MAPREDUCE API

Developed By:

Mayuri Keskar

Sri Annapurna Bhupatiraju

Kamlendra Kumar

Kavyashree

CONTENTS

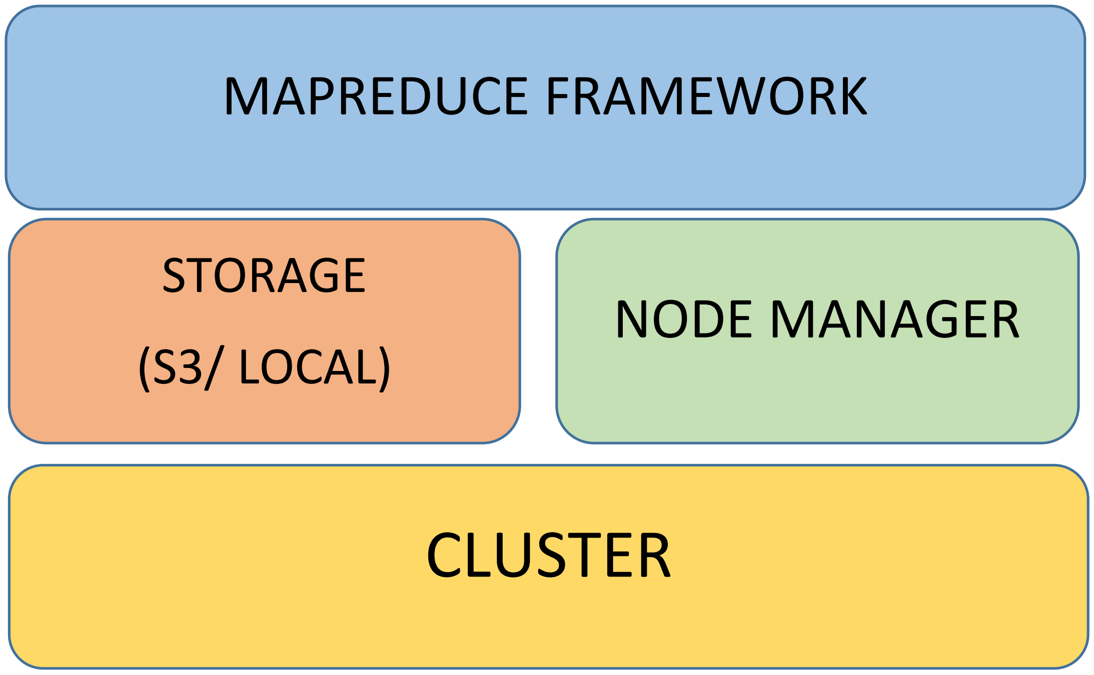
1. INTRODUCTION

The MapReduce API is a programming model and software framework built with an objective to facilitate and simplify the processing of large data-sets in parallel on clusters in a reliable and fault-tolerant manner.

In our project we tried to mimic Hadoop MapReduce API with functionalities like mapper, reducer, job tracker, task tracker, sorting and shuffling without using any Hadoop dependencies.

2. ARCHITECTURE OVERVIEW

The framework employs a master/slave architecture for both distributed storage as well as distributed computation. The architecture consists of mainly four building blocks (from bottom to top): Cluster, Storage, Node Manager and the MapReduce Framework.



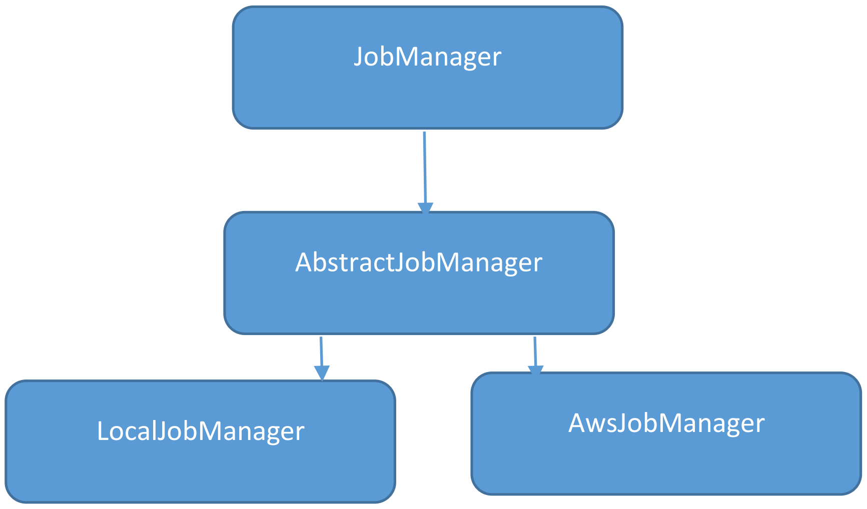
- **CLUSTER**:
  The Cluster is a host machine on which a task is run. Our API can be run on Local machine, on a set of host machines and on the EC2 clusters.

