Swapnil Pimpale (spimpale)
Romit Kudtarkar (rkudtark)

15-440/640 Project 3 Report

Collaboration Statement: All work submitted by us is our own.

Introduction

This report contains information about the design and implementation of MapReduce framework that we have written for Project 3.

Overview of Design

Our MapReduce framework utilizes a single master node and several worker/data nodes, all of which communicate to one another using java's native RMI API. Our framework supports running multiple map and reduce tasks on a single node, and allows the user to customize his/her mapreduce job by modifying the appropriate parameters in the configuration file 'config.txt'. Our framework also supports automated job restarts in the event of node failure.

Workflow

1. Initial bootstrapping and client submission of job from master node(covered in-depth in the '*Instructions for compiling and running our framework'* section.

2. Master node receives job submission request, copies client's input file(s) over to the HDFS, splits the input file(s) into chunks, and sends chunks with replication to every other node. The master also sends the client jar file to all the nodes.

3. Nodes receive their chunks and the client jar file and write them to their own HDFS directory. Master node then begins scheduling map jobs on these nodes.

4. Master node uses a 'greedy' scheduling algorithm, where it attempts to schedule the mapping of a chunk on a local node. That is to say, for every chunk, the master node gets a list of all nodes which already have that chunk on disk.

   For every node in this list, the master attempts to schedule the maptask for that particular chunk on a particular node in that list which has the least map load(i.e. it attempts to schedule a maptask for the node in the list which is currently

running the fewest number of maptasks). This is done to maximize throughput by ensuring that work is distributed as evenly as possible.

If all nodes in the list are running at maximum map capacity, the master then attempts to schedule the mapping of that chunk on a foreign node(i.e. a node that does not contain that chunk on disk) that is not running at maximum map capacity. If such a node is found, the master obtains the chunk from one of it's local nodes, and sends it to the foreign node, before scheduling the mapping for that chunk on said former foreign node. Once again, this is done to maximize throughput and to ensure that chunks don't have to spend time 'waiting' for their local nodes to finish their current maptasks.

5. For every maptask received by a slave/worker node, it creates a new thread to perform the mapping for the chunk associated with that maptask. Each map thread constructs an instance of the main class in the client's jar file, and repeatedly calls the map function of the object associated with the main class for each of the lines in the chunk for that maptask, collecting intermediate key-value pairs after each call. After the entire chunk is processed, these intermediate key-value pairs are sorted by key, partitioned, and sent to the appropriate reducer nodes. The intermediate key-value pairs are also written to a series of intermediate files on disk. A list of unique reducers associated with that particular chunk and job is sent to the master, to be used for scheduling reduces later.

   Each reducer node that receives these intermediate key-value pairs writes them to a series of reducer files on disk, to be processed for reduction later.

6. Once the master notices that all the maptasks for a particular job are done, it schedules the reduces for that job. It uses the list of unique reducers it obtained from every maptask and schedules reduces on each of these reducer nodes. Each reducer node starts a single thread that reads the reducer files it has on disk for that job, and processes those files, before writing the final key-value pairs to a single output file on disk in the HDFS.

Features of the Framework

- Provides several MapReduce IO classes: *IntWritable, ShortWritable, LongWritable, DoubleWritable, Text, OutputCollector* which the client can use when writing his/her map and reduce functions. Also provides the supporting classes *AddInterface*, and *KeyValue*, which are used by the MapReduce IO Classes.
- Maximal throughput: The scheduling algorithm tries to have each chunk be worked on ‘locally’, minimizing the overhead associated with sending chunks from one node to another for mapping. Furthermore, if all ‘local’ nodes for a chunk are running at maximal mapping capacity, the scheduling algorithm transfers the chunk from a fully-loaded ‘local’ node to a non-fully-loaded ‘foreign’ node to complete the mapping of that chunk. While there is some overhead cost associated with this transfer, this cost is a preferable alternative to waiting for maptask running on a ‘local’ node for that chunk to be finished, so that the pending maptask can be scheduled ‘locally’.
- Maximal throughput: Intermediate key-value pairs are sent directly from each node after completion of a maptask to the appropriate reducers. This is a preferable alternative to the option of waiting for all maptasks to finish before sending the appropriate reducer files to each reducer, since it reduces any overhead cost associated with the latter option. This is especially beneficial in situations where the network bandwidth can become a bottleneck.
- Parallelized mapping: Multiple mapper nodes, each of which runs multiple maptasks, each of which works on a separate chunk of the input file(s).
- Parallelized reduces: Multiple reducer nodes, each of which runs a reducer task for a single job, each of which works on reducing a separate subset of the intermediate key-value pairs.
- A heartbeat thread that monitors the status of each node at a regular time interval of 1 ms
- Ability to retry maptask/reducetask in the event of maptask/reducetask failure, with a threshold of 1 for the total number of retries

