Assignment: Final Project - Build your own Map-Reduce

<u>Team Members</u>:

Vedant Naik
Dixit Patel
Vaibhav Tyagi
Rohan Joshi
Priyanka Mane

Index

Objective

MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster. Conceptually similar approaches have been very well known since 1995 with the Message Passing Interface standard having **reduce** and scatter operations.

The objective of the project was to build a MapReduce**.**

Requirements

- To build a Map-Reduce system like Apache Hadoop M-R.
- The system should run both in parallel on Amazon EC2 and sequentially (or threaded) on our local machine. (described implementation in strategy)
- Sample interaction:

```
./start-cluster 8
./my-mapreduce WordCount.jar s3://foo/bar/input.txt s3://foo/bar/output
./stop-cluster
```

- The system should run the following programs, producing approximately the same output as Hadoop:
    - Hadoop Sample: WordCount.java (alice.txt as the input)
    - Hadoop Sample: WordMedian.java (alice.txt as the input)
    - Assignment 2.
    - Assignment 5.
    - Assignment 7.

Functionalities

- Implemented all utilities provided by the Hadoop framework in "coolmapreduce":
  - Mapper
  - Reducer
  - Job
  - Configuration
  - MapperHandler
  - ReducerHandler
  - Context

- In addition to this we also wrote our own files system implementation. It handles the following (explained in detail):
  - All **S3 file operations**. Example, writing to the S3 bucket, retrieving data from S3 bucket, etc. We have a FileSys class that handles functionalities of reading and writing to and from S3 bucket based on the input path.
  - **Operations between EC2 instances –** We are using Java's JSCP library for moving files between the different EC2 instances during the shuffling phase.
  - **Local file operations**, i.e., writing data to and from the mapper and reducer output and input locations respectively – we have implemented this functionality in the FileSys class. We are using the object Appendable to append serialized objects to disk files.

- We have also implemented a Shuffler in the MapperHandler and the first step of ReducerHandler.

Components
Following are the components of the system that we have built:

1. coolmapreduce

- Configuration – This file sets the cluster information for every job. In that we are storing the IP addresses of all EC2 instances that act as slave servers. Master server's DNS is also stored. By convention the last server DNS is taken as the Master DNS and rest all are considered slaves.
- Context – This class handles the logic for writing output from mapper and reducer programs. Since we are storing output values in folders associated with KEYs, we need to make sure the folder names do not contain special characters. But since the KEYs can contain any character and can be of any serializable type, it is hard to simply use the key.toString() as a folder name. To overcome this, we store the values in folders named as the hashcodes of the KEYs. We maintain the correlation between the hash-values and the original KEYs in a special map called MapperKeyMap (MKM). This is created during the map phase after a context.write(). In the reduce phase, the write method will write tab separated key value pairs from the reduce methods to a local file output/<JOBNAME>/reducer/part-XXXXX file.
- Job – This class sets a job specific parameters, for example jobname, etc. and implements waitForCompletion() method which runs the program on the cluster. The waitForCompletion() method starts off the Master program which instructs and coordinates with the Server instances running on the servers. The Master program synchronizes each "barrier" and makes sure that each Servers reaches up to a "barrier" and then move towards the next barrier.
- Mapper – This is an abstract class and every class that inherits this class must implement the map() method. It also implements the setup() and cleanup() methods so as to not mandate the implementations. If the user provides a setup() function, his function is called instead.
- MapperHandler – A MapperHandler gets a map job information and does pre-and-post cleaning of disk files. It calls the map() method of the job that writes the output on the disk. The data that is written to the disk is appended to the files based on KEY hashmaps. We are using the object Appendable to append the object in the serialized form in the disk files. After writing the map() output, it generates a list of KEYs (MKMs) and informs the master node about its KEYs for that job. The master does the load balancing and maps every key to a reducer. This step produces the BroadcastMKM (complete map of all hash-values to keys) and BroadcastKeyServerMap (which tells which key will be handled by which server in the reduce phase).
  Mapper handlers on each server then get these maps and begins the shuffle phase. In this phase, the mapper transfers all the valuesX.txt files for respective KEYs to the corresponding server. If the key is mapped to the same machine/node, it would be moved to the reducer input directory on the same machine else it SCPs' the file to

the remote location. Thus all output/<JOBNAME>/mapper/<KEY>/value<servernumber>.txt files are moved to the correct location as input/<JOBNAME>/reducer/<KEY>/value<servernumber>.txt files.

