Emulating Hadoop MapReduce Using Python

Team #40
Revati Pawar rpawar@usc.edu
Riten Bhagra bhagra@usc.edu
Samarth Saxena saxenasa@usc.edu

I. Introduction

In this project, we aim to emulate the Hadoop MapReduce framework by implementing basic mapper and reducer classes that allow users to extend the classes to provide their own map and reduce functions. The mapper function reads a text file and partitions the data into different output files using a consistent hash-based partitioner function. The reducer function takes a key and a list of values as input, similar to that in Hadoop, and combines the results into a final output. Additionally, we implement a partitioner, shuffler, and data grouper that take intermediate key-value pairs produced by map tasks and produce input for reduce tasks. We also create a number of data servers and distribute map and reduce tasks among the servers, similar to the Hadoop JobTracker-TaskTracker function for task scheduling.

Our goal is to demonstrate that users can build MapReduce jobs using the emulated framework and run the job to get the results. We show how multiple map tasks may be generated, one for each input split, and multiple reduce tasks may be requested. We also show how reduce tasks obtain their input key and list of values from the output of map tasks through the partitioner, shuffler, and grouper provided by the framework. Finally, we demonstrate that the tasks are indeed computed in multiple servers, although these servers may all run on the same machine. Overall, this project aims to provide a functional implementation of the Hadoop MapReduce framework and demonstrate its capabilities for processing large amounts of data in a distributed computing environment.

II. EDFS Architecture

Brief overview of the project architecture:

1. **Input File Splitter** : This component takes in the input file and divides it into smaller splits based on a fixed block size. The number of splits determines the number of mappers used in the MapReduce job.

2. **Parallel Execution** : The mappers and reducers are run in parallel using multiprocessing since they can be present on different systems in HDFS.

3. **Mapper** : The mapper component receives the splits generated by the input file splitter and applies the user-defined map function to each split. It generates intermediate key-value pairs where the keys are the line numbers and the values are the lines present in those splits.

4. **Partitioner** : This component takes the output of the mapper and partitions it into multiple files based on the key. The number of partitions determines the number of reducers used in the MapReduce job.

5. **Shuffler** : After the data is partitioned, the shuffler step is responsible for redistributing the data to different nodes in the cluster so that related data ends up on the same node. This is important because it reduces network traffic and enables more efficient processing.

6. **Data Grouper** : This component groups and sorts the different files to pass on to each individual reducer.

7. **Reducer** : The reducer component receives the sorted files and applies the user-defined reduce function to each partition. The output of the reducer is written to the output file.

8. **Job Tracker** : The job tracker component is responsible for coordinating the execution of the MapReduce job. It assigns tasks to task trackers, monitors their progress, and handles failures.

9. **Task Tracker** : The task tracker component is responsible for executing individual map and reduce tasks assigned to it by the job tracker. It communicates with the job tracker to report progress and handle any resource conflicts.

This is a high-level overview of the project architecture.

III. Design and Implementation of EDFS

Drive Link
Youtube Link

1. Splitter

Input file is split into multiple parts by the splitter function. This function decides how many splits are needed based on the block size. For example, if the block size is 512KB and the input file size is 2MB, then there will be 4 splits created.

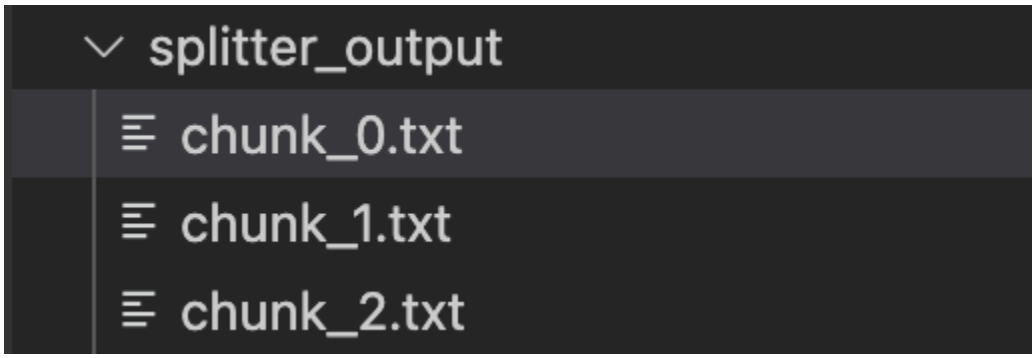
Splitting the input file into smaller chunks is important because it allows for parallel processing of the data. By dividing the file into smaller pieces, multiple mappers can process different parts of the file concurrently. This can lead to faster processing times since the work is distributed among multiple nodes in a cluster.

