TINY GOOGLE: A CLIENT-SERVER BASED PARALLEL SEARCH ENGINE

Sanchayan Sarkar, Supriya Hulsure

INTRODUCTION:

Tiny Google is a high data intensive search engine that enables people to search for words as form of queries. With the rise of the internet, distributed or cloud-based computing supporting data intensive applications is at the forefront. Tiny Google is an attempt to enable parallel based data processing on distributed servers in the elements network. It is a client-server basd framework that indexes text documents, maintains sorted index tables and allows searching word queries over a distributed set of servers. In this project we do two things: a) Create a client-server setup using TCP sockets for document indexing and searching in the elements network and b) Implement the functionalities of a) in the Hadoop cluster using Apache Hadoop.

SYSTEM DESIGN:

Client Server-Based Model using TCP Sockets

The system is based on a basic client server architecture communicating with TCP/IP [1] protocol. The system has three working modules: 1) Client (user) module 2) Master-server module and 3) Worker server module. It is as shown in Figure 1. The Clients (marked orange) are the users who is using the client application for searching and indexing. The Master Server (marked white) is the main module interacts with the clients and redirects their requests to each of the worker servers (marked green). All the operations take place in the worker servers. In our system design, any server in the elements network can be initiated as the master server and multiple servers (except the master server) can act as worker servers. Worker servers are allocated dynamically based on their availability.

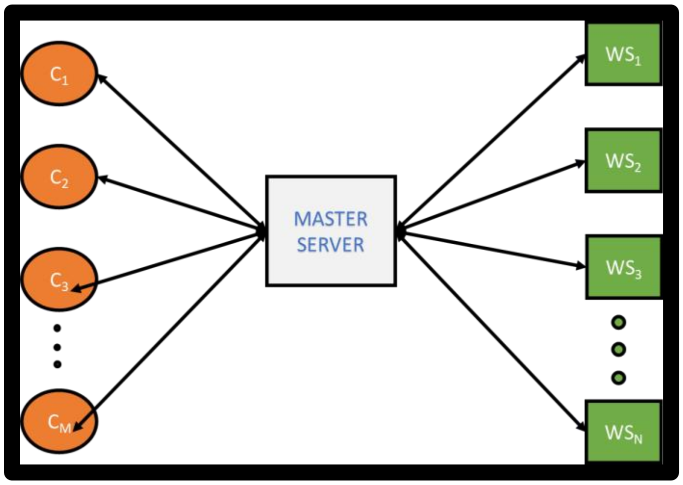


Figure 1 Client-Server Architecture

As far as the clients are concerned, any elements machine in the network can act as Client. The client only communicates with the master server which in turn communicates with the worker servers. All the data intensive operations are done on the worker servers. The following are the functionalities of each of the working components:

1. Client: Interfaces with the user asking for the type of operation they want to perform. It has two types of requests: INDEX and SEARCH. Upon user's choice it directs such requests to the Master Server.
2. Master-server: This is the main node which acts as a server to the clients and as client to the worker servers. It can communicate with both modules simultaneously. It takes in requests from the client and initiates connections with the workers through threads, for parallel processing. The operation request that the master server sends to the workers are: MAP, REDUCE and SEARCH. MAP and REDUCE are two operations request for the indexing operation while SEARCH is for the searching operation. Upon getting the results from the workers, it re-directs them back to the clients.
3. Worker Server: These are the nodes that actually perform the operations based on the request given from master-server. It has three functionalities:1) MAP, which creates a temporary index table for each of the file segments, 2) REDUCE, which creates the final inverted index table in a lexicographical order and 3) SEARCH, which searches for a particular word in the final inverted index table and returns a tuple of document indexes ranked by the frequency of the word occurrence in the document.

The functionalities of Client nodes and Worker Node can be better understood from Figure2.

