532 Project 1 Design Document

MapReduce Implementation using Java

Group Members:

- ★ Akhil Reddy Anthireddy
- ★ Prasann Desai
- ★ Muneer Ahmed

We used Java to implement the MapReduce library in this project where the user will also be able to create any user-defined Map and Reduce functions.

Project Overview:

The location and name of the data files (input, output and intermediate) are present in the config file. It also has the number of workers present in it. The user needs to define these before running the MapReduce application.

Once we run the MapReduce application, the master first splits the input data to distribute it among the map workers to perform Map function. The map workers then execute the user defined map functions and write intermediate results to the intermediate files. The map workers write the intermediate data in the form of key-value pairs.

We also handle faults by checking if the worker executing the map function gives a result within a certain timeframe or not. If it does not, then we re-start that mapper worker.

Once all the mapper workers have completed their execution the reducers start the reduce job. Each reducer aggregates for one key only. The reduce function takes the data from the intermediate files and then performs a user defined aggregation function for each key and writes the output to separated files, 1 for each key value. The workers run multiple reduce functions on a set of unique keys to generate the final output. The reducer iterates over each intermediate file and aggregates the data for the particular key to give the final output for that key. If needed we merge all these outputs into a final file.

Project contents:

We have implemented the following 3 test cases using MapReduce System:

1. *WordCount:* Frequency of every unique word in the document.
2. *WordLength:* Number of words in the document for each word length.
3. *Minimum and Maximum Temperature* for every year.

Following are the java files involved in the project so far:

- **Library programs:**

  a) *MRMaster.java* (Library): Functions as the master and splits the data and communicates and manages the workers. Also performs fault tolerance.
  b) *Mapper.java* (Library): Contains the general Mapper class interface.
  c) *Reducer.java* (Library): Contains the general Reducer class interface.
  d) *workermap.java* (Library): Executes the map function and writes intermediate files.
  e) *workerreduce.java* (Library): Executes the reduce function and writes output files.

- **Test case programs:**

  1. *mrwordcount.java:* Runs wordcount test-case using the map & reduce functions from the library programs.

     - **mapperwc.java**: Contains the user-defined implementation of the map function for word-count application.
     - **reducerwc.java**: Contains the user-defined implementation of the reduce function for word-count application.

  2. *mrwordlength.java:* Runs word length test-case using the map & reduce functions from the library programs.

     - **mapperwl.java:** Contains the user-defined implementation of the map function for word-length application.
     - **reducerwl.java**: Contains the user-defined implementation of the reduce function for word-length application.

  3. *mrweather.java:* Runs yearly-minmax temperature test-case using the map & reduce functions from the library programs.

     - **Mapperweather.java:** Contains the user-defined implementation of the map function for minimum and maximum temperature for a year.
     - **Reducerweather.java:** Contains the user-defined implementation of the reducer function for minimum and maximum temperature for a year.

- **Testing script:** *test_script_mr.sh*

Libraries:

MRMaster.java

This library executes the master which starts the mappers and reduces as new objects which are then executed using multi-threading. This is done using the process builder function which creates a new process for each function. The master also splits the initial input data into user defined parts (N) which will then be sent to each mapper. Once all mappers finish executing and creating intermediate files the reducers are then executed whose output is then merged. The master also handles fault tolerance, it does this by checking if the function timed out or if one process crashes or is unable to complete its execution. If any of the previous mentioned situations occur it reruns the mapper function for which it happens.

Mapper.java

This contains the function signature of the map function which can be adhered to by the user defined function for map.

Reducer.java

This contains the function signature of the reducer function which can be adhered to by the user defined function for reduce.

workermap.java

This library has the execute function which makes a worker execute the user defined map function and write the output to intermediate files.

workerreduce.java

This library has the execute function which makes a worker execute the user defined reduce function and write the output to output files.

Config File:

This file contains the number of processes, location and names of all the files (input, output and intermediate) needed to run or which are generated by the MapReduce system.

How do these programs work:

1. ***mrwordcount.java:***

   Once we set the N (Number of process) the master splits the data into N parts and sends 1 part to each mapper which then counts the number of occurrences for a each word in that data part and outputs a <key,value> pair where key is the word and value is the number of times it occurs in that part. These pairs are then written to an intermediate file.

   Once all the mappers have finished executing, the reducers start executing. Each reducer is assigned a single key(word). The reducer iterates over every intermediate file and for the key it was assigned it fetches the count form that file and aggregates from all files. The final count is then written to the output file with the key associated with it.

   Once all the rescuers finish executing the Master merges all the individual files to give a final output.

   If there is a failure of a function for any reason, the master handles this by checking for the failure and then reruns that function.

   OUTPUT FILE: KEY (Word), Total count of the occurrence of the word in the data.

2. ***mrwordlength.java:***

   Once we set the N (Number of process) the master splits the data into N parts and sends 1 part to each mapper which then counts the number of occurrences for a each word in that data part and outputs a <key,value> pair where key is the length of a word and value is the number of times word of the same word length occurs in that part. These pairs are then written to an intermediate file.

   Once all the mappers have finished executing, the reducers start executing. Each reducer is assigned a single key(length of words). The reducer iterates over every intermediate file and for the key it was assigned it fetches the count form that file and aggregates from all files. The final count is then written to the output file with the key associated with it.

   Once all the rescuers finish executing the Master merges all the individual files to give a final output.

   If there is a failure of a function for any reason, the master handles this by checking for the failure and then reruns that function.

   OUTPUT FILE: KEY (Length of Words), Total count of the occurrence of the words with length of key in the data.

3. ***mrweather.java:***

   Once we set the N (Number of process) the master splits the data into N parts and sends 1 part to each mapper which then counts the number of occurrences for a each word in that data part and outputs a <key,value> pair where key is the year and value is the temperatures which were recorded in that year. These pairs are then written to an intermediate file.

   Once all the mappers have finished executing, the reducers start executing. Each reducer is assigned a single key(year). The reducer iterates over every intermediate file and for the key it was assigned it fetches the minimum and maximum form that file and aggregates from all files. The final count is then written to the output file with the key associated with it.

   Once all the rescuers finish executing the Master merges all the individual files to give a final output.

   If there is a failure of a function for any reason, the master handles this by checking for the failure and then reruns that function.

   OUTPUT FILE: KEY (Year), Minimum and Maximum temperature for that year in the data.

How to run:

- Download the code from the "*project-1-group_akhilprasannmuneer*" folder.
- Run the following commands in terminal to execute the 3 test cases:
  - ○ chmod +x test_script_mr.sh (To set execute permissions for the testing script)
  - ○ ./src/test_script_mr.sh (Command to run testing script)
- Once the test cases run, check the output files for each map reduce job in the "***final_outputs***" folder.
- The input splits which the master makes can be found in the "***input_splits***" folder.
- You may use the config file to configure the MapReduce task. For example, you may change the value of N to test with different number of map/reduce workers
- If you wish to run each test case separately, then do the following:

  - For Word Count, run the following commands
    - javac src/**/*.java
    - java -classpath src application/wordcount/mrwordcount

  - For Word Length, run the following commands
    - javac src/**/*.java
    - java -classpath src application/wordcount/mrwordlength

  - For Temperature Min-Max, run the following commands
    - javac src/**/*.java
    - java -classpath src application/wordcount/mrweather