YASE Final Report

Team Member：Chaoyi Huang, Xinchao Shen, Huinan Yu, Mengli Li

1 Introduction

YASE is a mini search engine which has an index size of about one million web pages. It is designed to get relevant search results within a reasonable limit of time. It consists of four major components: Crawler, Indexer, PageRank and Search Engine. By the way, YASE stands for Yet Another Search Engine.

The milestones and labor division:

4.16 - 4.20 : Crawler - distributed crawler prototype (Chaoyi Huang)
Indexer - hadoop, EMR, Berkeley DB (Xinchao Shen, Huinan Yu)
PageRank - hadoop; map and reduce function (Mengli Li)

4.21 - 4.25: Crawler - robots.txt manager added (Chaoyi Huang)
Indexer - MapReduce functions; build inverted index database (Xinchao Shen, Huinan Yu)
PageRank - revise map and reduce; test on EMR (Mengli Li)
UI - design search interface html; learn css; print search result html (Huinan Yu)

4.26 - 4.30: Crawler - improve the URL frontier (Chaoyi Huang)
Indexer - produced first working inverted index (Xinchao Shen)
PageRank - handle dangling links and sink links; find out number of iteration time of converging (Mengli Li, Huinan Yu)
UI - show results in different result pages (Huinan Yu)
Search Engine - write search servlets, master and workers (Chaoyi Huang, Xinchao Shen, Huinan Yu)

5.1 - 5.9: Search Engine - ranking algorithm (Chaoyi Huang)
Indexer - significantly improved inverted index (Xinchao Shen)
PageRank - Normalization (Mengli Li)
UI - add side bars, spell check, filter unwanted result; voice recognition; add autocomplete feature on search box (Mengli Li, Huinan Yu)
Report (Whole team)

2 Architecture

Given seed URLs, the distributed crawlers start to crawl pages utilising URL frontiers, and produce two kinds of files: links and raw data. Indexer gets the raw data, run MapReduce to produce Inverted Index DB, Preview DB and Lexicon. Then it distributes the DBs on the EC2 cluster. PageRank takes links file from the crawler and run MapReduce and store the PR result on EC2s. Web Frontend takes queries from the user and passes them to Search Engine. Search Engine then contacts the corresponding node in the EC2 cluster to get the url lists and IDF of the keywords, and then get PageRank scores and Preview of these pages, rank them and send them back to the Web Frontend. The front end will present those results to the user with weather widget, amazon widget and ebay widget.

Since our DBs are fully distributed, in terms of storage and retrieving info, we can achieve scalability simply by adding more worker nodes in the system. However, if there are thousands of concurrent search requests to the master, it could be a bottleneck that we need to address.

Our design is partially resilient to faults. If one worker crashes, we will lose just the content on that particular worker, but everything else will be fine-- we can still retrieve info stored on other workers. However, if the master is down, then we have to fix it before we can make any further requests.
Following **extra credits** are implemented: Search suggestion; Spell checking; Voice recognition; Weather widget; Amazon/Ebay integration

