DISTRIBUTED SYSTEMS ASSIGNMENT-2 MAP REDUCE REPORT

SUBMITTED BY - RAHUL GUPTA

INTRODUCTION

In this assignment, we will be designing and implementing a distributed MapReduce system. The code implements MapReduce as a "library" that higher level applications can then use for data processing.

DESIGN DETAILS

Files in working directory

- config.yml - config file storing all necessary variables
- input.txt.py - input file containing text
- Output.txt - to store the final output of the map reduce framework
- Requirements.txt - txt file with the required dependencies
- Master.py - python file implementing master in the map reduce framework
- Server.py - server side python file for key value store
- Store.txt - txt file for key value store.
- Mapper subdirectory - contains mapper.py file with mapper python code and the intermediate mapper files are also made in this folder
- Reducer subdirectory - contains reducer.py file with python code and the intermediate reducer files are also made in this folder
- testCase_1.py - config file changes according to test case 1 mentioned in file
- testCase_2.py - config file changes according to test case 2 mentioned in file
- testCase_3.py - config file changes according to test case 3 mentioned in file
- testCase_4.py - config file changes according to test case 4 mentioned in file
- testCase_5.py - config file changes according to test case 5 mentioned in file

This code is compatible with python 3, dependencies are to be installed from requirements.txt.

Working Details -

- **For Test Cases -** run the testCase file - it sets config to details mentioned. Run server.py. Run master.py in a separate terminal.
- **For other cases -** change the config file accordingly and repeat same steps with server.py and master.py

Design -

- Master starts and splits the input file into chunks for each mapper.
- Master spawns mapper to work on each chunk parallelly.
- Mapper writes intermediate output to KV Store and returns control back to master
- When all mappers are done, master spawns reducers
- Reducers read mapper outputs from KV Store, apply the hash function and include selected keys in their inputs.
- Reducers output the processed file in their intermediate files.
- Master then combines the intermediate reducer output files.

Master Details -

For implementing the master, first I have made the imports necessary. I read the config file to get variables like port number and count of reducer and mappers.

I initialized a socket to connect to the key value store. I used the setup_mapper and setup_reducer function to create intermediate files according to the number of mapper and reducer.

I also removed intermediate files if existing from previous runs to avoid unexpected errors.

Master then splits input file to intermediate mapper inputs. I use the subprocess module to spawn mappers as processes and pass file paths and id as command line arguments, store them in an array and use a communication function to wait for all mappers to complete. Fault tolerance is also taken care of here which is mentioned in the Fault tolerance section below.

When mappers are done, master spawns the reducers using a similar method.

In the end when all reducers complete, master combines the intermediate reducer files to give the final output and closes the socket.

Mapper Details -

For implementing the mapper, first I have made the imports necessary. I read the config file to get variables like port number and count of reducer and mappers.

After getting the command line inputs using sys.argv I call the function word_count or inverted_index depending on config.

The function reads from a split input file and creates a list of tuples - (word, 1)/ (word, file_name) and stores into KV Store with mapper_id as key and the whole list as the value.

In the end before completion set mapper status in KV store to True or False depending on failure.

Reducer Details -

For implementing the reducer, first I have made the imports necessary. I read the config file to get variables like port number and count of reducer and mappers.

A single reducer reads each mappers output from KV store and processes using the function wc_parse and ii_parse depending on the application.

Hash function is also included in reducer to determine which word to take based on first letter ascii value modulo with id of reducer.

Reducer then writes to the intermediate file and sets its status according to failure.

KV Store Details -

KV Store implementation used from Assignment 1 submission with minor parsing modification.

For inverted index, the split files by the master are considered as separate files, named by mapper id example - file_1.

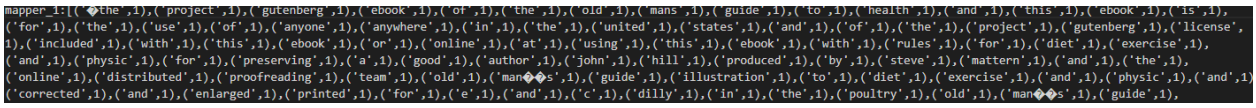
Test Case 1 -

Run testCase_1.py, includes an example with word count, 3 mappers and 2 reducers. Run server.py and then run master.py in another terminal.

