CS532: Project 1: Map Reduce Design Document

Introduction

The goal of this project is to implement a basic version of MapReduce running on a single server. We used the MapReduce Paper as reference to build this data processing system.The system takes a list of key/value pairs as input data, and produces a list of key/value pairs as output. The user defines the computation as two functions: Map and Reduce User defined functions.
We have implemented 3 diverse applications as use cases to test our MapReduce system.

1. Missing Time Series
2. Inverted Index
3. Word Count

These applications not only test the correctness of our system but also if the system is load balanced.

Description (Program Design)

IO Arguments Handling and Master Worker Invocation

IO Arguments Handling

The system accepts input information through a config file - 'app.config'. The system accepts input file information, output directory information, N and UDF class information through this configuration file. The main process doesn't accept any system arguments. The Main class invokes the Master and passes on the config file to the master. The Master in turn reads the config file, and initiates the workers and passes on the input file information and output directory information to the mapper and reducer workers respectively at initiation.

MultiProcessing

Once the Master instance is created, it invokes N worker processes. Each worker process is provided with a workerId, input file, communication port number and UDF class information. The master also initiates a communication thread for each worker process which listens to the worker's corresponding port and handles co-ordination based on the worker provided information. The master keeps tabs on global count of successfully completed processes at each stage. Once the global count reaches n and there are no inactive processes, the master re-initiates workers for the reducer stage and repeats the steps above or exits depending on previous process.

Coordination

Communication
The master and the worker processes communicate through sockets. In this system, the sockets are used for one way communication. The master and worker processes have separate threads which exclusively handle communication between the pair of processes. The Master keeps a list of active and inactive processes at any given point of time during the execution. These lists are initiated and updated whenever a process is made or ends. The master communicates with multiple workers at any point of time. It purely listens to the sockets while the workers provide regular updates to the master.

The worker process updates, for both the map and reducer stages, are random as long as the process is in execution. Once the process ends, the workers either provide a success symbol (1) to the master or provide a pre-decided error (-1 to -5) code to the master before exiting.

The Master coordinates with all processes by actively keeping tabs on the worker process health through sockets and responding according to the output. If the worker ends abruptly, the master initiates a fault tolerance protocol.

Error Handling
If the worker ends with an error code, the master stores the error code as process status and provides a list of process and status at the end of each stage before exiting without completing the task. If during fault tolerance or any other time, the process creation has a bug, the master will exit the system if the number of processes exceed N, to prevent crashing the system from increasing number of processes

Fault Tolerance
If a worker process ends abruptly, i.e. it sends null information through the socket, the process is moved from active to inactive lists by the master. The master regularly keeps checking the status of the active and inactive lists. The inactive list is only populated if the worker abruptly ends. If the master finds any processes in the inactive list, the master restarts the process.

Worker Execution Overview:

Mapper (Single Mapper Execution)
The worker calculates partition size and pointer and reads a specific chunk of data. This chunk of data is then split by a newline character-('\n') into an array of single lines from input. All lines in an array are complete. The user defined function class (UDF) is instantiated and each line in the array is passed on to the user defined function. The output hashmap of key value pairs from each function is stored together in an arraylist before each key is hashed and written to intermediate files.

Reducer (Single Reducer Execution)
At the reducer stage, the worker reads the intermediate files corresponding to its id and stores all the key value pairs in a hashmap where the values are stored as array lists. The user defined

function class is invoked and each key-value pair is passed on to the UDF reducer. The output of the UDF reducer is then written to the file.

Load Balancing

Partitioning
The number of partitions of the input file is equal to the number of workers (N) provided to the system. Each worker is passed on a worker id (ID) by the master. At mapping time, the worker then computes partition size and offset at which it should read the file using,
Partition_size = File_Size/N
Pointer = ID*Partition Size.

The worker then proceeds to read that specific chunk of file using a random access pointer. The last worker reads the file till the end of the file and not necessarily just it's chunk size.

