Cookiemonster MapReduce

By Karan Sikka & Samaan Ghani

Abstract

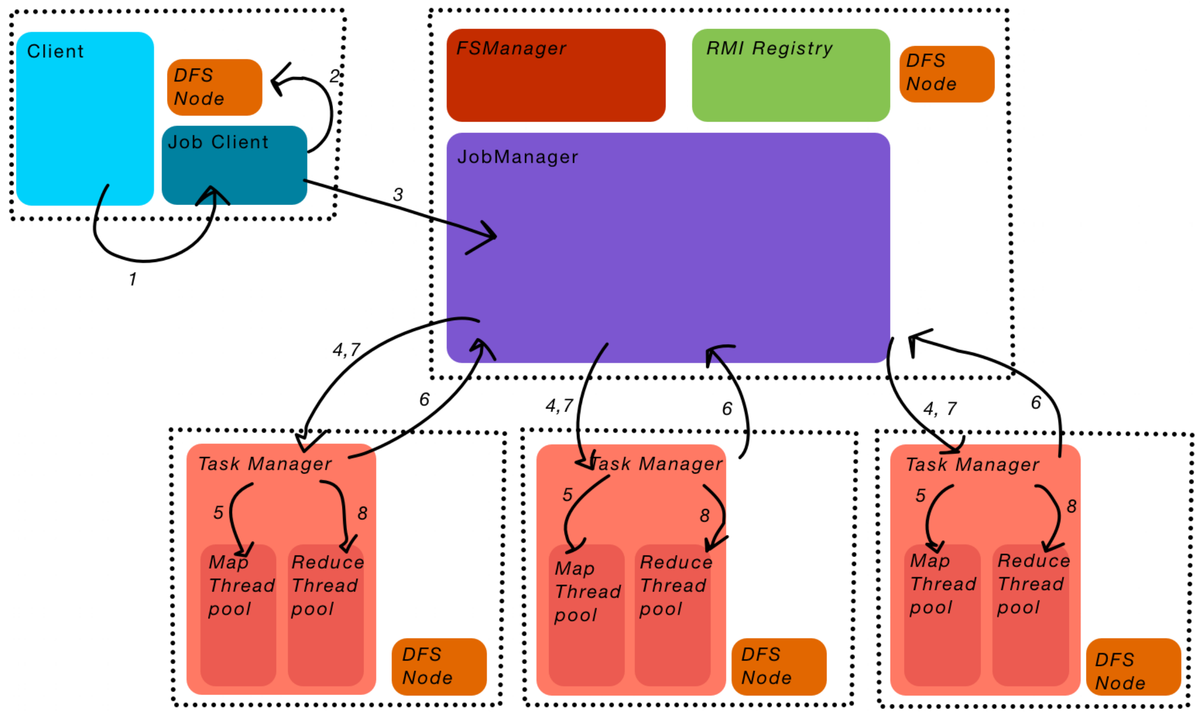
Imagine you have a lot of data - so much data that running a script to do simple filtering and counting would take too long on one machine. Or, the data may not even it on one machine. Cookiemonster MapReduce is a framework written in Java for distributing the data and computation across nodes. It helps you store massive amounts of data in a replicated and fail-safe fashion, and it makes it easy for you to write Java programs that process and combine the results of your computation in parallel, so that queries which previously took days now only take hours! Cookiemonster is a fully working system that was developed in an extremely short timespan (under a week), so naturally there are limitations. In this document, we seek to document our design, functionality, limitations and explain how we came to these design decisions.