Assumptions

- All input files are text files, and each line of a text file is a single record, which will be processed by a single call to a client-defined map function.
- All jobs will be submitted from the Master node. All client input files will be originally placed on the AFS, prior to bootstrapping.
- The Client jar will have exactly 1 main class, which has an empty constructor and exactly 1 ‘map’ method and 1 ‘reduce’ method.

- The ‘map’ method is of the form: map(<*Any MapReduce IO Class*> key, Text value, OutputCollector<*Any MapReduce IO Class*, *Any MapReduce IO Class*> output). Basically the ‘map’ method implemented by the client must take in 3 arguments and in this specific order:
    1) Some arbitrary key value, which the client’s map function must never use. The reason is that our mapreduce framework treats each line of an input file as a value in the key-value pair, so the actual key is not meaningful. You can see that both the wordcount and wordlength examples pass in *LongWritable* key as the key to the ‘map’ function, but neither example actually uses that key in the ‘map’ function. The first argument is merely a convention.
    2) The second argument must be of type *Text*. This is because the value for each call to the ‘map’ function is simply a single line of text read from the input file. The *Text* IO class is what our framework uses to wrap String objects, so this argument must be of type *Text*.
    3) The third argument must be of type *OutputCollector*. *OutputCollector* can be parameterized by any of the valid MapReduce IO classes defined by our framework, and it serves to collect the intermediate key-value pairs that result from the map operation.
- The ‘reduce’ method is of the form: reduce(<*Intermediate MapReduce IO Class*> key, Iterator<*Intermediate MapReduce IO Class*> values, OutputCollector<*Any MapReduce IO Class*, *Any MapReduce IO Class*> output). This is because the intermediate keys and intermediate values can be of any MapReduce IO type, since they are determined by the parameterized types of the OutputCollector object in the ‘map’ method As such, the first argument is the intermediate key, the second argument is an Iterator(parameterized by the type of the intermediate value) over a collection of intermediate values, and the third argument is an OutputCollector parameterized by any of the MapReduce IO Classes.

  Put simply, if I have a map function:

  map(LongWritable key, Text value, OutputCollector<*A*, *B*> output), where *A* and *B* are any valid MapReduce IO Classes.

  Then, my reduce function should have the form:

  reduce(*A* key, Iterator<*B*> values, OutputCollector<C, D> output), where C and D are valid MapReduceIO Classes, and A and B represent the same MapReduce IO classes in both the reduce and the map functions.

- The Master node never goes down/fails.

Failure Recovery

In the event of a maptask/reducetask failure, the thread running the task informs the node of it's failure. The node attempts to re-run the task again, and if it fails a second time, the node notifies the master node that a maptask has failed, and the master node proceeds to abort the job associated with that maptask, since the job cannot be completed.

In the event of node failure, the heartbeat thread informs the master node of the failure. The master node then proceeds to abort any jobs that have any maptasks or reducetasks associated with that node. While this may seem a harsh policy, there is good reason behind this choice. There are several cases to consider in the event of node failure, and they are not easy to handle. Let us consider the cases in the situation where we have a single job running.

Case 1 : Node fails during the map phase of a job

If a node fails during the mapping phase of a job it means the following:

- Any maptasks that have been sent to this node to be performed will need to be rescheduled and restarted on some other node. This is not particularly challenging to implement, and can be done.
- Any intermediate key-value pairs that have been sent to the failed node prior to it's failure will now need to be re-sent to a new node. Furthermore, the partition function must ensure that subsequent intermediate key-value pairs that originally mapped to the failed node are sent not to the failed node, but to the new node which receives the old intermediate key-value pairs sitting on the failed node. This is fairly complex, and implementing this would require each datanode to keep track of the current job state, individual task states, as well a mechanism to modify the partition function on the fly so that it appropriately map new incoming intermediate key-value pairs to the correct reducer.

