CS 6240 Report - Map Reduce Framework

Alwin Mathew, Ankita Muley, Navin Madathil, Revon Mathews

ABSTRACT

Map Reduce has a wide applicability in storing and processing huge datasets on commodity hardware. In this project, we implement a Map Reduce framework similar to Hadoop. From first principles. We expose a Map Reduce interface to clients, who can then use these interfaces to build and run their own Map Reduce programs.

1. INTRODUCTION

Our basic MapReduce framework processes unstructured data parallelly across the distributed cluster of processors or on a stand-alone mode. We have attempted to emulate the actual Hadoop architecture. The main aim of the project is to understand the working and implementation of MapReduce. The core idea behind MapReduce is mapping the data set into a collection of <key, value> pairs, and then reducing all the pairs with the same key.

In this project, we use basic java libraries to implement network communication, job scheduling, mapper API, reducer API etc.

In this report, we describe the architecture design and implementation of basic MapReduce framework that we have implemented. We have measured its performance and scalability, compared it with the results we got from Hadoop framework and also driven its future scope.

2. OVERVIEW

At the highest level, there are three independent entities:

1. A Job Client which submits the job.
2. The Job Tracker which co-ordinates the job with the task trackers. It's a Java application whose main class is Job Tracker.
3. The Task Trackers which runs the tasks into which a job has been split.

1. Job Submission

The process of job submission, checks the output specification of the job. For example, if the output directory already exists, if it is throws an exception. Having submitted the job, waitForCompletion() of the job. When the job is complete, if it was successful, we get the Job Completed message and total running time of the job. Otherwise, an exception is thrown that caused the job to fail.

The job submission process implemented by JobClient does the following:

1. Assign a new job ID to the job.
2. Establish a connection with the JobTracker.
3. Pass the client job to the JobTracker and retrieve input splits contexts instance.
4. Assigns task to the task trackers.

2. Job Initialization

When the Task Tracker receives a task, it puts it into an internal queue from where the Task Scheduler will pick it up and initialize it. The Task Scheduler first retrieves the input splits. It then creates one map task for each split.

3. Task Assignment and Task Execution

In our design, a task tracker will be able to run only one map or reduce task at a time. Task Scheduler after selecting the task, runs the map task on the task tracker. The task scheduler executes all map tasks before executing reduce task. If all the tasks in the task queue are processed, it will notify Job Tracker about the completion of map task queue and the reducer is started. On completion of the reduce task, its status is informed to Job Tracker.

4. Progress and Status Updates

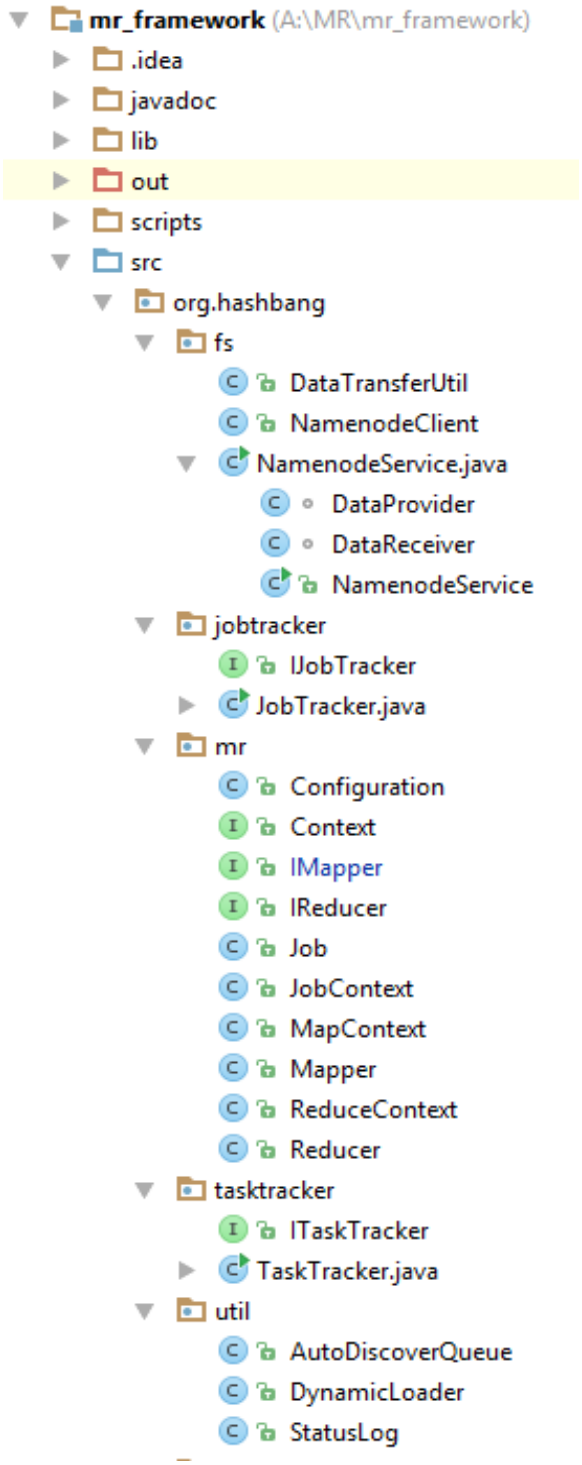
Status update and progress are logged on the console. Console log includes information about status of task assignment, task completion and job completion.

