HW 4 - Report

Table of Contents

1. Design Discussion

Following is the high-level description of how spark processes the data and steps taken by the spark to execute the program with how the respective functionality is implemented in hadoop jobs.

1. **sc.textFile(inputPath) :** SparkContext's textFile method creates a text file RDDs by reading the data files from input path as a collection of lines. By default, spark creates one partition for each block of file. MapReduce creates the map tasks as per the no of partitions in input data and pass each line as an input in map method.

2. **flatMap(HelperFunctions.flattenAdjacencyList) :** flatMap takes each line as an input from RDD and map it to 0 or more output items and overall transforms the RDD of strings to PairRDD. Here data will be partitioned by default hash partitioner. MapReduce has a map function (in Mapper of PreProcessing Job) which is doing the similar functionality. It also takes each line as input and emits 0 or more key value pairs.

3. **reduceByKey(HelperFunctions.reduceNodesByNodeId) :** This function merge the values for each key of PairRDD using the reduce function passed and transforms it to reduced PairRDD of same type. In MapReduce we used reduce function (in Reducer of PreProcessingJob) to combine the key value pairs emitted by the map function.

4. **persist() :** This function persist RDD in memory. Each node stores any partitions of RDD that it computes in memory and uses it for future actions to improve the performance of future actions. Here our node structure is not going to be changed so I have persisted it. In MapReduce, we are not persisting the data in memory instead we write the reduce output in file and the PageRankJob reads the data from these files.

5. **nodes.count() :** This function is an action which will return the no of elements in the dataset. For getting the node count, we used global counter in MapReduce and read it after finishing the PreProcessingJob.

6. **nodes.mapValues(value => 1.0 / noOfNodes) :** mapValues is a PairRDD function which passes each value in the key-value pair RDD through a map function without changing the keys and also retains the original RDDs partitioning. The output RDD is stored as a ranks RDD which is a variable RDD. It will be updated after each iteration of page rank calculation. In MapReduce, we are not running separate job for this instead we calculate the initial page rank in map function of PageRankJob's first iteration.

7. **nodes.join(ranks) :** In each iteration, this function joins the rank and nodes RDD. Rank is a variable RDD and which will be updated in each iteration, whereas nodes (nodeId & adjacencyList) is a fixed RDD. As both RDDs are partitioned by the hash practitioner, same key will reside in same partition in both RDDs so join will be done inside each par-

tition. In MapReduce, we have one data structure with nodeId, adjacencyList and pageRank so no need of join.

8. **flatMap(HelperFunctions.flattenRanks) :** This function emits the page rank contribution of each node along with delta with dummy key. It works on the joined RDD of (nodeId, (adjacencyList, rank)) and transforms it to RDD of (nodeId, rank) structure. In MapReduce we are doing this in map function of PageRankJob's mapper.

9. **reduceByKey(_ + _) :** This function adds up the page rank of a node. In MapReduce we have similar reduce function in PageRankJob.

10. **contribs.lookup("DANGLING~FACTOR") :** After adding up page rank contribution of each node we look up in RDD for dangling factor key and store it in delta. In MapReduce we store the delta in global counter and after finishing the job we fetch it from there to pass it in next iteration.

11. **contribs.mapValues({ rank => <newRank>}) :** This function updates the page rank sum of each node by applying alpha and delta to it. In MapReduce, we calculate this in map function of next iteration.

12. **ranks.saveAsTextFile(outputPath + "/AllRanks") :** If we want to write the page rank of all nodes in file then we can enable this call.

13. **map(node => (node._2, node._1)) :** For TopK by page rank, this function maps each (NodeId, Rank) pair to (Rank, NodeId) pair. In MapReduce, we have the very different logic but there also we have a tree map data structure which stores the topK elements by (Rank, NodeId) pair.

14. **top(kForTopK) :** This function returns the top k nodes by their page rank to driver.

In page rank calculation we perform multiple iterations over the same data. Spark gives very good performance for iterative programs. MapReduce reads data from disk in each iteration and after processing writes back to the disk and read again in next iteration. This involves heavy I/O operations whereas in spark, once we create RDD's from the disk we keep it in memory (until size not exceeds the limit) and transforms/perform the actions on the same RDD in each iteration.