Table of Contents

- Overall Architecture
- Distributed File System
    - Many FSNodes and One FSManager
    - File, Record, Replica Abstractions
    - “Fix” Replication
    - Local Record Cache
    - Reading/Writing Records
    - Reliability
    - Strengths and Weaknesses
- MapReduce
    - One JobManager and Many JobClients
    - Creation and Dispatching of MapTasks per Job
    - MapTask execution in MapWorkers on each JobClient
    - ReduceGroupers on each JobClient
    - Creation and Dispatching of ReduceTasks
    - Execution of ReduceTasks in ReduceWorkers on each JobClient (k-merge)
    - Storing and Retrieving Result
    - Strengths and Weaknesses
- API Guide
- System Administrator’s Guide

Overall Architecture

Diagram



0 - Start up the Cluster
1 - Create a Job Object and send to Job Client
2 - Write the files to the DFS
3 - Send the Job to the JobManager
4 - Create and assign MapTasks to all the Task Managers, start polling to see when the tasks are complete
5 - Task Managers receive MapTasks and assign them to the Map threadpool, after they complete, they merge the results based on the future reduce group
6- Polling realizes that all MapTasks are complete for a certain job
7 - Assign Reduce Tasks and wait for completeion
8 - Reduce and write output

⋯⋯ Sample configuration of machines

Distributed File System

For the MapReduce framework, we assume the presence of a distributed file system for reading input data, storing and sharing intermediate results between hosts, and storing output data. Note that it's suboptimal to use a replicated file system to store intermediate results, but since making one was required for the assignment, we decided to use it. If we had more time, we'd make a buffered shared memory region for storing temporary intermediate results and use sockets to stream the data to whoever wants to access it.

Bootstrapping the data

First, the user must get his or her data into the distributed file system. For sake of simplicity, when the user starts his or her MapReduce program, the user passes a path to the local directory containing the structured data files which will be used, and the program automatically invokes filesystem methods to bootstrap the data into the distributed file system. The required structured of the data is discussed in a later section.

The Structure of a MapReduce Program

A MapReduce program is a Java file with a main function, a Mapper class, and a Reducer class. The mapper class has a map function which takes a key,value pair, and returns an arraylist of key value pairs. The reducer class takes in a key and arraylist of it's values, and returns a key value pair. In our system, all keys and values must be strings. We did not have time to implement type-genericness.

How the MapReduce is implemented

Cookiemonster uses Java RMI for messaging between nodes in the cluster. When the user starts his or her MapReduce program, the user passes some data off to a "JobClient" which calls a remote method on the JobManager, which is responsible for starting, tracking, and knowing all other metadata about all the jobs in the system. The JobManager creates "MapTask" objects and delegates them to JobClients (via RMI). It then tracks their progress, (TODO) handling any failures, and when the MapTasks are all complete, it starts creating and dispatching "ReduceTasks" to JobClients. When ReduceTasks complete, the user is notified and can retrieve the results from the local filesystem (although they are also stored in the distributed file system). (Actually we didn't have time to put the results on the local fs, they are only in the distributed fs. But you can view distributed fs files on the node.)

In more detail, after all MapTasks on a single TaskManager complete, a postprocessing step is performed where the outputted <k,v> pairs are sorted and merged into an accumulating list of <k,v> pairs in their ReduceGroup. A <k,v> pair is deterministically assigned to a ReduceGroup as a surjective function based on the key k. That is, if <"dog", 1> is assigned to ReduceGroup 3, it is guaranteed that all <k,v> where k="dog" are also assigned to ReduceGroup 3. This system guarantees that when all MapTasks are finished, you have one list of <k, v> pairs per ReduceGroup, where the keys in one reduce group are mutually exclusive with those of any other reduce group. Therefore, all values for a key are contained in a single reduce group.

When ReduceTasks are scheduled on a ReduceWorker, which happens when all MapTasks complete for a job, they are fed in the values for some assigned key, which are obtained by reading and filtering the list of <k, v> pairs of the ReduceGroup for that key, which are stored in some file on the Distributed File System.

Distributed File System

Many FSNodes and One FSManager

In our system, one FSNode runs per machine, and one FSManager runs somewhere on the network to coordinate the FSNodes. We put the beef of the logic in the FSNode so that all FSNodes can do things in parallel without reading through the singular FSManager. The role of the FSNode is to act as a local filesystem and a client to the distributed file system. The role of the FSManager is to act as central metadata server for all the FSNodes and Files/Records/Replicas.
Any client of the Distributed File System connects directly to the FSNode of its host, rather than the FSManager. When reading, writing, or deleting, the FSNode may communicate with the FSManager to look up metadata, update the FSManager about the FSNode local state, or to tell the FSManager to fix the replication factor in case of the FSNode is out of space and needs to delete a replica. The FSManager sends heartbeats to the FSNodes, and if it detects a failure, it re-replicates and logs a message using the java.util.logging utility.

