Project 1

MapReduce Implementation

Supreetha Bangalore Somasundar

Rachana Narayan Acharya

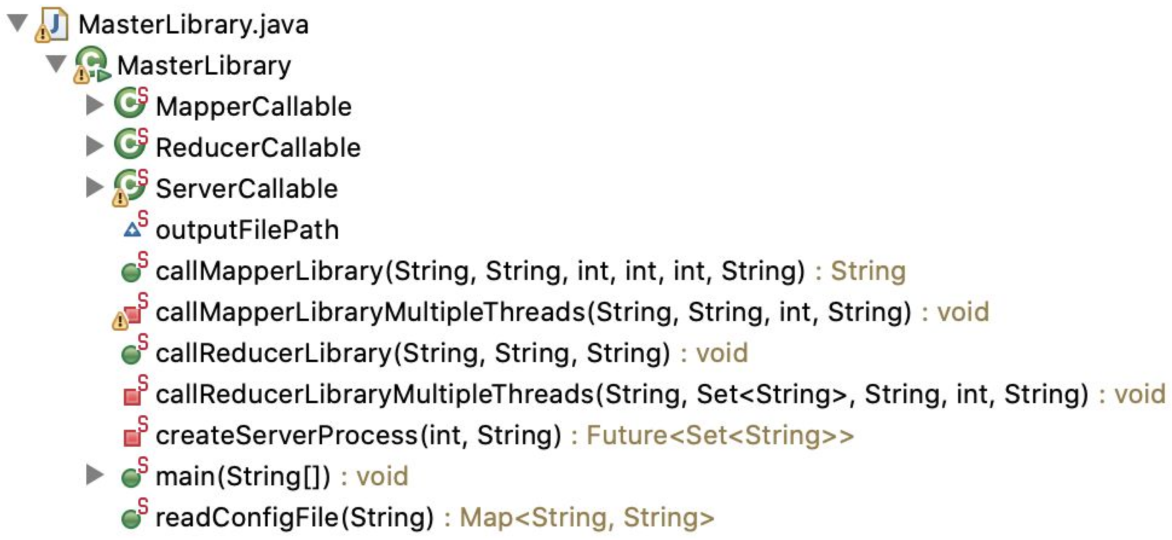
Smriti Murali

27th October 2020

Design and Working:

Master:

1. MasterLibrary.java:



```
MasterLibrary.java
  MasterLibrary
    MapperCallable
    ReducerCallable
    ServerCallable
    outputFilePath
    callMapperLibrary(String, String, int, int, int, String) : String
    callMapperLibraryMultipleThreads(String, String, int, String) : void
    callReducerLibrary(String, String, String) : void
    callReducerLibraryMultipleThreads(String, Set<String>, String, int, String) : void
    createServerProcess(int, String) : Future<Set<String>>
    main(String[]) : void
    readConfigFile(String) : Map<String, String>
```

The main method in this class is the beginning of the execution of this project. To make this project multi-process, we create thread pools and assign each process(mapper/reducer) to one particular thread.

a. MapperCallable: Creates threads that execute mapper processes
b. ReducerCallable: Creates threads that execute reducer processes

Fault tolerance is achieved by checking the exit code of the executed process. If the process turns out to have failed, we re-execute in the same thread.

To read the intermediate file names from the mappers, we make use of Sockets. Here again, we create multiple threads, each reading the output of one single mapper/reducer. The aggregated list of intermediate files is then sent to the Master.

a. ServerCallable: Creates multiple threads, each reading the output of one single process(mapper/reducer).

Mapper:

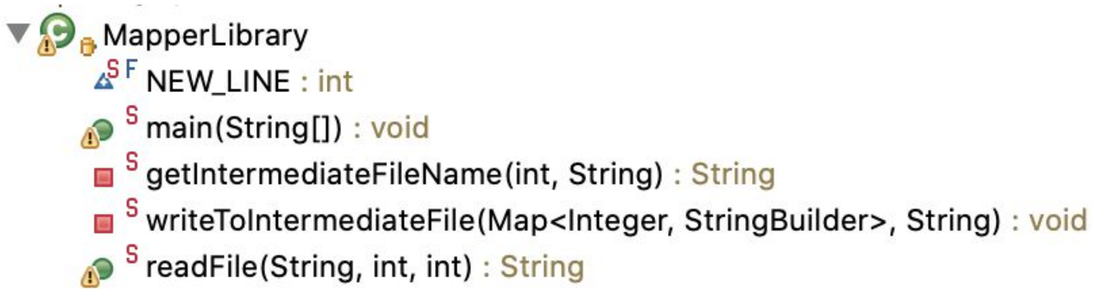
1. ### Mapper.java:

   We have defined a Mapper.java interface that has to be implemented by every Mapper UDF. It contains the definition of a map function that is to be implemented.

