CS539 Assignment 3

Hadoop and Map reduce

Ashish Jindal (AJ523)

1 Implementation Details

1.1 Parsing PGN files

I have used ChessPresso lib to parse the given pgn files. This is integrated into hadoop by overriding "FileInputFormat" and then implementing a new RecordReader using ChessPresso api. While parsing the large chess dataset, the library detected some corrupt records (Having invalid moves). So it throws an exception whenever an invalid move is found while parsing a game record. I have ignored such records while calculating the statistics. By implementing our own RecordReader we can split the pgn files into games (Default is by line) and distribute these games data to different mappers.

1.2 Maven for packaging

All the submitted projects have been created using maven and the target folder of the submissions have there respective jar files. As I needed to pack chesspresso jar in the project jar, so I used the maven-assimbly-plugin to package the jar. All the jars created are thus executable jars and have all the required dependencies.

1.3 Problem A

In this problem I send a key value pair of Colour and two integers from mapper. First integer has value 1 if the Colour corresponds to a win and 0 other wise. The Second integer is always 1. In the combiner and reducer I use the summation of first integer to get the count of wins corresponding to a Colour and the summation of second integer to get the total number of games. Dividing the two gives the required percentage and I output the result from the reducer.

1.4 Problem B

In this problem I create a new composite key class "PlayerWritable" which has two data objects one representing the color and other the player ID/name. This is kept as the key of the mapper and as the key here is WriteComparable so framework will use it's comparator to find which key goes to which reducer. This way all the players with same name are routed to same reducer.We send 5 counters in the corresponding value 4 of them representing the game result - wins/loses/draw/unknown and the last one for maintaining the total count. In the reducer we get one player-color key at a time and then we get the corresponding stats by summing the values and after that we write the calculated result into the file. The above explained functionality is common for both combiner and reducer.

1.5 Problem C

In this problem we use two map-reduce jobs. First we output move-count and "1" pairs from mapper which we then sum up in combiner and reducer to calculate the frequency of move-counts in the data. We then use a custom counter to keep track of number of games from the mapper which we use in reducer to finally calculate percentage. In the next map-reduce job we input the data obtained from previous job. In its mapper we reverse the keys to map a percentage to its move count and then the reducer gets the respective percentages in the sorted way which we dump in the file. To ensure that only a given range of values arrive at a particular reducer - for obtaining total ordering while sorting, we use a custom partitioner class. We set this class using the function "job.setPartitionerClass()" while creating the job. As the data was needed in a decreasing order so we wrap the mapper key in a "WriteComparable" class and implement a custom compare method to imply decreasing order of percentages.

Output for problem C-
Small dataset
output job 1 - folder outputSmallJob01
output job 2 - folder outputSmallJob03

Large dataset
output job 1 - folder outputLargeJob01
output job 2 - folder outputLargeJob03

1.6 AWS bucket details

Bucket name: aj523-cs539
1. Word count - Folder name "wordCount"
2. Problem A - Folder name "problem_a"
3. Problem B - Folder name "problem_b"
4. Problem C - Folder name "problem_c"

1.7 Output Problem A

Small Data set -
White 11028 .44
Draw 4582 .18
Black 9559 .38
Unknown 0 .00

Large Data set -
White 216483373 .50
Draw 16601112 .04
Black 203839727 .47
Unknown 0 .00