Our design principles were to maximize parallelism, minimize dependence on the FSManager, and tolerate/heal from FSNode failures. Our limitation is that if the Master goes down, the system is SOL. There is a lot more we could have done to optimize the file system's performance and reliability, but we were limited on time.

File, Record, Replica Abstractions

Our distributed file system is different from a general purpose one in that the abstractions are designed specifically for use by our MapReduce framework. We assumed that files were typically going to be very large and therefore we needed to partition them into Records. Rather than reading from files, we primarily needed to read from Records, as each MapTask ran on one Record.

We also were required to implement replication with a configurable replication factor. For this, we needed the concept of a "Replica", which is just a copy of a Record on an FSNode. In our system, there may not be two identical records on the same FSNode.

We do not support modifying existing Records or Replicas, but we do support deleting Replicas and telling the FSManager to re-replicate. This is to rebalance the replica distribution across machines in case one of the FSNodes runs out of space.

"Fix" Replication Strategy

We did not want to put the replication logic in the FSNode, because it seemed like it would be easier to do by an entity with knowledge of the entire system. So we put the logic in the FSManager. The way it works is that the manager has a method to check the replication factors of all the records, and if any record is under-replicated, it orders arbitrary FSNodes to create replicas. It does this until the user-defined replication factor is met. This strategy made it really easy for us to heal from FSNode failure, since if we detected failure, we could simply run the fixReplication method.

Local Record Cache

What happens if a client program requests a Record for which no replicas are stored on the local FSNode? The FSNode asks the FSManager which FSNode it is available on, and the FSNode then “downloads” a copy of the record to it’s local cache. The “downloading” is made possible because each FSNode is also running a multithreaded TCP Socket Server which can serve any file on the FSNode on request. The size of the local cache is configurable, and in the case that the FSNode is out of space, cached records which are not in use will be incrementally evicted from the cache. Notice that we intentionally made the system very lazy. The FSManager only reports the location of the FSNode with the record, and the FSNode downloads it directly from another FSNode. Also note this a potential security issue - but we didn’t care for the purposes of the assignment.

Reading/Writing Records

As previously mentioned, writing a File potentially produces multiple chunks. Also note that if you write and the system is nearly out of space, the FSNode will do something a bit funny. It will first write a Replica, tell the FSManager to replicate, then it will delete it’s local replica and tell the FSManager to replicate again, avoiding the current FSNode. This way, writing to the FSNode will write to the distributed system, but not store a local replica. This is clearly a suboptimal method of doing this, but it gets the job done. If we had more time, we’d remove the least frequently or least recently used replica from the FSNode instead.

Reliability

What happens if an FSManager goes down? Sorry not sorry.
But what if an FSNode goes down? There might be hope!
Since the FSMaster is presumably alive, the dead FSNode will be detected by a failed heartbeat. Then the FSMaster will update its internal state to reflect that the node and its replicas are no longer available. It will also fix the replication as described above.

Known Limitations

TODO
Currently no validation being done on file names. Ie, what happens if you use “..” or “/”? idk.

MapReduce

One JobManager and Many TaskManagers

The logic of the whole MapReduce facility is stored in a JobManager. It tracks the progress of jobs, as well as assigning map and reduce tasks to TaskManagers. We chose this model because there needs to be something monitoring the state of the whole system. The JobManager is able to assign tasks, monitor them, and even restart them in case a TaskManager fails. The JobManager can be run anywhere, but we recommend its own machine or own JVM so that it is not affected by any of the computation that is happening on other machines. We chose to have many TaskManagers, a TaskManager is bascially a local master for a worker node. A TaskManager receives map or reduce tasks and assigns them to threadpools and provides a way for the JobManager to poll them and ask for the state of the running tasks. A TaskManager is also helpful because after a round of MapTasks for a job is completed, it is able to start the Combine step on its own. This is helpful because each worker node should already combine its map output into a single file before the reduce step.

