ABSTRACT
Today, high performance hardware and modern distributed system developing tools gives one the probability to build a large distributed system in a much simpler way than several years ago. This work presents an implementation of high scalable distributed search engine being able to work on both cloud virtual machine and bare metal. The system is built with technologies including MapReduce\cite{3}, etcd, kubernetes as well as dockers. With this system, the whole Wikipedia article data \cite{1} (∼50M pages, roughly the number of pages Google had in 1999\cite{2}) is indexed within a reasonable running time and the system also shows acceptable query performance. Though this work is a simple prototype, it can be easily transformed to a high reliable and available system.

Categories and Subject Descriptors
H.4 \[Information Systems Applications\]: Miscellaneous

Keywords
Search Engine, Mapreduce, Distribute System

1. INTRODUCTION
With the development of the distributed systems as well as the parallel computation, search engine is growing more and more efficiency and high available. As MapReduce model matures, utilizing MapReduce to run process on distributed systems in parallel is increasingly used in real-world applications. In many cases, the map phase as well as the reduce phase reduce the time consumption compared to the case without using MapReduce.

Building a search engine usually requires the high scalability and consistency between the commodity machines that used as masters and workers. Cloud storage, which is similar with the Google File System, provided by Google online can be a proper storage platform for all data to be used in the project since all machines can visit this storage platform to get the resources without creating several same copies of the data on different machines. This project use the cloud storage when deploying on Google cloud while use the /data directory of the machine when running on the CIMS servers (We do not use GFS-like file system when running on CIMS servers because there is no HDFS\cite{8} or other similar file systems available on them). This help save the storage and improve the consistency as well as stability of the system. Recently, etcd, an open sources stable implementation of raft consensus algorithm, is produced and it can help solve the consistency problems in designing the whole search engine system by provide the hosts and masters a good intermediate platform to communicate and fetch information.

Also, the management of assign roles to machine can be redundant since assigning the machine to be host or master have a large amount of status to check before doing the assignment. Kubernetes is a large-scale cluster management. It is actually an open source version of Google’s Borg. This project add the kubernetes to help solve the machine management issues. It not only can assign the roles correctly without asking the programmer to check the availabilities, but also can help unify the requests sent from hosts toward master when there are several masters at the same time occupying different port. The usage of kubernetes does help simplified the programming process of the whole project.

In this paper we implement the search engine by making good use of the tools and architecture mentioned above to generate a high scalable product. The search engine is mainly divided into two part, the indexer and the queryer. The components and the implementation of indexer will be introduced in section 3. Also, the details of queryer will be clarified in section 4.

2. RELATED WORKS
With the size of data getting larger and larger, the processing of big data sets was challenged. To solve this problem, Google team came up with a kind of a programming model and an associated implementation for processing and generating large data sets, that is, MapReduce. This model implement computation mainly by two functions: Map and Reduce. Map function take as input a key/value pair and then generate a set of intermediate key/value pairs to be used in reduce function. In reduce function, it incorporates the intermediate pairs associated with the same intermediate key. This model is parallelly run on a large cluster of machines. The users can manage the whole system easily without concerned about the details about partitioning the data sets, assigning machines to the programs while running, dealing with machine failures, and setting the communication between inter-machine. This truly bring about the convenience for the programmers without any experience with parallel and distributed systems to manage the processes on the large distributed system.

This paper mainly makes good use of the MapReduce model to generate the Inverted Index as well as the TF-IDF (term
frequency-inverse document frequency) and add scores in it for further usage in search part. To get the inverted index, each document is parsed by the map function and generates the intermediate pairs: \(\text{word, document ID}\) pairs. The reduce phase takes all pairs for a given word, sorts the corresponding document IDs and generates a \(\text{word, list (document ID)}\) pair. Then, all output pairs are collected and forms the Inverted Index that is needed.[3]

Also, the MapReduce model can be and has already been used in many fields like Distributed Sort, Term-Vector per Host, Reverse Web-Link Graph, Count of URL Access Frequency, Distributed Grep and so on. A large amount of problems are easily expressible as MapReduce computations.

