Darrin Willis - dswillis
Bohan Li - bohanl

15-440 Writeup
Lab 3: MapReduce and Distributed Filesystem

https://github.com/darrinwillis/distributedSystems

Overview

Our implementation of MapReduce is structured around a few distinct systems interacting. Our Distributed File System (DFS) and MapReduce framework are controlled by a central master server, which dispatches the actual files, as well as the MapReduce jobs to a set of Nodes. Once the Nodes have collectively completed their tasks, the output files are gathered put back into Master's filesystem (In our case AFS) for easy access. RMI is utilized for almost all network interaction.

Configuration

All configuration is determined by a config file, which is automatically verified or generated upon system startup. It specifies all system properties including (but not limited to): Ports, Master address, Node addresses, partition size (in lines), and replication factor. This configuration file should not be changed while the system is running, as this could cause minor undesired behavior, or potentially permanent node disconnection.

System Startup

System startup is handled entirely by a bash script (./startSystem) and helper java program (Monitor.java) which bootstraps the system and immediately allows user interaction. It accomplishes this through dynamic Bash script generation, based on the config file. Each listed node causes a bash script to be generated based upon its listed address. The address is then used to ssh into it, change to the specified code directory, and start the node server. With key-based ssh (as can easily be achieved in AFS), this allows for a one-line system startup. Upon running this script, a terminal is run which allows the user to add files, add MapReduce tasks, and monitor various system statuses. The system can be ended by running another bash script (./stopSystem) in order to end all node and master servers cleanly. This stop script is automatically run at the beginning of the start script, disallowing duplicate systems to be run on the same set of nodes.

Distributed File System

The distributed file system is automatically cleaned at the beginning of each system startup. This entails all nodes and master cleaning out their local directory (which is by default located in /tmp/distributedFiles). The file system is then interfaced with entirely through the terminal, as previously mentioned. This terminal(./terminal) automatically starts upon system startup. The primary interface is to add a (text only) file to the DFS from the local file system (which may or may not be on a Node). Adding the file in terminal uploads it to the DFS master. The master server then partitions it by number of lines, and distributes it out to any online nodes. If the listed replication factor is higher than the number of online nodes, then all nodes will get a

copy of the partition. Importantly, this means that if 5 nodes are listed, with a replication factor of 4, but only 3 nodes are online, then only 3 nodes will get a copy of the file, and no other nodes will get a copy at any point. This was a design decision to minimize the wait associated with bringing a downed node back online (which could be long if a large number of files are added while it is offline), as well as one to reduce complexity. Each node maintains its partitions in a namespace within its specified local directory. The user can view a list of the Distributed Files at the terminal, though there is no feature for modifying or removing a distributed file.

Scheduler

The scheduling of these tasks is handled by a Scheduler thread located on MasterServer. All tasks to be performed are added to a task queue by the MasterServer. In addition we keep track of a NodeQueue. With every tick, the scheduler checks the first task in the task queue. If it is a mapTask, scheduler iterates through the NodeQueue checking for the first available node that has the FilePartition associated with the task and schedules the mapTask on it. If it is a reduceTask, it simply schedules it on the first available node in the NodeQueue. After a node in the NodeQueue receives a task, it is popped and pushed to the back of the line. Scheduler constantly schedules tasks until either the taskQueue or the NodeQueue is empty. This ensures that maps are always done locally, which is a great assurance of speed, especially considering our files are fairly optimally distributed, but comes with the drawback that if a subset of nodes happen to have very high demand, that it is possible to end up with unused cores on certain machines.

MapReduce

The actual mapping and reducing is handled by a SlaveNode object located on each NodeServer instantiation. Our scheduler sends task information via rmi to the NodeServer who then starts a thread for each task. This TaskThread then calls the doMap and doReduce function on its corresponding SlaveNode which handles the mapping and reducing. Once the operation is completed, the TaskThread dies. We have a TaskTracker running in the background on the NodeServer which detects this and tells the MasterServer via rmi that the task is finished. The MasterServer then checks whether or not all tasks of that type have been completed and acts accordingly. The details are listed below.

- **Mapping**
    - Each MapTask consists of a FilePartition that it parses for records to map. Then the job's map function is called giving us a list of key value pairs. The key is hashed to determine which reducer it is being sent to, and we write the key-value pair to the corresponding output file separated by a '~'. All these partial map output files are stored locally. Once a mapTask is finished, our NodeServer notifies the MasterServer making sure to include its address. Once all mapTasks scheduled for a job are finished, we add all of the reduceTasks to the task queue to be scheduled. A possible improvement would be to allow the application programmer to provide a comparator function to gain control over which reducers take which subset of the input. As it stands, using a hash function is a reasonable

solution.

