Project 1: MapReduce Design Document

Team Name: Team 9A
Team Members: Neeharika Karanam, Roshitha Bezawada, Sirisha Annamraju

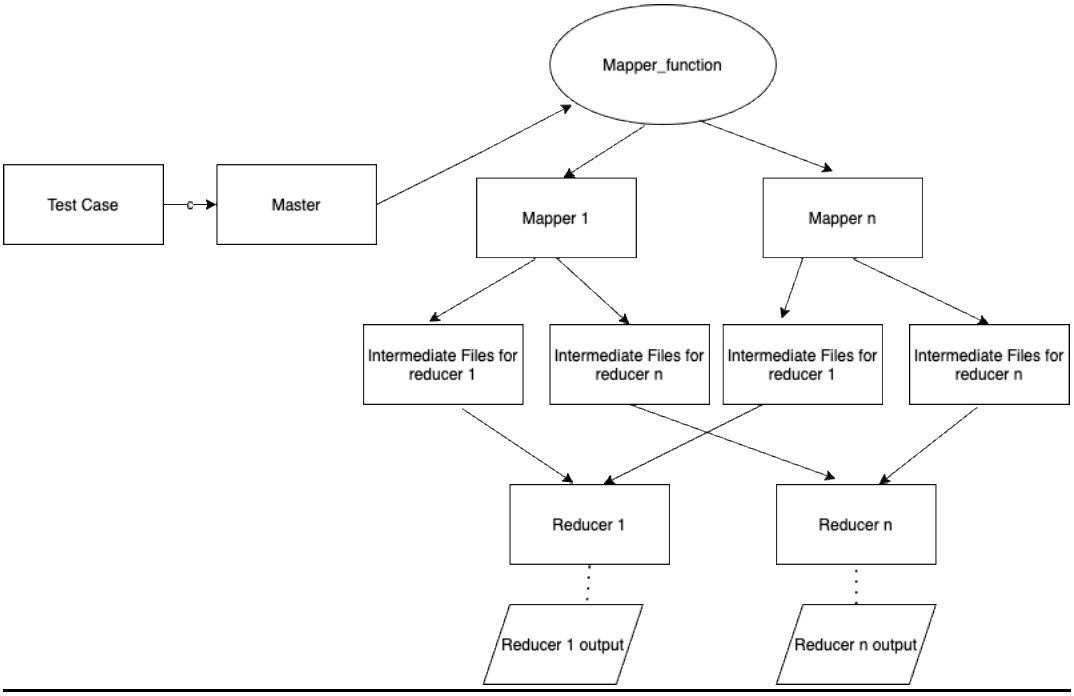
System Requirements

- Need to have JDK version 17 or above, Java SE 13 or above, Python3 and Pyspark installed in the system for the application to run.

Overview

MapReduce can be described as a processing technique as well as a program model for distributed computing. There are two important tasks in MapReduce are Map and Reduce. The main idea is to map the data into a collection in the form of a <key, value> pairs, and then we reduce the values over all the pairs with the same key. We have implemented the MapReduce framework using Java where the Map and Reduce functions will perform the MapReduce functionality on a distributed system for the user defined functions.

The control flow diagram of the MapReduce framework



Overall Program Design

As per the MapReduce architecture the user will define a function which triggers the master which contains both map and reduce script. The master will partition the given input files into multiple partitions and trigger the N mapper process which will invoke the user defined function for each

of the test case. Once all the N mapper processes are done executing then the intermediate output of the mapper will be provided to the N reducers where the output of each of the reducer is written to a separate file.

Input and output of the map function → <k1, v1> -> Map () -> list (<k2, v2>).
Input and output of the reduce function → <k2, list(v2)> -> Reduce () -> list (<k3, v3>)

How it works and Design Tradeoffs

The MapReduce job will split the input data files into various independent chunks which will be processed by all the different mappers tasks in a parallel fashion. In this framework the outputs of the mapper are sorted, and these sorted outputs are provided to the reducer tasks. The master will help start each of the mapper as a new process. Firstly, the RMI registry is started which helps to facilitate a communication between the master, the mapper, and the reducer. The master will add the UDF's into the registry. The mapper function will then launch the N mapper processes and it provides the intermediate files. Once all the mappers have been completed then the master will invoke the N reducer processes that will take the intermediate files as the input and then write to the N output files.

If the framework consists of N mappers and N reducers, then the mapper will generate N*N intermediate files in N different directories which will be the input for the N reducers and the reducer will provide N files as the output. For instance, when the first mapper has finished it will create an intermediate directory named as the "Reducer_1" which will have all the intermediate files created by the first mapper. In the same way the N mappers will produce N directories. Each of the directory is passed as an input to the reducer and the reducer will provide one output file for each of the directory i.e., it will have N output files in the end. In this design the master doesn't explicitly assign the intermediate files to the reducers whereas, the master will provide the locations of the N intermediate directories consisting of the intermediate files to the reducers. The major tradeoff of this framework is that the mapper produces N*N intermediate files.

In brief, the Master will invoke the Mapper process as well as the Reducer processes and these processes will in turn call the user defined map and reduce functions for the different test cases to run the application.

Parameters

Master

Along with the User Defined functions for map and reduce the master also needs certain other parameters as the input and they include.

- Number of mapper processes (N): The total number of mapper processes.
- Number of reducer processes (N): The total number of reducer processes.
- Input filename for the master: It is the input file for the master to perform the MapReduce operation.

- Output file directory for the reducer: It is the output file directory location for the reducer to save the outputs.
- Mapper object: It is the object of the user defined map class which will in turn be added to the RMI registry where the Master and the Mappers will be able to access it.
- Reducer object: It is the object of the user defined reduce class which will in turn be added to the RMI registry where the Master and the Reducers will be able to access it.