- <u>Reducer</u> – This is an abstract class and every class inheriting this class must implement the reduce() method. It also implements the setup() and cleanup methods() which are not implemented. If the user provides his own implementation, then his setup() and cleanup() are called instead. The Reducer follows the map phase.
- <u>ReducerHandler</u> - It gets the reducer job information from the master node. It merges all the value<servernumber>.txt files for each KEY and merges to a complete values.txt file (we used the object Appendable to merge serialized files). This is a complete file with all values for that key. Thus, while invoking the reduce method, we can simply use the fs.iter.FileReaderIterator.java class to get an iterable over all the values for any key. The reduce method then calls a Context.write which writes the output as per the logic in Context.java. At the end of the reduce phase, when all reduce calls and cleanup is complete, we move the output/<JOBNAME>/reducer/part-XXXXX to the output bucket in the output folder. The file is renamed such that the server number replaces the 0s. So, for example, server 2 will write the part file part-00002, server 10 will write part-00010, and so on.

2. Fs

- <u>FileReaderIterator</u> – This file reads the file from the disk as a generic iterator. This is a solution to the problem of being able to read and write collections to a file on the disk such that we can append to an existing file, and read using an iterable and avoid reading the whole file in memory. We use this to files written by the AppendableObjectOutputStream class.
  We do this by internally using a FileReaderIterator and implementing Iterator and Iterable interfaces. This class reads and caches one element in the file and overrides the next, hasNext() and iterator methods to let an object of this class be used as a normal iterator.
- <u>LoadDistributor</u> – This file does the load distribution on master based on the KEY file sizes. All the MKM merging and broadcast maps generation is done in this class. It also handles the logistics of the shuffle phase.
- <u>FileSys</u> – This is the class which handles all the disk operations of our program. Any read/write that needs to happen to a local disk or a disk on any other EC2 instance, happens through this class. FileSys.java is never instantiated and all methods are static. Methods in this class are well documented and can be referred for understanding how they are used.

3. I/O

- BooleanWritable – This file corresponds to the BooleanWritable class of Hadoop.
- DoubleWritable – This file corresponds to the DoubleWritable class of Hadoop.
- IntWritable – This file corresponds to the IntWritable class of Hadoop.
- LongWritable – This file corresponds to the LongWritable class of Hadoop.
- Text – This file corresponds to the Text class of Hadoop.
- Writable – This is an interface that extends Serializable class. All other classes implement this interface.

4. Master

It runs on the Master node. When a user starts a MapReduce job, it sends information to the data nodes in stages to start particular phases:

- MAPFILES: Send the file names to the slaves about which files to read and communicate end of transmission.
- FILESREAD: Wait for an ACK saying FILES_READ.
- MAPSTART : Send instruction to start reading the files.
- MAPFINISH : Wait for MAP_FINISH instruction from slaves.
- SendMKMs: It creates and sends the MKM's and master broadcast maps to all servers.
- SHUFFLEANDSORT: Instructs the shuffle of keys determined from the master broadcast map.
- SHUFFLEFINISH: Wait for an ACK for SHUFFLEFINISH.
- REDUCE: Send instruction to start REDUCE phase.
- REDUCEFINISH: Wait for REDUCEFINISH from all servers.

5. Server

This class handles incoming and outgoing network requests. On the master node, it sends requests to master node logic and on data node, it routes requests to mapper and reducer for the job.

6. Utils

- Constants – This class has information about all the global parameters required by coolhadoop. It provides flexibility to user to set up any path or constant based on the requirement.

Strategy

Mapping:

- Read input from S3 buckets
  Although Hadoop breaks files into chunks and works with these chunks from the map phase, to keep things simple, we treat each input file as a chunk.
- Identify each line as a Data Set
  We will be identifying each line as a data record and invoke a map function on it. For the purpose of testing, all input files are in the format .gz as we are only using a GZIPInputStreamReader. And for each line, we invoke the map function from the submitted .jar file.

Shuffle:

- Identifying Keys generated
  In the map phase, we write the context output for any key to a file within a folder dedicated for that key (output/JOBNAME/mapper/KEY/valuesX.txt) where JOBNAME identifies the MR job for which this mapper/reducer instance is running, KEY is the key for which the value is to be written and X is the server number on which the mapper instance is running.
  While writing values to any file, we use a custom object output stream class which helps us write lists of objects and read them without having to load the entire collection in memory. This is also used in the reducer.
  Since keys can be of any type and we want to create folders associated to these keys, we choose to create folders named by the hash code of the key. We maintain the co-relation between the hash code and the original key in a data structure named "MapperKeyMap", or in short, "MKM". We send these MKMs to the main server which then combines and broadcasts a global MKM for all the servers to refer to.
- Distribute keys among all instances
  Using the broadcast MKM we also generate a broadcast key server map. This map tells us which key will be handled by which server in the reduce phase.
- Move mapper output to the Reducer instance
  Using the broadcast key map, we move mapper output from output/JOBNAME/mapper/KEY/valuesX.txt to input/JOBNAME/reducer/KEY/valuesX.txt where in the destination, X stands for the server from which the file was generated. A KEY may have multiple such valuesX.txt files so once all the files are moved, just before the reduce phase, we combine all these files into values.txt for each key.