It's worth noting that the block size is configurable and can be set to different values depending on the size of the input files and the available resources. If the block size is too small, then there will be too many splits, which can lead to increased overhead in managing the splits. On the other hand, if the block size is too large, then there

may not be enough splits to make full use of the available resources. Therefore, it's important to choose an appropriate block size that balances these trade-offs.
In our implementation, we have set the block size to 2048, based on which the total number of splits will be determined. In total, 3 splits are generated which are stored under the splitter_output directory.

```
# spawn a subprocess to run the splitter.py script
server_process = subprocess.Popen(['python3', 'splitter.py', input_file])
server_process.wait()
print('The splits have been stored in the splitter_output directory.')
time.sleep(5)
```

```
Splitting the input file..

The splits have been stored in the splitter_output directory.
```



2. Parallel Execution - Mappers

In a typical Hadoop MapReduce system, the input data is split into multiple chunks, and each chunk is processed by a separate mapper. In our demo, the splitter generated 3 splits based on the block size. We want to process each chunk using a separate mapper, just like in Hadoop.

To achieve this, we use various modules including socket, threading, and subprocesses. This allows our program to handle multiple client connections simultaneously. When a client connects to a server, the server creates a new thread to handle the client's requests. This enables the server to handle multiple clients

concurrently, without blocking any other clients from connecting or processing their requests. In our case, we will have 3 mappers running on multiple servers.

The `handle_client()` function is responsible for receiving data from a client, extracting the chunk number and intermediate key-value pairs, and saving them to a file. It creates a directory for the output files if it does not exist and writes the mapper output to a file with a name based on the chunk number.

```
# This function handles the client connection and saves the mapper output to a file.
def handle_client(conn):
    received_data = b''
    while True:
        data = conn.recv(1024)
        if not data:
            break
        received_data += data
```

The `start_mapper_server()` function starts a mapper server on the specified port. It binds to the specified host and port and listens for incoming connections. Once a connection is established, it spawns a new thread to handle the connection and passes the connection object to the `handle_client()` function.

```
def start_mapper_server(port):
    host = "localhost"
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((host, port))
        s.listen()
        print(f"Mapper server listening on port {port}")
```

Finally, in the main section, it gets the list of input files and starts a mapper server for each file on a separate port. The code starts the mapper servers on ports 8000 and higher, with each mapper using a unique port number.

```
#Create Mapper Servers
server_processes = []
chunk_dir = "splitter_output"

# spawn a server process for each chunk file in the splitter_output directory
print('Creating mapper servers based on the number of splits generated..\n')
server_process = subprocess.Popen(['python3', 'mapper_server.py'])
server_processes.append(server_process)
time.sleep(1)  # Wait for server to start
```

```
Starting mapper server for chunk_2.txt on port 8000
Starting mapper server for chunk_1.txt on port 8001
Starting mapper server for chunk_0.txt on port 8002
3 mapper servers started.
Mapper server listening on port 8001
Mapper server listening on port 8000
Mapper server listening on port 8002
```

Overall, this allows multiple mapper servers to be created and handle data processing in parallel, improving the efficiency of the overall system. It can significantly reduce the processing time, especially for large input files. Instead of processing the entire input file sequentially, which can take a long time, we can process different parts of the file concurrently, which can speed up the processing time.

