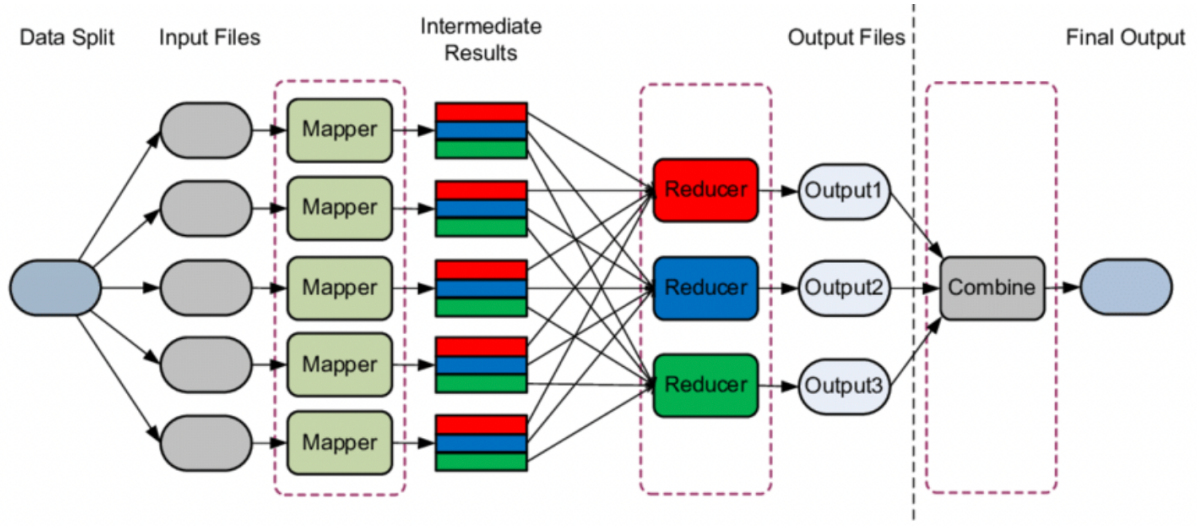
COMPSCI 532 - SYSTEMS FOR DATA SCIENCE - Fall 21
Project 1 - MapReduce

Team members: Sakshi Bhalerao, Snehal Thakur, Vishnupriya Varadharaju

Overview:
MapReduce library using Java has been implemented in this project. With the help of this library, the user can write an arbitrary Map and Reduce function to perform an executable MapReduce job on a distributed system. These user defined functions will be compiled along with the source code during Map and Reduce.
For this project, a total of three test cases have been taken to implement the user defined functions, which will be discussed in detail later.

Control Flow:



In the above architecture of MapReduce, the users will define the data and feed it as input to the MapReduce job. The MapReduce job then triggers multiple mapper processes which invoke user defined mapper functions on their corresponding partitions which write their results to intermediate files. Once, all the mapper processes are completed, the reducer processes will be triggered with the intermediate files as the input. The outputs of each of the reducer is written to a separate file which is then combined to give a single consolidated output.

Design and Working:
The master will start each mapper as a new process. Initially, an RMI registry will be started to facilitate communication between the master and the workers. Specifically, the RMI registry will be used by the workers to access the user defined map and reduce objects. The