5. Job Completion

When the job tracker receives a notification that the last task (reduce task) for a job is complete. It changes the status for the job to "Job Completed". Then, when the Job polls for status, it learns that the job has completed successfully, so it prints a message to tell the user and then returns from the waitForCompletion() method.

3. DESIGN

3. 1 Project structure



3.2 Design: High-level

The code is primarily distributed into 4 main logical packages:

(i) fs - FileSystem package

(ii) jobtracker – Job tracker package

(iii) tasktracker - Task tracker package

(iv) mr – Mapper/Reducer package

(v) util - Utility package

(i) fs - FileSystem package

This package is centered around the NamenodeService, and, provides file system services to the mappers and reducers. This provides the a data transfer utility which is leveraged by the framework components for file operations over the network. The APIs provided in this package abstract the physical location of the file being read or written. The NameNodeClient, for instance, connects to Namenode server and fetches data. For Map requests, the DataProvider class serves the request of input splits to the TaskTracker. For reduce, the DataReceiver class fetches data from the the reducer and writes it to the output directory specified by the job client.

(ii) jobtracker - Job tracker packages

This package is part of the core of the business logic of this project. It is where jobs and entire lifecycle is tracked and managed. It is the Job tracker who receives the job from the client thus becoming an entry point to the framework.

(iiI) tasktracker - Task tracker packages

This package is the other core component of the project. This is where the individual tasks are tracked and managed.

(iv) mr - Mapper/Reducer package

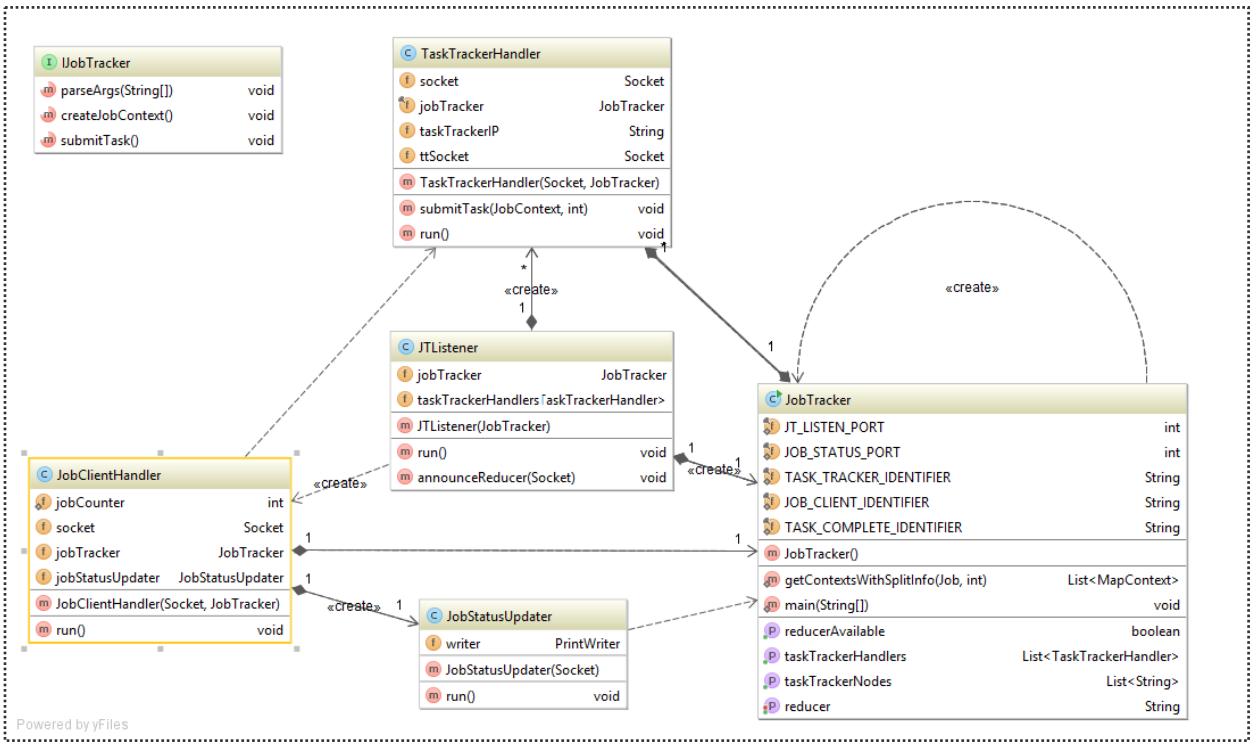
This packages host majority of the model classes that are consumed by the core business logic. More importantly, this package hosts the classes that we expect the consumers of our map-reduce project to extend. This package also contains the Job Class which is used to create a job and submit to the Job Tracker. The Configuration class provided in this package gives the user the abiity to change the block size of the input file. It also enables the user to define custom parameters to be used in the map/reduce phase of the job.