Once the mapper worker has read the chunk, the chunks are split using the end of line character ('\n'). If the first split doesn't start at a new line, it is ignored. The last split if it captures a line halfway, the partition is extended to complete the last sentence. This ensures that no line is missing since the first line is missing.

Key to Reducer Mapping:

Once the UDF returns a key-array value pairs hashmap to the mapper, the mapper has to assign each key to a specific reducer. For this particular task, we hash the key value using Java's internal hashing algorithm. The key is then assigned to the reducer with ID = output mod N. We assume that Java's hashing algorithm sufficiently randomizes the key hashes and thus divides the load among the processes fairly equally.

Design Tradeoffs :

1. **Number of partitions of the input file = Number of workers.**
   - Inorder to ease the system design, partition size = file size / N. This allows us to let the workers independently complete tasks without waiting for communication from the Master.
   - While this approach doesn't work very well for very large files if the number of workers is small, we have found that as long as the number of workers is high for large files, the load balancing is acceptable.

2. **The Master doesn't actively communicate with the workers but only listens.**
   - The master passes on all the required information to the workers as system arguments during initiation of the worker process and then actively listens to the worker to check it's health and restart it if it ends abruptly without errors.

3. **<u>As a consequence of our partitioning method, if a partition doesn't have atleast one new line character, it is ignored.</u>**

- Since the worker ignores the first sentence in the partition if it is broken, for small files if a partition exists within the size of a single sentence it gets ignored.
- We believe this is an acceptable trade-off since this happens only for small files and when the N is reasonably high. There is little need for load balancing in such scenarios.

4. **<u>Mapper key input is not null.</u>**

- There is no obvious way in Java to pass a null value to a function in Java as far as we know, so we are passing random int values as keys to the UDF mapper. But those keys are not used in our applications.

5. **<u>If load balancing hash code doesn't assign any key from a mapper i to a reducer j, then the corresponding M_i_R_j.txt intermediate file isn't created.</u>**

- This is simply to avoid having empty intermediate files and thus lower file access at the reducer.
- A consequence of this is that there may not always be N*N intermediate files.

6. **<u>Intermediate data is not being explicitly sorted before being passed on to a reducer, a hashmap is being used to handle groupby key instead.</u>**

- At the reduce stage, a worker with worker id I initiates a hashmap and proceeds to read all files which are corresponding to it. As it reads the files, the worker updates the hashmap which is a key - array (value) structure with each new key.
- This time complexity of the operation is O(1) for every key insert and thus is very less.
- We note that since the hashmap accepts the key and values as Objects, the memory complexity of this operation may not scale for very large files.

7. **<u>During comparison of our output with the spark/python outputs, we perform multiple steps of cleaning with respect to quotation marks, braces, etc.</u>**

- To test the correctness of our system for the 3 applications we have chosen, we opted to write python/spark codes for the same applications. With regards to the two systems, the outputs differ in terms of representation, therefore we perform several steps to clean the spark/python output for comparison with the output we obtained from our system.

8. **<u>While testing for fault tolerance</u>**

- The intermediate files are overwritten once the workers are created and executed again after being dead. Therefore there is no need to delete them after the workers are killed.
- Edge case: If the workers are killed just after being created and before assigning any work, the system fails to work and is stuck.
- We are killing all N processes at the mapper stage and rebooting the workers by creating and executing all of them again before the reduce stage.

<u>Execution:</u>

To run the system for one application, the user must

1. Create their udf java file by implementing UDF interface and save it in src/mapReduce/
2. Update app.config file with input file, output file, N and UDF class name
3. Run the following command on the command line:

```
py execute.py
```

To execute automated testing to check for the correctness of the applications and fault tolerance, run the following command:

```
py testfile.py
```

This command will execute the testing for the applications by comparing the spark/python outputs with the output we obtain from our system and also test if the workers are rebooted after being killed by invoking kill_Proc in killProcesses.py