Creation and Dispatching of MapTasks per Job

A user creates a job object, supplying a map function, reduce function, local path to input data, number of reducers, and number of mappers. This gets passed to the JobClient who then bootstraps the files to the DFS and sends the job object to the JobManager. The bootstrapping step just uses the DFSNode (each computer has its own) to write the input files to the DFS. When a job is received by the JobManager, it assigns it a job id so that they have a unique identifier. Then it retrieves the relevant Record objects from the DFS that were previously added by the JobClient. A Record is basically like a file pointer to pieces of the original input files. This is helpful because we are not passing around the file itself and wasting system memory, instead we can use these objects. The Record also breaks down the file into pieces that are required for each map, so that the input file is broken into chunks. To read we can just call Record.read() which then actually reads in the file. Next, after getting all the Records we create MapTask objects which contain all the information necessary to complete a maptask. This includes the job which contains the map function, a record, mapTask id, status etc. Then we dispatch these mapTasks to the TaskManagers, giving an equal number of tasks to each TaskManager, and overflow tasks to the last one.

MapTask execution in MapWorkers on each TaskManager

Once the TaskManagers receive the MapTasks, they start each of them on threads, and keep track of the thread Futures (Java Object) in an array. MapWorkers read in the input and create a key value pair using the map function that is provided by the user. The they add the pair to a TreeMap. We selected a TreeMap because it forces the keys to be ordered. We wanted this implementation because it is very helpful for the later Combine and Reduce steps. After the

MapTask has completed, the MapWorker writes key values to the DFS from the TreeMap so that they are ordered from least to greatest.

Combine execution in MapWorkers on each TaskManager

After all the MapTasks pertaining to a particular job have completed on a TaskManager, the TaskManager signals for a combine. This combines all relevant files for a reduce group into one file. Each reduce task is assigned several keys, using the hashcode of the key modded with the number of reducers. This is the ReduceGroup. So that a single reducer only has to worry about one file for each TaskManager output, we merge them together before a reduce. Of course we can't assume that all the different map outputs will fit in memory at once, so we perform a k- way merge. This requires taking a value from each file and storing it in a TreeMap, again used for the ordering of keys. They we remove the least element and write it to a buffered writer, and replace that value by adding a pair to the tree from the same file from which the pair we just wrote came from. This ensures that at one time, there are only k pairs in memory and that the output file is sorted. We then write this to the DFS.

Creation and Dispatching of ReduceTasks

After all of the combining is done, the JobManager sends ReduceTasks to the TaskManagers. Each TaskManager receives one ReduceTask. A Reduce task consists of the job, which contains the reduce function, and a reduceGroup. The reduce group tells it basically which files in the DFS it cares about, as a result of the combine, it is the same number that was assigned in the Combine. For instance if we are doing a word count, then we could split the reduce groups on the first letter of each key. For instance group 0 gets A-G, group 1 get H-P, group 3 gets Q-Z.

Execution of ReduceTasks in ReduceWorkers on each TaskManager (k-merge)

Now a the TaskManager gives the Reduce task to a thread. It gets all the relevant Files on the DFS and again just like the first merge creates a TreeMap with one pair from each file. It then gets the smallest pair, and remembers the key. It starts accumulating the values that match that key. So after removing a pair and starting this accumulation, it gets a pair from the same file as the one removed and adds to the TreeMap (same as before). But what is interesting is that once a new key is seen, because all the separate files were sorted we are guaranteed that there are no more values associated with that key in this whole job. So we can call reduce with that key and accumulated values and write the final result to the DFS. Again we do this to minimize memory usage on a machine. But we must now continue with the new key and keep going until all of the files have been read. This happens in parallel with other reduceTasks so when all of the have completed there will be several result files, each of which contain key value pairs. But this is our result because each file has only unique pairs and so the final answer is just split among files.

Known Limitations of MapReduce

We make the assumption that a single MapTask can fit into memory. This is a pretty reasonable assumption because the MapTasks are pretty small. We also use a lot of in memory data structures which would cause all state to be lost if the facility went down. Ideally we would use a database of some sort or also distribute the data structures. The output of MapTasks are also stored in a ArrayList. We realize that for certain implementations of map this could be bad for memory usage and the real Hadoop uses a context to provide a stream based approach instead.