- **STORAGE**:
  The Storage is a permanent, reliable solution for the overall process. When the API is used to run a MapReduce job on the local machine, the storage data is stored on that local machine itself. Whereas, when the API is run on the EC2 cluster, the S3 works as the storage. The input for the Job is picked up from the S3 bucket and the final output of the job is written back to the same or different S3 bucket.

- **MAPREDUCE FRAMEWORK**:
  The MapReduce Framework is the software layer implementing the MapReduce Paradigm.

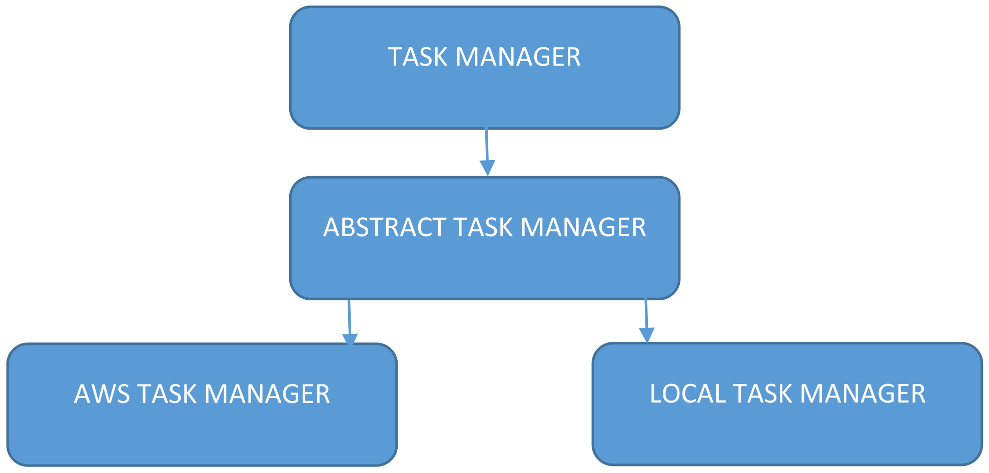
- **NODE MANAGER**:
  The Node Manager provides an infrastructure for controlling the various tasks of the job. It is basically of two types: Job Manager and Task Manager.

  1. *Job Manager*: The JobManager controls the master. The JobManager knows where the slaves are located. Depending on the type of the task the JobManager sends actions to the slaves. The architecture of Job Manager is as follows:



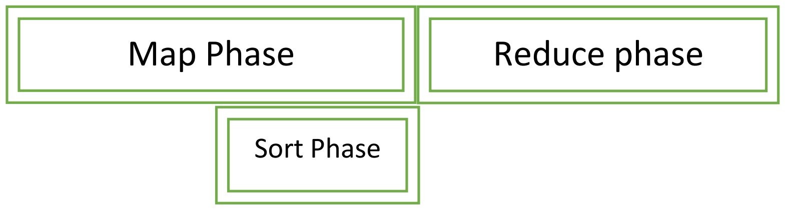
     The JobManager is an interface which is been implemented by AbstractJobManager. The AWSJobManager and LocalJobManager extend the AbtsractJobManager.

2. *Task Manager*: The TaskManager controls the slaves of the system. The TaskManager performs the tasks as directed by the master and sends the response back to the master through the Slave Channel.



The TaskManager is an interface which is been implemented by AbstractTaskManager. The AWSTaskManager and LocalTaskManager extend the AbtsractTaskManager.

3. MAPREDUCE TIMELINE



Job start --Job Finish

**Reduce Phase starts after Map and Sort Phase is completely finished.

Map Phase

When the client submits the job, the following details need to be provided to the master- map class, reduce class, jar name, job name, and input and output paths for the job. In our framework, we are calling map task for each input file and writing the output of map task to a temp file on the master/S3. When the slaves finish map task for all input files, all output files returned from each map task will be sorted and shuffled for reduce task which is explained later. So in Map phase all the slave nodes are dedicated for map tasks, the number of map tasks executed at a time is equal to the number of slave nodes running.

This is the Map Task execution timeline:

*RUNNING_MAP***:** Starts map task on a slave node. Calls map function for each line of input file.

MAP_COMPLETED: Map task is finished on slave node, master receives the "MAP_COMPLETED" status.

MAP_FAILED: If for any reason map task fails on a slave node, master receives "MAP_FAILED" status. When master receives this status it sends the file which was failed to process on the slave node to another slave node.

