Tiny Google Project Report

Rav Singh, Ishrat Ahmed

In this project, we designed and implemented an application to index and search large documents. The implementation is done in two ways: 1) Java-Based Implementation, and 2) Hadoop-based implementation.

Java-Based Implementation

This is a Socket based implementation where we have a master server that handles all the connections and request and clients issue an index or a search request. The main components of this system are described below:

Master Server

The master server is implemented in a multi-threaded manner. The first thread is a MasterServerThread that waits for connection from either a client or a worker node. This thread then spawns a new thread to handle all of the communications between the Master and the connecting machine. The Main thread waits for the arrival of a request in the WorkQueue (this queue holds incoming index/search request). Then the following checks occur.

If the number of workers is zero, we respond back to the client saying that there are no workers available and to try again later.

If the request is an index request, the master server checks the document for its indexed status. If it has been indexed, then it responds back to the client stating that it is already indexed. Otherwise, the master server proceeds with the indexing request.

The server then checks for the current active request type and either proceeds further with the job creation or it waits for the jobs to be completed. This is done so that no two same requests are handled at the same time (e.g, search request is not handled as the time as the index request).

For both indexing requests and search requests, the master server then creates a Job Coordinator thread. To handle all of the Job-related communications.

Job Coordinator

The job coordinator creates jobs and sends them to the registered workers This class defines a set of mappers and reducers. These number depend on the size of the file and the number of available workers. For example, if the file is small, then only one worker would be selected to handle the mapping task. Furthermore, even if we have many workers, we limit ourselves to 4 reducers be relatively efficient on the reducing step. (And it was simpler to encode). The workers are randomly selected, so as to not purposefully overburden a particular worker.

Once the Job Coordinator class specifies the workers and reducers, it sends the job to each of them and then and waits for an acknowledgement of their receipt of the job. This acknowledgement is to avoid deadlock between two workers. More specifically, a deadlock can occur if 2 requests come in and a Job

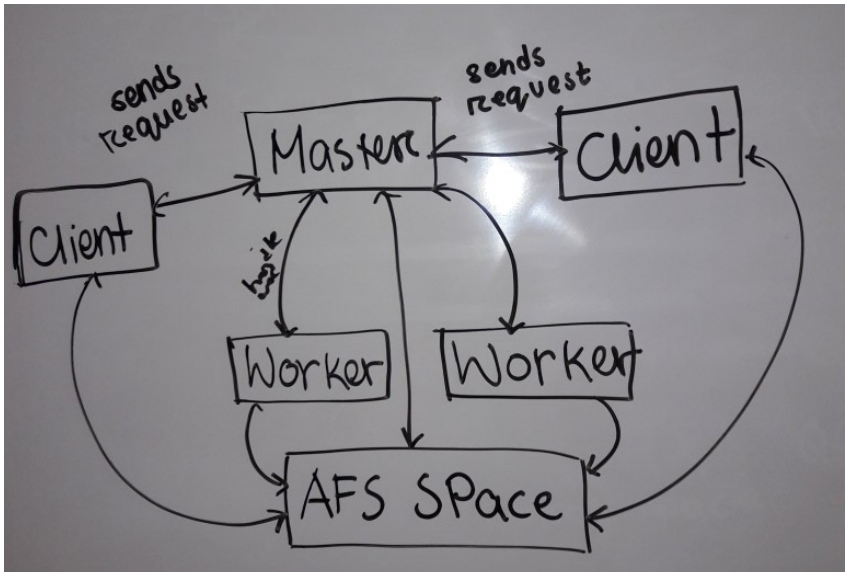
coordinator ends up selecting 2 workers, both doing a reduce task for one request and a map task for the other. If the reduce tasks arrive first to the workers, then they will be waiting for the map tasks to finish, which will never happen.

Also, to maintain consistency in the communication, the only objects we send over TCP/IP are Job, Request, JobAck (Job Acknowledgements), and RequestAcks.

Finally, one caveat to our socket-based implementation is we leveraged the afs space for our shuffling. To explain, instead of sending messages over the network directly to the worker that will do the reducer step, we save a file to the afs drive, (located in the */Jobs folder) This simplifies the shuffling process so the reducer can wait or all of the files necessary to be present before proceeding, instead of waiting on messages.

Workers:

These workers will communicate with the only with the master to receive job requests and then handle all of the shuffling through the afs space. It will be better described later on.



