Buzz!t : A Distributed Search Engine

Bowen Bao, Yunqi Chen, Linwei Chen, Haoyun Qiu
Department of Computer and Information Science
University of Pennsylvania

I. Introduction

Buzz!t (AKA:Buzz it!) is a distributed search engine which majorly focus on searching prevalent topics such as entertainment, arts and sports.

II. Project architecture:

Buzz!t includes 4 major parts: a distributed crawler, indexer, PageRank and a search engine interface. Crawler offers web pages crawled from internet for the indexer and PageRank, and the search interface uses the combined results of indexer and PageRank for queries and content display. Here are the details of each parts:

Crawler:

The macro view of this part is a Mercator-design crawler with master-slave structure, which includes a master node to dispatch jobs and monitor their status and several worker nodes responsible for crawling. Each worker is multithreaded and includes these major components:

- Url Frontier: HostHeap for maintaining politeness; frontend url queues for holding urls and determine url priority; several backend url queues contains urls of only one host for each.
- DUE (Duplicate URL Eliminators): Determine if a url has been visited.
- Host Splitter: Redirect urls of hosts not belong to this worker node to other nodes.
- HTTP Fetcher: Fetch web page from internet.
- Link Extractor: Parse and extract links within an html document.
- Url Filter: Remove urls include disallowed links and manual black lists.
- Content Seen: To determine if a url's content has been crawled before.
- Database: Store page content for Indexer and sub-links for PageRank.

Indexer:

The primary goal of indexer component is to create inverted index of the documents crawled by crawler part. The inverted index consists of a term(keyword) as the key, the document ID it appears in as well as the number of times(word count) it appears in that particular document.

In addition, in order to give ranking suggestions for UI display, the indexer component maintains a data structure containing the weight of every term to different documents which represents the importance of a term to the document it associates to. The weights mainly takes into consideration of tf-idf, proximity factors. The weighing mechanism of the indexer component also implements stemming concept to normalize words belonging to same etyma to a single word. More advanced function would be lemmatizing words in a document and performing weighing computations over each classification.

If time permits, indexer should also adapt its ranking strategy according to the geo-location where a search query occurs. Using this as a baseline, we extend our indexer with multiple feature scores, including phrased based indexing, query term selection, time-based ranking etc.

PageRank:

The PageRank component is responsible for performing link analysis on the web pages retrieved by the crawler. More specifically, this component takes in the links crawled and extracted by the crawler and constructs a graph where nodes represent the URLs and edges represent links. Then the URL ranks are calculated and updated iteratively using PageRank algorithm until they converge. Spark is used as the basis of PageRank implementation, and the output of PageRank is a mapping from URLs to their page ranks.

Several extensions are preferred and will be implemented. For instance, because not all linked pages are crawled (e.g. due to robot.txt restrictions), we are expecting a fairly high number of dangling links. Accordingly, in each iteration pages without outbound links are removed in order to normalize the number of outbound links on pages with dangling links.

Search Engine and UI:

The frontend is a search form for the user query and weighted results listing after user's search. Behind that there is a query parser to validate, parse and classify the queries in order to look up the results. And the backend will be several EBS storages of the combined results of indexer, PageRank and manual adjustment.

III. Rough milestones:

11.25
Crawler: Deploy on EC2 with master and worker nodes and do testing. Offer a large corpus of web pages for indexer and PageRank.
Indexer: Stemmer support, Test on large corpus, Deployment on EC2, Extension to multiple feature scoring algorithms.

12.2
Crawler: Improve performance and test. Prepare for the extra credits like web-page priority, consistent crawling, same-content document detection. Offer bigger corpus web pages.
Indexer: Benchmark suite with multiple feature scoring ranking algorithms.
Search Engine: Basic UI.

12.9
Crawler: Improve performance and test. Extra credits.
Search Engine: spell-check, dynamic search.

12.10~demo
Bugfix, running and optimization.

IV. Bitbucket repository:

git@bitbucket.org:cis555final/mysearchengine.git

V. Division of labor:

Crawler: Haoyun Qiu
Indexer: Bowen Bao, Linwei Chen
PageRank: Yunqi Chen
Search Engine: Linwei Chen (Frontend UI), Bowen Bao (Ranker and Image search), Yunqi Chen (Mobile client, auto completion), Haoyun Qiu (Web server structure and build/test flow)