Google's cloud storage is a scalable distributed file system for large distributed data-intensive applications. It offers fault tolerance, delivers high aggregate performance when running on the hardware. The cloud storage has been driven by observations of the Google application workloads and technological environment, both current and anticipated, while sharing many of the same goals as previous distributed file systems. The file system can meet the customers' storage needs very well.[5]

In this paper, we need Google's cloud storage (the data directory when using CIMS) as an unified file view for all machines, so that all machines can share a uniform data file from the file systems which can avoid many inconsistency while changing the data.

The engineers in Google raised the concept of the Borg system[10], which is a cluster manager that runs a number of jobs from different applications across clusters each with hundreds of machines. This system admits, schedules, starts, restarts and monitors all the applications that Google runs. Also it achieves high utilization by combining admission control, efficient task-packing, over-commitment and machine sharing with process-level performance isolation. Borg supports high-availability applications with runtime features that minimize fault-recovery time, and scheduling policies that reduce the probability of correlated failures. Kubernetes is an offspring of Borg system. In contrast to Borg, Kubernetes is accessed exclusively through a domain-specific REST API that applies higher-level versioning, validation, semantics and policy, in support of a more diverse array of clients. This paper we utilizes the Kubernetes to solve the load balance in the search engine.

Consensus algorithm allow machines to work as a coherent group that can survive the failures of some of the machines. So it plays a important role in building reliable large-scale software systems. The existing implementation, Paxos, is difficult to understand and requires complex changes to support practical systems. Diego Ongaro and John Ousterhout in Stanford University designed a new implementation of consensus algorithm called Raft[7]. Raft provides a better foundation for building practical systems. It separates the key elements of consensus and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Also Raft uses overlapping majorities to guarantee safety when changing the cluster membership. Etcld is one of the implementation of Raft and this search engine takes advantage of etcld to improve the fault-tolerance among the machines and the strong consistency of the data.

3. INDEXER
In this section, we describe a Hadoop-like (mainly Hadoop v1) job management system and how to use this system to generate the related data required by the searching module.

First of all, here is a list of data we need to generated:

- Inverted Index
- IDF (inverted document frequency)
- Document text
- TF-IDF (with title bonus)

Considering the huge data we need to process, it’s apparently not enough to use only one machine. Fortunately, we realized that the whole process is highly scalable: the whole dataset can be divided into several independent groups, and only IDF need to acquire global information (because it need to know the whole dataset size). Besides IDF, other data can be generated parallely, which means we can use any number of machines to compute them.

3.1 Job Management System
To fully take the advantage of high scalability, we implement a job management system to coordinate such computation process. The design of our homebrew system is kind like Hadoop. We defined 2 kinds of hosts, job tracker and worker, which includes a task tracker and several task handlers. Job tracker receives job definition (in YAML format) sent from user. A job consists of many tasks that can run parallely and the job tracker distribute the tasks to task tracker. Task tracker calls different kind of task handler to execute those tasks and track their running state.

In general, we can have any number of workers, but that comes a problem that how to configure and track states. To make it easier to scale out, we make the active task tracker send heartbeat signal to job tracker in order to let it know its existence and running state, just as what hadoop does. Initially, there is no workers available. Then, new worker starts up and introduce itself to job tracker and the latter would distribute tasks to it when there is unfinished task available. When a task tracker failed, the job tracker would notice heartbeat timeout, then it would mark the affected tasks failed.

We run the whole Wikipedia job with the system running on NYU CIMS's servers and use circus[4] to monitor the processes like Figure 1 shows. The system can also run on Kubernetes cluster with provided configure files and Dockerfile, shown as Figure 2. Kubernetes provides many useful features for this system. For example, the job tracker now runs as a service with its own domain name, which enables task tracker find it without setting job tracker’s address explicitly. Also, workers can be automatically scaled out with Kubernetes’ auto-scale feature.
3.2 Search Data Generation

With the job management system, we can now generate the search data efficiently with the following 3 kind of tasks.