After the map phase is complete, master sorts the data in all the temp files where map task has written and creates separate file (referred as Key file in next sections) for each key which has <Key, Value> pairs for one particular key.

Reduce Phase

Master sends out one key file at a time to each slave node for reduce task. Reduce task reads all file data and converts it into <Key, Iterable<value>> and calls the respective reduce function. After reduce task is complete, the output is written in a part file and is sent to master. The number of reduce task running at a time is equal to the number of slave nodes running for the job.

This is the Reduce Task execution timeline:

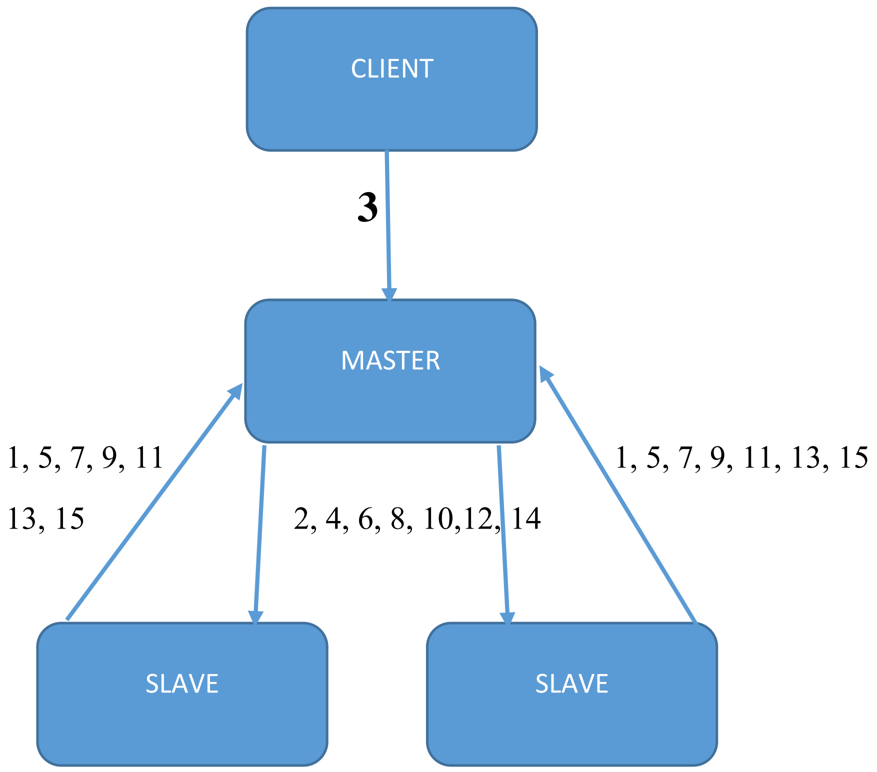
RUNNING_REDUCE: Starts reduce task on a slave node. Converts data in file to <Key, Iterable<value>> and calls reduce function for each key.

REDUCE_COMPLETED: Reduce task is finished on slave node, master receives the "REDUCE_COMPLETED" status along with output part file from reducer.

*REDUCE_FAILED***:** If for any reason reduce task fails on a slave node, master receives "REDUCE_FAILED" status. When master receives this status it sends the file which was failed to process on the slave node to another slave node.

4. IMPLEMENTATION

4.1 Job Execution Flow



ACTIONS:

1. *SLAVE REGISTRATION*
   This message is sent by the slaves to the master once they are up. The master receives this message and registers the slave ids of each slave.

2. REGISTRATION_ACK
   Once the master registers each Slave, it will then send an acknowledgement to each slave, notifying it that the registration is complete.

3. *JOB_SUBMISSION*
   The Client sends this request message to the master to start the Job. The client also sends the following details in the Job request- Jar name, Job Name, Driver Class, Mapper Class, Reducer Class, Output Key Class, Output Value Class, Input Path and Output Path.

4. *START_MAP*
   The Master will send this message to all the slaves to start the map phase. It will encapsulate the name of the file which the slave needs to read in the request message.

5. *PART_MAP_DONE*
   Once a Slave is done reading the file, it will write the data of the file in a temp file on its local machine and send this message to the master asking for the next file. The master sends one file at a time to the slave to read. The temp file generated after reading each file is sent back to the master/S3 bucket.

6. START_SORT
   Once all the files are read by the mapper slaves, the master starts the sort phase by sending a file to each slave and then sending the Start_Sort message. Once a slave receives this message, it reads the file and store it in List.

7. *PART_SORT_DONE*
   This message it sent by the slave to the master once it is done reading the file given by the master. When the master receives this message, it checks if it has a file that is not read, and sends that file to the slave with the message – START_SORT.