(iv) util - Utility packages

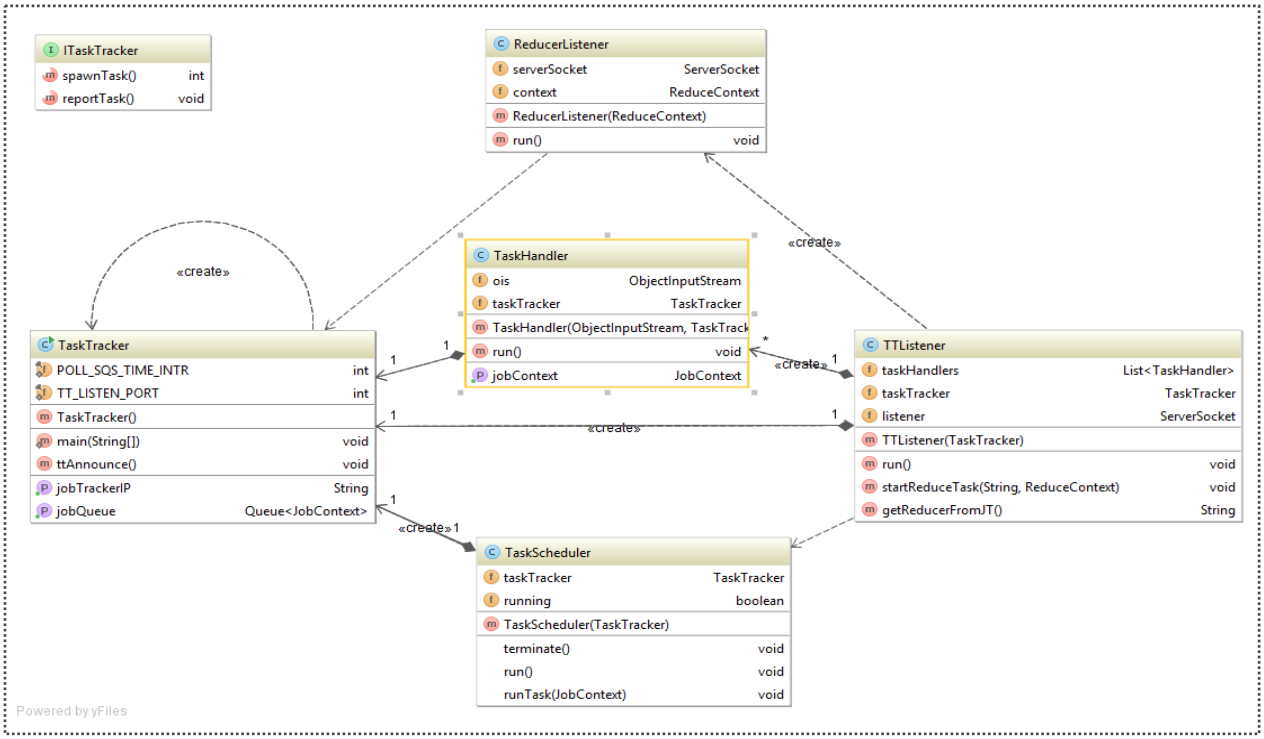
Last but not the least we have the util package which houses utility logic that the other modules can consume. Utility package houses the AutoDiscovery feature of the framework. It is used by the Job tracker to announce its IP to the potential task trackers. Util package also includes a status logging class which logs the job/task status to the Job tracker which inturn is provided to the user for tracking the status of the job. The packages additionally provides a dynamic class loader. This is a critical component in the framework, as the user provided classes need to be loaded to the JVM on job/task trackers before they can be used.

3.3 UML DIAGRAM

3.3.1 JobTracker



3.3.2 TaskTracker



4. IMPLEMENTATION OVERVIEW

As asserted above, the jobtracker package forms the core of the business logic of our project. There are two main processes in this package, which maintain the life cycle of all mappers and reducers.

JobTracker

The first of these is the JobTracker, which on its inception spawns a thread (JTListener) that listens on a server socket for message that are destined to the JobClient, TaskTracker and even messages that track the completion of tasks. Further details on how the listener processes these messages are provided in a subsequent section.

TaskTracker

The next process, again at the heart of our project is the TaskTracker. The TaskTracker on its inception spawns two threads - TaskScheduler and TTListener which schedule tasks and listen for tasks respectively. When TTListener receives information about a number of tasks to handle, it spawns TaskHandler instances which handle the same.

Clients who wish to use our MR Framework need to extend the Mapper and Reducer classes which are provided in the mr package.

- *map(Long key, String value, MapContext context)*
- *reduce(KEYIN key, List<VALUEIN> values, ReduceContext context)*

These classes operate on key value pairs which can correspond to arbitrary classes that are Serializable. This is important because instances of the primitives on which the Mappers and the Reducers operate need to be transferred over the network.

NameNode Service

The NameNode Service is responsible for handling all file system interaction. When the NamenodeService process starts it instantiates a server socket on port 9999. Here it listens for calls from mappers and reducers to provide and persist data respectively. Data to be provided to the mappers is handled by a DataProvider thread, and data to be persisted (the output of the reducers) is handled by a DataReceiver thread.

5. IMPLEMENTATION DETAILS

When the JobTracker starts, it pushes its IP to a global queue service to notify the TaskTracker of its IP. In order to facilitate this we leverage Amazon SQS. Amazon Simple Queue Service (SQS) is a fast, reliable, scalable, fully managed message queuing service. SQS can be used to transmit any volume of data, at any level of throughput, without losing messages or requiring other services to be available.

JTListener: The lifecycle of a job begins at the JobTracker via the JTListener thread. Whenever a request comes in to schedule a new Job, the JTListener dispatches a new JobClientHandler to process that job.

JobClientHandler: The JobClientHandler computes the number of splits required to process the job. A limitation of our current system is that it uses a fixed predefined size - one block - as the amount of data that

each task must process. Hence if the input file had a size of 50 blocks we would use 50 tasks to process the same. The advantage of such a design however is that the data required for each task is localized, and can be retrieved from the NameNodeService in a single request thereby facilitating fast processing of the data. Once the required number of splits is calculated, a new MapContext with all the relevant details such as - Job object, the name of the input file to process and the section of the file to process is instantiated. For each MapContext a TaskHandler is created on the TaskTracker which processes the task.

JobStatusUpdater : The JobStatusUpdater is a simple thread that sends updates to the client regarding the status of the job and tasks. It runs on port 9092 and listens for updates from the JobTracker and TaskTrackers. These updates are directly sent to the client job. This way the user gets a real time feed of the number of map tasks that have been completed for a given job and whether the job has finished execution. It also gives the total execution time.

Similar to the JobTracker, when the lifecycle of a TaskTracker begins it instantiates two threads the TaskScheduler and TTListener.

TaskScheduler: The TaskScheduler continually polls the TaskTracker's job queue. Whenever a task becomes available the TaskScheduler fetches all necessary data to execute the task. This includes the data required to process the task and other job meta data such as working directory. After building this "JobContext", it then goes ahead and runs the task.

TTListener: The TTListener as outlined in the overview listens for new task requests on port 9091. Whenever it receives a request to process a set of tasks, it retrieves from the Server socket the number of tasks to process and the MapContexts pertaining to each task. It then adds this task to the task trackers job queue so that the scheduler can pick up the task and execute the same.

The NameNodeService is responsible for handling all file system requests. Whenever data is requested by a Mapper or a Reducers the NameNodeService schedules a DataProcessing thread to handle the request. The data processing threads (DataProvider/DataReceiver) leverage the utility class DataTransferUtil to help process these requests. The DataTransferUtil has a number of handy methods such as readFileToBuffer, readStreamToBuffer, writeFromBufferToStream which abstracts a lot of the complexity of implementation from the data processing threads.

Whenever a client wants to leverage our MR framework he must extend the Mapper or the Reducer classes which are a part of the mr package. This package contains implementation details of how KV pairs are fed and read from the Mappers and the Reducers. The clients on their part after processing a key value pair (in the mapper say) write the result into the respective MapContext/ReduceContext.

Our framework then reads the data from these context and then processes them as necessary. For instance, every time a key value pair is output by the Mapper, our framework groups the values output by the client by their corresponding keys.

6. PERFORMANCE EVALUATION

In this section, we analyze the performance of our Mapreduce implementation. We have written an extensive test suite for our framework which tests it rigorously for varying input sizes ranging from a few 100 megabytes to 5 gigabytes. All our tests were done on ec2 instances which were dynamically instantiated before the tests.

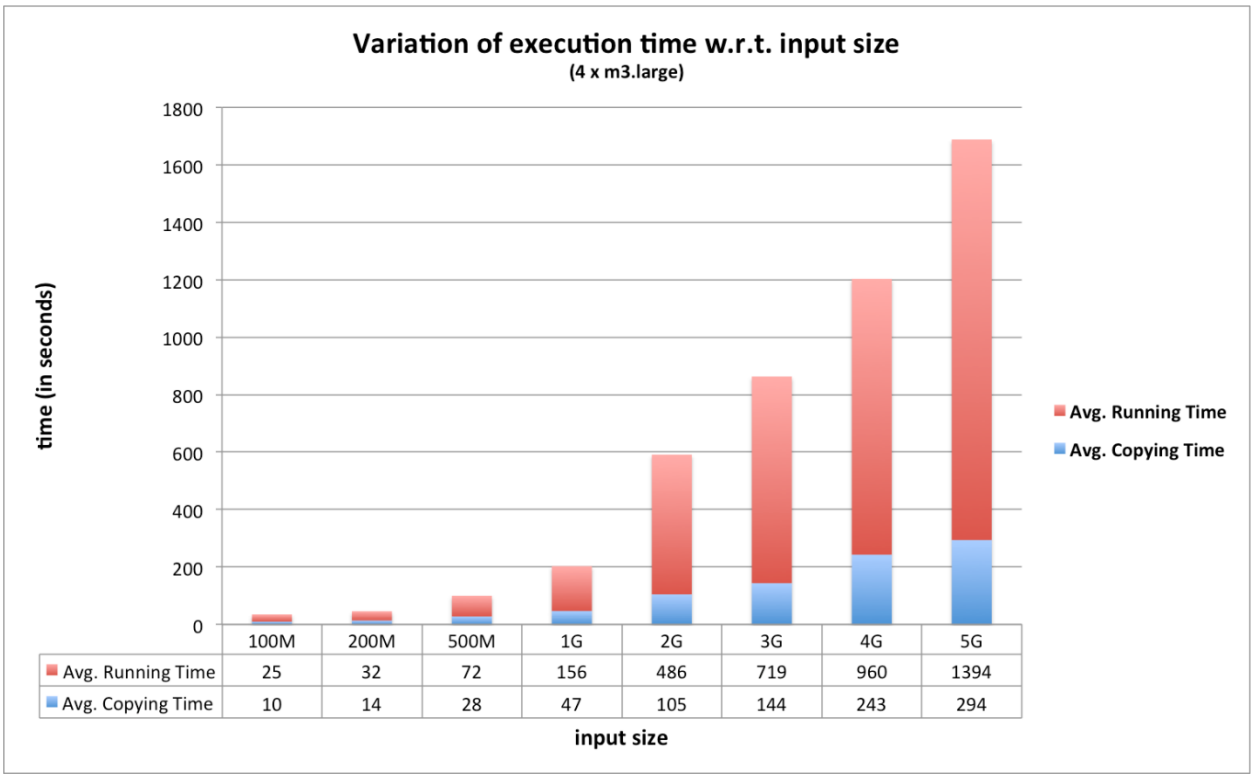
The following table shows the different instance types we used. (fig. 1)

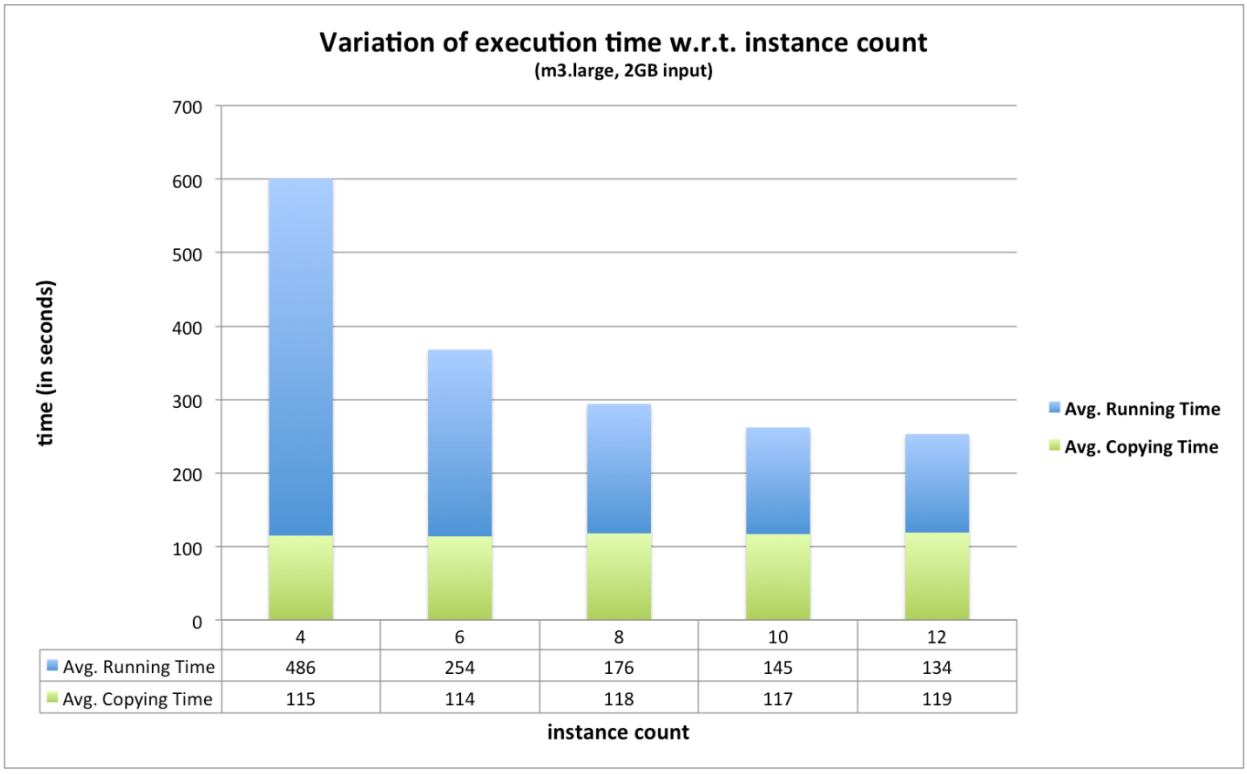
(fig. 1)