3. Mapper

The splits created in step 1 determine how many mappers are used. If there are 3 splits, then there will be 3 mappers, and each split will be sent to a separate mapper. The generic mapper function is called, which generates intermediate key-value pairs. The keys are line numbers, and the values are each line present in those splits.

In a MapReduce system, the input data is divided into smaller chunks called "splits", and each split is processed by a separate "mapper". In our case, we have split the input file into 3 chunks, so we need 3 mappers to process each chunk.

When we create the mappers, we create 3 instances of the mapper class, one for each split. These mappers will be responsible for processing their respective splits, generating intermediate key-value pairs.

We create 3 mappers (m1, m2, m3), and each mapper is responsible for processing one of the 3 splits, parallelly on a different server.

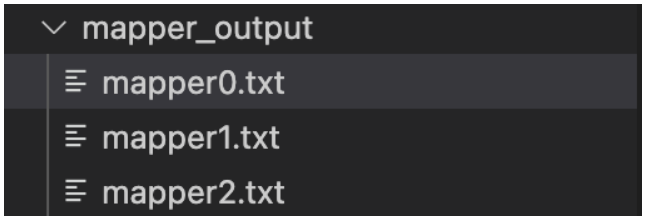
We have a class called Mapper, which processes a chunk of text by creating key-value pairs for each word in the chunk. It then connects to a server and sends the key-value pairs. The UserMapper class extends the Mapper class and overrides its method to run the process_chunk() method. In the main section, it gets the filename of the chunk to be processed and the port number of the server.

```
# Process the chunk to create key-value pairs
def process_chunk(self):

    # Read in the chunk file and preprocess the text
    with open(self.chunk_filename, 'r') as file:
        data = file.read()
        data = data.translate(str.maketrans("", "", string.punctuation)).lower()
        words = data.split()

        # Create key-value pairs for each word in the chunk
        key_value_pairs = [(word, 1) for word in words]
```

The Mapper will generate intermediate key-value pairs by processing the input data, and these pairs will be stored in memory until they can be passed further to the partitioner.

```
#Run the Mappers
# start a mapper process for each chunk file
for i, chunk_filename in enumerate(os.listdir(chunk_dir)):
    port = 8000 + i
    chunk_path = os.path.join(chunk_dir, chunk_filename)
    server_process = subprocess.Popen(['python3', 'mapper.py', chunk_path, str(port)])
    server_process.wait()
print('The mapper outputs have been stored in the mapper_output directory.')
time.sleep(10)
```

```
Starting mapper for splitter_output/chunk_2.txt on port 8000
Connected to server localhost:8000
Connection from ('127.0.0.1', 64291)
Output saved for chunk 2
Starting mapper for splitter_output/chunk_1.txt on port 8001
Connection from ('127.0.0.1', 64292)
Connected to server localhost:8001
Output saved for chunk 1
Starting mapper for splitter_output/chunk_0.txt on port 8002
Connected to server localhost:8002
Connection from ('127.0.0.1', 64293)
Output saved for chunk 0
The mapper outputs have been stored in the mapper_output directory.
```



So, in summary, for each split, we create a separate mapper instance, and each mapper generates intermediate key-value pairs.

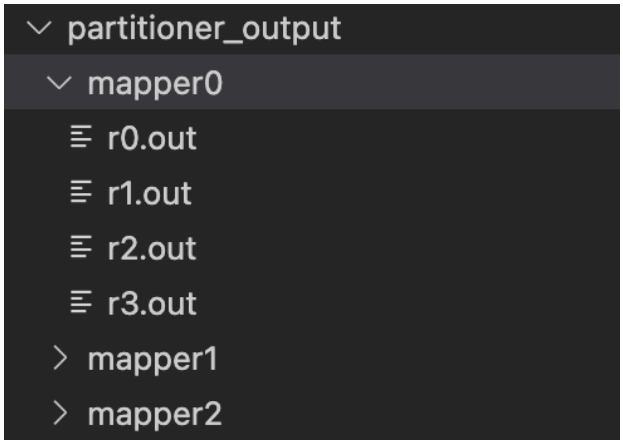
4. Partitioner

In a MapReduce system, the partitioner takes in the intermediate key-value pairs generated by the mapper and distributes them among the reducers. The partitions generated by each partitioner will be determined by the number of reducers.