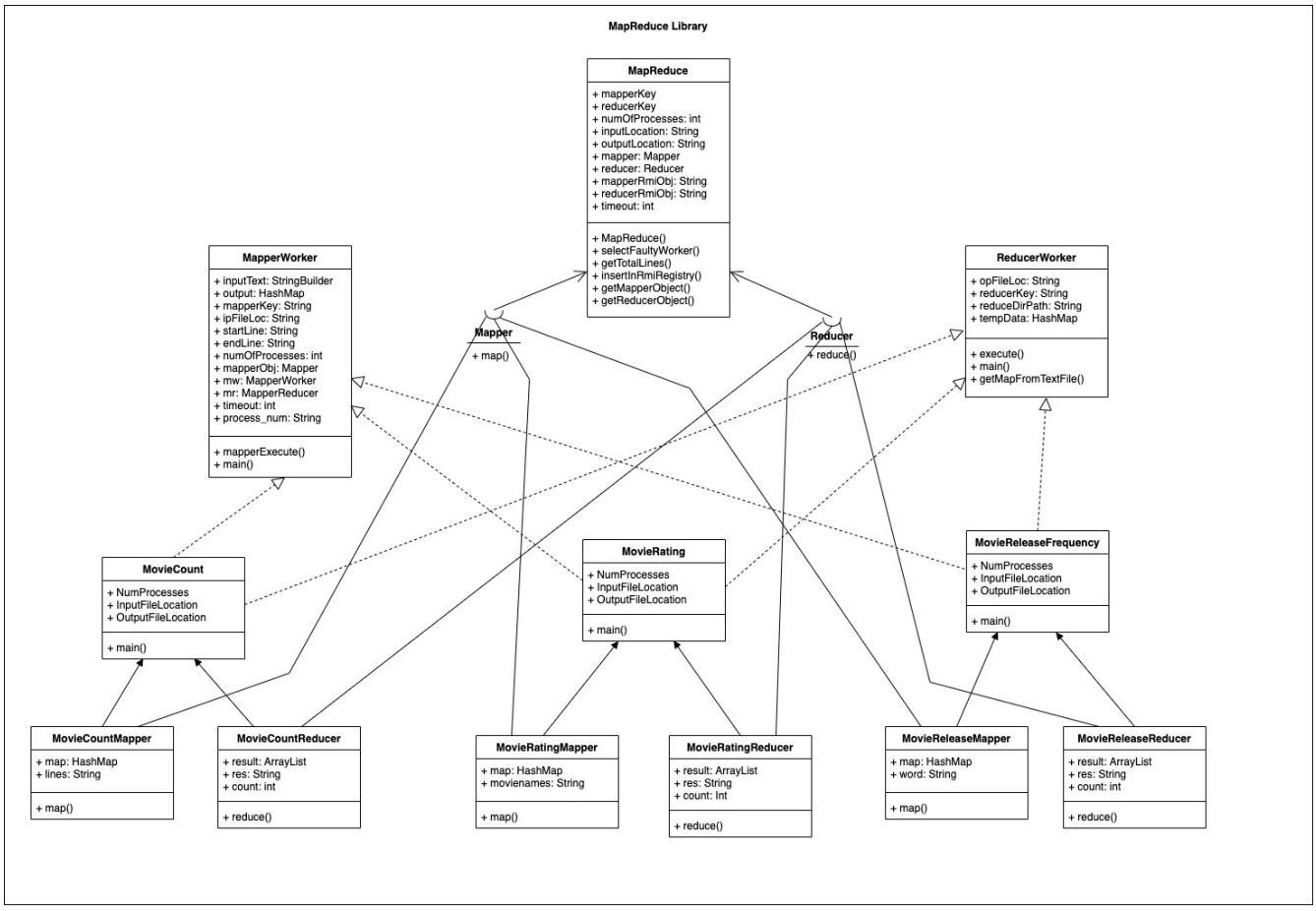
master (src/MapReduce/utils) will add the user defined mapper and reducer objects to the RMI registry. The mapper will then launch N mapper processes. Each mapper will receive an id, number of processes, location of the input file, the starting and ending line numbers of the text file it has to work on. Upon completion of all mapper processes, the master will start N reducer processes that will write to N output files.

Intermediate File Generation:

N Mappers and N reducers are created. Each mapper will produce N intermediate files, corresponding to each of the N reducers (i.e., if there are M mappers and R reducers, each mapper would produce R output files).

Let's consider an example, where, the first mapper creates a directory "mapper_0" with N files - "reducer_0.txt", "reducer_1.txt"...to "reducer_N-1.txt". Reducer N will then read "reducer_N-1.txt" from each of the directories "mapper_0", "mapper_1" until "mapper_N". This way the master does not need to explicitly assign intermediate files to the reducers. The master will only need to give the locations of the N remote directories "mapper_0", "mapper_1" …, that contain the intermediate files to the reducers. However, the overhead here is that there will be a total of N*N intermediate files formed.

