CS6240 - Large Scale Parallel Data Processing.

Homework 1 - Report

09/22/2019

Name: Akshay Kulkarni
GitHub Repo: https://github.ccs.neu.edu/cs6240-f19/akshaykulkarni-Assignment-1.git

MapReduce Implementation:

```
Mapper<Object, Text ,Text ,IntWritable>{

      // Map function input is each line in csv.

      map( Object , text t ){

            emit( x, 1 )
      }

      combine( node x, [c1, c2,...]{

            // same as the reducer.

      }

      // Each c is a partial count for word x

      reduce( word x, [c1, c2,...] ){
            total = 0 \\for each c in input list
            total += c
            emit( x, total )
      }
}
```

The file(s) from specified input folder are taken by hadoop job-runner and are distributed into different chunks into hdfs then the partitioner assigns specific chunks to mappers. The map functions are fed input line by line. Each line is an edge which is split on the delimiter and the second value (node) is taken as a key and assigned a value of 1. This (Key, 1) tuple is emitted by the map function to a combiner function which in this case is identical to the reducer which computes the final local counts, the keys and their respective values are assigned to the responsible reducer by hadoop partitioner. The reducer takes in all the values for a specific key (node) and computes and emits the global count.

Spark-Scala Implementation:

```
// creating schema from predefined Edge-Class encoder
val schema = Encoders.product[Edge].schema

// loading the csv in a Spark DataSet (similar for nodes)
Val edges = spark.read.option("header", "false")
                        .schema(schema)
                        .csv(args(0))
                        .as[Edge]

Val TotalCount = nodes.join(follower_counts,Seq("User"),"leftouter")
                        .sort(desc("Count")) // sorts in desc of user
                              .na.fill(0,Seq("Count")
                                .write.format("csv").save(args(1))
```

Scala is relatively easier to implement the program in the abstraction and versatility of the Spark API makes the program as compact as any pseudocode. I decided to implement Spark DataFrames in my version as they offer better performance, optimization and memory management plus give you easy to use functions to manipulate data better, My program reads in the csv file on the lfs using a spark csv parser and creates a dataframe with the predefined schema with typecasting with spark_se, after which it is possible to simply call methods like groupby() and count() (even sort()) by column to compute followers for all nodes and then save the output back to lfs.

edges.explain() ***output:***

```
== Physical Plan ==
*(1) FileScan csv [Follower#2,User#3]
      Batched: false, Format: CSV, Location:
          InMemoryFileIndex[file:/home/akshay/Desktop/Twitter-dataset/data/edges.csv],
                PartitionFilters: [], PushedFilters: [], ReadSchema: struct<Follower:int,User:int>
```

Follower_counts.explain **output (for normal count):**

```
== Physical Plan ==
*(2) HashAggregate(keys=[User#3], functions=[count(1)])
+- Exchange hashpartitioning(User#3, 200)
   +- *(1) HashAggregate(keys=[User#3], functions=[partial_count(1)])
          +- *(1) FileScan csv [User#3] Batched: false, Format: CSV, Location:
             InMemoryFileIndex[file:/home/akshay/Desktop/Twitter-dataset/data/edges.csv],
                          PartitionFilters: [], PushedFilters: [], ReadSchema: struct<User:int>
```

**Note- The bonus question Output of the result in order of the number of followers and users with no followers is in the Spark output folde on github*

Running Time Measurements:

Execution Name & Run	Run Time (M:S)	Data to Mappers (MB)	Data Transfer Mappers->Reducers (MB)	Data from Reducers to Outputs (MB)
MapReduce-Run 1	01:37	1319.43	92.93	87.52
MapReduce-Run 2	02:00	1319.44	92.93	87.52
Spark- Run 1	01:25			
Spark- Run 2	01:22			

Both Spark jobs were achieved on m4.large machines which are lower spec and cheaper than the m5.xlarge machines.

Speedup-

In my opinion, the Map-Reduce program should expect to have a decent speedup due to the more compute power that a cluster provides. Both runs were comprised of 20 Mappers and 10 Reducers and the non-sorting implementation of the Follower count problem does not have any inherent sequential processes therefore in theory it can experience a t/n parallel run-time. On the other hand, the sorted version's speed would depend on the implementation, if sorting were done at the reducer end, that would entail quite a network cost and also would have to be done by a single worker, but if sorting is implemented in more elegant ways such as implementing appropriate partitioning and using custom keys and comparators then that could be better but theoretically still bounded by N.

AWS FIles-

MapReduce-

Logs:
https://github.ccs.neu.edu/cs6240-f19/akshaykulkarni-Assignment-1/raw/master/AWS-logs/fc-hadoop-run1.pdf
https://github.ccs.neu.edu/cs6240-f19/akshaykulkarni-Assignment-1/raw/master/AWS-logs/fc-hadoop-run2.pdf
Output:
https://github.ccs.neu.edu/cs6240-f19/akshaykulkarni-Assignment-1/tree/master/AWS-Outputs/MapReduce-1
https://github.ccs.neu.edu/cs6240-f19/akshaykulkarni-Assignment-1/tree/master/AWS-Outputs/MapReduce-

Spark-

Logs:
https://github.ccs.neu.edu/cs6240-f19/akshaykulkarni-Assignment-1/raw/master/AWS-logs/fc-spark-run1.pdf
https://github.ccs.neu.edu/cs6240-f19/akshaykulkarni-Assignment-1/raw/master/AWS-logs/fc-spark-run2.pdf

Output:
https://github.ccs.neu.edu/cs6240-f19/akshaykulkarni-Assignment-1/tree/master/AWS-Outputs/Spark-1
https://github.ccs.neu.edu/cs6240-f19/akshaykulkarni-Assignment-1/tree/master/AWS-Outputs/Spark-2