We are partitioning the output of the mappers into a specified number of files based on the number of reducers. For each mapper file, the function creates a new directory with the same name as the mapper file under "partitioner_output". Within this directory, it creates the specified number of output files, one for each reducer.

```
#Run the Partitioner
print('The Partitioner is running..\n')
server_process = subprocess.Popen(['python3', 'partitioner.py', str(reducers)])
server_process.wait()
print('The partitioned outputs are stored in the partitioner_output directory.')
time.sleep(10)
```

The code then reads through each key-value pair in the mapper file, calculates a hash value for the key to determine which reducer it should be sent to, and writes the key-value pair to the corresponding output file. The hash value is calculated using the MD5 hash function, and the modulo operator is used to ensure that the hash value falls within the range of the reducer count.



So, if there are 3 mappers, each generating 4 partitions (the user input is considered 4 here, it can be a different value as well), there will be a total of 3 * 4 = 12 partitions. So, mapper0 will have 4 partitions (r0.out, r1.out, r2.out, r3.out) and similarly for each mapper output.

5. Shuffler

In a MapReduce job, the intermediate key-value pairs generated by the mapper are not directly sent to the reducer. Instead, they are first shuffled and grouped based on their keys and then sent to the reducer. This process is called shuffling, and it is responsible for ensuring that all the values associated with a particular key are sent to the same reducer.

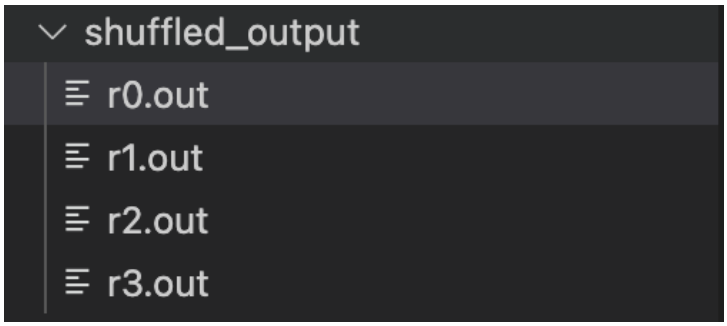
Once the partitions are created, the shuffler takes over. The shuffler is responsible for ensuring that all the values associated with a particular key are sent to the same reducer. It does this by shuffling and grouping the intermediate key-value pairs based on their keys and sending them to the appropriate reducer.

We use a function to shuffle the output of the partitioner into new files in the shuffled_output directory. The code uses the glob and os modules to find the list of subdirectories in the partitioner output directory, and then creates shuffled output files for each reducer. To shuffle the data, the function iterates over each file and

reads its contents line by line. It then writes each line to a new output file, such that lines with the same key are grouped together. This is achieved by iterating over each reducer directory and writing lines from files with the same index as the current output file index. The shuffling is based on the hash value of the key, which determines which reducer the key belongs to.

```
#Run the Shuffler
print('The Shuffler is running..\n')
server_process = subprocess.Popen(['python3', 'data_shuffler.py'])
server_process.wait()
time.sleep(10)
```

For example, if there are 4 reducers, and the key "apple" appears in partitions mapper0/r0.out and mapper2/r3.out, the shuffler will group all the values associated with the key "apple" and send them to the same reducer, say reducer r2.out. This ensures that all the values associated with a particular key are processed together by the same reducer.