API Guide

To create a task that runs on our Map Reduce Facility, the user needs to provide a Map function that extends the Mapper abstract class that is provided with the facility:

```
public abstract class Mapper implements Serializable {

     abstract public ArrayList<Entry<String, String>> map(String key, String value);

}
```

So the user provides a input file of key values, if there is no key, for instance for WordCount, they can write the map function to ignore the keys. They must implement the map function that meets the above spec and the rest of the Mapper class will handle all the things needed to take that data and incorporate it into the framework. This was necessary because only the user knows what map function he or she wants to implement. We do the same for the reduce function:

```
public abstract class Reducer implements Serializable {

    abstract public String reduce(String key, ArrayList<String> value);
}
```

The user provides a reduce function for a key mapped to an ArrayList of values. The reduce basically receives collected values and then performs the reduce function. The user must also provide a main method where he or she creates the Job object and passes it to the JobClient. It should look something like this:

```
public class WordCountProgram{

    public static void main(String[] argv){
        File localdir = new File(argv[0]);
        String nodename = argv[1];
        String reghost = argv[2];
        int regport = Integer.parseInt(argv[3]);

        Mapper m = new WordCountMapper();
        Reducer r = new WordCountReducer();
        Job job = new Job(localdir, m, r);

        JobClient.submitJob(job, reghost, regport, nodename);
    }

}
```

The user includes the local directory from which the DFS should get files, the RMI registry host and port, as well as the node it is running from so that it can get the correct DFSNode from the registry. We also bind the Mapper and Reducer to the job.

System’s Administrator Guide

AKA How to setup and configure the cluster

0. Compile source files

```
mkdir bin
javac -cp src -d bin src/**/*.java
javac -cp src -d bin src/**/**/*.java
```

1. Set up the cluster

RMI Registry (write down port and hostname)

```
rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false <rmiport>
```

Write a config file

```
# map/reduce worker config

map_slots 3
reduce_slots 3
mapr_tmp_dir /tmp/cookiemapred/

# file system

dfs_working_dir /tmp/cookiedfs/
record_size 6000000
max_node_size 120000000
replication_factor 3
heartbeat_interval 5
```

Note that the temporary directories you specify must exist beforehand, and as a good practice, should be empty.

There are 5 types of programs you must run to set up the cluster.
1. rmiregistry (one)
2. FSManager (one)
3. FSNode (many)
4. JobManager (one)
5. TaskManager (many)

Note that each program in the list depends on the existence of the previous programs.

Here we go through an example sequence. Note that one FSNode should be named “Master”, as an odd quirk. This is because JobManager is hard coded to connect to that FSNode. In our tests, we ran the *Manager type programs on the node which we named Master.

We recommend you use screen or a similar tool to help you view and manage these long-running processes.

On a node which we choose to name “Master”:

```
java -cp bin cookiemonster.dfs.FSManagerImpl <path to config> <rmi host> <rmi port>
java -cp bin cookiemonster.dfs.FSNodeImpl Master <rmi host> <rmi port>
java -cp bin cookiemonster.mapreduce.JobManagerImpl <path to config> <rmi host> <rmi
port>
java -cp bin cookiemonster.mapreduce.TaskManagerImpl Master <rmi host> <rmi port>
```

On a node which we name “Charlie”

```
java -cp bin cookiemonster.dfs.FSNodeImpl Charlie <rmi host> <rmi port>
java -cp bin cookiemonster.mapreduce.TaskManagerImpl Charlie <rmi host> <rmi port>
```

You can repeat the procedure for setting up Charlie for as many nodes as you like, so long as you give unique names.

Now the cluster is set up!

2. How to run a mapreduce job

Given a java program written in accordance with the API Guide, running is simple.

```
java -cp bin examples.WordCount.WordCountProgram exampledata/ Master
127.0.0.1 6666
```

This will submit the job to the cluster (assuming RMI registry is still running.)
# TODO The output will show you the progress, polled every 5 seconds.

When done, get the results from the specified data output path.