Since our mapreduce framework aims to maximize throughput, it was designed in a manner whereby a mapper would immediately send it's intermediate key-value pairs to the appropriate reducers upon completion of a map task. If a node failed and it had intermediate key-value pairs sent to it for reduction, it would be very difficult to manage the issue of re-sending them to the appropriate new reducer node, since it would require keeping track of a very large amount of information, as well as adjusting the partitioning function on the fly.

Case 2 : Node fails during reduce phase of a job

If a node fails during the reduce phase of a job it means the following:

- Any ‘reduced’ output, i.e. any intermediate key-value pairs that have already been reduced will need to be reduced again. Furthermore, any intermediate key-value pairs that would have been reduced by this node will have to be transferred to another node for reduction. This means that any intermediate key-value pairs that originally mapped to this failed node will have to be re-sent from their ‘original’ nodes(which performed the mapping and generated those pairs) to the new reducer node. Once again, this would require each datanode to keep track of the job state, individual task states as well as a mechanism to modify the partition function on the fly so that it appropriately maps

Since we do not maintain intermediate states for jobs and tasks, it is very difficult for us to restart/reschedule a job midway, which is why we decided to reschedule the job completely in the event of a node failure.

Testcases

We provide 2 simple examples for our framework: WordCount and WordLength. WordCount simply counts the number of occurences of each unique word in a file/series of files. WordLength counts the word length frequencies(how many words of length 3 appeared, how many words of length 4 appeared etc.) in a file/series of files. We also provide 3 input directories containing text files which the examples can be run on.

Instructions for compiling and running our framework

1) Copy all the provided scripts and directories over to your afs home directory
2) Run the following command: chmod 700 buildscript.sh
3) Run the buildscript: ./buildscript.sh . This will create our mapreduce framework jar: framework.jar and two example jars: wordcount.jar and wordlength.jar.
4) Edit the configuration to add/remove nodes you plan to run mapreduce on, the location of the HDFS(Hadoop Distributed File System, default is ‘/tmp/sr0’), the port for the RMI registries to run on, the maximum number of maps that can be run on each node, the maximum number of reduces that can be run on each node, the number of records/lines per chunk, and the replication factor for the chunks.

5) Run the slave_node.sh scripts on the worker/slave nodes and the master_node.sh script on the master node. This will set up the RMI registries on these nodes and set up any node parameters for the nodes from the config.txt file. Please note that the slave nodes should be started before the master node.

6) You can now submit a job to the framework. The job should be submitted from the master node. Jobs can be submitted using commands of the following form:

java -jar framework.jar startjob <jobname> <Clientjarfile> <Inputdirectory> <OutputDirectory> <NumberOfReducers>

This tells the framework to start a job with a job name specified by <*jobname*>, which runs the map and reduce functions defined in the main class of the client jar file specified by <*Clientjarfile*>, using the directory specified by <*Inputdirectory*> as the path to the input files needed for the job, and using the name specified by <*OutputDirectory*> as the name for the output directory where the final output files will be stored for each reducer. Note that this final output directory is a subdirectory of a separate directory we create for the actual job submitted by the user.

After the job completes, the final output files(each reducer will have 1 output file) will be present in the directory:

<*HDFS_DIR*>/<*jobName*>-<*jobID*>/<*OutputDirectory*>/finaloutput , where <*HDFS_DIR*> is the distributed file system directory path specified in the config.txt file(default is /tmp/sr0), <*jobName*> is the name of the job submitted on the commandline, <*jobID*> is the unique job ID for the job, and <*OutputDirectory*> is the directory specified by the user when submitting the job.

Here's an example of how to submit a job:

java -jar framework.jar startjob firstjob wordcount.jar inputdir3 outputdir 3

This tells the framework to start a job with the name 'firstjob', using the map and reduce functions specified by the main class of 'wordcount.jar', using the files in 'inputdir3' as the input files, using the name 'outputdir' as the name of the outputdirectory, and using a total of 3 reducers.

Assuming that *HDFS_DIR* is set to it's default value of '/tmp/sr0', the resulting final output file(s) for that job will be present on the appropriate nodes in the directory: /tmp/sr0/firstjob-1/outputdir/finaloutput .

Say we run a second job then, with the same paramaters for the job submission as before but a job name of *secondjob*. The resulting final output file(s) for that job will be

present on the appropriate nodes in the directory:
/tmp/sr0/secondjob-2/outputdir/finaloutput .

Side Notes

- Chunk IDs are unique across jobs
- Intermediate and reducer files are not stored on the AFS, they are stored locally on each node in /tmp and get cleaned once the job is completed/aborted.