CS532 Project 1: Map Reduce

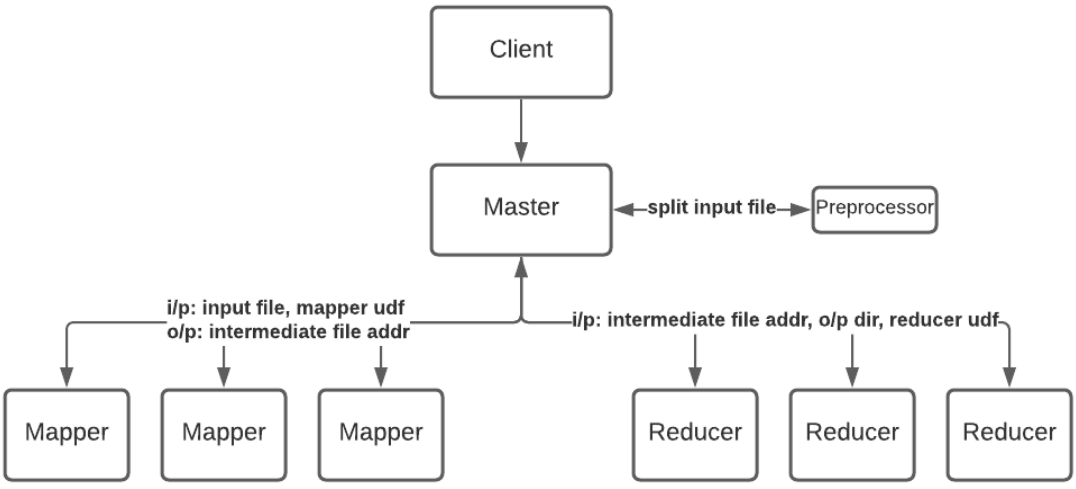
Design Document

Group Members:

- Pragya Sarda
- Priyanka Singla
- Sai Rishita Golla

System Overview:

There are 4 components in the system: Master, Preprocessor, Mapper, and Reducer.
The client (test case like word count in our case) sends the input file, the mapper udf and reducer udf to the master. Master calls the preprocessor to split the input file into n files (n = number of mappers/reducers set by client). The master then calls the multiple mappers and sends the preprocessed input file to each mapper which produces a list of key, value pairs. Multiprocessing in Python is implemented to simulate the distributed environment of the MapReduce system. Once all the mappers are executed the master calls reducers to complete the reduce phase to output the final key, value pairs. The system is fault tolerant towards one worker failure. Choosing Python over Java was a design tradeoff that was made during the design phase of our map reduce implementation. Although the multiprocessing library of Python is quite new and provides less functionality as compared to Java's, we made a tradeoff to choose Python for quicker development and simpler code base.



Master:

Client invokes the master to execute the map reduce task. The client sends the input file location, output file directory, number of mappers/reducers, mapper udf and reducer udf to the master. Master calls the preprocessor to split the input file into n files (n = number of mappers/reducers). The master then calls the n mappers, sending one file to each mapper. This is done parallely using the Multiprocessing. In case of failure of a mapper, master restarts that particular mapper with the input file corresponding to that mapper. Once the mapper phase is complete, the reducer phase is started. The master starts *n* reducers in parallel and similar to the mapper phase, it restarts a reducer in case of failure (only if the output file has not been generated by the reducer). It returns to the client once the execution is completed.

Mapper Worker:

Mapper worker reads the part of the file assigned to it by the master. It takes udf mapper function, number of reducers and intermediate directory path from master. It executes the udf mapper function over the file chunk and fetches all the < K, V > pairs. It then takes the hash of the key so that each key is mapped to only one reducer by all mappers. While mapping these keys to reducers, a list of hashed reducers is being maintained. The result of key value pairs are then written in intermediate files following the below mentioned directory structure.

```
/intermediate_dir/Reducer_id/Mapper_id.pickle
```

Each reducer will have a separate directory in which all mappers will write < K, V > pairs for that particular reducer. So there will be an R number of folders in the intermediate directory, where R is the number of hashed reducers. Each R folder would have M files produced by each of the mappers.

For fault tolerance, whenever mappers dies or does not respond, it will be restarted even if it completed the execution.

Mapper notifies the master about the intermediate file locations and list of hashed reducers it mapped the keys to.

Reducer Worker:

Master invokes the reducer by sending the intermediate file locations.

The data corresponding to each reducer is parsed by reading all the files in the intermediate directory. A dictionary is then created with a unique key and list of values as follows - [{k1:[v1,v2,v3]}, {k2:[v1,v2,v3]}, {k3:[v3]}]. For each of these unique keys a reduce function is executed which takes a user defined reduce function as the input parameter. The executed function returns the key and reduced output computed based on the list of values. The output for each reducer is written in a separate file. Following is the structure of the output file for each reducer which contains the final key, value pairs.

```
/output/test_folder/reducer_id-timestamp
```

In case of reducer failure, the worker is restarted and the reduce function corresponding to the worker id is re-executed.

Test Cases:

Three test cases have been implemented to test the MapReduce system - word count, sum per customer, and words with length.

Word Count

MapReduce returns the frequency of each word occuring in the input file.

Sum per Customer

For this test, the input file contains the amount spent by a customer on various orders. The resulting output file contains the customer id as key and sum of all the purchases for a given customer as the value. That is, the MapReduce system is used to sum all values for each given key.

Words per Length

Test case to find how many words are there for a certain length. For example,
Input: [Leonard, Sheldon, Amy, Penny, Raj, Bernadette, Howard]
Output:

[7:2] #words of length 6 are: Leonard, Sheldon
[3:2] #words of length 3 are: Amy, Raj
[5:2] #words of length 5 are: Penny, Howard
[10:1] #words of length 10 are: Bernadette

Run the code:

In MapReduceBazinga, test script named “run_tests.sh” will run all test scripts for all three test scenarios:

- With a single process
- With multiple processes
- With multiple processes and one fault