Output in master terminal.

```
PS C:\Users\rg12\Downloads\Map reduce> python master.py
SPLITTING INPUT FOR MAPPERS
.....STARTING MAPPERS..........
MAPPER_1 IS RUNNING..........
MAPPER_2 IS RUNNING..........
MAPPER_3 IS RUNNING..........
MAPPER_1 status set
MAPPER_3 status set
MAPPER_2 status set
.....MAPPERS DONE..........
.....STARTING REDUCERS..........
REDUCER_1 status set
REDUCER_2 status set
.....REDUCERS DONE..........
DONE
```

Snap of KV Store(only a small part)



Snap of output.txt(only small part as file is very large)

```
'project',3
'gutenberg',3
'ebook',6
'of',132
'the',228
'old',28
'mans',3
```

Test Case 2 -

Run testCase_2.py, includes an example with inverted index, 3 mappers and 2 reducers. Run server.py and then run master.py in another terminal.

Output in master terminal.

```
PS C:\Users\rg12\Downloads\Map reduce> python master.py
SPLITTING INPUT FOR MAPPERS
.....STARTING MAPPERS..........
MAPPER_1 IS RUNNING..........
MAPPER_2 IS RUNNING..........
MAPPER_3 IS RUNNING..........
MAPPER_2 status set
MAPPER_3 status set
MAPPER_1 status set
.....MAPPERS DONE..........
.....STARTING REDUCERS..........
REDUCER_1 status set
REDUCER_2 status set
.....REDUCERS DONE..........
DONE
PS C:\Users\rg12\Downloads\Map reduce>
```

Snap of KV Store(only a small part)

store.txt

```
1   mapper_1:[('the','file_1'),('project','file_1'),('gutenberg','file_1'),
    ('to','file_1'),('health','file_1'),('and','file_1'),('this','file_1'),('
```

Snap of output.txt(only small part as file is very large)

output.txt

```
30   'diet',["'file_1'", "'file_2'", "'file_3'"]
31   'exercise',["'file_1'", "'file_2'", "'file_3'"]
32   'physic',["'file_1'"]
33   'preserving',["'file_1'", "'file_2'"]
34   'a',["'file_1'", "'file_2'", "'file_3'"]
```

Test Case 3-

Run testCase_3.py, includes an example with word count, 5 mappers and 3 reducers. Run server.py and then run master.py in another terminal.

Output in master terminal.

```
PS C:\Users\rg12\Downloads\Map reduce> python master.py
SPLITTING INPUT FOR MAPPERS
.....STARTING MAPPERS..........
MAPPER_2 IS RUNNING..........
MAPPER_1 IS RUNNING..........
MAPPER_5 IS RUNNING..........
MAPPER_4 IS RUNNING..........
MAPPER_3 IS RUNNING..........
MAPPER_2 status set
MAPPER_1 status set
MAPPER_4 status set
MAPPER_5 status set
MAPPER_3 status set
.....MAPPERS DONE..........
.....STARTING REDUCERS..........
REDUCER_2 status set
REDUCER_1 status set
REDUCER_3 status set
.....REDUCERS DONE..........
DONE
PS C:\Users\rg12\Downloads\Map reduce>
```

Snap of KV Store(only a small part)

store.txt

```
13    reducer_1_status:true
14
15    reducer_2_status:true
16
17    mapper_4:[('this',1),('ebook',1),('is',1),('f
      1),('to',1),('check',1),('the',1),('laws',1),
```

Snap of output.txt(only small part as file is very large)

output.txt

```
38    'their',9
39    'descendants',1
40    'being',4
```

Test Case 4 -

Run testCase_4.py, includes an example with inverted index, 5 mappers and 3 reducers. Run server.py and then run master.py in another terminal.