The line of code is very high in MapReduce compare to Spark. MapReduce has a very strict API that doesn't allow for as much versatility. Since spark abstracts away many of the low level details it allows for more productivity. Also things like broadcast variables and accumulators are much more versatile than Distributed Cache and global counters. At the same time we can easily understand the data processing and movement done behind the MapReduce execution whereas its bit tough for Spark execution.

Spark uses resilient distributed dataset (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel. Spark persist an RDD in memory (or any other level), allowing it to be reused efficiently across parallel operations and they automatically recover from node failures. Spark has concept of transformations which are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset. The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently.

2. Performance Comparison

2.1 Running Time

I have used SAXParser in HW3 and HW4 to parse the input initially. But after running spark program with SAXParser, I found that it is taking so much time in only parsing the data so wrote my own parser using jsoup. To compare the performance of Spark and MapReduce, I have executed HW3 programs using new parser.

I have used alpha = 0.15 for my page rank calculations (Source: Wikipedia page of page rank algorithm). Though the value of alpha is not hardcoded and you can configure in Makefile.

Below is the screenshot of AWS EMR successful run:

- Wiki-full-html input run on m4.large machines (1 master, 5 workers (1st) & 10 workers (2nd)) using Spark program:

HW4 Cluster	j-353B3MWY9M9PP	Terminated All steps completed	2017-03-18 13:07 (UTC-4)	30 minutes	24
HW4 Cluster	j-1EMQA3CLAWVY9	Terminated All steps completed	2017-03-18 00:35 (UTC-4)	21 minutes	44

- Wiki-full-html input run on m4.large machines (1 master, 5 workers (1st) & 10 workers (2nd)) using MapReduce program:

HW3 Cluster	j-HR9WPODIMMSC	Terminated All steps completed	2017-03-18 11:56 (UTC-4)	52 minutes	24
HW3 Cluster	j-OX1KNO591TS3	Terminated All steps completed	2017-03-18 11:15 (UTC-4)	32 minutes	44

Below is the total running time of full Wikipedia input on both configurations:

(Time in Seconds)

Execution	6 m4.large Machine	11 m4.large Machines
Spark	1371	757
Hadoop - MapReduce	2685	1545

As we can see in above table, Spark is the clear winner here. Running time of the spark execution is almost half of the running time of Hadoop execution in both running configuration. The main reason behind the better performance of the spark is the way it processes the data. Spark does everything in memory (in-memory caching abstraction - RDD) while MapReduce persists the full data set on disk after running each map and reduce jobs. Spark has transformations which are lazy, In that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset. The transformations are only computed when an action requires a result to be returned to the driver program. This

design enables Spark to run more efficiently. Spark can launch task much faster than MapReduce. MapReduce starts a new JVM for each task whereas Spark keeps executor JVM running on each node so it is much faster.

2.2 Top-100 Wikipedia Pages

The top-100 Wikipedia pages with highest page rank along with their page values sorted from highest to lowest are provided at below paths. The results of the page ranks are almost the same. The pages are ordered perfectly the same in both execution but there are very minor differences in the page rank values (at 9^{th} decimal point). This minor difference can be because of our double to long conversion in MapReduce program while storing delta or can be dependent on double precision calculation of the specific environment. As the differences are very minor and page rank order is perfectly same we can ignore them.

For MapReduce execution:
4_AWSOutputFiles/HW3-MapReduce/simple-dataset 6-machines
4_AWSOutputFiles/HW3-MapReduce/wiki-full-dataset 6-machines
4_AWSOutputFiles/HW3-MapReduce/wiki-full-dataset 11-machines

For Spark execution:
4_AWSOutputFiles/HW4-Spark/simple-dataset 6-machines
4_AWSOutputFiles/HW4-Spark/wiki-full-dataset 6-machines
4_AWSOutputFiles/HW4-Spark/wiki-full-dataset 11-machines