MapReduce: Design Document

Team 9

- Parin Jhaveri (32601659)
- Tejas Chheda (32673487)
- Sharmistha S. Gupta (32698753)

Introduction

The objective of this project was to implement MapReduce from scratch. MapReduce is a distributed computing model that takes as input a set of data and breaks it down into key/value pairs as an output.
We implement three applications to test the model, namely:

1. **Word Count**: The objective is to output the number of occurrences of each word that appears in a text file. The N output files created have the word as the key and the number of occurrences of the word as the value which is separated by a space.
2. **Inverted Index**: The objective is to output the indices at which each of the words present in a text file occur. In the case of the output for this implementation, the line indices (first line in the input file is considered to have an index equal to zero) are considered to be the document indices. The N output files created have the word as the key and comma separated line indices as the value.
3. **Distributed Grep**: The objective is to look for a pattern in a text file and output the lines that contain the specified pattern. In the UDF, we have hardcoded the pattern to be searched for as “map”. In the output file, any such lines which have the word “map” will be present. The N output files created will have the line as the key and the value here will be empty.

These applications are present as user-defined functions (UDFs) and can be defined in the form of their respective Map and Reduce functions.
We generate the outputs of the above applications using our implementation of MapReduce as well as Python, and then compare the results to determine the correctness of our model.

Detailed Design Description

COMMUNICATION

The Worker processes communicate with the Master via sockets. The Master opens a server socket with only one port number and all the Worker processes connect to that single port as Clients. The Master never sends any data through the sockets, it just listens on that port continuously and waits for the workers to send messages.
The Workers send a WorkerStatus Object which contains the status message (SUCCESS/FAIL), Worker ID, and the path of the file (only in case of Mapper) of where the

intermediate output has been written to. If the status of the message is “FAIL”, then the Master carries out Fault Tolerance and respawns the corresponding worker process.
The Master via the Process library keeps tabs on every Worker Process that is still running and if any of them crashes an EOFException is thrown which is then caught by the Master. The Master then goes ahead to check the one process which has crashed and carries forward the fault tolerance.

MASTER PROCESS OVERVIEW

1. The final script, *test_script_automated.py*, calls the command *"java -cp src/ Main"* through *os.system()*, which calls the main() function residing in Main.java. This in turn invokes the Master.
2. The Master first loads properties from the config file. The properties specified include:
   - number of workers, N
   - the input file path (file from which each mapper will read its own partition)
   - the output file path (directory where each reducer will store its output)
   - the UDF class
   - Whether or not you want to test the application by crashing the worker
   - Whether or not you want to test the application by forcing a runtime exception
3. It then performs a setup and cleanup by deleting and re-creating the intermediate and output directories.
4. Next, it creates a socket server to initiate communication with the workers.
5. **Partitioning:** The master then counts the number of lines present in the input file and partitions them amongst the N number of Mappers. It calculates an offset to determine the maximum number of lines that each worker should read. The last mapper process will be asked to read the rest of the lines which may be greater than the offset. The offset is calculated by rounding down (total number of lines)/N to the nearest integer.
6. The Master then spawns N mapper processes and passes as String arguments that the mappers will need:
   - Worker ID
   - Socket Port to Connect to
   - Worker Type (Mapper/Reducer)
   - Input File Path (Single Input File for the Mapper / Comma Separated String of all the intermediate file paths that a particular Reducer should read from)
   - UDF Class
   - Start Line (Line index the Master should read the input file from)
   - Offset (Number of lines the Master should read from the Start Line)
   - Output File Path (Which Directory the Reducer should write to. Mapper writes to the intermediate directory)
   - Number of Workers, N
   - Which Node/Worker ID should crash (Fault Tolerance Testing, see that section for more details)
   - Boolean Value if we want to test by forcing a Worker to Crash
   - Boolean Value if we want to test by forcing an exception to be thrown

The master assigns a worker id to each of the workers and keeps track of them and its arguments by keeping activeWorkers object which has a mapping of the process spawned as key and WokerDetails as value. The Master acts as a listener for all the workers and communicates with them through socket communication over a single port.

7. Once each mapper is done with their job, the master receives the intermediate file locations from each of the mappers. Each Mapper writes to 'N' intermediate files based on a hash function (further details given below).
8. Once all the mappers are over, these file locations are then mapped and assigned to each reducer by the Master in a way by assigning one reducer one of the 'N' intermediate files that it should read from, from each mapper output (as each mapper writes to 'N' intermediate files)

WORKER PROCESS OVERVIEW

We use the Factory Design Pattern to determine the type of Worker that the Master wishes to spawn by using the WorkerType as the decision condition which is one of the String arguments passed when spawning the Worker Process.