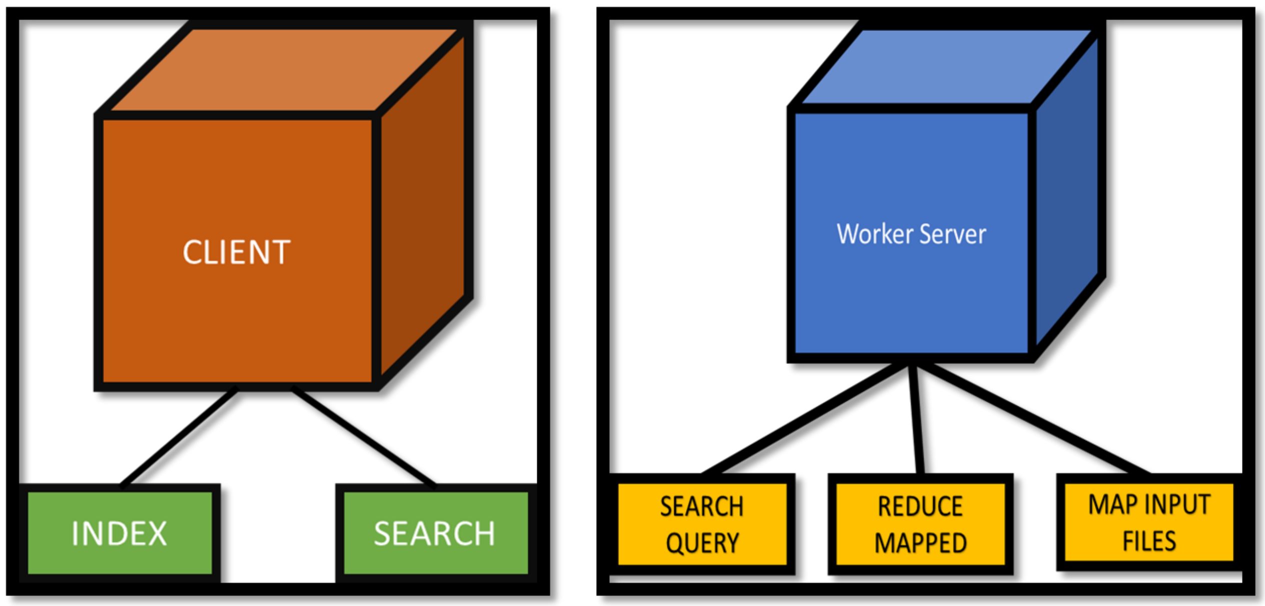


Figure 2 Client Node and the Worker Node

<u>Hadoop Based Model</u>

In this part of the project, we have implemented a simple search engine using MapReduce framework on Hadoop. MapReduce is a very efficient programming model [4] for generating and processing big data sets with parallel, distributed computing on clusters. Hadoop is an open source implementation of MapReduce model of programming. Hadoop uses the Hadoop Distributed File

System at its backbone [5]. The HDFS architecture uses a NameNode and a DataNode concept whereby the NameNode acts as a sort of master node having all the files and directories in a namespace and individual files are split and their blocks are replicated in the DataNodes. The data processing happens at the Data Nodes. Additionally the data nodes also hold a lot of metadata for the data splits.

METHODOLOGY

<u>Implementation Details of socket-based Client-Server System</u>

All the programs are implemented using Python 2.7 in the element machines of the network. For keeping track of the servers and their port numbers, a hash map of name of the servers along with a predefined port number was created. Another assumption that we have taken is that the masterserver component must be executed first followed by the worker server components. This is to make sure that the masterserver can listen to any new servers that want to join the system as a worker node. The application has three parts:

- Client.py:- It is the client user interface of the application and asks the user for choosing either an request for Indexing or Searching. Upon such a request, it connects to the masterserver.py and presents the output.
- Masterserver.py:- This is the masterserver which sets up with a predefined initial set of servers. It's main function is to read the client request and take the following actions:
    - If request is **INDEX** then it runs two algorithms:

      MAP

      Input: $[File_1, File_2, \dots , File_F]$
        - For each File, obtain starting and ending line number pairs <Start$_i$,End$_i$ > pair for each worker server. In this process, we are indicating that each file will split will be processed by each worker server.
        - Start threads, $t_i \; \forall i = n , n \in Worker\ Servers$ with a $< Start_i , End_i >$ pair , where each thread connects to the worker server W_i for mapping operation. In the worker server, it will create a temporary file containing the index table for that particular segments $Segment_i \; \forall i \in [1, Number\ of\ Files * Number\ of\ Worker\ Servers]$. If there are existing segments already present due to some previous index request, it will just add these segments and not overwrite them.
        - Wait for the threads and return.