**Reformat Task:** In this task, the raw Wikipedia data is divided into N groups. A group denote a independent part of data that can further plug into a single searching host described in section 4. For each group, its raw data is cleaned, which including removing control symbol and keywords from Mediawiki source file, and divided into M shards again. A shard is a part of data processed by a single CPU core.

**Mapreduce Task:** This task is basically a local mapreduce computation model executor. The number of workers equals to the CPU cores.

**Integration Task:** This task will convert the partial results from several mapreduce tasks to the data format that can be used in the searching module.

With these task, we can generate the search data in an efficient and distributed way, like Figure 3 shows: First, we feed the whole English Wikipedia article data into the reformat task and it produces N groups formatted data that can be fed into the following mapreduce task. In each group, the formatted data is split into M shards, where M equals to the number of CPU core. This design is to utilize the multi-core CPU’s power in modern machine. In this work, the whole reformat task only runs on a single machine, because of the connection limitation of the wikipedia archive server. However, if there is no such limitation, it can be easily configured to get feeds from different archive servers and run this task on multiple machines (though in practical, search engine’s data comes dynamically from crawler and there would be no need to do such process).

Then, the formatted data would be feed into 3 mapreduce tasks in order (the basic algorithm is based on the assignment 4’s description):

1. **Inverted Index Task (#1)** In the map phase, the whole article is tokenized and for each token (or term) it would emit a <docId, term> pair. In reduce phase, a partial inverted index table of each part is generated. Our algorithm is a bit different from the one in [3]. The advantage is, in this way, we can easily get a partial document frequency for each reduce part, which can be used to compute the global IDF in the following task.

2. **Inverse Document Frequency Task (#2 & #3)** In the map phase, the input is the output of all of the inverted index tasks. It only use the partial document frequency in each input file and emit them. In the reduce phase, we only use one reducer and it generates a global IDF. When mentioning all of the inverted inverted index tasks, we are saying the whole results come from all of the groups, which extremely increases the I/O cost. So, we decide to use a cascade way to get the global IDF by first getting the IDF in each group and then use another mapreduce task to generate the global IDF, which largely improved the performance (the result shown in the following section).

3. **Document Task (#4)** Its map phase is basically an identity function, the data from reformat task is emitted directly into the reduce task with key DocId. In the reduce task, the IDF vector got from IDF task is used to compute the TF-IDF vector of each document and the document text is split into sentences. To further improve performance, we compute the documents’
TF-IDF in batch, which do matrix-vector multiplication rather than vector-vector. The empirical result is shown in the evaluation section.

Finally, the results of the previous mapreduce tasks are converted into the format compatible with the searching module.

### 3.3 Evaluation

#### Table 1: Preliminary Result on 1% Dataset

<table>
<thead>
<tr>
<th>Method</th>
<th>Running time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-process</td>
<td>1130</td>
</tr>
<tr>
<td>Multi-process System</td>
<td>287</td>
</tr>
<tr>
<td>Batch Doc. TF-IDF Generation</td>
<td>281</td>
</tr>
</tbody>
</table>

We first test our system with 1% of the whole dataset (157MB compressed) running on our single local machine (Intel Core i7 quad-core, 8 threads, 16GB memory and SSD). It shows nearly 4 times faster than our previously assignment 2’s single process indexer, which we consider is quite reasonable. It also shows the batch TF-IDF computation gives 6s improve on the whole process.