Output in master terminal.

```
PS C:\Users\rg12\Downloads\Map reduce> python master.py
SPLITTING INPUT FOR MAPPERS
.....STARTING MAPPERS..........
MAPPER_1 IS RUNNING..........
MAPPER_5 IS RUNNING..........
MAPPER_4 IS RUNNING..........
MAPPER_3 IS RUNNING..........
MAPPER_2 IS RUNNING..........
MAPPER_1 status set
MAPPER_5 status set
MAPPER_4 status set
MAPPER_3 status set
MAPPER_2 status set
.....MAPPERS DONE..........
.....STARTING REDUCERS..........
REDUCER_2 status set
REDUCER_1 status set
REDUCER_3 status set
.....REDUCERS DONE..........
DONE
PS C:\Users\rg12\Downloads\Map reduce>
```

Snap of KV Store(only a small part)

store.txt

```
4
5   mapper_3:[('wwwgutenbergorg','file_3'),('if','file_3')
6
7   mapper_1_status:true
8
9   mapper_2_status:true
```

Snap of output.txt(only small part as file is very large)

output.txt

```
38  'their',["'file_1'", "'file_2'", "'file_3'", "'file_5'"]
39  'descendants',["'file_1'"]
40  'being',["'file_1'", "'file_3'", "'file_4'", "'file_5'"]
```

Test Case 5 (fault tolerance) -

For this test case fault tolerance rate is set to 50%, meaning there is a 50 percent chance that a mapper or reducer can fail. If it does not fail in a single run, repeat till failure is visible in the command line.

Output in master terminal.

```
PS C:\Users\rg12\Downloads\Map reduce> python master.py
SPLITTING INPUT FOR MAPPERS
.....STARTING MAPPERS...........
MAPPER_2 IS RUNNING...........
MAPPER_1 IS RUNNING...........
MAPPER_3 IS RUNNING...........
MAPPER_2 status set
MAPPER_1 status set
MAPPER_3 status set
mapper_2 failed, RESTARTING......
MAPPER_2 IS RUNNING...........
MAPPER_2 status set
mapper_2 failed, RESTARTING......
MAPPER_2 IS RUNNING...........
MAPPER_2 status set
.....MAPPERS DONE...........
.....STARTING REDUCERS...........
REDUCER_1 status set
REDUCER_2 status set
.....REDUCERS DONE...........
DONE
PS C:\Users\rg12\Downloads\Map reduce>
```

Snap of KV Store(only a small part)

store.txt

```
    1),('the',1),('other',1),('in',1),('severe',1),('weather',1),('
2
3   mapper_2:[('longer',1),('life',1),('by',1),('john',1),('hill',1
    ('almost',1),('no',1),('restrictions',1),('wwwgutenbergorg',1),
```

Snap of output.txt(only small part as file is very large)

output.txt

```
17    'anyone',1
18    'anywhere',1
19    'in',83
20    'united',2
```

BONUS PART -

PART - 1 Using Key Value Store

KV Store used for communication between mapper, reducer and master.

PART - 2 Fault Tolerance

For fault tolerance, there is very less chance of failure in a local machine so I have simulated a faulting mechanism in mapper.py and reducer.py. I generate a random number between 1 to 10 and check the fault tolerance percentage mentioned in the config file. I set the completed flag according to this faulting output. In the master, when I have spawned mapper and reducer, I have stored the id as well. After spawning them once I check flags using the id and call the failed component again until it completes.

Message Formats -

Requests to the key value store are as follows -

SET Request - 'set–[key]--[value]' key is the word and value is the whole list.

GET Request - 'get–[key]' , expects a response as the encoded value as a string.

Delimiter for the requests changed to '--' in the kv store.

The list of values is then parsed to take each tuple from the list and perform required computation in reducers.

Limitations and Error handling -

One clear limitation is that MapReduce is not applicable in real time jobs because it is inefficient. It is only good for batch jobs. When mappers are running the reducers are idle and vice versa, => there is a lot of ideal time which can be used better.

It can only be used for simple applications and not for complex algorithms or working on graphs.

In my implementation a shared state is required, that is the configuration file to store the common variables which are accessed at different steps.

Error handling has been taken care of if it is in mapper or reducer. Due to the KV store, any error in the KV store will result in failure. Any errors in the master will also result in failure, the only solution is to restart the master in that case.

REFERENCES

- https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf