Parallel Data Processing using Mapreduce
PASS MapReduce API & Framework
CS6240 Spring 2016

Ajay Subramanya, Prasad Memane, Smitha Bangalore Naresh, Swapnil Mahajan

Introduction:

We are a team of four trying to build a MapReduce framework analogues to hadoop.

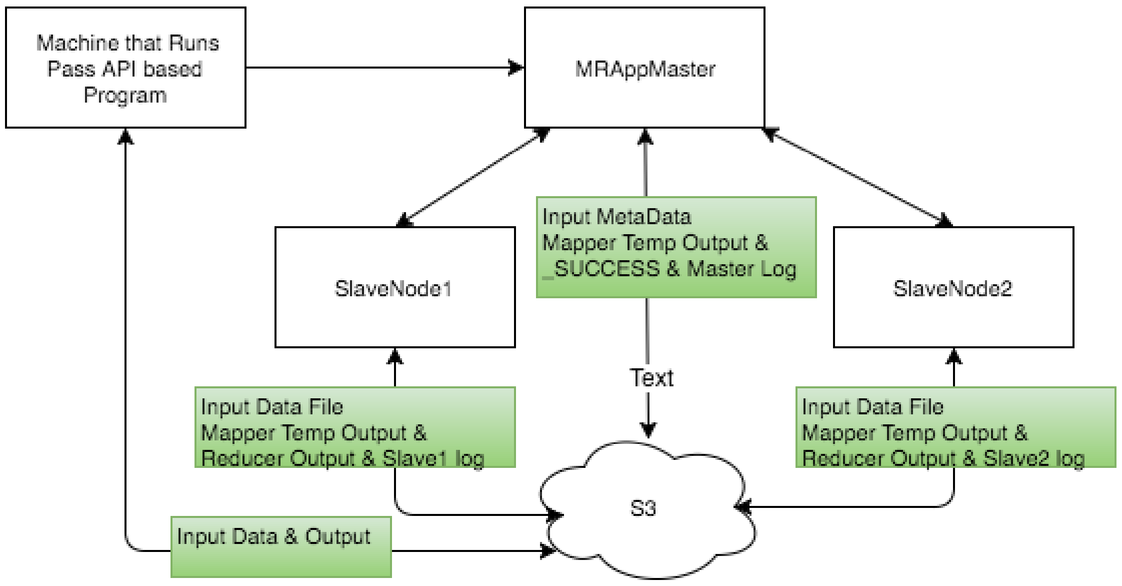
Why? : We have used hadoop extensively to write jobs that process huge datasets, we understand how to use the hadoop's APIs but we now are trying to dig deeper and figure out what happens beneath the hood.

We call this API - PASS API based on our initials.

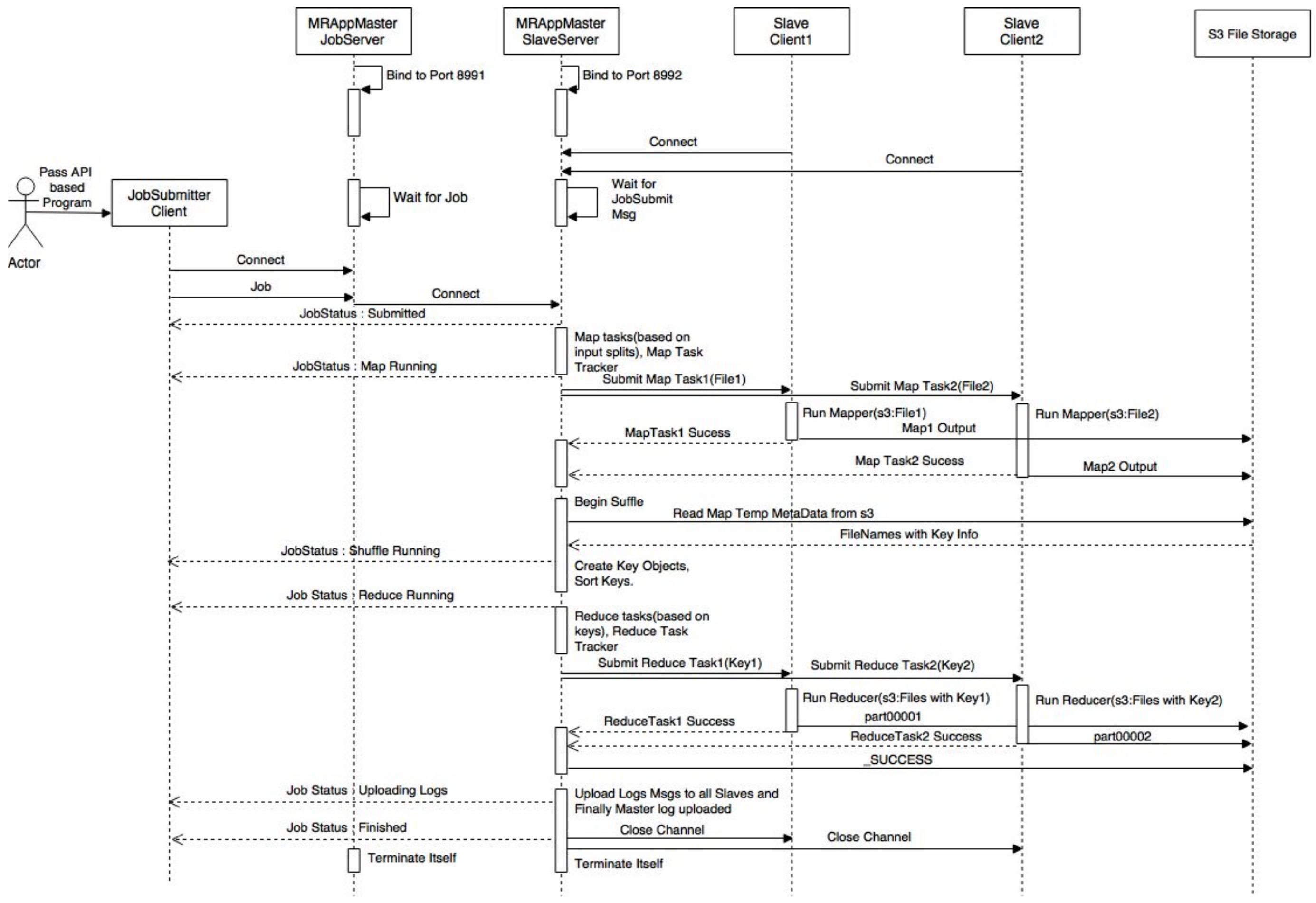
Design:

1. **Brainstorming Design:**
   Design based on YARN approach and Hadoop MapReduce API. Before coming up with the design for implementing MapReduce framework we brainstormed. For running on distributed system we tried to understand YARN 's components and how they start master and slaves and interact with each other. For understand the MapReduce Hadoop API we refered online sources and understood all the methods that our PASS MapReduce API needs to implement.

2. **System Design:**
   Main actors in our PASS MapReduce model are as follows:
   a. Local Machine/Any Machine: To run user jar on local(pseudo setup mode) or ec2(cloud mode where provisioning setup is provided)
   b. S3: shared FS
   c. Master & Slave machines


Machine that Runs Pass API based Program
MRAppMaster
SlaveNode1
Input MetaData Mapper Temp Output & _SUCCESS & Master Log
SlaveNode2
Text
Input Data File Mapper Temp Output & Reducer Output & Slave1 log
Input Data File Mapper Temp Output & Reducer Output & Slave2 log
S3
Input Data & Output

3. **Activity Diagram/Workflow**



Link to Diagram(For better view) :
https://drive.google.com/file/d/0B9Lnpdbxpk9bSkNHRnU4QWxvZE0/view?usp=sharing

Building Blocks:

1. **MapReduce-API:**

   Job : The job class is the main point of entry to the API for the user mapreduce program. It has two methods - submit and waitForCompletion.

   a. submit : accepts a user job and models it into a Job object and sends it to the MRAppMaster. Here the main components are validation and network communication. Validation is making sure the inputs and the parameters supplied by the user are as required by the contract of the API. Once the validation is complete we then send the job over the network (not exactly if we are running in pseudo) to the master. Here we connect to the master and send the validated job object.
   b. waitForCompletion : Once the job is submitted to the master - it is polled from time to time to check for the status of the job. How this works is the master keeps the API informed of the status as and when it changes , but if the API does not hear from master for over a minute the API sends a status request to the master, and the master replies with the status.

   Mapper : This is a class extended by User Defined Mapper which provides similar functionality as Hadoop Mapper class, Setup, map, and cleanup. It also has run method which is called by slave for assigning a specific Map Task.

   MapperContext : MapperContext object is used by each Mapper instance, which will hold the references for
   1. reader object, that reads the data from the input file, generates a Long key and returns the key and read data row to the map() method where it is processes.
   2. Writer object, this is used when map() / cleanup() calls write with key and value to write the map() output to be used by reducer phase. Writer writes the objects instead of simple text, as this facilitates the read operation in ReduceContext

   Reducer : This is a class extended by User Defined Reducer which provides similar functionality as Hadoop Reducer class, Setup, map, and cleanup. It also has run method which is called by slave for assigning a specific Reduce Task.

   ReducerContext : ReducerContext object is sued by each reducer instance, which will hold references for
   1. All the related files for a particular key on which current reduce() is being performed

2. Reader : for each key, MapperContext() writes the data in a key -> List<Values> format. Reader reads the files for a single key, and consolidates all the values and passes its Iterable to the the reduce method where all the values are reduced in either user defined reduce(), or the reduce() method defined in Mapper.
3. Writer: this object holds the reference to a local file part-r-0000X that will be used to write the output of current reduce(). After reduce is finished, this method writes the key as a string and value as a String to this file.

Custom Data Types :

Similar to the Hadoop writable objects, we have defined our own primitive datatypes, to be used by the API and to ease the porting of existing Hadoop codes to suite to PASS API.

2. **MRAppMaster:**

   Main class of the Master. MRAppMaster spawns 2 servers i.e. JobServer and SlaveServer. SlaveServer opens a netty server socket to communicate with Slaves. JobServer opens a netty server socket to listen to incoming job requests.
   Master and Slaves are up and running and established communication before a user can submit the job. If user submits the job any time before error message is thrown.

   To handle the job effectively 3 state machines are implemented.
   JobStateObserver : Maintains the different phases during running the Job(Submitted, Map, Shuffle, Reduce, UploadLogs, Finished, Error)

   SlaveServerStateMachine: Maintains all the SlaveChannels, JobSubmitterChannel and SlaveHeartBeatInfo table. When channel is idle heartbeat of 60 secs from every slave is received and Master keeps tracks of the active slaves.

   TaskStateMachine: Used to store, track, schedule, scheduleNext, update the tasks within a phase. Each phase is associated with List<Tasks> and TaskTracker.

   SlaveServer and slaves wait idly till Job Server receives Job. Once the JobServer receives Job it connects to SlaveServer and sends a Job object. Input Split is obtained and a Queue of MapTasks are created for each input split. Then MapTask tracker is setup to track the time and status for each task. First set of Map tasks is sent to slaves and removed from Queue. Slave executes the task and replies with the TaskResult object and task tracker is updated for the taskid and next task in the Queue is scheduled.
   Once all the map tasks are finished total time for map tasks is computed and logged. Begin the shuffle phase. We are using s3 as a shared filesystem. Obtain metadata of all the files in MapTemp output and get the keys form the filenames. Sort the keys by forming mapKeyOutputClass objects. Time taken for this is logged.
   Now we have a set of keys and reduce tasks are formed and reduce task tracker is setup. First set of reduce tasks from the Queue is dispatched to slaves and upon

receiving TaskResult task tracker is updated for the taskid and next task is the Queue is scheduled.
Once all the reduce tasks are finished successfully _SUCCESS file is uploaded and time taken is logged and Upload log Task is sent to slaves and on success master writes its log and sends finished status to JobSubmitter and closes slave channels and jobsubmitter channel and finally itself.
On error in any of the above tasks then error is returned to the user.

3. **Slave**: This is used by the master to execute tasks. A task may be a Map Task, Reduce Task or Upload Logs task. There is a channel established between each Slave with the master. The master sends down a task object every time it wants to some job done. The Slaves understand the task object and do the necessary task that the master requested and write the result to s3 and inform master about the success or failure of a task.

   There are mainly three parts to the slave - Handlers, Java Reflection and S3 Wrapper

   a. Handlers : There are mainly three handlers - SlaveHandler, MapHandler, ReduceHandler and finally LogHandler. Slave handler is used to send the initial message to the master saying that it was spawned and ready for tasks.
   b. Java Reflection : Used to load the user jar dynamically and then load the user written mapper or reducer class and then the methods in those classes.
   c. S3 Wrapper : Used by the slave to write intermediate and final results to s3.

4. **Common:** Is used to hold class that are or have the potential to be used in more than one module. This is added as a dependency in all the modules. The most significant Class in this would be ‘Task’ which models a task that the master sends down to the slave.

5. **Provisioning:**
   Provisioning takes care of spawning the EC2 machines and tagging them as master and slaves. It initiates the network communication between them. Also it makes sure to run the respective scripts on the master and the slave machines. These scripts downloads the respective jar and log4j.properties file from S3 and runs the jar with the appropriate paramaters. Once this is done, the Provisioning will ask the user on his local machine to input the command to run their mapreduce program. This program is run in a separate JVM on the local machine and the mapreduce tasks are performed on the EC2 cluster spawned before. Once the mapreduce tasks are completed with a success or failure, the EC2 instances are terminated. However, the logs are uploaded to AWS S3 and can be viewed in case something goes wrong.

Why netty.io ?

We initially thought that we may have to handle huge volumes of data flowing between components. And somehow using plain sockets did not appeal to us. We did not want to go through the pain of debugging network logs and working at socket/TCP level. We were looking for a library that abstracted these concepts and just gave us methods to read and write data from a channel. We were introduced to netty in the paper “Scaling spark in the real world - Performance and Usability ” by Michael Armbrush et al. They were using netty for the following reasons.

- Zero copy I/O : Instruct the kernel to copy data directly from on-disk files to the socket, without going through the user-space memory. This reduces not only the CPU time spent in context switches between kernel and user space, but also the memory pressure in the JVM heap.
- Off-heap network buffer management : Netty maintains a pool of memory pages explicitly outside the Java heap, and as a result eliminates the impact of network buffers on the JVM garbage collector.
- Multiple connections : Each Spark worker node maintains multiple parallel active connections (by default 5) for data fetches, in order to increase the fetch throughput and balance load across the nodes serving data.

The above features impressed us , and we started digging deeper into understanding netty. To be honest it was a steep task as there are limited tutorials, except for the ones by the developers of netty which is limited. Although the examples presented by them was helpful for us in a way that we were able to model our client server architecture using it. But beyond that using handlers and various other functionalities we had to refer the API docs and figure things out.

We started out building a POC which later transformed into our A9 - dsort project. Since we were comfortable with it after a9 we continued using it for the project, although we may not be leveraging all of the above mentioned features, but it gave us a clean framework to fit in our components.

What works, and what doesn't?:

1. What works?:
    a) Basic MapReduce functionality with framework to run the jobs on distributed framework.
    b) Having the API work for pseudo-distributed and distributed setup based on user preference.
    c) Successfully able to run WordCount and A3 with minimal modifications to existing hadoop programs.
    d) HeartBeat mechanism from slaves so master knows which slaves are active.
    e) JobSubmitter receive status of the Job from MRAppMaster and MRAppMaster informs about the state changes.
    f) Time taken for each task is tracked and logged along with the total time to complete all the tasks in a phase.

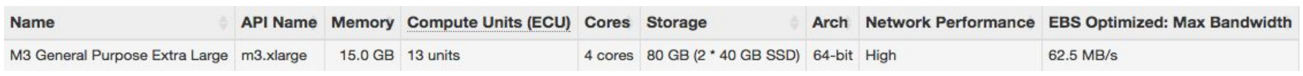
g) Logs uploaded to s3 at the end.

2. What doesn't?:
   a) Combiner
   b) Chaining jobs together
   c) Split a large file into manageable block size
   d) Replication of intermediate data
   e) Retry mechanism(But can be easily implemented with the framework we have built and data is stored in s3)
   f) Shuffle phase should ideally begin after 80% of Mapper phase . In Our API it waits till Map phase is completed
   g) Composite Object as a key

Comparison:

Cloud Configuration:

Master: m3.xlarge (1 node)
Core: m3.xlarge (2 nodes)
Zone: us-east-1



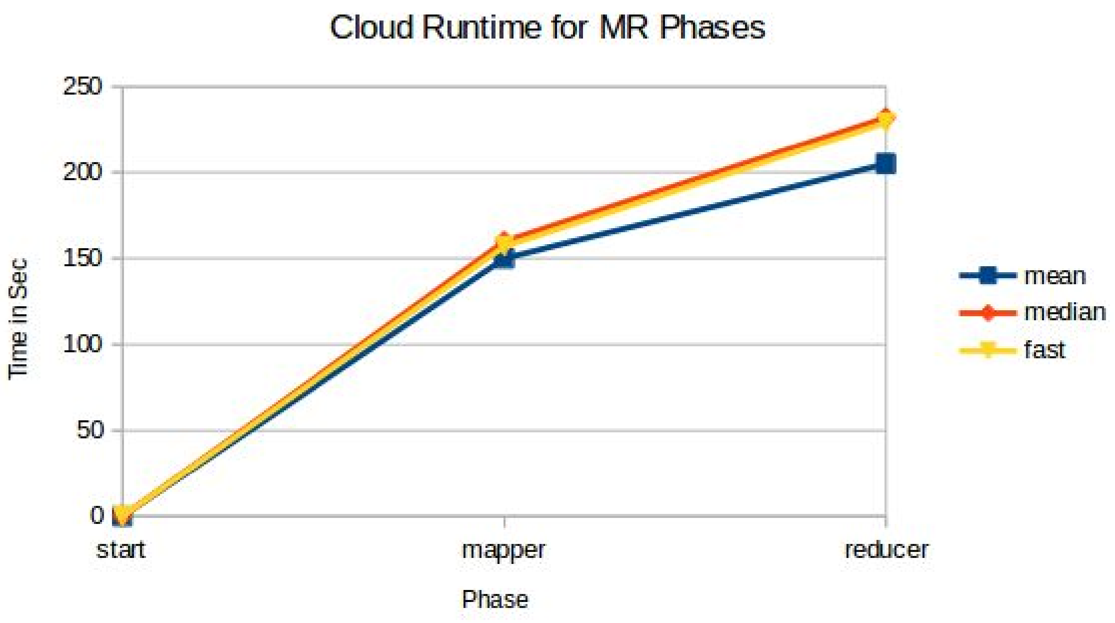
Name	API Name	Memory	Compute Units (ECU)	Cores	Storage	Arch	Network Performance	EBS Optimized: Max Bandwidth
M3 General Purpose Extra Large	m3.xlarge	15.0 GB	13 units	4 cores	80 GB (2 * 40 GB SSD)	64-bit	High	62.5 MB/s

Reference: http://www.ec2instances.info/

Local Configuration:

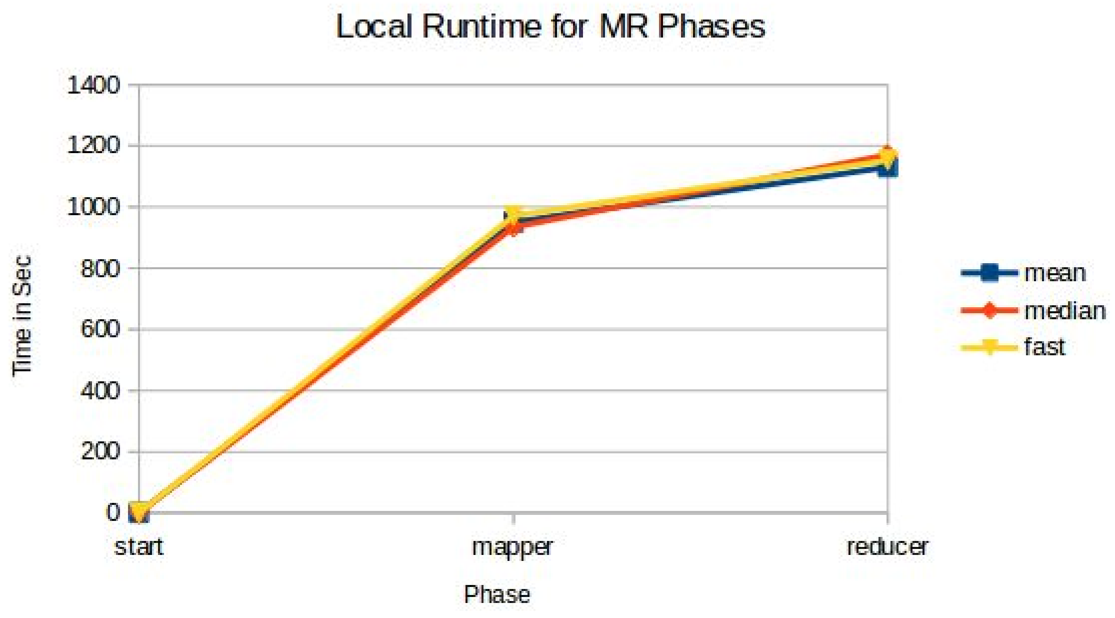
Processor: Intel i5-4210U @1.7GHz
RAM: 8 GB
JVMS Spawned: 4 (1 Master + 2 Clients + 1 for User's mapreduce program)
Average Network Bandwidth: 17.98 MB/s

