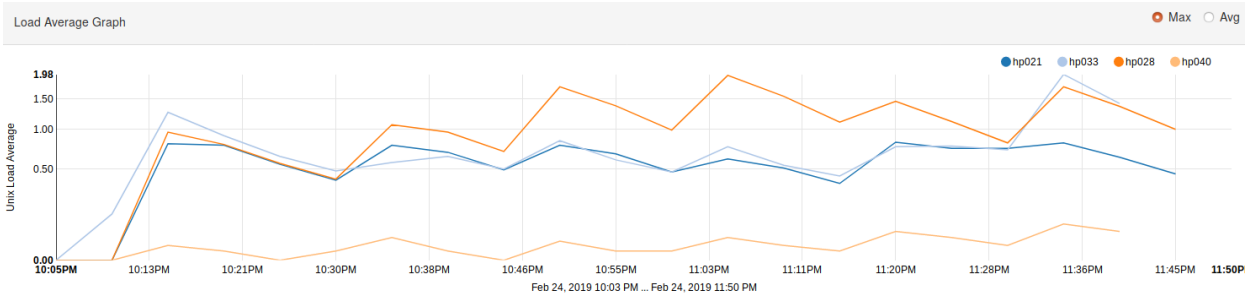
Running Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

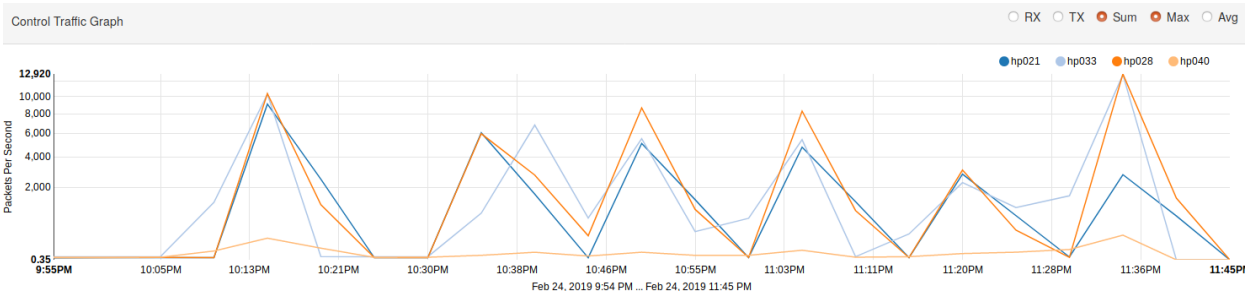
Completed Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20190224163258-0002	homework 1 part 2 - wiki basic	20	32.0 GB	2019/02/24 16:32:58	claudio	FINISHED	3.0 min
app-20190224163006-0001	homework 1 part 2 - wiki basic	30	32.0 GB	2019/02/24 16:30:06	claudio	FINISHED	1.8 min
app-20190224162735-0000	homework 1 part 2 - wiki basic	30	32.0 GB	2019/02/24 16:27:35	claudio	FINISHED	1.8 min

The picture above shows how the completion time was affected when killing one worker with respect to the baseline: from 1.8min to 3.0 min.



MAX load for each node in the cluster during time



Control Traffic in terms of packets per second during time