Reducer

- Similar to the custom appendable output object stream, we have appendable input objects which help us read the valuesX.txt files without having to load them in memory.
- We can do so through iterables. It is this iterable that we send as an argument along with the key and the context to the reducer. This is done using reflections.

Running on EC2

- We maintain a publicDNSFile which contains all the server DNS's started by the start-cluster.sh script.
- We use the last server as the Master server on which the User Programs main() runs and calls the Master program. All the other servers are treated as slaves and the Master instructs the servers accordingly.
- Master ships the job to all the servers and instructs them to start the job.

Running locally

For Local we run on a single server where the Master and the slave Server run together. The publicDNSFile contains two entries for the Master and slave server.

Non-functional

- Not loading all or a lot of Data into Memory
  Since this framework will be used to run other map reduce programs, we want to keep as much memory free as possible for the map reduce jobs. We achieve this by not loading data directly into memory.
  Instead, we write collections to a file on the disk and read off of it.
- Moving large chunks of data between EC2 clusters
  In the shuffle phase, we want to move data between EC2 instances based on which EC2 instance will work as a reducer for which key. We move these files around using JCH which internally uses FTP.
- Using reflections to invoke methods from .jar files
  For every object we need to get an instance of the class and then invoke it. This is done with the help of reflections.

Challenges Faced

1. Network interactions between Data nodes (Server) and master node (Master). The Server is designed as a multi-threaded server and keeps accepting connections till its lifetime and keeps static objects whichever required for its particular phase. It talks on port 1210 if distributed EMR mode, and incremental ports when local as 1210, 1211, 1212, etc. The servers start and wait for the master to connect to it. First time when the Master connects to the servers, the servers talk to themselves and establish communication channels. The Master instructing them to talk with barriers is also another challenge which took a while to design and implement.

2. Writing and reading serialized data to and from disk. - Reading and writing complete single data objects was easy but maintaining collections of data objects needed some work. We managed to do this using the ObjectAppendable read and write classes.

3. Load balancing and equally distributing work to every data node. - Using round robin, we distribute files between mapper instances. In the shuffle phase, we move mapper outputs to reducers in the similar fashion using KEYs.

4. Starting mapper and reducer classes using reflections. We designed our Mapper/Reducer with protected access-modifiers. When used our program, we modified them to public since we weren't able to call protected methods of one package from the other package. We ended up making them public to avoid confusions w.r.t access specifiers.

5. Co-ordination among all the different components such as file operations. Maintaining integrity and consistency across all the components. - Earlier on, we had distributed different components of the project amongst all team members. We also agreed upon a bunch of API calls between these components. But still, integrating all the components required some effort.

6. Keeping the functionality as similar to Hadoop as possible. - Our aim was to make the system behave like a normal Hadoop system. In that, we avoided using shared memory on S3. We avoided loading a large amount of data in memory. We maintained effective communication between all EC2 instances.

7. Using SCP for a large number of files, when the file sizes are huge caused a few issues because there is a limit to the number of sessions a channel can keep open. - We handled the exception that occurs when SCP fails to send a file by waiting for a couple of seconds and trying to put the file on the channel again. We repeat this until the file is sent. Since JSCP is able to send this file at some point, it guarantees that we are not stuck in an infinite loop.

Allocation

Vedant Naik
Responsibilities:
- Creating initial stubs
- File Operations to local disk, remote EC2 and S3.
- Designing Data Structures and folder structures for Mapper output, Reducer input, Load Distribution and shuffle phase, as well as setting up code conventions followed throughout to help make integration easier.
- Shuffling logic, overall flow design.
- Integration and debugging.

Files contributed to:
- Context.java
- MapperHandler.java
- ReducerHandler.java
- FileSys.java
- FileReaderIterator.java
- LoadDistributer.java
- Constants.java

Dixit Patel
Responsibilities:
- Designing the server.
- Socket programming for achieving the communication between server and master.
- Invocation and Reflection using Java
- Designed testing framework
- Integration and debugging.

Files contributed to:
- Configuration.java
- ReducerHandler.java
- MapperHandler.java
- Server.java
- Master.java

Vaibhav Tyagi
Responsibilities:
- Designing the flow
- Designing the code conventions
- Designing Data Structures and folder structures for Mapper output, Reducer input, Load Distribution and shuffle phase, as well as setting up code conventions followed throughout to help make integration easier.
- Shuffling logic, overall flow
- Integration and debugging.