1. PASS Cloud Runtime for different phases of MapReduce for A3:



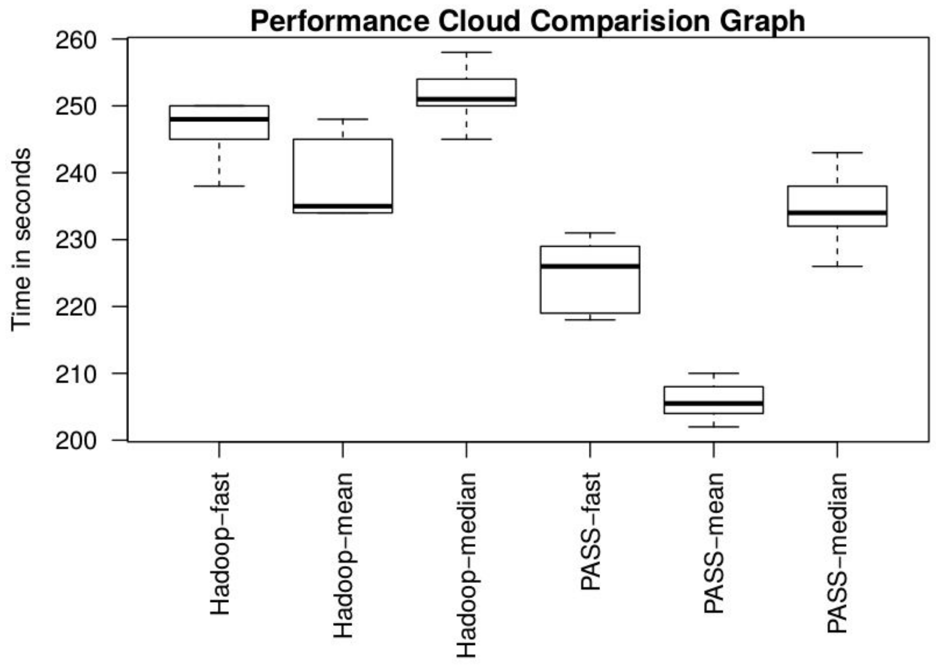
To point out an important feature in the above graph, the shuffle phase is not considered, since our design does not have a separate shuffle phase like traditional Hadoop MapReduce API. The shuffle phase in our design can be considered as the time required by each map-task to upload the key-files to S3 and time required by each reduce-task to download the respective key-files from S3. Hence this time gets incorporated into the mapper and reducer phases. This time is highly influenced by external components like AWS S3, Internet speed and Network bandwidth.

2. PASS Local Runtime for different phases of MapReduce for A3:



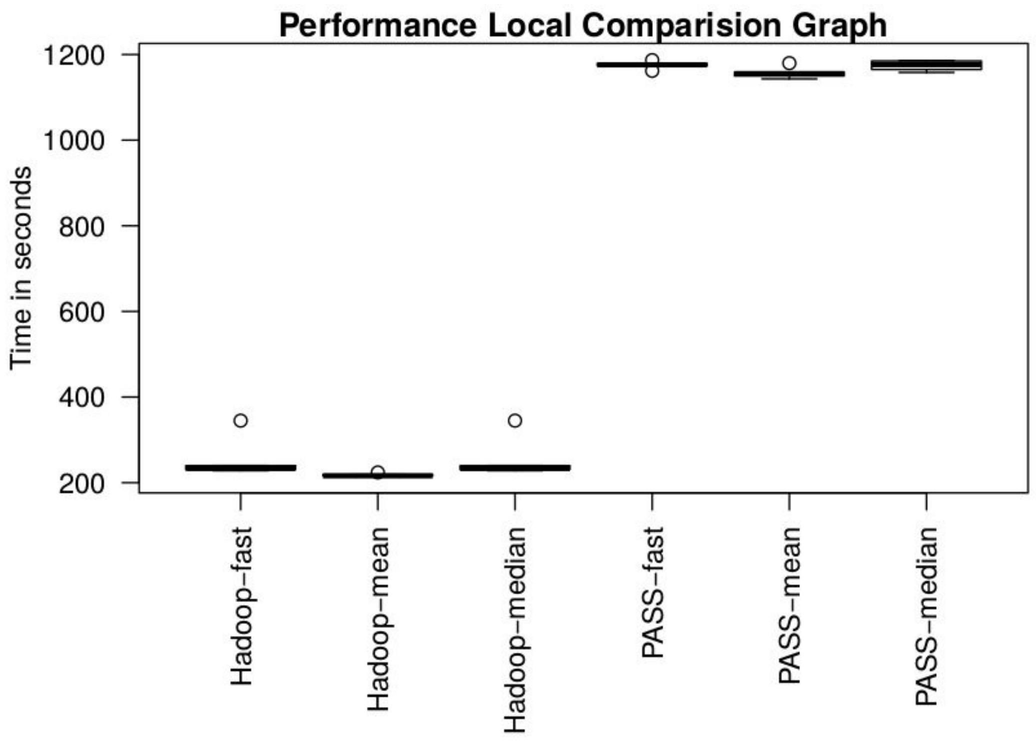
The local runtime is highly influenced by the user's local machine configuration , HDD and internet speed.

