CS744: Report of Assignment1

Group 15: Zihang Meng, Pan Wu, Yiwu Zhong

February 14, 2019

1 Part1

Install and deploy HDFS and Spark system.

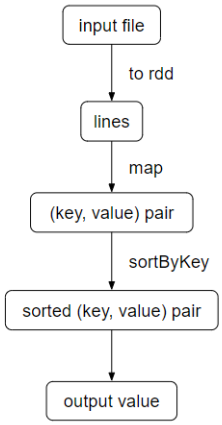


Figure 1: Lineage of Secondary Sort Algorithm

2 Part2

Here, we will write a simple Spark application that, inputting the data in a cvs file, we need to sort the data firstly by the country code alphabetically (the third column) then by the timestamp (the last column). The RDD lineage is as Figure 1 shows. Our solution is following:
1. Read data: Firstly, we load the data from the file in HDFS, and generate the base RDD (Lines).
2. Map: then we use a map operation to generate a new RDD which contains (key, value) pairs. Note that the key is a tuple (cca2, timestamp).
3. SortByKey(Reduce): after that, we sort the pair RDD using sortByKey operation, and we get the sorted RDD.

4. Map: At last, we map the sortedRDD to extract the values and output the result to a file in HDFS.

3 Part3

In this part, we implement the PageRank algorithm, which is an algorithm used by search engines like Google to evaluate the quality of links to a webpage. The algorithm can be summarized as follows: set initial rank of each page to be 1; on each iteration, each page contributes to its neighbors by rank(p) / number of neighbors; update each page's rank to be 0.15 + 0.85 * (sum of contributions); then go to next iteration.

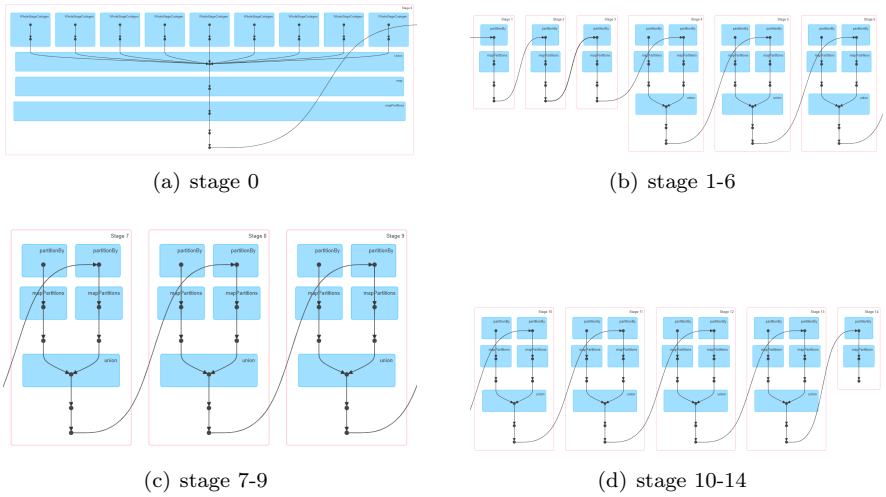
3.1 Task1

Task: Write a Scala/Python/Java Spark application that implements the PageRank algorithm.
Solution: Our implementation is as follows:
1.Read the input files from HDFS and do preprocessing (such as deleting some lines not needed)
2.Parse url pairs from the data. Use source url as key and destination as value.
3.Initialize the url rank to be 1.
4.Do a for loop, where in every iteration, we first compute the contributions of each url to its neighbors and then reduceByKey to get the updating rank of each url.
5.Write the output to given path.
After implementing this algorithm, the RDD lineage is as Figure 2 shows. And we test it on Berkeley and Wiki dataset respectively. We spent 1.92 minutes on Berkeley dataset and 9.4 minutes on Wiki dataset.

3.2 Task2

Task: Add appropriate custom RDD partitioning and see what changes.
Solution: We run our code on Berkeley and Wiki dataset, varying the number of partition. The results are shown in Figure 3. According to the curve, we can find that with the increment of partition number, the running time decrease first, then reach a minimum (partition = 30 where the partition number is 2 times of the CPU cores number we used), and finally increases gradually.