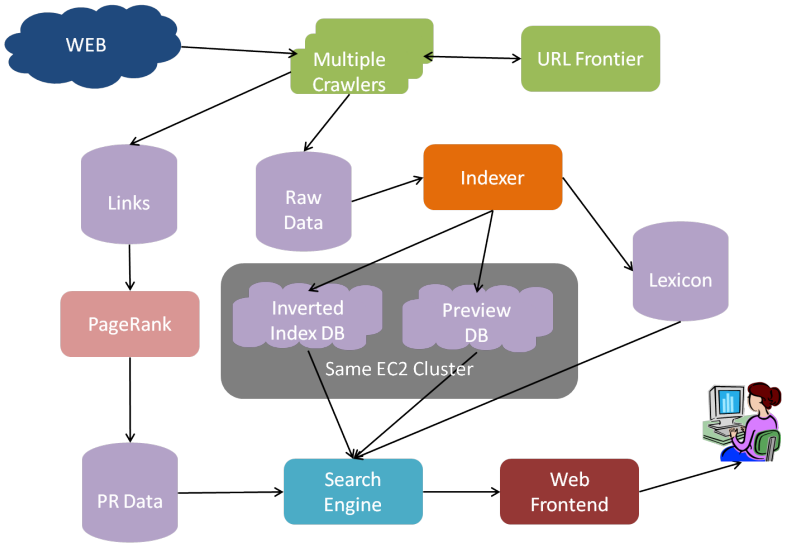

Figure 1. YASE search engine Architecture

3 Implementation Details

3.1 Crawler

The distributed crawler has following main components:

- **Multiple URL Frontiers:** Every URL frontier is an on-disk queue on top of Berkeley DB. According to Heydon and Najork's *High-Performance Web Crawling*, using 3 times as many URL frontier queues as crawling threads will get high speed. This is indeed a very important design if we add delay between two successive requests. Because after comprehensive profiling our crawler, we found most of CPU cycles are used to find the next available URL if we only use a single URL frontier. Therefore, we decide to split the URL frontiers so that each host's URL can only be store in one URL frontier. By doing this, we don't need to spend too much time to poll URL with same host from the queue and put them back if delay time is not pass.
- **Synchronized Data Structure:** Because we use multiple threads to crawl, it is important to have some synchronized I/O methods. In our crawler, there are SyncFroniterWriter, SyncLinkWriter and SyncRawWriter. The URLFilter is also synchronized. It is responsible for checking if the URL is already visited. We use in-memory Least Recently Unit to store visited URLs. That is because URLFilter is frequently used and we don't want it to be a bottleneck. But since the memory is limited, we should store the hash value (16 bytes) of URL by using MD5. This solution works perfectly for millions of pages scale.
- **Inter-crawler communication:** If a crawler finds URLs that do not belong to itself according to hash value, it will send them to their crawler once a while via socket communication.
- **Robots.txt Management:** Since this component is used almost every time we crawl a URL, it is very important to have a efficient design. Here we use an in-memory Least Recently Unit to store parsed robots.txt. Therefore we could not only have efficient access to robots.txt for frequently visited site, but also avoid to use large amount of memory.

3.2 Indexer

Indexer includes following parts:

- **Stemmer**: Read each line of the output of the crawler, stem all the words in it using lucene, and then store the results on S3 in plaintext
- **InvertedIndexer**: Run inverted indexer on EMR taking input from S3, and then output the results back to S3. The format of the output is as follows:

word	url1_TF_PageLength_PageTitle	url2_TF_...	...

  - the underscore "_" is just a representation of the unique separator used. The one actually used was "@-!"
  - the order of urls is sorted descendingly by their TF using a fixed-size priority queue
  - When constructing each line, StringBuilder is used instead of String "=+" operations. This saves tremendous amount of memory and CPU time since we don't have to create duplicate string objects for every word-url pair.
- **IndexSplitter:** Split the results of EMR onto different EC2 instances and store them locally using Berkeley DB**.**
- **ContentSplitter:** Similar to IndexSplitter, it creates "Preview DBs" and split them on each EC2 instance so that search engine can look up the preview of a web page by its url.
- **IDFCalculator:** Produces a lexicon and calculate the IDF of each word so that Search Engine can make use it during ranking

3.3 PageRank

- **Input file:** The input file for page rank is a text file called "links.txt" which contains one source domain, initial PageRank, and some destination out links for each line**.**
- **File Conversion:** The first step for PageRank part is to convert input file to the format we want. This conversion includes merging same source domain, removing hog links from destination links, and handling with the dangling links and sink links(randomly generate 200 outlinks for dangling links and sink links).
- **Run MapReduce on EMR:**The second step for PageRank is to design the map and reduce function and run it on EMR.