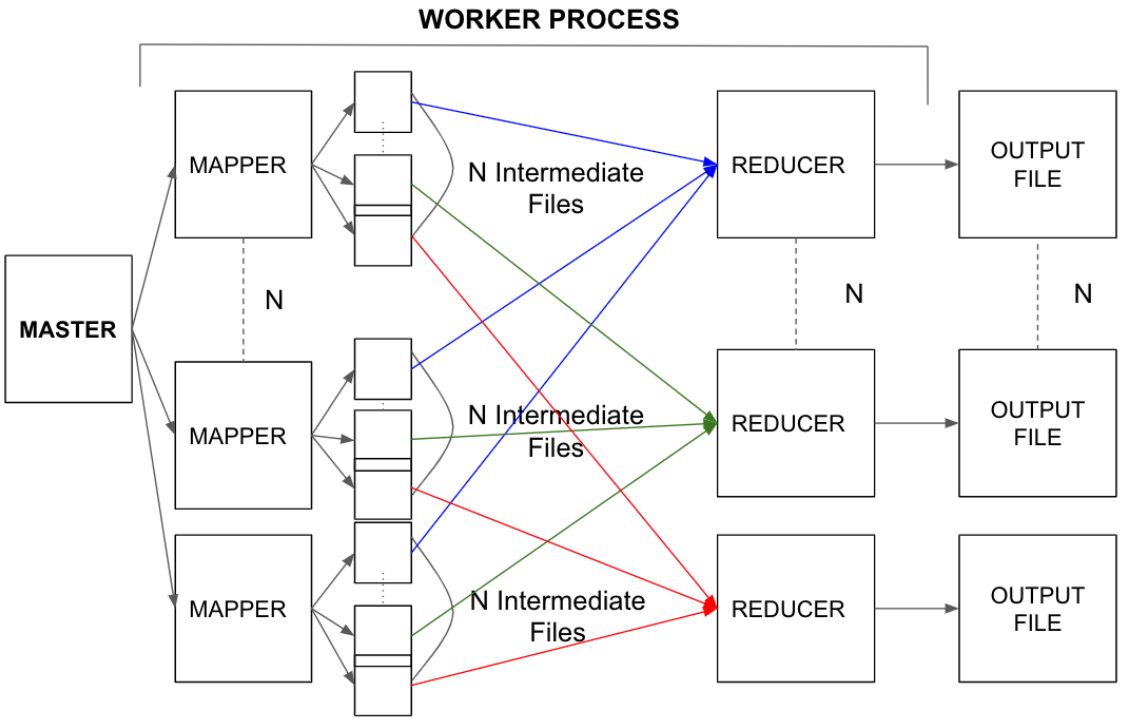
Mapper

1. Mappers establish a connection with the Master using the Port number it received as a String argument. This is done using Sockets in Java.
2. The Mappers reads the partition of the file assigned to them by the master, where the 'startLine' corresponds to the line index of the input file that the Mapper should start reading from, and the 'startLine + offset' is the last line index.
3. The Mappers create a <key, value> pair where the key is the line index and the value is the corresponding line.
4. Next, they invoke the 'map' method in the UDF Class using Java reflection, and the output is an Output object which has a hashmap as its member.
5. **Hash Function:** The output of the UDF is written to intermediate files. A hash function is used to determine which intermediate file to write to. The hash function we have implemented adds the ASCII value of the characters of the key and returns its mod N value as the hash key (where N is the number of reducers). This ensures that each key is assigned to one reducer only. Each mapper writes to N intermediate files based on the key. Thus, a total of NxN intermediate files are generated.
6. The mappers then send the intermediate files to the master through sockets.

Reducer

1. Reducers establish a connection with the Master using the Port number it received as a String argument. This is done using Sockets in Java.
2. Each reducer reads the intermediate files sent by it to the Master. It stores the key-value pairs in-memory.
3. It then invokes the 'reduce' method in the UDF Class using Java reflection, and the output is an Output object which has a hashmap as its member.
4. Each reducer then writes the output to one output file. Thus, 'N' output files are generated.

The figure given below shows the Master, Mapper and Reducer Processes, how each Mapper produces N intermediate files, how each Reducer reads the same indexed file from each of the 'N' intermediate files produced per Mapper, and how the Reducer writes the output finally to the 'N' files.



COMPARISON WITH PYTHON OUTPUT

The output files generated for each of the three UDFs as a result of our MapReduce implementation are compared with the output files generated through a Python implementation which are stored in *src/test_scripts*.

Each of the N workers generate an output file, thus giving us a total of N output files. These files are concatenated into a single file called *outfile.txt* which is stored in the respective UDF directories in *output/*.

A function called *compare()* checks whether all the lines present in *outfile.txt* are present in the output file generated using the Python script and vice versa. If this check passes, our comparison is said to have been successfully completed and the program terminates and the concatenated file, output.txt, is deleted.

Note: If the UDF application is Inverted Index, the ordering of indices might be different in the output files generated using our MapReduce model and the Python implementation. Thus, we use another function called *compare_inverted_index()* which sorts the lines in both the files before comparing them, thus checking only for the presence of the relevant indices and not getting impacted by the order of occurrence.