      REDUCE

      Input: $[Segment_1 Segment_2, \dots Segment_S]$
        - First obtain the number of splits $< start_{alphabet_i}, end_{alphabet_i} > \; \forall i \in [1, Number\ of\ Servers]$. Essentially we want each server to produce the final inverted index table for a certain range of alphabets. So our final inverted index tables will be in lexicographical order.

        - For each split, $< start_{alphabet_i}, end_{alphabet_i} >$ start threads simultaneously to connect to the worker server W_i which will read in all the input segments (outputs of the mapper function) and produce the final inverted index table for that particular split. All the final inverted index splits will be written to disk.
        - Wait for all the threads to finish and return. By this time, we have the final inverted index table ready but as a combination of several splits based on the number of worker servers.
    - If the request in masterserver is **SEARCH,** then run the following algorithm:
      SEARCH
      Input: $[Split_1 Split_2, \ldots, Split_M]$,$[Word_1, Word_2, \ldots, Word_W]$
        - BusyList=None
        - For each Word, $Word_i$ we first check in which split it belongs to. Let this be S
        - If Worker server, W_i is not in BusyList, then start thread to connect to this worker server for searching this particular word in the assigned split, S. Add this thread to BusyList. Add the result to a shared queue
        - If the thread finishes, remove that thread from the BusyList and repeat the previous two steps until all words are searched for.
        - Wait for the threads to finish and return the queue to the client. This queue has the search results

An interesting aspect of this approach is the Indexing and the Searching are completely different separate from each other. The parallelization done in the indexing approaching for both mappers and reducers are dependent on the number of worker servers while the parallelization done for searching is based on the number of words. Hence, even if we have search words which are part of the same split, we can still search for them parallelly.

- Server.py – This is the worker server script that runs in the worker server nodes. It has two components: a) MAP: On getting this request it will read the file from the given starting and ending line numbers and create a map of words and their frequencies for that segment. It will write the segment to a shared temporary directory on disk and b) REDUCE: This will read all the segments from the temporary directory “temp” and merge the segments according to the lexicographical split. The output is an inverted index table for that split. It will write the table on a shared directory “index” on the disk.

Rather than sending the data of file splits from the master server to the worker server, we made use of the AFS system which is a distributed file shared system over this country. Since all the workers and the master are aware of the shared location of the “temp” and the “index” directory, we are directly accessing them. We took this approach because a) It will be less time consuming as we do not have to send large packets over the network and b) It will be safer as it does not have to take into account any failure that could have otherwise occurred while transmitting data.

For **communication protocol** we have used TCP as it has an established connection- oriented protocol where the delivery of the stream of bytes in ensured without any loss. However, for sending and receiving data, we developed functions that structures a stream of byte into series of packets to ensure an ordered delivery. For more reference see [2] and [3].

Implementation using Map-Reduce in Apache Hadoop

We have implemented the same functionalities of the Client-server model in using Hadoop. They are as follows:

- **Indexing of Documents**: Directory path of documents is given as an input to the program to implement master-index data structure which consists of all the different terms(words) from input files with a posting list. The posting list for every term has a node for each document which contains that term in the form of $< D_i, F_i >$. D_i is document id and Fi is frequency of that term in Document D_i. This Master-index is used when user wants to search keywords in these documents and gets output as list of documents ranked by the frequency of that keyword in these documents.

  Any MapReduce program works in two phases – a) Map and b) Reduce. In mapping phase of this program, map function parses each document and emits a sequence of (word, document id) pairs. The corresponding reduce function accepts all key value pairs emitted by map function for a given word and produces (word, list((document id, frequency)) pairs. The output of reduce operation is a set of inverted indices (Master-index) for given input documents.