6. Parallel Processing - Reducers

In a MapReduce system, the use of multiple reducers can help to improve the efficiency of the data processing by enabling the parallel processing of intermediate key-value pairs generated by the mappers. For example, if we have multiple reducers, each reducer can be responsible for processing a subset of the intermediate data generated by the mappers. This can help to reduce the overall processing time, as each reducer can work independently and in parallel to process its subset of the data.

We have implemented a Reducer server function that listens on specified ports for incoming connections from the Mapper, receives the sorted file from the Mapper, and then creates a new thread to handle each client. The function `handle-client` handles each client by calling a `UserReducer` object on the sorted file received from the client, which performs the word count, and then closes the client socket.

```
# Define a function to handle each client that connects to the server
def handle_client(client_socket, sorted_file):

    # Use the lock to prevent multiple threads from printing at the same time
    with lock:
        print(f"[*] Received file {sorted_file}")
        #Call Extended Reducer
        reducer = UserReducer(sorted_file)
        reducer.word_count()
    client_socket.close()
```

The function `start_reducer_server` creates a socket object and binds it to the specified port, listens for incoming client connections, and accepts them with a new thread.

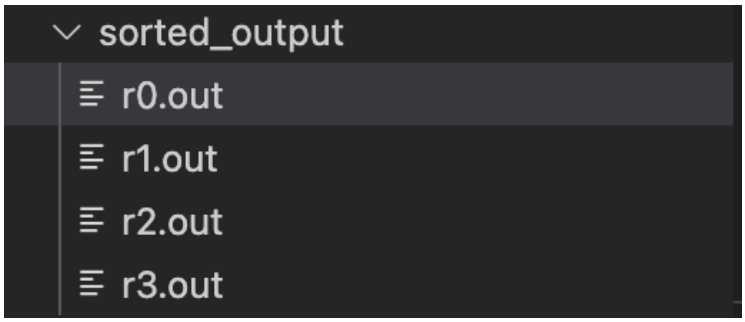
```
# Define a function to start the reducer server on a specified port
def start_reducer_server(port):

    # Create a socket object and bind it to the specified port
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(('0.0.0.0', port))

    # Start listening for incoming client connections
    server.listen(5)
    print(f"[*] Listening on port {port}")

    # Continuously accept incoming client connections and handle them with a new thread
    while True:
        client, _ = server.accept()
        sorted_file = client.recv(1024).decode('utf-8')
```

The main function gets the number of reducers, calculates the ports for the reducer servers, and starts a new thread for each reducer server. Finally, it waits for all threads to finish before exiting.

```
Setting up 4 reducer servers..

[*] Listening on port 5040
[*] Listening on port 5041
[*] Listening on port 5042
[*] Listening on port 5043
```

7. Data Grouper

The Data Grouper sorts the input file into chunks, writes each chunk to a separate file, and then merges the sorted chunks into a single sorted output file. It then sends the sorted output file to a reducer server via a TCP socket. Finally, a combiner is called to combine the output from the reducers into a final output file.



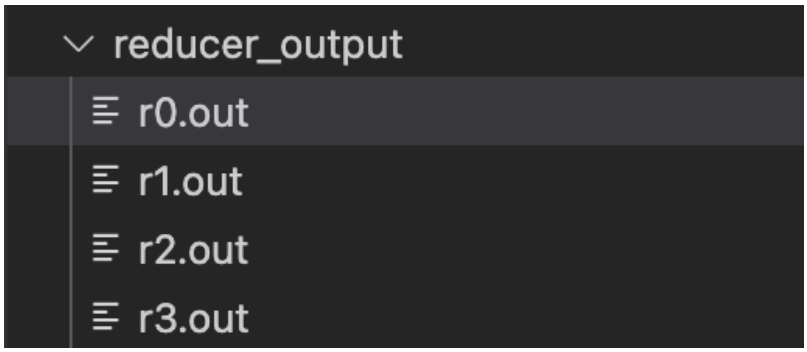
8. Reducer

Once the intermediate key-value pairs have been shuffled and sorted, they are sent to the corresponding reducer based on the partition name. For example, all partitions ending in "r1" are sent to reducer "r1". The reducer receives these key-value pairs and performs some computation or transformation on them, depending on the specific task.