By default, Spark reads data into an RDD from the nodes that are close to it. In our task, the links are repeatedly joined with the ranks and each join operation requires a full shuffling over the network. With partitions, the links which have the same hash code are on the same node. And these partitions help the program parallel distributed data processing with less network traffic for sending data between executors. So the partitioning benefits our processing procedure.



(a) stage 0

(b) stage 1-6

(c) stage 7-9

(d) stage 10-14

Figure 2: Lineage of PageRank Algorithm

More specifically, the number of partitions shouldn't be too small or too large. If the number of RDD partitions is too small, some cores may be idle and this doesn't make a big difference on computation time. If the number of RDD partitions is too large, the number of tasks will also be too large and these tasks increase the overhead of managing them. So too many partitions also increases the execution time.

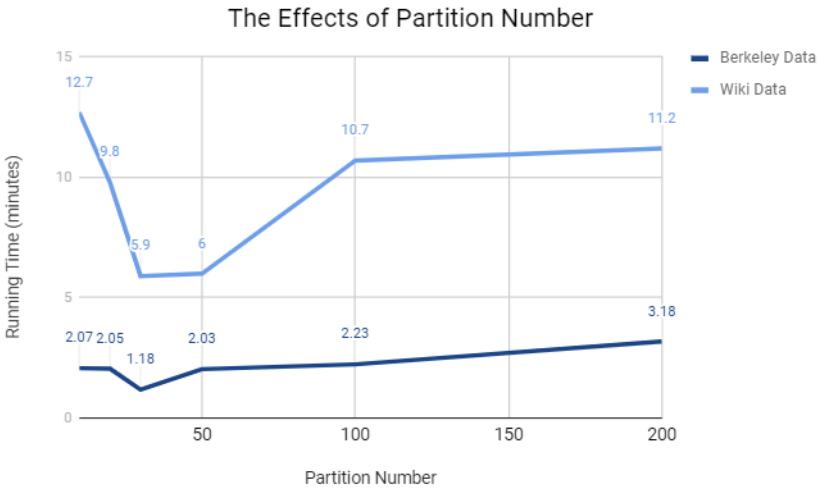


Figure 3: The Effects of Partition Number

3.3 Task3

Task: Persist the appropriate RDD as in-memory objects and see what changes.
Solution: We choose test our code on both datasets with 30 partitions. For Berkeyley dataset, we spent 1.2 minutes without persisting, and 1.18 minutes with persisting. For Wiki dataset, we spent 5.9 minutes without persisting, and 5.4 minutes with persisting. Caching or persisting are the optimization techniques for interactive and iterative Spark computations. In our task, we run the Pagerank program as an iteration task and we need to reuse the data repeatedly. After caching data in the memory, the intermediate results are saved in memory, reused in each iteration through memory instead of creating I/O with disk, and thus reduces the computation overhead. So the computation time will be reduced after caching data into memory.

3.4 Task4

Task: Kill a Worker process and see the changes. You should trigger the failure to a desired worker VM when the application reaches 50% of its lifetime
Solution: We choose the setting where we use 30 partitions and appropriate cache. The original running time is 5.40 minutes.

We first tried killing the worker 1 and clear the cache of it after the application runs for 3 minutes. Spark gives the following error and then go back to Stage 0 to run: "ERROR TaskSchedulerImpl: Lost executor 1 on 128.104.223.189: Remote RPC client disassociated. Likely due to containers exceeding thresholds, or network issues. Check driver logs for WARN messages." After restarting from stage0, it takes extra 4.67 minutes to finish the task.

Also, We had another try where we killed the worker after running for 4.50 minutes. Spark also went back to stage0 to run. This time it only took extra 3 minutes to finish.

The total time with killing the worker and clearing the cache is longer than original running time. But note that it takes less time to complete the task when Spark went back to stage0 than that of without killing and clearing. That's because some RDD in other 2 workers still existed and the Spark only need to re-compute those RDD which are cleared or killed. As the result, the running time from stage0 to the very end is decreased.