8. *PHASE1*
   This message is sent by the master to slave when the master does not have any other file left to be sent to the slave. When the slave receives this message, it sorts the data collected by reading all the files and generates samples according to Parallel Sorting by Regular Sampling algorithm.

9. *PIVOT_REQUEST*
   This message is sent by each slave to the master along with the samples generated in the previous phase. The number of samples generated by each slave is equal to the total number of slaves in the framework. On receiving this request, the master first waits for the samples from all the slaves and then generates pivots from the samples. The number of pivots in always one less than the number of slaves.

10. *PHASE2*
    This message is sent from the master to all the slaves along with the pivots generated in the previous stage. Each slave receive the pivots and partitions its data based on the pivots.

11. *PARTITION_REQUEST*
    This message is sent from each slave to the master. The slave also sends the partitioned files to the master/S3 bucket generated in the previous stage before sending this message.

12. *PHASE3*

    On receiving this message, the master redistributes the partitions such that each slave receives the chunk assigned to it. The slave receives the partitions, reads all the files and writes it back to the temp file based on the keys. The values of each key is written to a separate file.

13. *SORT_DONE*

    The files generated in the previous stage are sent to the master/S3 bucket by the slave along with this message notifying that the sort phase is completed.

14. START_REDUCE

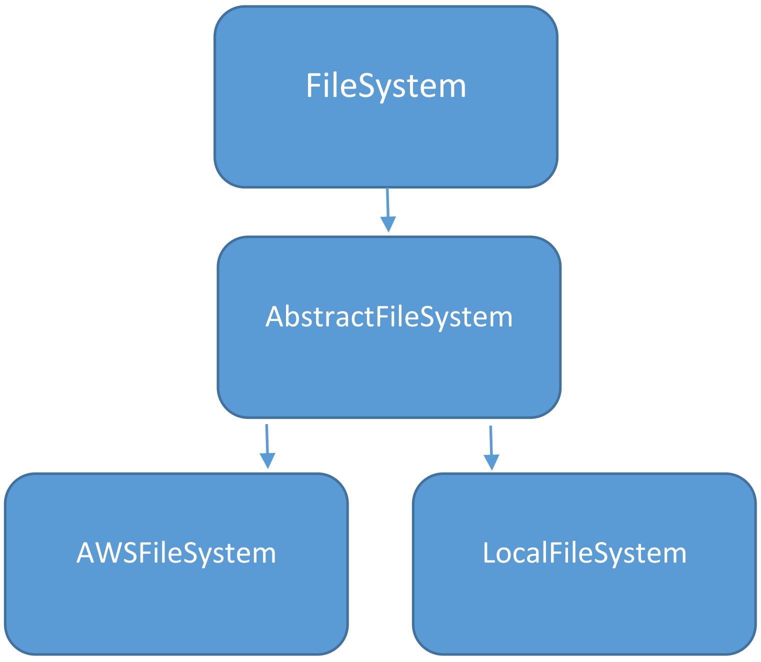
    The master starts the reduce phase once it is notified by the slaves about completion of the sort phase. The master sends this message to all the slaves along with one file. Since now, each slave starts the reduce job.

15. *PART_REDUCE_DONE*

    Once a slave is done reading and processing the file sent by the master, it writes the output to S3 and sends this message to the master. On receiving this message, the master sends the next file in the buffer to the slave.

The job gets completed once all the files are read and processed by the slaves and final output it written to the S3 bucket/ local output folder.