#### Table 2: Running Time (sec.) on the Whole Dataset

<table>
<thead>
<tr>
<th>Phase</th>
<th>4 srvs</th>
<th>8 srvs*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inverted Index</td>
<td>2418</td>
<td>1209</td>
</tr>
<tr>
<td>Generate IDF (cascade)</td>
<td>159</td>
<td>80</td>
</tr>
<tr>
<td>Generate IDF (cascade, accumulate)</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>Generate TF-IDF</td>
<td>1339</td>
<td>670</td>
</tr>
<tr>
<td>Integrate</td>
<td>264</td>
<td>132</td>
</tr>
<tr>
<td>Total</td>
<td>4204</td>
<td>2114</td>
</tr>
<tr>
<td>Generate IDF (original way)</td>
<td>622</td>
<td>622</td>
</tr>
</tbody>
</table>

Next, we generate the whole dataset (12.9GB compressed, 51.4GB uncompressed) on the NYU CIMS’s server. We deploy 4 workers on server from Snappy2 to Snappy5, each has 20-core Intel Xeon CPU and we only use 8 cores of each machine and the job tracker is running on server linserv1. It shows that processing the whole data on this architecture uses 1 hour and 20 minutes with our best setting. Also, the cascade IDF computation method improved the performance at a large scale. What’s more, it can be easily concluded that if we have 8 machines rather than 4, the whole running time can be shorten into about roughly 40 minutes.

### 4. QUERYER

All the Wikipedia data is seperated into several parts and stored on the different machines, so a main queryer is responsible to collect the results in the different datasets and a series of hosts are necessary to operate the sub-queryer, doc server and index server.

#### 4.1 Main Queryer

To realize the low latency of this search engine, we store the indexes of the query items in the cache to avoid sending duplicated requests to the remote hosts. If the customers search a word, the main queryer firstly finds the item in the cache. If it’s not in the index cache, then the main queryer will send requests to the index servers of the list of hosts and combine the index results in the order of the score. Meanwhile the index results with docID and the score of the item in this document should be stored in the cache for future search. Once the queryer gets the indexes of the item, it will fetch the results of the snippets with the corresponding docID. For low time costs, the system will only fetch the snippets of the documents which will be presented in the current page (each page shows ten results and the default page is 0).

In practical system, the main queryer doesn’t know the number and the addresses of the remote hosts, so tracking the hosts is necessary for the queryer. The hosts send the heartbeat with their addresses and dataset ids to the main queryer, then the queryer will add the host into the list if the host doesn’t exists, otherwise update the time stamp of the heartbeat of the host. When some host goes offline or loses the internet connection, the main queryer will notice that the host is time out and remove it out of the host list.

#### 4.2 Host

Each host has a sub-queryer, N doc servers and N index servers. It’s a totally practicable search engine with all the necessary functions and features.

**Sub-Queryer:** There are two handlers, MainHandler and RequestHandler, in the sub-queryer. MainHandler processes the search from the front end and fetches the complete results from RequestHandler to show the results at the front end. RequestHandler send the words to the doc servers and index servers, then get the doc results and index results respectively. Sort the index results from different servers by score and combine the sorted index results with the doc results. Finally it returns the combined results.

**Doc server:** Doc Server can be a significant component of each host since it provides each search with the snippet of each document. In general, the Doc Server mainly receives the query request which includes the document ID and the query word. With these parameters, the Doc Server
combines the detailed document data (title, URL and snippet) fetched from document store and send these information back to the host's querier. The related terms in this content should be emphasized and the snippet should be a relevant chunk of text.

However, the content of snippet should not be the randomly selected from the documents. To get the most relevant content for snippet, our system uses a similar approach like [9]. It first calculate another set of TF-IDF for each sentence in the document. This set of sentence's TF-IDF is stored in cache for later use, which help save a large amount of time. When the querier request is received, the snippeter will calculate each sentence's score with its TF-IDF as well as the term's TF-IDF. The first sentence of each document will add an extra bonus since it is always more important for the whole document. After that, the sentence with the highest score will be picked up and the sentences around it will also be chose and combined to form the final snippet and returned.