Network Structure:

The Image above highlights the networks structure of our implementation. Essentially showing that the clients and workers all talk to the master and there is no communication between clients and workers unless it is through the afs space or through the Master server.

Non-Networked Files:

We have a document indexer that creates document ids for us to make sure that we don't index the same document multiple times and for the Inverted Index to use.

We also have a Java file called IIinterface.java that handles all the whole Inverted Index structure. The way it is organized is by creating 26 folders for each letter of the alphabet and then inside of each folder

will be a text file. This text file is named in a particular format, which is "$term-$documentID-$count.txt" and the contents of this file is the canonical path of the file.

The FileHandler.java is there to help with reading in and counting the number of lines. WordCount represents the structure to store the results of the indexing and search processes as word-count pairs. Utility supplies functions and static constants for use.

Indexing:

When an indexing request comes in, the Job coordinator sets the number of mappers and reducers for the work. The number of mappers is based on criteria of each mapper getting around 100 lines to map each or the whole set of workers getting an equal share of the work. (If the number of lines per worker exceeds 100.) The reducers are going to be from 1-4 workers based on if there are 1 or more workers, ideally 4 if we have more that 4 workers.

Once the job has been received by the relevant workers, all of the workers will begin their map process. Here they first read in the lines of the document they are going to process. As they are reading it in, all punctuation, numbers, extra spaces, and Stop Words are removed. Stop Words are a list of the most common words (pulled from here: https://gist.github.com/sebleier/554280), typically used in NLP tasks, and we are removing them from the string to limit the number of entries in the Inverted Index Structure. Once they mapper is done counting the words, they will have a WordCount object representing all of the words and counts for each word. We then have the mappers save their work in a folder called "*/Job/$jobId". During this write, a lock file is created to make sure that reducer doesn't read in a file when it is not ready to.

Once all of the map tasks are done, the reduce tasks begin, or they were already running if the worker just had a reduce task. This will wait for all of the map tasks to get done and then the reducer reads in all of the WordCount obects that were created. Merges them together, using a supplied merge function in WordCount. Once the merging is complete, we save the entries into the Inverted Index structure using IIinterface.java's adds entry method.

Search:

The search process is mostly the same as the indexing. We create map tasks that instead look through the Inverted Index structure using interface's supplied methods of reading' the structure. We split the work based on characters and have set lists of what ranges of characters each map task will entail. The mappers just check to see if the term that the II entry has is contained in the search query (This also means that if 'a' is an entry in our II structure, then words like 'apple' will match). Once the mapping is done we create a file that just lists all of the entries and the one reducer will look at all of the files and rank and retrieve the results.

The ranking processing is mainly checking to see which documents have the highest occurrence of the search terms combined. And then returns that one document. This process doesn't necessarily let the user know of all of the documents that match, but just the one document that has the highest occurrence.

Hadoop-based implementation

For the hadoop-based implementation, we implemented an index mapper and index reducer to perform index operation. We also implemented a query mapper and query indexer to do a search. For both cases,

we extended the MapReduce class. The execution of a MapReduce job begins when a client submits a job configuration with specified map, and reduce functions along with the location for input and output data. We used StringTokenizer to token the input file and the reducer aggregated and displayed the result. For querying, the query element is passed through the argument. If a match is found, the term with filename and its number of occurrence is printed out.

In order to perform search, first index has to be created on the input file. The output of this index is given as an input path to the search query.

Metric Used:

We decided to use completion time of each task to be a metric to determine how well the index/query operations work. We measured the nanoseconds of client sending in the request to the master and then record the nanoseconds when a response is received. The method used is Java's System.nanoTime(). The difference indicated the total time to complete the task, including network communication for the Java-based implementation.

Java-based Usage:

To use this package, we need multiple terminal windows open. One in the oxygen element server and two more in any of the elements servers. In all of them 'cd' into the jars folder and make sure that an 'ls' lists the following:

Master.jar

Client.jar

Worker.jar

Stopwords.txt

If you see an 'InvertedIndex' folder or a 'documentList.txt' delete them as it will not allow to index the same file multiple times unless they are not present.

In the oxygen terminal, we will run the Master. Run:

java -jar Master.jar

And the master should be up and running, creating the necessary files and directories.