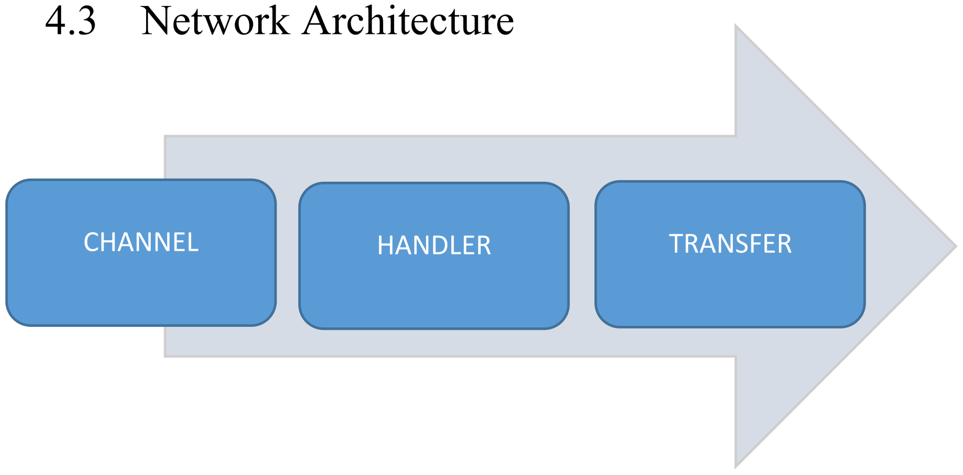
4.2 File System Architecture



Filesystem is used to control how data is stored and retrieved. Without a file system, information placed in a storage area would be a large body of data with no way to tell where one piece of information stops and the next begins. By separating the data into pieces and giving each piece a name, the information is easily isolated and identified. The filesystem in our project is either Local File system or S3 based on the mode of execution.

1. *FileSystem*: Interface which has read and write operations to files that Local file system and S3 would be performing.
2. *AbstractFileSystem*: Implements common methods of FileSystem interface that are used in both Local Filesystem and S3.
3. *AWSFileSystem*: Specific to read/write file operations in AWS S3.
4. *LocalFileSystem*: Specific to read/write file operations in Local.

4.3 Network Architecture

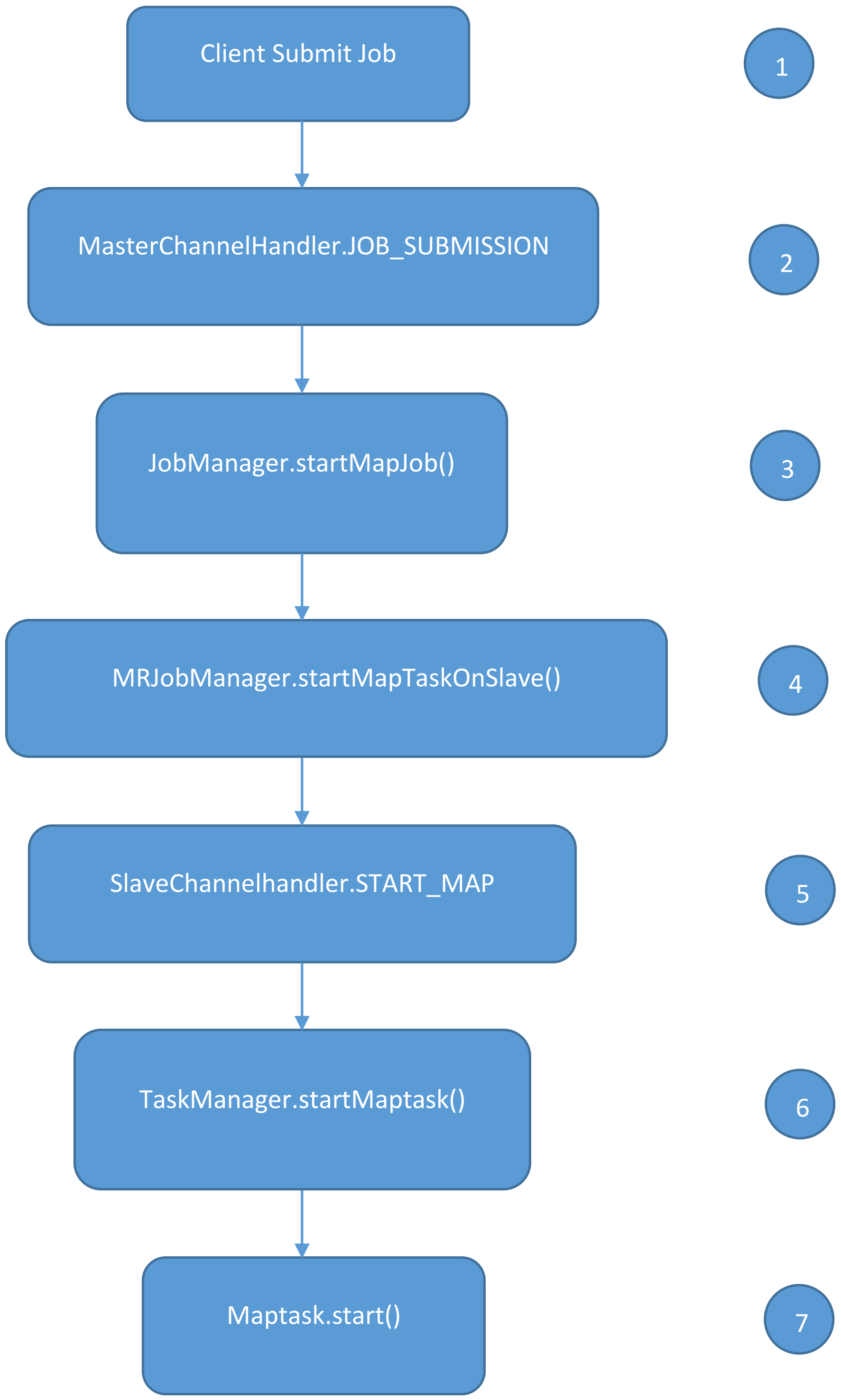


The MapReduce API is built on the crux of the Netty architecture. Netty uses its own buffer API instead of NIO ByteBuffer to represent a sequence of bytes. This approach has significant advantage over using ByteBuffer.