   ```
   public List<Pair<String, String>> map(String key, String value);
   ```

   The map function takes in a key and a value. The key can be null. It generates a list containing processed key-value pairs. It is analogous to the map function in the paper which defines the function as (k1, v1) => list (k2, v2).

2. ### MapperLibrary.java



```
MapperLibrary
  NEW_LINE : int
  main(String[]) : void
  getIntermediateFileName(int, String) : String
  writeToIntermediateFile(Map<Integer, StringBuilder>, String) : void
  readFile(String, int, int) : String
```

   The library handles all the responsibilities of reading the partition of the file and processing each row of the partition. The mapper UDF class, the file name, and the offsets of the file partition which the mapper instance is required to process are passed to the main function. The partitioning logic is present in readFile function and we are reading complete rows of the file. Then each of the rows in the partition is passed to the mapper UDF map function using Java reflection. The output is maintained in a buffer and finally, it is written to the intermediate files. The number of intermediate files generated depends upon the number of reducer processes. The output generated by the UDF map function is mapped to one of the intermediate files by hashing the key and computing the modulus with the number of processes.

3. ### Mapper UDF:

   There are 3 mappers UDF's defined for 3 applications.

   - MapperUDF: Word-Count application. Removes punctuation and splits the line on spaces. Return (word, 1) for each word as the output from the function.

- MapperInvertedIndexUDF: Inverted-Index application. Removes punctuation and splits the line on spaces. Returns (word, docId) for each word as the output from the function where docId is the scene in which it occurs.
- MapperEmployeeSalariesUDF: Average Salary across ages application. Reads a csv row and splits the row on a comma. The fields containing the age and the salary are fetched. Returns (age, salary) for each row as the output from the function.

Reducer:

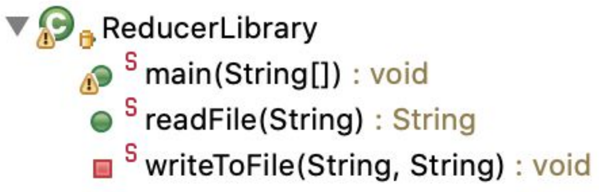
1. Reducer.java:

We have defined a Reducer.java interface that has to be implemented by every Reducer UDF. It contains the definition of a reduce function that is to be implemented.

```
public List<String> reduce(String key, List<String> values);
```

The reduce function takes in a key and list of values associated with it. It processes the list associated with a key by computing the sum or average and returns the output in the form of a list of strings. It is analogous to the reduce function in the paper which defines the function as (k2, list (v2)) => list (v2).

2. ReducerLibrary.java



```
ReducerLibrary
    S main(String[]) : void
    S readFile(String) : String
    S writeToFile(String, String) : void
```

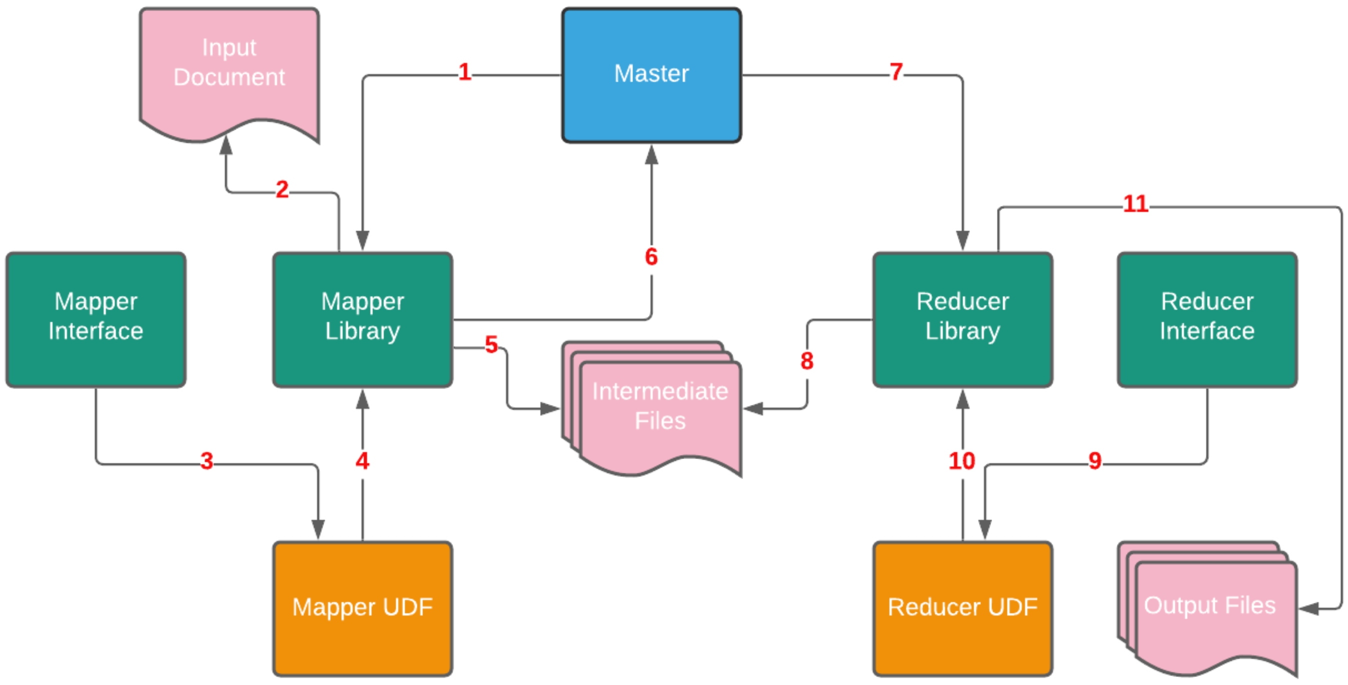
The library handles the responsibilities of reading the intermediate file, grouping by keys, and calling the reduce function for each of the keys. The reducer UDF class, the intermediate file name, and the output file path on which the reducer instance is required to work on are passed to the main function. The readFile function reads the passed intermediate file. A hashmap is constructed to maintain the mapping of a key to its values. The key and its values are passed to the reduce function defined in the reducer UDFs. The processed output returned for the key is then written to the output file in the writeToFile function.