3. Performance Cloud Comparison (Hadoop Vs. PASS) for A3:



As stated above the shuffle phase is highly influenced by external components. Also, the runtime is for m3.xlarge machines for which the network performance is high and max bandwidth is 62.5 MB/s. Also the intermediate key files being generated after each map-task are of size 800kb on an average for A3. All these factors count into a lower runtime of A3 for our design as compared to Hadoop. However, this is also the bottleneck of our design. When the size of intermediate key files produced increases, the upload and download time will increase resulting into a higher overall runtime. The runtime will increase linearly with the increase in the size of the intermediate key files produced at each map-task.

4. Performance Local Comparison (Hadoop Vs. PASS) for A3:



The high runtime for local for the PASS-API as compared to the Hadoop is because, our system is dependant on the internet speed and mainly because a separate JVM is spawned for every instance of client, one for master and one for the local program. Hence the number of slaves spawned is the most prominent bottleneck of our design, since every slave runs in its own JVM and hence the additional overhead of spawning and maintaining the JVM.

5. Results comparison for A3:

Sample PASS-API result for A3

Month	Airline	Mean Ticket price in $
1	AA	513.6952712415
2	AA	505.0345978166
3	AA	503.9135312861
4	AA	500.7823133527
5	AA	499.6444374566

Sample Hadoop-API result for A3

Month	Airline	Mean Ticket price in $
1	AA	513.6952712415
2	AA	505.0345978166
3	AA	503.9135312861
4	AA	500.7823133527
5	AA	499.6444374566

6	AA	496.0579495592
7	AA	496.251921597
8	AA	495.6274112503
9	AA	491.5351910347
10	AA	491.1364271366
11	AA	498.4790232754
12	AA	508.9446672752

6	AA	496.0579495592
7	AA	496.251921597
8	AA	495.6274112503
9	AA	491.5351910347
10	AA	491.1364271366
11	AA	498.4790232754
12	AA	508.9446672752

Porting from Hadoop MapReduce API to PASS API:

1. The driver class of mapreduce program should instantiate the Job class of PASS API instead of the Hadoop MapReduce API and set the required parameters the same as traditional Hadoop MapReduce API the only difference being instead of setting by class the PASS-API's classes are set by class names as strings.
2. The users mapper should extend the PASS API Mapper and Reducer classes and use the PASS API defined generic data types.

PASS API	Hadoop MapReduce API
CustomInteger	IntWritable
CustomString	Text
CustomLong	LongWritable
CustomDouble	DoubleWritable

3. Use the PASS API's MapperContext, ReducerContext instead of Context object from Hadoop MapReduce API for mapper and reduce respectively.
4. If Custom Key or Custom Value classes/objects are being defined, then those classes should implement Serializable, KeyInterface and Comparable. Also, the toString method should be overridden.

Enhancements that can be done in our API:

1. For reducer, currently we are reading all the files for a key and providing the values for the key as an iterable to the reduce method. This results in memory overhead since we are holding all the values for a key in the memory. This can be improved by reading and passing every object/row from the file to the reduce method one at a

time, instead of loading all the objects/rows in memory and then passing it to the reduce method.
2. Writing generic interfaces for all user defined data types to be implemented. This will enable setting up a contract that a user-defined data type ,i.e., Composite Key or Composite Value has to follow.

Using HDFS over S3 as shared FS:

One of the major enhancements we could possibly do is use HDFS as a shared file system over S3. Using S3 has the following drawbacks ,

- The user of the API should have an AWS account, if not the API would not be very useful.
- For huge datasets there would be considerable amount of network traffic. HDFS also would have network traffic but that would be within the intranet and not over public internet.
- If S3 is down for some reason then the API would not work at all.
- Lots of configurations needed for authentication and access.

While there are a lot of issues there are some things that go it it's favor,

- It was easy for us to setup and use s3 , the java SDK that AWS provides does work decently well.
- Since we are not implementing replication of data and having fixed block sizes , it was just used as a shared file system , and nothing more. We wrote output of a phase to it and read it back in the next phase.
- Since we were delegating the task of data transfer to AWS S3 SDK, our implementations of mapper and reducers were simplified to a great extent. This in comparison to our distributed sort implementation (A9 - previous assignment) reduced lines of code by at least half.