MapReduce UML Class Diagram:



Input Parameters from User:
The user should also provide certain parameters to the library as input, apart from the user defined map and reduce functions. These parameters include –

- *Number of processes (N)* – Indicates N mappers and N reducers
- *Input file location* – Specifies the location of the text file on which the MapReduce operation will run on.
- *Output file location* – Specifies the directory where the N output file from the MapReduce operation will be located.
- Object of the user defined mapper class which is to be added to RMI registry for Master and mapper worker to access.
- Object of the user defined reducer class which is to be added to RMI registry for Master and reducer worker to access.
- *Timeout for each worker* – the worker will be stopped, if this time limit is exceeded. This is done to mitigate stragglers.

Function signatures of map and reduce functions:

```
HashMap<String,List<String>> map(String,String)throws
RemoteException List<String> reduce(String,List<String>)
throws RemoteException
```

Sorting of keys using MD5 hash
The mapper will compute, to which intermediate file it has to write a specific key. This is done by hashing that key from 0 to N-1. In this library, the key is hashed using the MD5 hashing algorithm. This is done to ensure that all the mappers write to the same intermediate file in that particular remote directory.
Consider an example where, mapper 2 and mapper 3 encounter a key “Parasite”. This key will get hashed to number 1. As this key is hashed to a value of 1, mapper 2 will write the key value pair to mapper_2/reducer_1.txt and mapper 3 will also write the key value pair to mapper_3/reducer1.txt. This way, reducer 1 itself will perform the reduce operation on all the occurrences of “Parasite”. Thus, there will be no need for shuffling of keys by the reducers.

Fault Tolerance:
Two scenarios have been considered for fault.

1. *Worker Failure* – Worker was unable to complete a task, as that particular process got killed.
2. *Stragglers* – A specific worker takes very long to complete its process.

To induce these faults, a random worker is selected from the N workers (i.e. one mapper and one reducer) to be faulty. Whenever a fault occurs, the Master will deal with it by restarting the failed worker.
To deal with stragglers, user can define the time limit for each worker, beyond which the worker (mapper or reducer) is stopped. In this library, it has been set to a default of 30 seconds.

Test Cases:

Three test cases have been taken, upon which MapReduce jobs will be performed. The test cases which contain the business logic would be executed when we run the MapReduce source code. The user defined test cases are Movie Count, Movie Rating and Movie Release Frequency. For each of these test cases, there are three classes, which are the test class with the main method running the entire logic of the test case. This is done by invoking the user defined Mapper and Reducer functions. The user defined mapper functions are invoked to run on each row of the partition assigned to it. The output of this user defined map function is then stored in an intermediate file present in the local file system. This is in the form of a key and a list of values associated with each key. Once all the mapper functions are executed, the reducer will start. Each user defined reducer function handles the information belonging to a single key only and then it writes the output to a separate file. The final output of the MapReduce job is to group all the outputs together from the reducer.

1. **Movie Count**
    - The test case Movie Count gives the number of occurrences of each movie present in the file “movie count.txt”.
    - There are a total of three classes present which are MovieCount, MovieCountMapper and MovieCountReducer.
    - The MovieCount class builds on the MapReduce and MapReduce specification classes. In this class, the number of processes which are two, the input file location as well as the location where the output must be stored are specified. The MapperKey as well as the ReducedKey are also specified.
    - The MovieCoutnMapper class extend the MapperCount class and implements the Mapper and Serializable interfaces.
    - The MovieCountMapper class is executed over a portion of the input file. It gets its input in the form of a key and value pair.
    - In the MovieCountMapper class, the input is converted to lowercase and then split into movies based on new line character.
    - HashMap using MD5 is used which has a Key of String Data Type and a value of List of Strings.
    - Each time a movie occurs in the input, a “1” is appended on to the value of that particular movie.
    - The output of the MovieCountMapper class consists of a key which is a unique movie in the input and it’s corresponding value which is a list of the occurrences of that movie in the input.
    - This output is then stored in a temporary intermediate file and then the MovieCountReducer class takes the value for that particular key as input and deletes the intermediate file.
    - The MovieCountReducer class then aggregates all of the values corresponding to that particular key and then will give the output as a single key value pair, with the movie as the key and the count as the value.