Fault Tolerance

We have tried to mimic two types of faults:

1. Exception Thrown Due to Runtime Exception
2. Node Crashes

We have tolerated both of these faults for just one node.

Exception Thrown:

- Here we force an exception to be thrown if the key ‘exception_worker’ is set to true in the config.properties file.
- If it is set to ‘true’ then a random number generator will generate an integer in the range of 0 to N-1. This is because our Worker ID for Mapper or Reducer will always be from 0 to N-1. This number generated will be the node for which an exception will be forced to be thrown. Let’s call it node_to_crash.
- This node_to_crash is passed as a String argument to the Worker Processes (Reducer and Mapper part)
- If the node_to_crash is equal to the Worker ID and if the ‘exception_worker’ is set to true, a custom exception is forced to be thrown. This mimics an unfortunate Runtime Exception caused by for example a Heisenbug.
- This custom exception is caught and a ‘FAILURE’ status message is sent to the Master.
- The Master detects this status and then respawns a Worker Process with the same Worker ID.

Node Crashes:

- Here we simulate a node crash by calling the System.exit(0) command to kill the process if the key ‘crash_worker’ is set to true in the config.properties file.
- If it is set to ‘true’ then a random number generator will generate an integer in the range of 0 to N-1. This is because our Worker ID for Mapper or Reducer will always be from 0 to N-1. This number generated will be the node for which an exception will be forced to be thrown. Let’s call it node_to_crash.
- This node_to_crash is passed as a String argument to the Worker Processes (Reducer and Mapper part)
- If the node_to_crash is equal to the Worker ID and if the 'crash_worker’ is set to true, System.exit(0) is executed to kill the process.
- When there is only one Worker left, the Master checks if the process is still alive by calling the Process API isAlive(). If it returns false, that means the process has been killed.
- The Master then respawns the Worker Process with the same Worker ID.

Note: When testing for the fault, then during the time when Mapper Processes are created and when the Reducer Processes are created, both times a particular node will fail.

Design Trade-offs

Following are the design choices made by us for our implementation.

1. **N = Number of workers = Number of partitions of the input file**
   The master calculates an offset to determine the number of lines that are to be read by each worker. This is calculated by rounding off (total number of lines)/N to the nearest integer.
   While this may not work optimally for large input files, we have observed that if for large files there are large number of workers then the load balancing is acceptable
2. **N*N Intermediate Files**
   We decided to have this implementation as this would not cause different processes to write to the same intermediate file. Then we would have to deal with locking issues. We believe by each mapper writing to N intermediate files it would be faster as there would be no threads in the waiting state. Also it reduces the chances of deadlocks.
3. **The master acts as a listener for the workers**
   The master initiates the worker processes and communicates relevant information in the form of system arguments during the initiation. It then keeps track of the worker processes by being an active listener using Sockets.
4. **Mapper Key Input is Not null**
   We require the line indices for the application of Inverted Index UDF. So we pass that always as the key value.
   We had to create these line indices as we just have one single input file
5. **Partitioning Input File Requires Master to Access Entire File to Count Number of Lines**
   This is the only way that we could design so that we can go ahead with the calculation of (total number of lines)/N.
   The other way would be to provide the total number of lines in the config.properties, but that would require too much user input, so we decided against it.
6. **Intermediate data is not Sorted when being Read by Reducer**
   We instead implemented a Hash Map in Output object which could be updated in O(1) time whenever a key is accessed when the Reducer is reading the intermediate files. This complexity would increase if we were to do the sorting too.
7. **We Clean the Input Data of Punctuation Marks and Special Characters in both, the python script and when the Mapper reads a line**
   If we didn't do the cleaning then Spark was creating different outputs then what we expected MapReduce to create. Also we were using commas as delimiters in our intermediate file which could lead to further issues.
8. **Fault Tolerance: Killing Processes within the Library itself**
   If we had killed processes from the python script then we would have to add a lot of delays to the library code.
   Instead, to randomize the process of killing nodes, we have included a random number generator which decides the node we want to force a fault in.

How To Execute

The user must execute the following steps in order to run the MapReduce implementation for a particular application and compare the results with that of a python script.

1. Define the application in the form of a UDF class with a map and a reduce function. The class should be stored in *src/TestCases*.
2. Specify the number of workers, N, the input file path, the output file path and the UDF class in *resources/config.properties.*
3. Write a python script for the application, generate an output file and store it in *src/test_scripts.*
4. In case the input file (*data/demo.txt*) needs to be changed for different inputs, the python scripts in test_scripts/ must be re-executed to generate the updated outputs for comparison with the Java Map Reduce outputs (Example commands to be run from test_script directory: *python3 word_count.py, python3 inverted_index.py, python3 distributed_grep.py*)
5. In the script *test_script_automated.py,* append the new Test Case in the udfList variable.
6. Run the command *python3 test_script_automated.py* in the terminal from the root directory.