- **Reducing**
  - Each ReduceTask takes in a list of Nodes that have partial output files from a MapTask of the same Job. This information is included in the ReduceTask by the MasterServer in addition to a nodeid which determines the input files for the ReduceTask. For each ReduceTask NodeServer iterates through the given NodeList, and downloads (using our FileIO) all partial map output files associated with its nodeid. Once all the files it needs are local, the ReduceTask is sent to SlaveNode which parses the inputs and does the reducing. Finally, the output is stored locally, and NodeServer notifies Master that the reduce is complete additionally giving it the location of the reduce output.
- **Final Concatenation**
  - Once all ReduceTasks are finished, MasterServer concatenates the output of each reduce into the output file specified by the user. It downloads (using our FileIO class) all of the partial reduce outputs from the corresponding node. This output file is then output onto the local machine of master, which in our case is always AFS. This allows for an easy way to read the output.

Robustness

Whenever communication is blocked between a node and master, master stops allocating any new tasks to it until it comes back online. Its status is monitored every 10 ms with a heartbeating thread in master. Files partitions are each allocated onto the node with the fewest partitions, providing a fairly robust allocation, which is O(n*p) where n is the number of nodes and p is the number of partitions. This will be a fairly low number in almost all circumstances, making this acceptable.

Compiling / Running

Simply run `make` to compile all necessary code. `make clean` to clean out all `.class` files.

To start the system, run ./startSystem

To stop the system, run ./stopSystem

To run the terminal, run ./terminal

To run a sample WordCount, once the system is up, in terminal type
start WordCount out.txt in.txt 3
To start a WordCount on in.txt, and output to out.txt, using 3 reducers

Sample MapReduce Jobs

WordCount - This is a simple MapReduce routine which counts all words in a Distributed File.

LongestLine - This returns the longest line in a Distributed File

SysAdmin's Guide

In order to start the system, all nodes must have similar local file systems, such that they can use the same local directory to store local files. All nodes must also have a common directory which contains the .class files for the system. In order to run the system, simply run the ./startSystem bash script from any machine on the system, after providing key-based authentication such that all nodes can be ssh-ed into. If SSH is not option for your configuration, then simply running a node server with "java NodeServer &" will correctly connect it to the system. Enabling key-based ssh is vastly preferable, however, as it will fully automate the system. ./stopSystem will end the master server and any reachable nodes. Any nodes which cannot initially reach master which shut down on their own. The system can be monitored from the terminal, which has functions: nodes, files, and monitor. 'nodes' and 'files' displays a minimum amount of information which gives an overview of the current system status. 'monitor' provides a comprehensive readout of each partition on each node and the currently running map or reduce task. This terminal also allows for 'add' of a file and 'start' of a MapReduce routine. All functions are documented internally in the terminal.

Programmer's Guide

When programming a mapreduce job, all you need to do is provide 3 functions. map(key,val) takes in document name and a record and outputs a list of key value pairs in String[key,value] format. getIdentity simply returns the reduce identity. reduce(key,vals) takes in a key and a list of values and returns a single string which is the desired reduction of the list of values. See WordCount for an example of use of this framework.

Possible Improvements

Caching

Our framework does not make use of any caching at any level. However, caching could easily be put in place by allowing Nodes to keep a certain amount of copies of files which are gathered from other nodes during the reduce phase.

Improved Read/Write

Our framework does not allow for the reading of the middle of a Distributed File, or modifying any file that has already been created. Appending to a file would be fairly easy to add,

as well as reading in the middle of a file. Arbitrary modification would be very difficult to add, and probably shouldn't exist.

Reduce as soon as Maps Finish

Currently, we wait for all maps to finish before starting to Reduce. To save time, we should have reducers which begin as soon as each of their maps is finished. This is non-trivial however, and more difficult to do robustly.

Sort the Final Output

Currently, out final output isn't sorted, it is just a combination of reduce outputs appended one after another. This is more similar to MapReduce as stated in Google's original paper, as our output is simply a collection of outputs which are individually sorted, but randomly concatenated to each other.

Recover from Failure

When a node disconnects from our system in the middle of doing work, we do not reschedule the work that was lost.

Bugs

- Sometimes, when the number of reducers is high (>10) our system does not output the correct output.
- When dealing with huge files, our system may hang while creating MapTasks