Map

The input parameters have 2 values: String of key and String of value. It will provide an output in the form of an HashMap of String and a list of String

Reduce

The input parameters have 2 values: String of key and a list of String of the values. It will provide an output in the form of a list of String.

Shuffling of keys

The mapper uses hashing of a key to an integer from 1 to N to determine which intermediate file to write a particular key. This will ensure that the various mappers will write to the same intermediate file in its respective remote directory. For instance, when the mapper 1 and mapper 2 get a key say "this" then it will get hashed to the number 1. As this key has been hashed to the number 1 then the mapper 1 will write it to "reducer_1/epochTime_intermediate_process_1.txt" and the mapper 2 will write to "reducer_1/epochTime_intermediate_process_1.txt". So, the reducer 1 will operate on all of the occurrences of the key 'this'. Therefore, there is no shuffling of keys amongst the reducers will be required.

Fault tolerance

The Straggler problem has been considered as the fault at the reducer end. Stragglers is the situation where one worker will take very long to complete. We consider a N reducer to be taking more time than required to replicate the straggler effect. A time limit of 5 seconds has been set as default and if this time limit has been exceeded then the reducer process will be destroyed and restarted.

Test Cases

To perform the MapReduce framework, we have considered 3 different user defined test cases namely WordCount, SearchWord and WordLength. These test cases will consist of the business logic that would be executed when the Master is being executed. Each of these user defined test cases will consist of three classes: main class, map class and a reducer class. The main class will run the test case by calling the Master class which will in turn invoke the user defined map and reduce functions.

1. **WordCount:** The basic idea of the test case WordCount is to count how many times a particular word is present in the set of input files.

    - The three classes are WordCounter(main), WordCounter_map(map), WordCounter_reduce(reduce).
    - The WordCounter is the main class which will run the Master class. We then define the number of mapper processes as 4, the input file location, the output file directory and also define the user defined mapper and reducer functions.
    - The WordCounter_map will extend the Mapper_obj class which consists of the user defined mapper function which is executed over a partition. The inputs are in the form of key and value pair.
    - The output of the mapper function is in the form of a Hashmap which will consist of the word as the key and a list consisting of the number of occurrences of the word. This output is written to the intermediate files of the associated reducer directory.
    - The WordCounter_reduce class will extend the Reducer_obj which will then sum up all the values that are corresponding to a key and will then produce an output of a single key value pair where the key is the word, and the value is the count of the occurrences.
    - The reducer function will create a directory "reducer_output_WC" which is specific to WordCount is created under the output file directory and write the output into "reducer_output_WC".

2. **SearchWord**: The basic idea of the test case SearchWord is to check in how many files a particular word is present in the set of input files.

    - The three classes are SearchWord(main), SearchWord_map(map), SearchWord_reduce(reduce).
    - The SearchWord is the main class which will run the Master class. We then define the number of mapper processes as 4, the input file location, the output file directory and also define the user defines mapper and reducer functions.
    - The SearchWord_map will extend the Mapper_obj class which consists of the user defined mapper functions which is executed over the partition. The inputs are in the form of a key and value pair.
    - The output of the mapper function is in the form of a Hashmap which will consist of the word as the key and a list consisting of the file names which consist of the word. This output is written to the intermediate files of the associated reducer directory.
    - The SearchWord_reduce class will extend the Reducer_obj which will then sum up the number of occurrences of the word in different files that are corresponding to a key and will then produce an output of a single key value pair where the key is the word, and the value is the count of the occurrences.
    - The reducer function will create a directory "reducer_output_SW" which is specific to SearchWord is created under the output file directory and write the output into "reducer_output_SW".

3. WordLength: The basic idea of the test case WordLength is to check how many words are there of a particular length in the set of input files.

    - The three classes are WordLength(main), WordLength_map(map), WordLength_reduce(reduce).
    - The WordLength is the main class which will run the Master class. We then define the number of mapper processes as 4, the input file location, the output file directory and also define the user defines mapper and reducer functions.
    - The WordLength_map will extend the Mapper_obj class which consists of the user defined mapper functions which is executed over the partition. The inputs are in the form of a key and value pair.
    - The output of the mapper function is in the form of a Hashmap which will consist of the length of the word as the key and a list consisting of the words which are of the same length as that of the key. This output is written to the intermediate files of the associated reducer directory.
    - The WordLength_reduce class will extend the Reducer_obj which will then sum up the number of words of the same length in different files that are corresponding to a key and will then produce an output of a single key value pair where the key is the word, and the value is the count of different words.
    - The reducer function will create a directory "reducer_output_WL" which is specific to WordLength is created under the output file directory and write the output into "reducer_output_WL".

How to run

A single shell script to run all the test cases is in the root directory name "run_me.sh". This will help in executing the user defined test cases implementing the MapReduce framework and will run them.

./run_me.sh

For executing the spark implementation of the test cases which are present inside Spark/Code.

Python3 filename.py

Outputs

The MapReduce framework will provide an output of N files, which is based on the number of processes. The spark implementation will provide one single out file for each of the test case.

The output for the MapReduce framework is present under the "Reducer_output" folder and the individual test case outputs are under:

- "Reducer_output/reducer_output_WC" (for Word Counter)
- "Reducer_output/reducer_output_SW" (for Search Word)
- "Reducer_output/reducer_output_WL" (for Word Length)

The output of the spark implementation of the test cases is under “Spark/Output_files” folder. All the output folders are present in the root directory.

Since we have one faulty reducer for each of the test case which will lead to a delay of 5 seconds for each of the test case. Therefore, the total time required to run all the test cases will be around 20-30 seconds.