Files Contributed to:

- Configuration.java
- Job.java
- FileSys.java
- LoadDistributer.java
- IO

Rohan Joshi

Responsibilities:

- Setting up the clusters
- Starting cluster and stopping clusters
- Networking communication between EC2 nodes
- SCP and file transfer between nodes.
- Testing Framework

Files contributed to:

- start-cluster.sh
- Stop-cluster.sh
- my-mapreduce.sh
- FileSys.java

Priyanka Mane

Responsibilities:

- Reporting and documentation
- Testing
- Integration and debugging.

Files contributed to:

- IO
- Report.pdf
- Configuration.java
- LoadDistributor.java

Every member was initially allocated a module/part of this assignment. We worked on these parts (as seen from the Files section in the table above)
Later, everyone worked together to integrate all parts, and later debug and fix error.

Test Program Assessments

1. **WordCount**

   Did this test program require any new functionality to get to work?
   - This program required us to implement the basic Hadoop classes.
   - We needed to change the imports.
   - We needed to change access modifier for map and reduce methods from protected to public.
   - We compress the input files to .gz

   How did performance compare to Hadoop?
   The program is marginally slower on our implementation as compared to Hadoop.

   Did you produce the expected output?
   Yes, we produced the expected output.

   Execution Time
   ~ 3 minutes

2. **WordMedian**

   Did this test program require any new functionality to get to work?
   - We wrote our own version of WordMedian and tested it.
   - We compress the input files to .gz

   How did performance compare to Hadoop?
   The program is marginally slower on our implementation as compared to Hadoop.

   Did you produce the expected output?
   Yes, we produced the expected output.

   Execution Time
   ~ 3 minutes

3. **Assignment 2**

   Did this test program require any new functionality to get to work?
   - We needed to change the imports.
   - We needed to change access modifier for map and reduce methods from protected to public.

   How did performance compare to Hadoop?
   - Our implementation : ~ 30 minutes
   - Original EMR implementation : ~ 10 minutes

   Did you produce the expected output?
   Yes, we produced the expected output.

4. **Assignment 5**

   Did this test program require any new functionality to get to work?
   - We needed to change the imports.
   - We needed to change access modifier for map and reduce methods from protected to public.

   How did performance compare to Hadoop?
   - Our original (EMR) implementation was slow to begin with since our reduce calls have O(n^2) complexity.
   - Our current implementation does not speed the program up.
   - Implementation time: ~ 1 hour 45 mins on complete dataset
   - Original EMR implementation time : ~ 2 hours

   Did you produce the expected output?
   Yes, we produced the expected output.

5. **Assignment 7**

   Did this test program require any new functionality to get to work?
   - We had to change the way the models were being written and retrieved to and from the S3 bucket.
   - Instead of using the FileSystem class of Hadoop which essentially gives input and output streams to the SerializationHelper of Weka, we wrote two custom methods in FileSys which provide these streams.
   - We had to change the Routing code in Reduce phase to use these methods in place of Hadoop's FileSystem.

How did performance compare to Hadoop?
Our original (EMR) implementation was slow to begin with since our reduce calls have O(n^2) complexity. Our current implementation does not speed the program up.

Did you produce the expected output?
Yes, we produced the expected output.

Note: For Assignment 7, we used a test data set which was smaller than the given A7history data. The output for this test data was as expected.

Conclusion

We set out to implement a system that mimics Hadoop. We feel we have achieved the following:

- We use the distributed nature of the EC2 instances to run map, shuffle and reduce phases in parallel.
- We make use of the local disk space on these EC2 instances and no shared memory (S3 bucket).
- We do not load data directly into memory; freeing the heap space for the MR job submitted.

Overall, we feel this is a good implementation of a parallel data processing system which is not as powerful as Hadoop, but does manage to run our MR jobs.

References

https://aws.amazon.com/documentation/ec2/
http://epaul.github.io/jsch-documentation/javadoc/
http://www.programcreek.com/2013/09/java-reflection-tutorial/
https://examples.javacodegeeks.com/core-java/reflection/java-reflection-example/
https://github.com/is/jsch
http://stackoverflow.com/questions/15768516/why-does-an-sftp-connection-still-exist-after-the-jsch-channel-has-been-closed
http://unix.stackexchange.com/questions/136165/java-code-to-copy-files-from-one-linux-machine-to-another-linux-machine
http://unix.stackexchange.com/questions/136165/java-code-to-copy-files-from-one-linux-machine-to-another-linux-machine
http://stackoverflow.com/questions/2094637/how-can-i-append-to-an-existing-java-io-objectstream
http://stackoverflow.com/questions/2094637/how-can-i-append-to-an-existing-java-io-objectstream
http://stackoverflow.com/questions/2149785/get-size-of-folder-or-file