**Index server:** Index server of the search engine provides the index score results which is decided by the relevance between the query words and the content of each document. So getting the proper score is a fundamental function of the search engine. Firstly, we need to find the dataset where the TF-IDF and index stores. With the existing TF-IDF file and index file, we can get the TF-IDF vector of query words. Then, according to the query item, we process the accurate search to get the unbiased scores of the query item in each document. Also we will process the blur search, in the contrast the scores in blur search will have a lower weight. After that, the results in accurate search will be integrated with the result of blur results. The integrated results will be sorted to generate the final index results, which will be send to sub-querier if requested.

### 4.3 Automatic Configuration

To avoid the inconsistency of the configuration between hosts as well as master, this project make use of etcd, a distributed and consistent key-value store for shared configuration, to realize automatic configuration of different machines. Etdc is a kind of application of raft consensus algorithm. In this application, only servers need to be provided and the etcd will elect the leader, that is, the master, itself and other servers will be the hosts. The master and hosts only communicate with etcd to load or fetch information from the table etcd provided and the existence of the servers will be checked out by the etcd according to heartbeat sending.

### 4.4 Evaluation

We deployed our project on CIMS's *linserv1* (Main Querier) and *snappy2-5* (Host) to test the querier on 4 groups out of total 8 due to the computation power limit. We find out that the efficiency of query can be acceptable and cache does shorten the time consumption when search the same term for the second time. The outcome of querier is as follow:

The efficiency of cache can be obviously seen from the Table 3. Almost all time consumption of first query can be limited within 0.5 seconds which is acceptable for us. In all, in the querier part, we didn’t have enough time to do the algorithm optimization but we did spare our efforts to accelerate the query speed and enhance the scalability of the querier.

### 5. Future Work

As we say in the previous sections, this project is just a prototype that can hardly satisfy the requirement in production environment. Here we list some potential ways to improve the whole system.

First of all, considering the job management system, it is actually prone to worker crash. Once a worker crash or isolated from the master, the whole job would failed. To fix this, we can automatically reschedule this task to a different worker just in a manner similar to [3]. However, the isolated worker may be still alive and writing things to the output path. So more procedures need to be taken in such case, for example, to write output in a temporary location and clean them up after the whole task finished. What’s more, the job tracker also suffers Single Point of Failure (SPOF) problem, which can be solved by having multiple replica on and sync their states with etcd or Zookeeper[6].

Second, for the searching part, there can only be one host for one data group, which apparently can not reach the increasing query need. Fortunately, adding the feature of multiple replica can achieve by just modify the group selection algorithm for one host. For example, when a new host start, it can check the number of replica of each group, and choose the one with least replicas.

### 6. Conclusion

This project implements the search engine by utilizing MapReduce, Etdc and Kubernetes to improve the fault-tolerance and high-availability. Also making use of the file system on CIMS similar to HDFS helps store the data. MapReduce help generating the TF-IDF, indexes and docs on distributed system in parallel, which reduce the time to gen-

<table>
<thead>
<tr>
<th>query</th>
<th>first search (s)</th>
<th>second search (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misaki</td>
<td>0.30668</td>
<td>0.08919</td>
</tr>
<tr>
<td>join</td>
<td>0.39387</td>
<td>0.08747</td>
</tr>
<tr>
<td>save</td>
<td>0.36508</td>
<td>0.08699</td>
</tr>
<tr>
<td>chase</td>
<td>0.40641</td>
<td>0.08675</td>
</tr>
<tr>
<td>sai</td>
<td>0.34785</td>
<td>0.08618</td>
</tr>
<tr>
<td>results</td>
<td>0.38783</td>
<td>0.08178</td>
</tr>
</tbody>
</table>
erate the files greatly; Etcd improves the fault-tolerance of the search engine and solve the consistency problem between main queryer and hosts(sub-queryer); Kubernetes helps us solve the management issues; using index cache during the query raises the efficiency of searching. With these technologies, this search engine is high-scalability and efficient in both indexer and queryer parts.

7. REFERENCES


