Design Document - MapReduce

Disha Singh | Pranjali Ajay Parse

1. Program Design

The following is a summary of the various groups that are used in the project:

- **MapReduce.java**: Contains the mapreduce interface that the test cases implement
- **Master.java**: This file contains the implementation of the master. This is where the workers are created, as well as the global synchronization barrier. For the mapper and reducer phases, we build separate processes.
- **Worker.java**: This file contains the implementation of the workers. Both file reading and writing operations are handled here. For the mapper and reducer phases, we constructed separate processes. This is accomplished using a mechanism known as *heartbeats*, in which all workers send a message to the master at regular intervals (2 seconds), and the master assumes a worker has crashed if no message is received during that period.
- **WorkerHeartBeat.java:** Extends the thread class, that implements the heartbeat mechanism, allowing workers to send messages to the master.
- Test cases that implement this interface — **WordCount.java, URLFrequency.java, and DistributedGrep.java**

2. The User Defined Functions (UDFs):

For MapReduce, the existing implementation supports multiple workers. It supports the following test cases:

- **Word Count [WordCount.java]**: Count the frequency of each word in the user input file **[input/wordcount_input.txt]**
- **Distributed Grep [URLFrequency.java]**: Find the occurences of the word "distributed" in the user input file **[input/distributedgrep_input.txt]**
- **URL Frequency count [DistributedGrep.java]**: Count the frequency of each URL in the user input file **[input/urlfrequency_input.txt]**

3. Systems Specifications:

- Windows - 64 bit system
- Gradle 6.7 - the environment path variable also needs to be set (Download: link)
- JAVA (JDK 14) (Set the path of JDK in gradle.properties file)
- JAVA_HOME: path variable set in the environment variables.

4. How to run the project and tests:

To run project and test scripts together:

Go to the main project directory and simply double click on the file:
DOUBLE_CLICK_TO_RUN_PROJECT.bat

To run this project, simply follow these instructions:

```
./gradlew run
# runs three UDFs on our implementation of the MapReduce Framework by starting the RunUDFs.java in start folder
```

To start the test script, use the following command:

```
python test/testScript.py
# runs three tests comparing output of map reduce for the three UDFs with actual expected output
```

To run the implementation for your own UDF:

- Add the mapper function and reducer function in the src/main/java/udf folder
- Add config file with no of workers you need and the location of input/output/intermediate files in the metadata folder
- Add starter code in start/runUDFs.java file

5. How the tests work:

The tests are started by running testScript.py present in the test folder. This file then runs **wordCountTest.py, distributedGrepTest.py** and **URLFrequencyTest.py**. Both of these test files use ***actual output*** to refer to the output of our MapReduce program, and ***expected output*** to refer to the actual results used for verification. The respective test python files compute these predicted results. In this way, we can see if our distributed task implementation produces the same results as a non-distributed, single-process task implementation.

Note: Since the config files are located in the root directory of the project, the tests must be run from there rather than from the test folder.

6. How the implementation works:

The following function declarations are part of the MapReduce interface that we implemented:

- The mapper function of each worker takes a row from the data partition allocated to that worker and outputs a list of key value pairs, which are then written to intermediate files.
- The reducer function takes a list of values for a specific key and returns a string as the final result. This values list is created after the combiner function is applied to the intermediate file data.
- The getSeparator function is used to obtain the separator that is used to separate rows in the input for each test case.

The execution begins with the formation of a Master process. The master refers to the given config file, which contains the desired locations for the input, output, and intermediate files, as well as the value N, which specifies the number of workers. After that, the master takes an object from the test case that implements the MapReduce interface defined by our library, serializes it, and saves it to a file. The worker processes are then given unique identifiers. The master creates a socket to facilitate communication with all of the workers, and the master is constantly looking for messages from the workers. All of the workers are given the file containing the serialized object, as well as the communication port and file paths. The workers are all set up to start with the map process, and a global barrier is set up on the master side. The input files are used as inputs, and the intermediate files are used as output files in this process. The workers read-only access the serialized object, deserialize it, start the worker heartbeat thread, and then use that object to access the user-defined map function. Following that, the task defined in the UDF is executed. Each worker completes its task by sending a message to the master on the specified port, indicating that it has completed it. To ensure global synchronicity, the master acknowledges and keeps a count. The first step ends when all of the workers have given the master a message indicating that their tasks have been completed.

After that, the second phase begins, with workers being created and initialized for the reduce phase. The same procedure as before is repeated, with the workers now serving as reducers. The input files for this process are the intermediate files, and the output files are the final output files.

Note: We recognize that the serialized object file would not be accessible to all workers in a pure distributed manner, but we can still submit the serialized object to the worker nodes. As a result, the code can be generalized with only minor changes.

7. Design choices & Tradeoffs:

We needed the heartbeat mechanism between our master and workers to enforce fault tolerance. It could be implemented in two ways: the client could send heartbeats at regular intervals, or the master could inquire about the status at regular intervals. We preferred the choice of having the worker send the heartbeat because it needed a smaller number of ports. In the global barrier stage, the worker will simply connect to the port that the master is listening to and write the message there. Also, after completing milestone 2, this was a more viable choice to pursue. We had to come up with a threshold after which the master decides that the worker process has died and respawns it when enforcing the heartbeat mechanism. We assume there will be no network delay in our system, so we raise the limit to 2000 milliseconds, which is the time it takes for workers to send heartbeats. In a distributed system, this value may be changed depending on the network latency.

We had the option of either writing the combiner output to files or keeping it in memory. We chose the in-memory combiner output method for this project because we assumed the combiner output would fit in memory. This also aided in determining when the combiner function should be called. It can be called either after or just before the reducer phase. Calling before the reduce process was the best choice since we were using an in-memory approach.

We were posed with the question of how to generalize row separation for various types of resources when implementing different test cases. For eg, a row can be anything separated by whitespace in a word count, but it is typically a new line character in URL Frequency. As a result, we added a new method to the MapReduce.java interface called getSeperator() that returns the separator for each test case.

We chose approaches that favor ease of execution in several areas of the project. These could, in theory, be made more functional. As a result, we've made minor productivity tradeoffs in exchange for convenience.