  For the map part, the input key: "source domain" ; value: "<PageRank> dest domain1, dest domain2 ,..."
  The output for each outlink: key: "dest domain"; value: "source domain <PageRank> <number of outlinks>"
  For the reduce part, the input key: "dest domain"; value:"source domain <PageRank> <number of outlinks>" "another source domain info" "..."
  The output key: "dest domain"; value: "<new PageRank> source domain1, source domain2 ..."
  We calculate new PageRank for each domain in reduce function by the formula:

  $$PR(u)=cvBuR'(v)Nv+(1-c) \qquad (c=0.85)$$

  In each iteration, the map got the input files from the output files of the last reduce. Considering the efficiency of PageRank, we decide to do 10 iterations because it is good for both accuracy and time efficiency.
- **PageRank Normalization:** The last step is to handle the text file after map and reduce. We normalize the PageRank result to 0-10 and store the PageRank data in a text file in the format of pairs of link and corresponding PageRank score.

3.4 Search Engine & Interface

We designed the UI with a tool named UIKit. We wrote css to embellish the front end html. On the search interface, we included a button for voice recognition. On the search result page, each result has title, crawl time, url and brief preview. Each page contains 10 search results. The spell checking and search suggestion was also implemented. We also included weather widget, amazon widget and ebay widget.

4 Ranking

In the ranking algorithm, we take following factors into account:

4.1 Cosine similarity between TF-IDF vector of query and webpage: We also consider the length of webpage as a feature in TF-IDF vector, so that longer webpage won't have unfair advantage over short webpage.

4.2 Words in URL or Title: We boost the ranking of a webpage if it's URL or title contains the keywords. Also, we check if the domain name matches the keywords so that the homepage of a website will display at the very top position. The boost factor is calculated by adding all matched words' IDF values and then multiply the original score to get the final score.

4.3 Length of URL: The shorter URL usually means higher hierarchy for one website. The final score should be divided by square root of the URL length.

4.4 PageRank: We simply multiply PageRank's value to original score.

5 Evaluation

5.1 Crawler evaluation

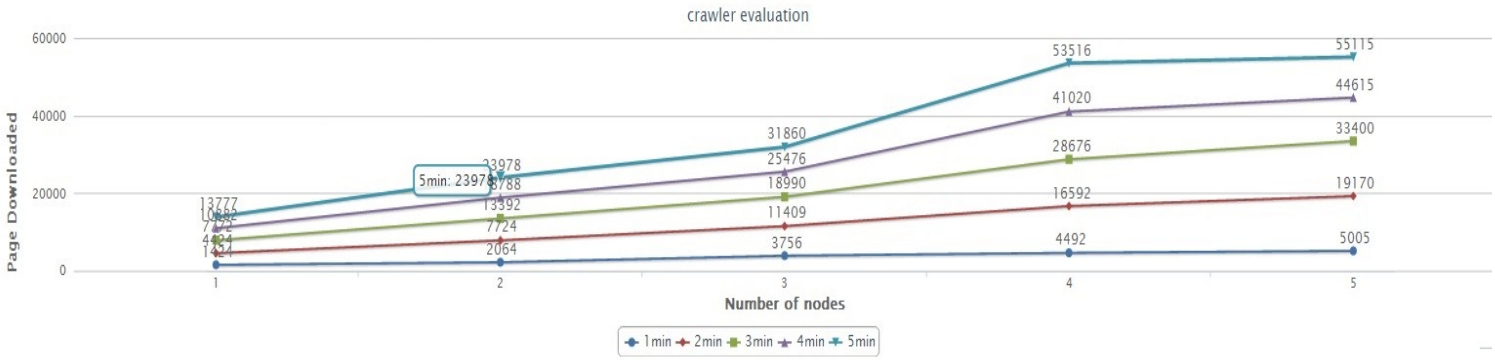

Figure 2. Crawled pages on 5 EC2 c3.large nodes with 30 threads each

This graph shows that more EC2 nodes bring us more pages within same amount of time. But the speed per node is not always increased. This is because there are lots of network traffic and I/O operations when sending/receiving URL to/from other crawlers. In our test, we have more than one gigabyte URLs in one crawler after downloaded one millions pages. This might suggest that we don't even need to send URLs to other if every crawler has a long enough seed URL list. Discarding others' URL won't result in insufficient URLs for any crawler.