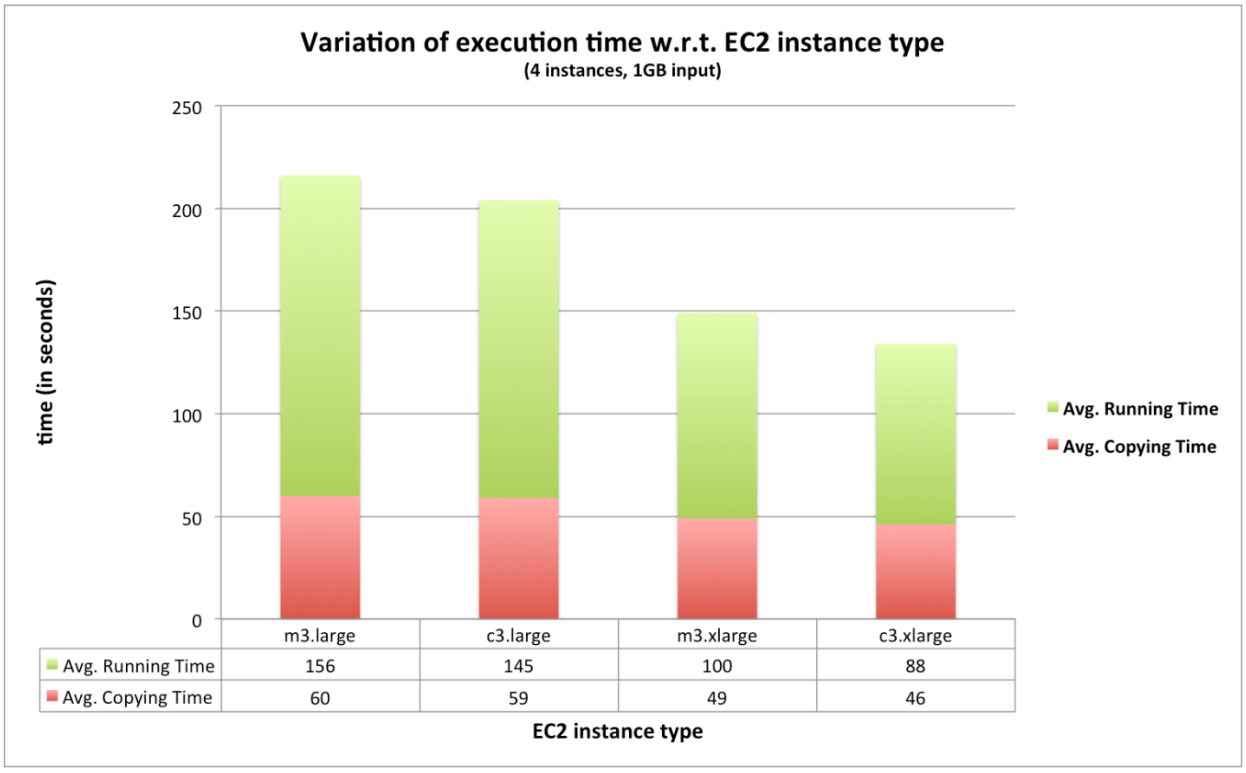
Instance Type	CPU Type	No. of vCPUs	Memory (GB)	Storage type	JVM max. memory (GB)
m3.large	Intel Xeon E5-2670 v2 at 2.5GHz	2	7.5	SSD	6
m3.xlarge	Intel Xeon E5-2670 v2 at 2.5GHz	4	15	SSD	12
c3.large	Intel Xeon E5-2680 v2 at 2.8GHz	2	3.75	SSD	3
c3.xlarge	Intel Xeon E5-2680 v2 at 2.8GHz	4	7.5	SSD	6