2. **Movie Rating**
    - The test case Movie Rating gives the rating for each of the movie present in the file “movie rating.txt”. The rating for each movie is calculated out of 100.
    - There are a total of three classes that are present. These are MovieRating, MovieRatingMapper and MovieRatingReducer.
    - The MovieRating class builds on the MapReduce and MapReduce specification classes. In this class, the number of processes are three. Also, the input file location and the location where the output is to be stored are also specified.
    - The MovieRatingMapper class extends the MovieRating class and implements the Mapper and Serializable interfaces.
    - The MovieRatingMapper takes in the input and converts it to lower case. It then splits the input based on the presence of a new line character. From each line it then splits based on the presence of a colon (‘:’) and then creates a key value pair of the movie, with its corresponding rating.
    - The output of the MovieCountMapper is then stored in an intermediate file. The MovieCountReducer takes the input corresponding to a key as the input value. HashMap using MD5 is used which has a Key of String Data Type and a value of List of Strings. Once the MovieCountReducer takes in the value, the intermediate file is deleted.
    - The MovieCountReducer then sums the value (i.e. the rating) with the corresponding value for that particular key (i.e. the movie). It then formats all the values corresponding to that key and stores it into the corresponding output file.

3. **Movie Release Frequency**
    - The test case Movie Release Frequency gives the count of the number of movies released in different years as per the input file “movie release frequency.txt”.
    - There are a total of three classes that are present which are MovieReleaseFrequency, MovieReleaseMapper and MovieReleaseReducer.
    - The MovieReleaseFrequency class build on the Mapper and Map Reduce specification classes. In this class, the number of processes are three. Also, the input file location and the location where the output is to be stored are specified.
    - The MovieReleaseMapper class extends the MovieRelease class and implements the Mapper and Serializable interfaces.
    - The MovieReleaseMapper takes in the input and converts it to lower case. It then splits the input based on the presence of a new line character. From each line it then splits based on the presence of a colon (‘:’) and then creates a key value pair of the year, with its corresponding movie name.
    - The output of the MovieReleaseMapper is then stored in an intermediate file. The MovieReleaseReducer takes the input corresponding to a key as the input value. HashMap using MD5 is used which has a Key of String Data Type and a value of List of Strings. Once the MovieReleaseReducer takes in the value, the intermediate file is deleted.

- The MovieReleaseReducer then increments a unique counter every time, the input value has a key with a particular year. It then formats all the values (i.e. the count of the movies for that particular year) corresponding to that key (i.e. the year) and stores it into the corresponding output file.

Code Execution:

To execute the three test cases, the shell script "run-test-cases.sh" should be run from the source directory. This compiles all the library code with the test cases and then runs them.

Execute *./run-test-cases.sh* or *sh run-test-cases.sh*

The output of the MapReduce Library for the above three test cases, can be verified with the output from the PySpark implementation of the same. The PySpark implementation is present in the src/spark folder. Each python corresponds to a test case.

Execute *spark-submit filename.py*

On running the bash script, *OutputComparison.py* is also executed. This compares the output of MapReduce Library and the PySpark implementation for each of the test case and tells if the results are matching or not, which can be seen as output in the terminal.

Note – The code is best implemented on a Mac system.

Final Output:

The MapReduce library will give N output files, based on the number of processes. The PySpark implementation will give one output file for a single test case execution.

The output for the test cases are present in docs/ directory. The output for the PySpark implementation is present in the "spark_output" folder in the root directory.