5.2 Indexer evaluation

Note: all nodes are m3.2xlarge

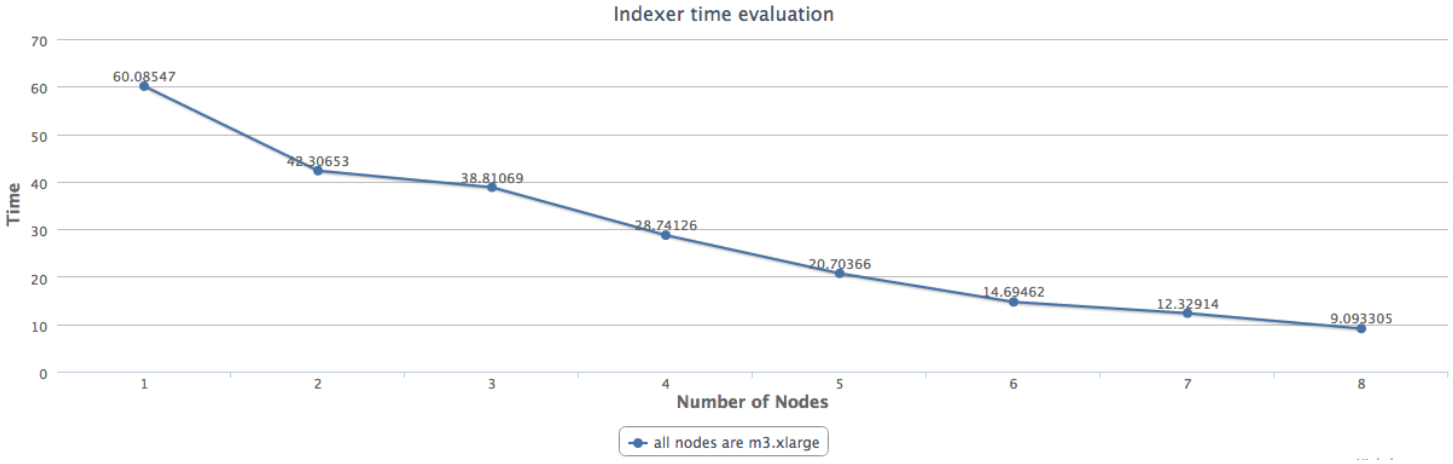

Figure 3. Indexer time evaluation as node varies from 1-8

As we can clearly see from the graph, increasing the number of nodes does help improve the performance of the indexer, but at a diminishing rate. Running on 6 - 8 nodes is quite cost-effective since we would not experience much performance gain by adding extra nodes.