The Master and Slave have a channel. Once all the clusters are up and running, the slave channels get registered in the master. The MasterInitializer and SlaveInitializer initialize the channels when they become active. The MasterChannelHandler manages the requests which a master receives and the SlaveChannelHandler manages the requests which a slave receives. By using the Netty framework, one does not have maintain a list of IP address explicitly.

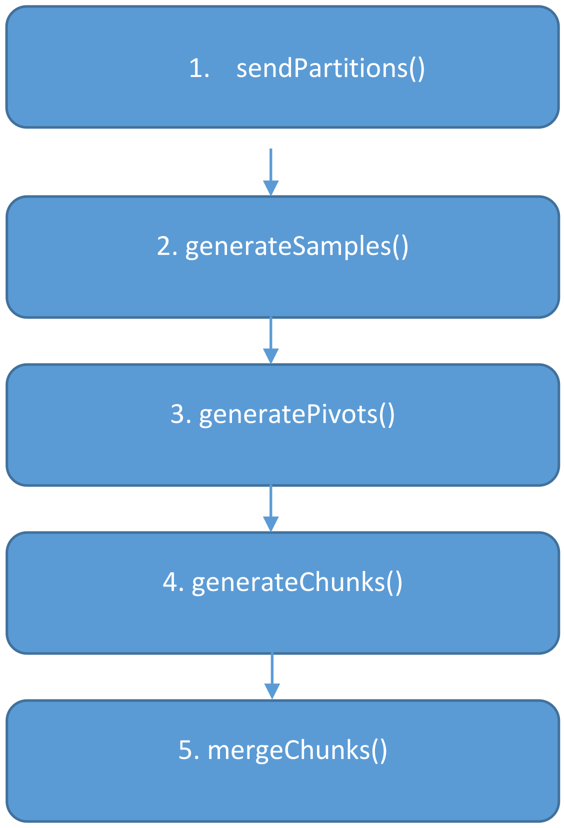
4.4 Task

Maptask


Client Submit Job
MasterChannelHandler.JOB_SUBMISSION
JobManager.startMapJob()
MRJobManager.startMapTaskOnSlave()
SlaveChannelhandler.START_MAP
TaskManager.startMaptask()
Maptask.start()
1
2
3
4
5
6
7

a. Client submits job to client
b. Master knows a job is submitted and calls startMapJob() and keeps track of the job
c. Interface of JobManager
d. Starts Maptask on slave
e. Slave recognizes that it has to execute Map task
f. TaskManager keeps track of Map task on the slave and invokes MapTask on slave
g. MapTask is the class where the MapTask is executed. The task calls the map function for each input line of the input file