3. Reducer UDF:

There are 3 reducer UDF’s defined for 3 applications.

- ReducerUDF: Word-Count application. The key is now a word. The values consist of all the occurrences of the word. The values are summed up and returned. This gives the frequency for each word.
- ReducerInvertedIndexUDF: Inverted-Index application. The key is a word. The values consist of all the document ids the word occurs in. If the document contains many occurrences of the word, the list of values would contain duplicate values. The reducer removes the duplicate values and returns the list.
- ReducerEmployeeSalariesUDF: Average Salary across ages application. The key is the age. The list of values would contain different salaries obtained by employees of a particular age. We summed up all the salaries and divided by the number of values to obtain the average salary.

How does it work?



Steps:

1. The master calls the mapper library with the appropriate parameters.
2. The mapper library instance reads the assigned partition from the input document.
3. The mapper UDF implements the map function defined in the Mapper Interface.

4. The mapper library calls the mapper UDF by passing each row of the partition to the map function.
5. The mapper library then generates the intermediate files from the outputs of the mapper UDF.
6. The mapper library then notifies the master of its completion and the path of the intermediate files.
7. The master calls the reducer library with the appropriate parameters.
8. The reducer library reads the assigned intermediate file.
9. The reducer UDF implements the reduce function defined in the Reducer Interface.
10. The reducer library calls the reducer UDF by passing the key and all of its associated values.
11. The reducer library then generates the output files from the outputs of the reducer UDF.

Design TradeOffs

1. Ad-Hoc design for the partitioning of files:

   We pass the offsets to the Mapper process which denotes the partition of the file it should work on. The partition sizes are computed by dividing the size of the files with the number of processes. The offsets are computed from that. Hence, the offset might point to any word in the middle of the row. In order to remove any inconsistencies, we have written a code to read complete rows if the offset ends up somewhere in between the row. If the start offset is somewhere in between the row, then we move the start offset to the next row. I feel this is not very scalable or would not work in a distributed system as the mapper instance would not have access to anything ahead of before the file partition than it is supposed to work on.

How To Run?

The script to run the 3 applications is present in src/main/resources/. It is called runScript.sh.

```
sh src/main/resources/runScript.sh
```

The script would run each of the application by passing the respective config file to the master. The mapper runs for each of the application and generates the intermediate files which would be present in the public/ folder. Once the mapper finishes, the reducer runs by reading the intermediate file assigned to it and generates the output files which is also present in the public/ folder.

The output generated is compared to the output generated by the spark code on the same file for the same application. The test script returns appropriate output depending on the comparison. The spark outputs and the spark code are also present in the public/ folder. The test scripts are present in the main project.

Conclusion

This project was a good learning experience to know about how to go on implementing a map-reduce application. We learned how to spawn different processes, reflection in Java, and also how fault toleration works. To go about designing the project, we discussed different ideas and weighed the pros and cons to see which one would work best.