- **Searching Keywords** – One or more keywords are given as an input to this module and this module will find these keywords in Master-index table produced form Indexing module and return a ranked list of documents which contains these keywords. Map function reads the master-index file and searches for keywords. It emits the inverted index of those keywords to ranking module which sorts the documents in inverted index posting list of that keyword according to the frequency of occurrences of that term in that corresponding document. The reducer function emits (keyword, list(document ids)) pairs. This is the output of searching process in this Hadoop implementation.

Ranking and Retrieval Mechanism

For the ranking and retrieval during the composite search process we have used the frequency as the primary indicator for sorting the index results. If the $< term_{i,} frequency_i > and < term_i frequency_j >$ after the searching process, we simply add the frequency $frequency_i \ and \ frequency_j$ and that particular term to the resultant list. Otherwise, we simply put all the unique terms to the resultant list, in a decreasing order of their frequencies. We used the same ranking mechanism for the Hadoop based model as well.

EXPERIMENTS & RESULTS

Experimental Setup

The master-server have been taken as *neptunium.cs.pitt.edu* elements machine.

For the input variable worker servers, we started with 2 servers (nodes) namely *hydrogen.cs.pitt.edu* and *germanium.cs.pitt.edu* as out first test case. Following that, we added *rhenium.cs.pitt.edu, selenium.cs.pitt.edu* and *antimony.cs.pitt.edu* incrementally to evaluate our test cases.

The client was located at *oxygen.cs.pitt.edu* (but in all purpose, it can be located at any server).

In scenario 1, for the input variable search keywords we started with 1 keyword "lover". Following that we included "is", "lovely", "person" and "and" incrementally to evaluate our test cases.

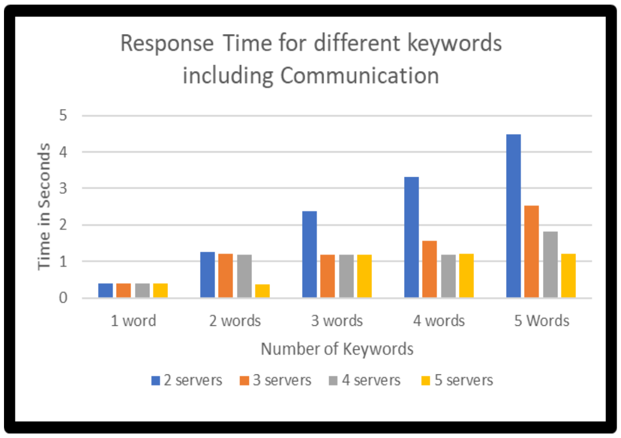
In scenario 2, for the input variable search keyword we have started with 1 keyword "lover" and then we have repeated it incrementally.

For the output(dependent) variables, we have recorded the communication response time and server response time. Communication response time is the search time taken by the entire process from client request to client output while the server response time is the time taken only for the parallelization process of searching.

For the Hadoop implementation, we repeated the same experiment and recorded the time from sending the function call to the return of the function call.

Results

The results obtained can be seen from figures Figure3 a) and Figure 3b) and figures Figure 4 a) and 4b).