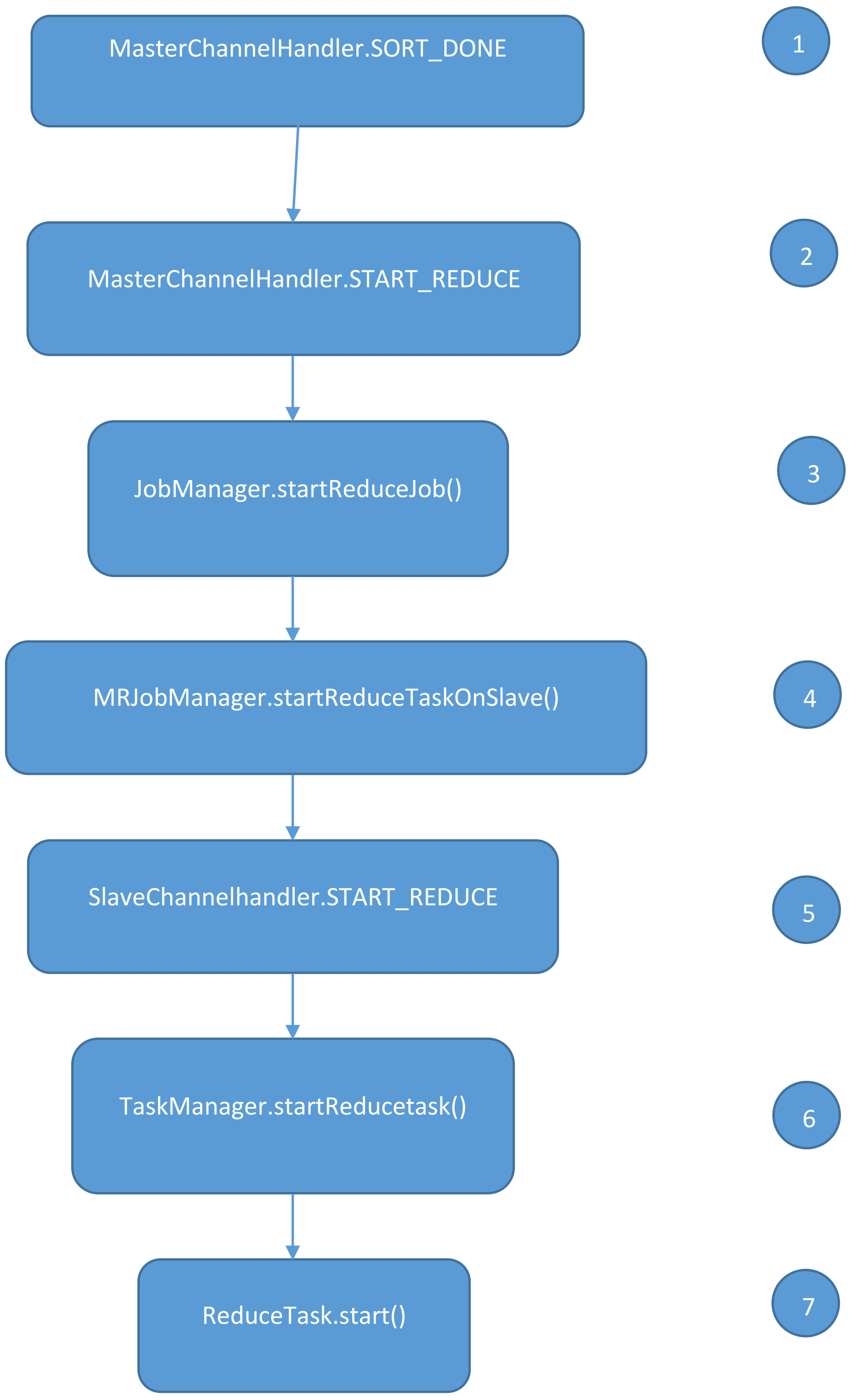
SortTask



1. sendPartitions():
   Master sends the map output files to each slave one at a time so as to ensure load balancing.

2. generateSamples():
   a. Each slave will collect its files into a list of key value pairs and sorts this collection.
   b. Then, they generate samples from this collection.
   c. Number of samples = Number of slaves
   d. Each slave sends the samples to the master
3. generatePivots():
   a. The master then receives the samples from all the slaves till,
      Total number of samples is not equal to Number of slaves * Number of slaves
   b. It then generates the pivots based on the keys.
   c. Number of pivots = Number of slaves – 1
   d. It then broadcasts these pivots to all the slaves
4. generateChunks():
   a. Each slave receives the pivots and partitions its local data into chunks.
   b. Number of chunks = Number of slaves
   c. The slave will then retain its own chunk number and transfer the remaining chunks to the master/S3 bucket.
   d. For example: If there are 3 slaves, Slave 1 will keep the "1_1" file and will transfer "1_2" and "1_3" files to the master.
5. mergeChunks():
   a. Master receives the chunks from each slave and transfers the required chunks to their respective slaves.
   b. For example: All "*_1" to Slave 1, All "*_2" files to Slave 2 and All "*_3" files to Slave 3.
   c. Each slaves will receive the chunks and then add it to a global list which will be sorted and then written to a file based on key.
   d. Each key file is then transferred to the master/S3 bucket for the reduce task.

ReduceTask


MasterChannelHandler.SORT_DONE
1
MasterChannelHandler.START_REDUCE
2
JobManager.startReduceJob()
3
MRJobManager.startReduceTaskOnSlave()
4
SlaveChannelhandler.START_REDUCE
5
TaskManager.startReducetask()
6
ReduceTask.start()
7

a. Master receives SORT_DONE message after map phase is finished and the map output data is sorted and grouped by Key.
b. Master calls startReduce() and keeps track of the job status.
c. Interface of JobManager
d. Starts Reducetask on slave
e. Slave recognizes that it has to execute Reduce task
f. TaskManager keeps track of Reduce task on the slave and invokes ReduceTask on slave
g. ReduceTask is the class where the ReduceTask is executed. The task generates <KEY , Iterable<value>> for each file it receives and calls reduce function for each key and writes the output to a part file.