We did look into the API's that HDFS provides for data transfer and storage, *org.apache.hadoop.fs.FileSystem* is a generic class to access and manage HDFS files/directories located in distributed environment. This could be accomplished as one of the future work/enhancement to this project which would allow for users to run their mapreduce jobs using our API in pseudo-distributed mode.

Conclusion:

We would like to conclude that working on this project made us appreciate the effort that going into building a system like hadoop. We have tried to simplify the implementations wherever possible, such as not having equal inputs splits or data replication. Although, we may have a simple implementation of hadoop API, we now know where the the design improvements that need to be done and how they could be done. For instance we already have a framework for implementing retry mechanism, composite key and Job Chaining but did not finish it due to time constraints.

Effort:

Smitha Bangalore Naresh:
Brainstormed the design with the team to come up with the architecture based on YARN and Hadoop API. Designed, Implemented and Tested MRAppMaster which is responsible for accepting the job and forming tasks and distributing tasks(Map, Reduce) to all connected slaves and does Shuffle by itself. MRAppMaster has main components Job Receiver, SlaveServer, StateMachine and TaskScheduler. Implemented Task State Machine and Job State Machine and TaskTracker which takes which takes the job from start to finish and finally informs the job submitter. Also provided the framework to measure time taken by each individual task and incorporated netty's heartbeat mechanism. Tried to understand java reflections and wrote a basic prototypes. Understood netty much better which helped to implement a clean program without any resource leaks. Involved in bug fixes across components by performing system tests. Made sure most of the classes in have extensive JavaDocs. With Swapnil tried to get composite key functionality working. Due to time constraints feature could not be tested so was not checkin. Contributed to writing the report as well as diagrams in report like system design and activity diagram etc.

Ajay Subramanya:
Worked with the team to understand the project requirement, brainstormed ideas and possible implementations of the mapreduce API. Once we had a fair idea as to what needs to be done, I worked on setting up the maven project, the idea was to have each abstraction as a module , so that they can depend on each other if needed. So I had a maven module called common which would house all the common methods and universal state machines. The common module is added as a dependency in all other modules. All other modules such as Master, Slave , Mapreduce-API and provisioning depend on each other in some way or the other. After the project was setup I worked on the Job class in our API, this involved validating the user inputs , creating a job object and sending over the job object to the Master. Then I worked on building out the Slave, here I had to accept the Task object that MRAppMaster sends and use Java Reflections which Smitha had built a prototype of. Using the prototype I built a class that accepts the jar file loads it to memory and then extracts the classes in it and finally runs the necessary methods in it. Contributed to the makefile, README and Report and other minor tasks.

Prasad Memane:
Worked with the team to understand the project requirement, brainstormed ideas and possible implementations of the mapreduce API. Worked on Provisioning which was built by Ajay for A9. Incorporated Provisioning to work with the current MapReduce design. It

involves spawning of ec2 machines, tagging them as master and slaves and initiating the network connection between them. It involves spawning a separate JVM to run the user's mapreduce code. Worked with Swapnil to write the scripts for spawning the ec2 cluster and running the user mapreduce code. Also brainstormed to decide which graphs to plot and worked on the same. Also figured out the runtime and comparison of the traditional Hadoop Mapreduce API with our PASS API and the entire design to find the good points and caveat, bottlenecks of our system design. Contributed to the README and Report.

Swapnil Mahajan:

Brainstormed with the Team in the beginning, to figure out the available options for architecture and implementation. Designed the PASS API. Implemented Mapper and Reducer classes which would be extended by user program for writing custom Mapper and Reducers. Implemented MapperContext and ReducerContext classes those will be used by Mapper and Reducers to store the runtime context of the current Map/Reduce phases.
Designed the Interface and implemented the primitive data types equivalent to Hadoop Writable (CustomLong, CustomString, CustomInteger and CustomDouble) that can be used to easily convert existing Hadoop Mapper / Reducer to PASS API cormat.
Designed and Implemented the scripts

1. buildAndCopyToAws.sh, which builds all the jars, generates the scripts required by Master and Slave nodes for bootstrapping , and starts the provisioning script.
2. pseudoSimulator.sh : This script checks for the OS version, OS X, or Linux, and calls the specific version script that simulates the pseudo distributed environemnt.
   It first starts a terminal and runs Master in a JVM, once the Master is started, it starts separate terminals for each slave which is very handy for tracking the progress.

Worked on README.txt and generated all the performance graphs required by the report.