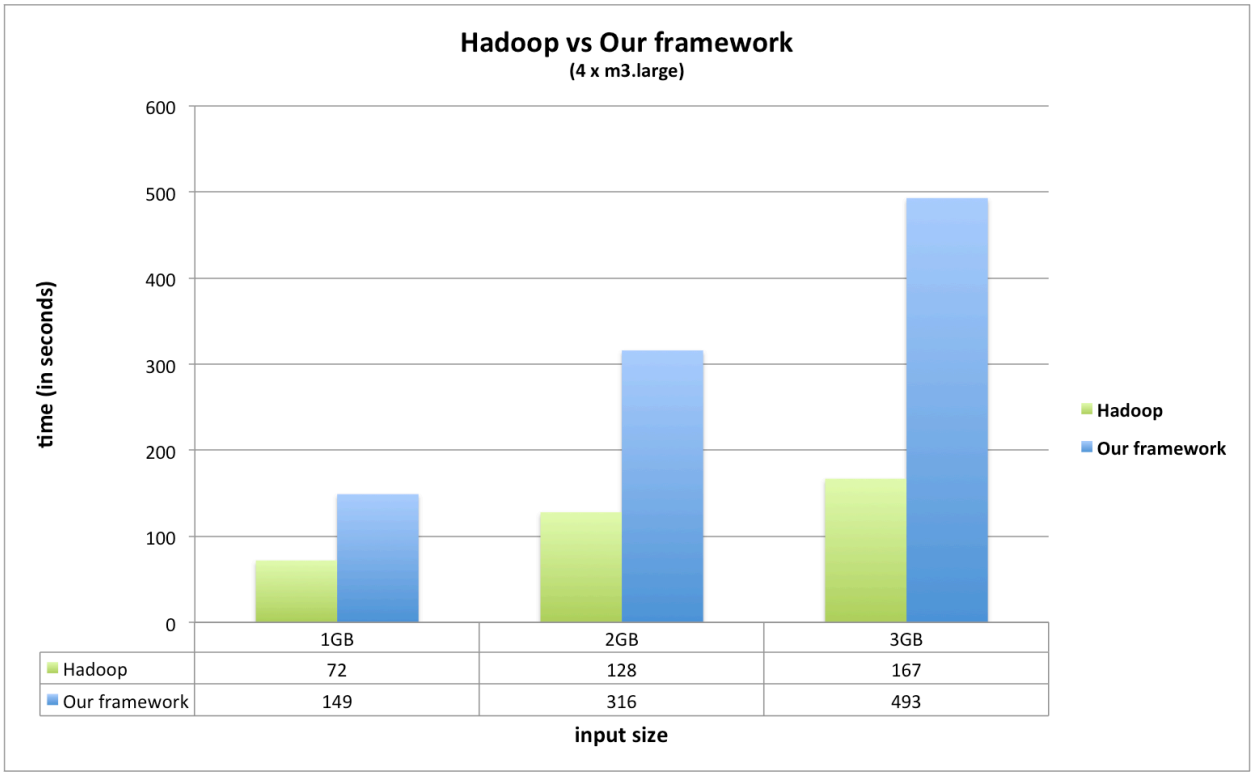
To thoroughly analyze the performance, we included cases which test the framework by scaling out and scaling up the hardware, and also feeding inputs of varying sizes. Scaling up is done by using machines with higher processing power and physical memory. Scaling out is done by increasing the number of machines (task trackers) thus helping the framework to effectively distribute the tasks.

We have run all our tests with the MedianPurchase(Assignment A2_V2). All our tests were conducted on machines with SSD as we did not want the disk read/writes to become a bottleneck. The results are shown in the graphs below.




Variation of execution time w.r.t. instance count
(m3.large, 2GB input)
time (in seconds)
700
600
500
400
300
200
100
0
Avg. Running Time
Avg. Copying Time
4
6
8
10
12
Avg. Running Time
486
254
176
145
134
Avg. Copying Time
115
114
118
117
119
instance count


Variation of execution time w.r.t. EC2 instance type
(4 instances, 1GB input)
time (in seconds)
250
200
150
100
50
0
Avg. Running Time
Avg. Copying Time
m3.large
c3.large
m3.xlarge
c3.xlarge
Avg. Running Time
156
145
100
88
Avg. Copying Time
60
59
49
46
EC2 instance type



6. POSSIBLE IMPROVEMENTS

i) Single reducer

In our implementation, we set the number of reducers as one. As a future enhancement we could optimally choose the number of reducers .

ii) Parallel execution of tasks per task tracker

Even though all the tasks to be executed are received by the task tracker, our current implementation only permits executing one task at once. As a future improvement, we would like to incorporate some parallelism into the task execution.

iii) File Splitting

The file splitting logic ignore the last and first line of the block. This needs to be rectified in the future iterations of the framework.

iv) Compression

Even though we send the intermediate files to the reducer compressed, we have not implemented the same for the input file. We had it in our scope but the sending of block chunks over network made it a little more complicated and we did a scope reduction.

7. CONCLUSION

This project was developed to understand the basics of a MapReduce framework. It is a basic prototype of the actual framework and can be improved in terms of performance, functionalities and accuracy.

We have identified some key areas where there are scopes for improvement as described in the previous section.

8. REFERENCES

HDFS Architecture Guide
https://hadoop.apache.org/docs/r1.1.2/hdfs_design.html
Apache Hadoop Main 1.2.1 API
http://hadoop.apache.org/docs/r1.2.1/api/