4.5 Status

File Operation Status:

1. NOT_STARTED: File is not processed yet by map or reduce task.
2. IN_PROGRESS: File is being processed by map or reduce task.
3. FAILED: File processing failed.
4. DONE: File is processed by map or reduce task.

Job Status:

1. RUNNING_MAP: Map phase is in progress
2. MAP_FAILED: Map phase failed.
3. MAP_COMPLETED: Map phase completed
4. RUNNING_REDUCE: Reduce phase in progress
5. REDUCE_FAILED: Reduce phase failed.
6. REDUCE_COMPLETED: Reduce phase completed
7. RUNNING_SORT: After Map phase is completed, sorting phase starts where it sorts all map output data.
8. SEND_SORT_COMPLETED: Sort phase completed.
9. PHASE1: Master waiting for samples.
10. PHASE2: Slaves waiting for pivots.
11. PHASE3: Master waiting for sorted data.

Slave Status

1. IDLE: Slave not doing any task
2. MAP_RUNNING: Map task in progress.
3. MAP_FAILED: Map task failed on slave.
4. REDUCE_RUNNING: Reduce task in progress
5. REDUCE_FAILED: Reduce task failed on slave.
6. PART_MAP_DONE: Map task completed on slave.
7. PART_SORT_DONE: Sort task completed on slave.

8. SORT_RUNNING: Sort task in progress.
9. PART_REDUCE_DONE: Reduce task completed on slave.
10. PHASE1: Slave generating samples.
11. PHASE2: Slaves waiting for pivots.
12. PHASE3: Slave sorting the final data.

PERFORMANCE ANALYSIS

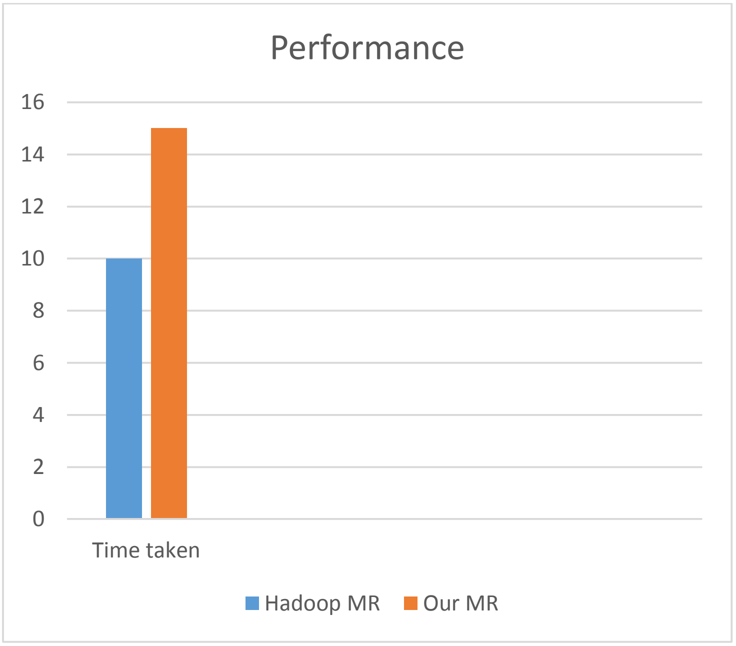
Our API-

A2 Assignment was run.

Time taken to run EC2 cluster of one master and 2 slaves: 13 mins

Hadoop API-

Time taken to run EMR cluster of one master and 2 slaves: 10 mins



Configuration:

2.7GHz Intel Core i5

8GB Mac Book Pro

HOW TO USE OUR MAPREDUCE API

1. Our MapReduce Framework expects input data to be present in “input” folder and Output directory to be specified as “output” in Local file system or S3.

2. Map and Reduce input type is expected to be string.

3. Required parameters when submitting a job are jar name, job name, mapper class, reducer class, input path(bucket name if running on AWS) and output path(bucket name if running on AWS).

FUTURE SCOPE

1. The existing design assumes that the key value pair sent by the mapper as well as reducer is a pair of String, String. But this feature can be made more generic by having a generic Output key Class and generic output value class.

2. Though the framework deals with a few fault tolerance situations, an extensive mechanism to deal with it is needed.

3. Improve the efficiency of the file transfer.

REFERENCES

1. http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html
2. https://en.wikipedia.org/wiki/MapReduce
3. http://www.jcraft.com/jsch/examples/
4. http://tutorials.jenkov.com/java-reflection/index.html
5. http://www.java2s.com/Code/Jar/h/Downloadhadoop0202coresrcjar.htm