To run the worker, go to one of the other terminals and run

java -jar Worker.jar & ...&

We can repeat the 'java -jar' command for the number of workers that we want, so if we want 3 workers then we can do

java -jar Worker.jar & java -jar Worker.jar & java -jar Worker.jar

The workers can come in and out at any time, but the work will not be completed if a worker leaves in the middle of a Job.

To run the client, do the following in another terminal:

java -jar Client.jar {index||search} {file to index using a relative path | quoted search term}

We rely on the path to the file for indexing to be relative so a sample index is:

java -jar Client.jar index ../src/TrashFiles/tinyClient.java

And it should pull the correct file. As for searching the following is a sample search:

java -jar Client.jar search "Some complex search query that wont work"

The client will wait 5 seconds before executing the request to let the user make sure and cancel if it is incorrect.

To close each of them, do "CTRL-C". Check to make sure that all of the components are properly closed.

When closing the Master, the workers are automatically closed, but the client may not close by itself.

Hadoop Instructions:

1. Login to Hadoop Server: had6110.cs.pitt.edu

2. To start, the input files need to be copied into HDFS. To do this,
    a. Make a directory:

        hadoop fs –mkdir [project foldername/input foldername]

        hadoop fs –mkdir [wordcount/inputdemo]

    b. Copy the input files in this directory from the Hadoop server (cd into the directory where input files are stored)
    hadoop fs –put [src location] [dest location]
    hadoop fs –put inputdemofiles/* wordcount/inputdemo

3. To compile the Java program:
    a. Compile first (cd into the location where 'src' (containing Java files) is kept

        javac -verbose -classpath /opt/hadoop/share/hadoop/common/hadoop-common-2.7.2.jar:/opt/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.7.2.jar -d class src/*.java

        it will create class files in the 'class' folder
    b. Create jar file:

        jar -cvf [jar_name].jar -C class/ .

        jar -cvf wordcount.jar -C class/ .

    c. Run Index of the document

        hadoop jar [jar_name].jar [classToRun] [arg 0 : input path] [arg 1 : output path]

        hadoop jar wordcount.jar hadoopIndexer [wordcount/inputdemo] [wordcount/indexoutput]

this creates indexing of all the documents present in inputdemo folder (after step 2).

To check the indexing, we do:

hadoop fs –cat wordcount/indexoutput/part-00000

d. Run Search: the output path on the precious step is the input path for this step, so

hadoop jar [jar_name].jar [classToRun] [arg 0 : input path] [arg 1 : output path] [arg 2: query]

hadoop jar wordcount.jar hadoopQuery [wordcount/indexoutput] [wordcount/searchoutput] [Rosalind]

To check the searching results, we do:

hadoop fs –cat wordcount/searchoutput/part-00000

For both steps c & d the output folder name has to be changed since it throws an exception if the folder is already present.

4. To check for any folder in HDFS, do
hadoop fs –ls [project_name]
hadoop fs –ls wordcount

This will all the variation of input/output files if present

To check for the contents of the index/search result
hadoop fs –cat [project_name]/[file_name]/part-00000
hadoop fs –cat wordcount/searchoutput/part-00000

Test of Java-Based Implementation:

To get the data shown, we indexed the three files, twelfthnight, tamingoftheshrew, midsummersnightsdream and the times for indexing with a 1-3 workers is shown in the excel file. The time was measured using Java.nanoTime() from the client side when they were sending the requests and when they receive a response from the master concerning that result of the request.

The indexing time seems to follow an exponential decay pattern as the number of workers increases. Also, it seems that our implementation will be as fast as the hadoop version when we get up to around 6 workers.

We also ran one search of "freedom night" on the three indexed documents and this request took around the same amount of time for 1 through 3 workers to complete the search.

Test of Hadoop-Based Implementation:

We tested our implementation in Hadoop server (had6110.cs.pitt.edu).

It seems that hadoop keeps a consistent time among every request it receives. This may be the case because it speculatively executes more workers to potentially get the job done faster. Therefore, hadoop

can attempt to meet a soft real-time deadline of 15-30 seconds. This is also supported by the fact that hadoop takes around the 16seconds to execute a search when our system takes around 1 second.

Conclusion:

Through this process we can say that hadoop provides a response in a consistent amount of time of between 15-30 seconds, and is easier to implement in. This means that if the requirements of a user fits those specifications, then hadoop would be a good tool to use. Unfortunately, if this is too slow for some reason, then manually implementing a solution could provide better results, despite the difficulty in implementing such a system.