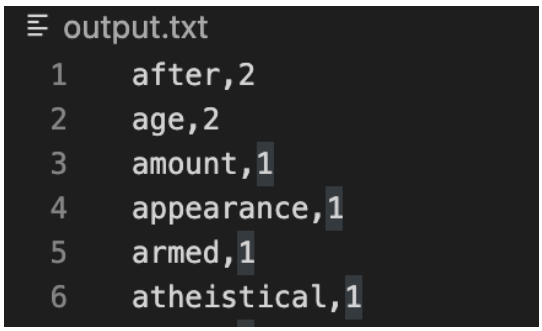
```
The data is sorted and the files are sent to different reducer servers for processing..

[*] Sent sorted_output/r2.out to reducer server on port 5040
[*] Received file sorted_output/r2.out
[*] Sent sorted_output/r3.out to reducer server on port 5041
[*] Received file sorted_output/r3.out
[*] Sent sorted_output/r1.out to reducer server on port 5042
[*] Received file sorted_output/r1.out
[*] Sent sorted_output/r0.out to reducer server on port 5043
[*] Received file sorted_output/r0.out
```

For example, if the task is to count the number of occurrences of each word in a large text file, the mapper will produce intermediate key-value pairs where the key is a word and the value is 1 for each occurrence of that word. The shuffler will group all of the key-value pairs with the same key and send them to the same reducer. The reducer will then receive all of the key-value pairs for a given word and sum up the values to get the total count for that word.

Our function defines a class called Reducer, which takes a path to an input file and a method that returns the input file path. The class also has a subclass named UserReducer, which extends Reducer and has a method called word_count that counts the occurrences of each word in the input file and writes the result to an output file in a directory called 'reducer_output'



Once the reducer has finished processing all of the key-value pairs assigned to it, we call a combiner function which combines the contents of all the output files produced by the reducer. It outputs the final result in the form of key-value pairs, which can be written to a file or sent to another system for further processing. In the case of the word count example, the output is a set of key-value pairs where the key is a word and the value is the total count for that word across the entire input file.



V. Learning Experiences

- This project provided an opportunity to learn about distributed systems and how they can be used to process large amounts of data efficiently. Through implementing and working with the MapReduce framework, we gained knowledge about how distributed systems can be used to break down complex tasks into smaller, more manageable ones.
- We gained experience with techniques like data partitioning, shuffling, and sorting to optimize processing performance.
- As with any complex project, this one presented challenges in debugging and troubleshooting code. We had to develop strategies for identifying and addressing bugs in distributed code, which required a deep understanding of the underlying system and a comprehensive approach to testing.
- The project provided an opportunity to learn about how to design and implement distributed systems that can take advantage of multiple servers to process data more efficiently. We gained practical experience in using Python's socket and threading modules to create a system of worker nodes that could work in parallel to process data, distributing the workload and optimizing performance.

VI. Conclusion

In summary, HDFS uses the MapReduce framework, where input files are split into multiple parts, and each split is processed by a separate mapper. Intermediate key-value pairs are generated and sent to the partitioner, which determines the number of reducers to be created. The data shuffler and data grouper are responsible for shuffling and sorting the data before sending it to the reducers. Each reducer then performs computations on the key-value pairs and outputs the result.

In conclusion, we have successfully designed and implemented a simplified version of the Hadoop MapReduce framework in Python. Our framework provides a distributed computing solution for processing large data sets by distributing the workload across multiple nodes, which significantly reduces processing time.

We have discussed the various components of our framework, including the mapper, partitioner, shuffler, data grouper, and reducer and explained how they work together to process data in parallel with multiple mapper and reducer servers.

Overall, our framework provides a scalable and efficient solution for processing large data sets, and we believe that it can be extended to handle more complex tasks in the future. We hope that our work inspires further research in this area and helps advance the field of distributed computing.