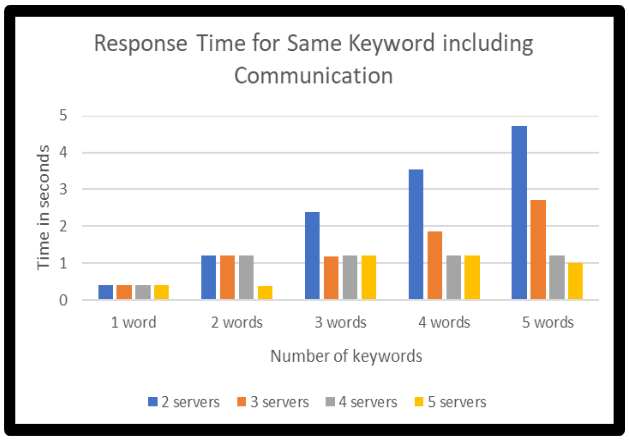
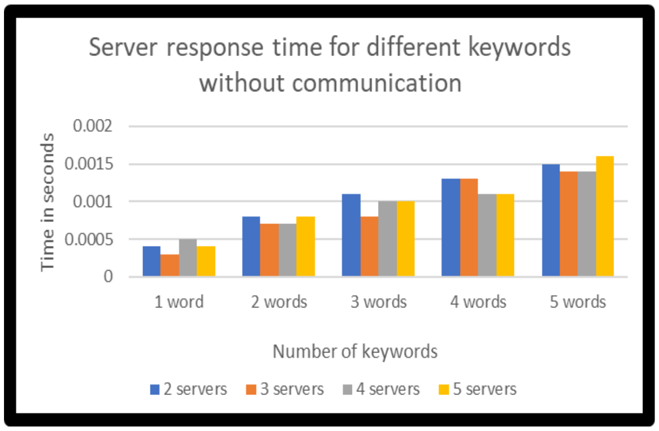


Figure 3 a) Response + Communication Time for different keywords b) Response + Communication Time for the same keyword

As we can see from Figures 3 a) and Figure 3 b), the response time steadily increases with the increase in the number of words for the same server (node) configuration. However, for the same word configuration, the response time for different keywords decreases with an increase of the number of worker servers (worker nodes).

However, from Figures 4 a) and 4) we can see that the actual server response time in general increases with the number of words but for the same word configuration, the server response time doesn't show a gradual decrease with the increase in number of servers which is an unexpected behavior.



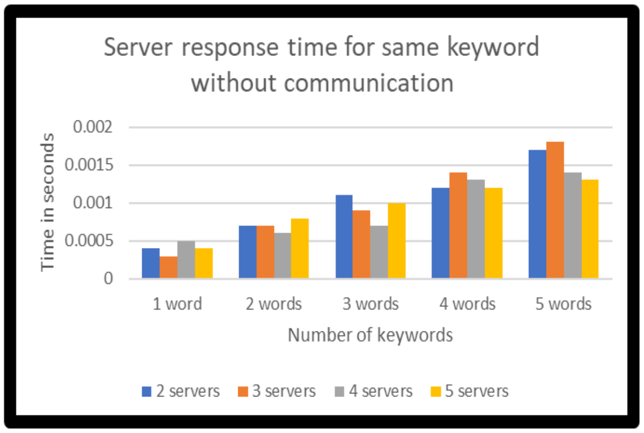
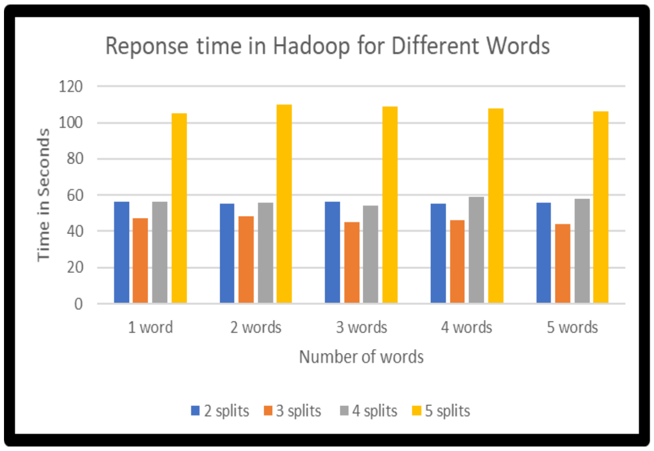


Figure 4 a) Server response time for different keywords and b) Server response time for same keywords. These do not include the client-server communication time

In our Hadoop implementation, we observe from Figures 5a) and 5b) that the response time roughly remains the same with the increase in the number of splits as well as the number of words. However, for the number of splits=5, we observe an sharp rise for all the words. It's timings are much higher compared to our client-server based model and therefore it wouldn't be a proper metric to compare the two mainly because there is not much control over the timings of the Hadoop application while there is a great deal of flexibility in recording the response times in our client server application. However, it can be seen that other than the 5th split, the response times remain relatively the same for all the words



