Design Document of Tiny Google

Mehrnoosh Raoufi and Alireza Samadianzakaria

1- Introduction

Tiny google is a distributed search engine that can index multiple documents at the same time and have a distributed index. This project has two parts: the first part is the implementation of a distributed search engine using socket programing and the second part is implementation using Hadoop. In this document we explain the design of the project as well as the setting of the experiment to show the performance. Both parts are implemented in a single java project however, the codes related to each section is in different packages.
In section 2 of this document we discuss the design of the first part of the project as well as the client-server architecture. In section 3 you may find the design of the Hadoop implementation and in section 4 we discuss the experiment to measure the performance of both systems as well as the improvements that we have made for the hadoop section.

2- Socket Based Tiny Google

In this part we use a master-slave architecture to distribute the indexing task and searching task among different machines. Each client connects to the master node and sends its requests, then the master asks the helpers (slaves) to perform the tasks and returns the result to the client. In the design section we discuss the design of the master and helpers, in the second subsection we discuss the API that masters gives to the clients.

Design

If the command is indexing a file or directory, it breaks the file(s) to sections and assigns each section to one helper, then the helper is responsible to scan and tokenize that document. After that the helpers pass the frequencies related to each term to the helper that is responsible to that term. For instance one helper is responsible to keep the inverted index for the term “x” and it receives all the (doc,frequency) pairs related to this term from other helpers. We use hash code to determine which helper is responsible for which term. This ensures load balancing among the helpers.
So each time that the master receives index request for some files the followings will happen:

1- master assigns one file to each helper
2- helpers count the terms in each file
3- helpers sends each other the terms they have counted (each term will be sent to the related helper node
4- helpers merge the lists getting from each other and adds it to their index.
5- helpers sends the message success to the master.

6- master returns success to the client.

Each time that the master receives a search query, it breaks it into terms, sends each term to the responsible helper for that term, sends

The socket based part of the project uses three packages of the project:

1- SearchEngine
2- Master
3- Helper

The SearchEngine consists of classes related to logic of our searching. There are two main classes in the search engine: TokenScanner and Indexer. TokenScanner gets a string or a file and returns its tokens. It will separate words using spaces, parentheses, brackets, commas, and etc. The indexer is responsible to get a document and use TokenScanner to scan it and the make an inverted index of the document. It can also store itself in a file in which each line is:

Term doc1 frequency1 doc2 frequency2 …..

It can also gets a query or an array of terms and find all the documents that have all of these terms. It will give each doc a score based on the following:

$$Score(q,d) = \sum_i f_{q,t_i} f_{d,t_i}$$

In which d is the document and q is the query.
The index is in the memory by default; however, the client can ask the master to save it to the disk or load an index from the disk.

The Master node has two classes with main method. a) ClientSample class, which is an interface for the client to connect to the master and in order to run it we need the IP address and the port of master. b) Master class, which needs a config file (consisting all the information related to the helpers and the port that the master is going to listen to.

The Helper package has one class with main method, the argument needed to run a helper is only the port that it is listening on for the master to get connected. Then it gets the other informations from the master (ie. the port it needs to listen for other helpers, the id of the helper, other helpers' IP:Port)

Client-Server Architecture

The master node is also the server for the clients. Each client connects to master and sends commands to the master. The master runs the commands one by one. Therefore parallelization is not in the mean of doing multiple commands simultaneously; however each command such as indexing will be parallelized using the helpers.
The followings are the commands and the possible outputs sent by the master to the client:

- indexdir <directory address>: this command asks the master to index a directory, the master will skip the documents in the directory that are already indexed. It will return a success message after completion
- indexdoc <doc address>: it asks the master to index a single doc.
- Search <query>: it can be used to search a query
- Reset: it will reset the index
- Save <directory>: it will save the index in a directory address (it can be an afs address)
- Load <directory>: it will load the index from a directory.

2- Hadoop Based Tiny Google

There are two different sections in the hadoop based part:

1- Indexing

In the indexing part the mappers read the documents and sends the key,value pairs to the reducers. The keys will be the terms, and the values are the doc and frequency (1). This value will be passed using a custom Writable class called TermFrequency that has a Text and IntWritable inside it.
Then the reducer will make the inverted index. We have also used combiner to combine the results of a mapper as a method for optimizing the performance. Without combiner the mapper will sends a message for each time that it sees a term. For instance if it sees term x in doc y, 5 times, then it sends (x,y,1) 5 times. However if we use the combiner the mapper will send (x,y,5) instead. Which makes the communication less and the sorting shuffling step faster.
To run the indexer the user needs to specify the input folder, output folder and a boolean that shows if we are going to use combiner.
The index will be saved in files produced by the reducers and it is in the form:

Term doc:freq doc:freq doc:freq…..

2- Searching

The searching part is a separate hadoop job. It needs the address of index folder, output folder and the searching query.
The searching query will be passed to mappers as a hadoop conf parameter. Each mapper then reads the index files and for each line that represents the term that is in the query it send the following key value to the reduce phase:

(doc, (term, freq))

Where doc is the key and (term, freq) is the value stored in TermFrequency data structure. The frequency will be the frequency of the term in the doc multiplied by the frequency of the term in the query.
Then in the reduce step, for each doc(key) we see if all the terms of the query are appeared. If yes we write doc,score as a line in the result. Otherwise, we write doc,-score. Therefore, the

user can also see those docs that have some parts of the query. (they will be those results with negative score)

3- Experiment

Our metric for the experiment is response time for indexing because the heaviest task in this search engine is the indexing part.
For the first part first we run using one helper and one master in a single machine as the base case. Then we increase the number of helpers in a single machine and compare them to see the effect of parallel processing in a single machine. In the second experiment we experiment using different number of machines and see the effect of increasing the number of helper machines. To see if the changes in time are really different for the slowest setting and the fastest setting; we experiment using these two settings several times and run paired t-test to see if the difference is statistically significant.

For the hadoop part we are comparing standalone map reduce versus map-reduce with combiner. Since the hadoop cluster may return different runtimes for the same task (due to its shared capacity) we repeat this several times and then run two sample t-test to see if the difference in the response time is statistically significant.