5.3 PageRank evaluation

Note: all nodes are m3.xlarge

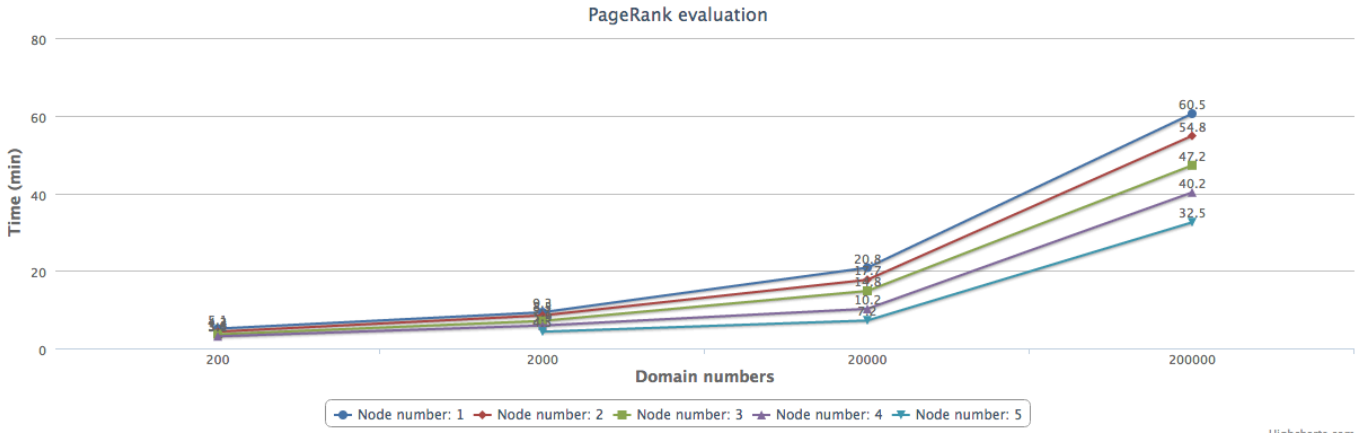

Figure 4. PageRank time evaluation as domain number and time varies

Time to run PageRank: As we can clearly see in the picture above, as domain number increases, PageRank needs more time to run. In real situation, since we crawled 1 million pages, so we need to calculate PageRank for 200,000 domains. Obviously, increasing the number of nodes does help improve the performance of the PageRank, but not too much. That is because most of time for running PageRank is spent on IO process. So it is not so necessary to increase the number of nodes to improve the efficiency of PageRank

Number of iterations: We test a set of number of iterations (1,5,10,15,20) PageRank needed to converge. And we found when the number reaches 10, PageRank won't change too much, so that we think 10 is enough for the PageRank to converge and it's a good choice for both time efficiency and accuracy.

5.4 Search engine evaluation

Number of query words	Range(ms)
1	50 - 2000
2	200 - 7000
3-5	4000 -10000
5-20	8000 -17000

Table 1. Searching time range for different number of query words

In conclusion, the time ranges widely for different situation.
For one word searching, the more common a word it is, the longer time it needs to search. If a word only contains digit, it often needs more time to search.
For a phrase contains a couple of words, the more keywords it contains, the longer time it needs to search. If a phrase contains many key words and all of them are common words, it needs long time to search.
Cache: We designed cache for the searching content. Once the user search a word or a phrase, our search engine would store the searching results including title, url, time and preview and stored the data as a text file in the cache. So that if the same searching content is searched next time, it only needs about 100ms to show the results.

6 Conclusions

We believe our project is a success. It achieves every goal in our design specification and surpasses our initial expectation.
Crawler and indexer were optimized in almost every aspect we could think of. Our implementation can crawl one million web pages within 2 hours, index them in less than 10 minutes, calculate the PageRank within 30 minutes and got the searching result in 1000ms on average.
We think if we had more time, we would firstly add synonym support. e.g. if you search “upenn”, it will automatically add “penn” in the search keywords. And then make the system more robust so that it’s more resilient to faults.
We learned many things in this project, not only the web technology, but also the importance of team work. Besides, we also learned some lessons from the project. For example, initially indexer was running unbearingly slow even on 16 m3.2xlarge machines. It would timeout after 10 minutes for “not reporting status to master”. Even setting the timeout to 10 hours would not solve this issue. We knew something was not right. After profiling the code on a small sample of the data, we found out that StringBuilder was consuming all the CPU time and memory. So we rewrote the MapReduce implementation and corrected the way StringBuilder was used. The improvement was significant. Now it is lightning fast--less than ten minutes for one million web pages! It was a huge lesson learned. Another example is for PageRank part, initially we have no idea to handle the uncrawled links, at first we just assume the in link for uncrawled links as out link, but as a result, the page rank for sink nodes are very high and not sensible. So we try many different methods, and finally decide to randomly generate 200 out links from crawled links as the out links for uncrawled links.