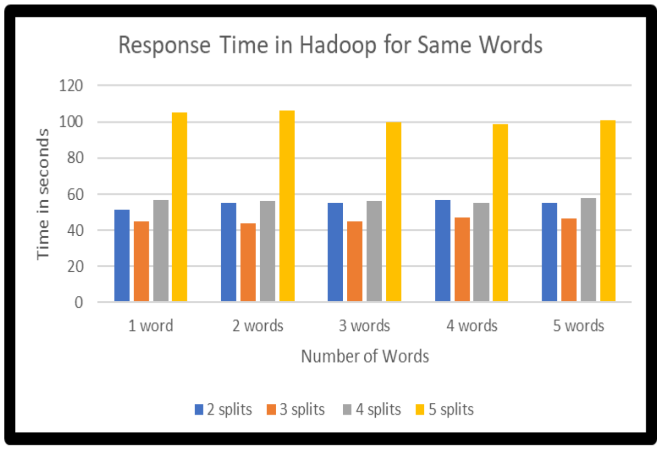


Figure 5 a) Response Time for different keywords and b) Response Time for Same Keywords

DISCUSSION

As we have seen from the observations, the communication time increases with the number of words for a particular server configuration. This can be easily explained because the master server can only search as much words parallelly as the number of servers configured. If the number of words exceeds the number of servers, inevitably the time taken will also be more. Also, the fact that the response time decreases with the increase in servers is also explainable as it is obvious that more the number of serves, more words can be searched parallelly thereby decreasing the response time. Also, another interesting observation is that if the number of words is same as the number of servers or less than it, then for that word figuration, there is not much change even if we increase the number of servers.

However, the actual response time of the server for just the parallel process doesn't exhibit the same pattern which is unexpected. However, this might have two probable reasons: 1) There is an I/O operation in the search process which reads in the split file information. This can potentially dominate the actual time of searching as they are not of the same magnitude. Hence, the results. 2) The input file size is not that big in general to require parallelization. Therefore, even if there are expected change in search processing it might not be recorded discriminatively.

When it comes to the Hadoop implementation, we can see from Figure 5a) and 5b) that there is an increase in the time for all the number of words. Also, there does not seem to be any general increase or decrease with the change in splits or the change in words. One reason can be that the time it takes for Hadoop to create the splits is greater in order of magnitude than the time it takes for Hadoop to actually search for the words parallelly. Another reason probable reason might be that the size of the input files are not big enough to observe a statistically significant response. Also, it will be clearly inappropriate to compare the two architectures from the response times as the degree of control in observing the time in our Client-Server architecture is far greater than the one in Hadoop.

CONCLUSION

In this project, we have implemented two things: a data intensive search engine application using a client-server model framework based on AFS file system and a search engine using Apache Hadoop using the HDFS file system. In the first part, we created multiple worker servers and a master server that parallelizes indexing and searching requests amongst the worker servers. We have found out that by increasing the number of worker nodes, we save time as it distributes the work efficiently. A similar behavior can be observed in Hadoop as well. Indexing with much bigger size data can reveal better behavior trends. In future work, we would like to implement a multiple client interface for searching and indexing. Currently we have a multi-client interface; but we haven't used synchronization mechanisms to order the client requests. In future, we would use synchronization to ensure reliability of the system. Another aspect that we are thinking of extending is giving results based on not just the searched keyword but also on closely related keyword. We are thinking of using Natural Language Processing techniques to do that in future.

ACKNOWLEDGEMENT

We would like to thank Dr. Daniel Mosse and Dr. Taieb Znati for giving us an opportunity to work on this project.

REFERENCES

1. V. Cerf and R. Kahn, "A Protocol for Packet Network Intercommunication," in *IEEE Transactions on Communications*, vol. 22, no. 5, pp. 637-648, May 1974.
2. https://www.geeksforgeeks.org/socket-programming-python/
3. http://stupidpythonideas.blogspot.com/2013/05/sockets-are-byte-streams-not-message.html
4. Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.
5. https://www.aosabook.org/en/hdfs.html