VII. Implementation

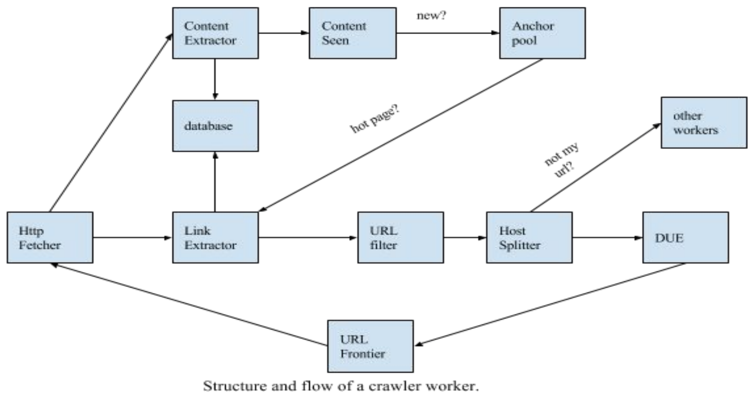

Figure 1, Crawler Architecture Overview

Crawler

Crawler should be polite, robust, scalable and efficient. URL Frontier is responsible for maintaining politeness. Within it there is a Host Heap based on each active host's next-crawl timestamp; a frontend url queue for waiting urls; a backend url queue with crawling urls of each active host. We improve the url ordering in frontend queue with Similarity based algorithm[1] by maintaining another queue for hot urls. A hot url is which the url or title contains words in anchor pool or its content contains over 10 of them. And each sub-link of hot page has a 20% chance to become a hot url and enqueue the hot queue. Backend queue will first look for urls in hot queue, then in normal queue when no hot url exists, so that we can crawl more contents we want with limited resources.

As only a small subset of pages can be crawled and indexed, redundancy should be removed. We implement Content-Seen to remove pages with similar content and check HTML's canonical and lang tag to drop duplicated and non-english pages. By that about half of pages could be removed. For robustness, we've handled complicated cases include various of malformed urls, content-encoding (i.e. gzip, deflate), cookies issues, different response codes and so on. For performance, worker is multithreaded and URL Frontier has been tuned to make threads busy; DUE and Content-Seen are in-memory with Rabin fingerprint algorithm[2] to control memory footprint. Furthermore, caching and batching are widely used in crawler: robots.txt are cached; Host Split urls are cached and batching for other workers; database sync are also batched to make disk I/O not an issue. Crawler clean web page (10% size of raw html) for Indexer and store sub-links for PageRank which make their works easier. Our crawler is scalable with nodes running on EC2 and Berkeley DB on AWS EBS is utilized as the storage.

Indexer

Figure 2 presents the architecture of our search engine. Our search engine provides two types of search: image search and text search. We support image search by first conduct an image recognition based on pre-trained VGG convolutional neural network[3]. We retrieve the recognized text of that image as query and treat as normal text search. In text search, our query result consists of two parts: general ranked web pages and knowledge entity. We will explain below what each part is, and how we implemented it. All components in this part runs on Amazon EC2 and is implemented in Java and Apache Spark.

[1] Cho, Junghoo, Hector Garcia-Molina, and Lawrence Page. "Efficient crawling through URL ordering." (1998).

[2] Rabin, Michael O. *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[3] Very Deep Convolutional Networks for Large-Scale Image Recognition K. Simonyan, A. Zisserman arXiv:1409.1556

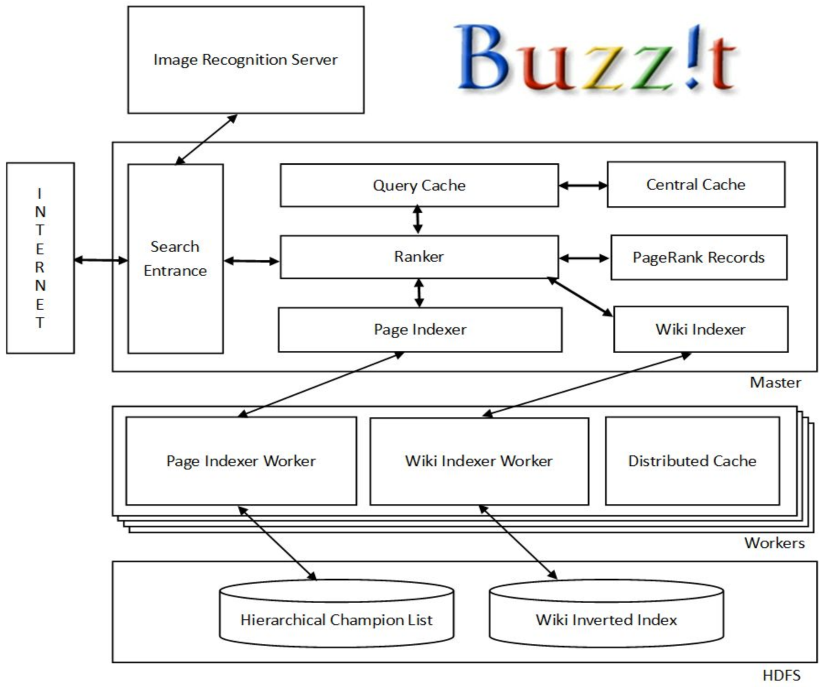

Figure 2, Components Inside Indexer and Ranker

We rank each of our crawled web pages by the following equation:

$$\begin{aligned} w_i = & \sum_{x_j \in Content(i)} \mathrm{TfIdf}(x_j, doc_i) + \alpha \sum_{x_j \in Title(i)} \mathrm{TfIdf}(x_j, doc_i) + \beta \sum_{x_j \in Url(i)} \mathrm{TfIdf}(x_j, doc_i) \\ & + \gamma \sum_{bi\,j \in Doc(i)} \mathrm{TfIdf}(x_j, doc_i) \end{aligned} \tag{1}$$

where

$$\begin{aligned} \mathrm{TfIdf}(x_j, doc_i) &= \mathrm{NormalizedTf}(x_j, doc_i) \times \mathrm{Idf}(x_j) \\ &= \Big(a + (1-a)\cdot(\mathrm{count}(x_j)/\mathrm{max\_count}(doc_i))\Big) \times \Big(1 + \log(\frac{N}{n_j})\Big) \end{aligned} \tag{2}$$

and α, β, γ are preset constant weighing parameters. This allows us to tune our ranker till we find a satisfying configuration. Each x_j is a word in the query. We implemented a stemmer along with a stopword list for words in documents and queries so as to enhance the accuracy of matching. Notice that this is the weight of document provided by indexer, our final result combines these weights with the corresponding PageRank factor of each page. We implement this computation in Apache Spark and store the result of Tf-Idf distributed across HDFS, so that we will not need to recompute it everytime we restart our cluster.

Now we have a way to rank our documents, we want to rank them efficiently. Our approach basically consists of two parts: multi-layer caching and hierarchical champion lists.

For multi-layer caching, as in Figure 2. , we provide our search engine with an in-memory central cache for a limited amount of popular queries. This cache operates on a least visited removal basis. Our second layer cache spreads across all nodes in the cluster. We implement this level of caching by utilizing the Apache Spark RDD object.

To increase the ranking speed, we implement champion list as posting list, which is each word possessing a list of documents in which it appeared, but in a sorted and limited length fashion. Now we compute weights for documents only if it appears in one of the query words' champion list. This approach greatly improves efficiency as the number of documents which we are computing is greatly reduced, yet it prohibits certain drawbacks such as we might not retrieve enough web pages. In order to solve this problem, we implement hierarchical champion lists, which will compute longer champion lists if it discovered that the total number of results are less than expected.

Knowledge Entity Retrieving

Unlike crawled web pages which are fuzzy and sometimes unreliable, wiki corpus has the advantage of being precise and trustworthy. Our approach in this module utilizes this property, and focus on retrieving the most exact matching to the user query. The method is rather straightforward and simple. We index each wiki entity only by their title, and we retrieve entities only if all query terms are matched. In this strict matching policy, there still might be multiple matched results. We simply apply a greedy selection by selecting the entity of smallest title length, as this in turn is what the Tf-Idf approach would have chosen.

PageRank

PageRank is an iterative algorithm that involves many joins and hence can benefit from Spark RDD partition. Therefore our PageRank is implemented on top of Spark.

There are two sets of data we are keeping track of, namely 1) url -> its neighbors, and 2) url -> its rank. The core algorithm per se is simple and elegant:

a) Initialize all pages' rank to 1.0;

b) for each page p, distribute p's rank among its neighbours (another interpretation: p is voting for its neighbours, where the vote weight each neighbor receives is dependent on p's rank and the number of neighbors p has);

c) for each page p, collect the contributions it has received, applying a decay factor such that

$$new\_rank\_of\_p = (1 - factor) + factor * weights\_received\_by\_p \quad (3)$$

We let the factor be 0.85, as suggested by Page and Brin. The page p's rank is then updated.

Step *b)* and *c)* shall be applied multiple times until the ranks converge (it has been proven that they eventually will). We've experimented with different number of iterations and found that usually 10 iterations are good enough. Although we ultimately set up the number of iterations to be 20 for some extra accuracy, the difference between 10 and 20 iterations is trivial.

All the work in the PageRank module are done on Amazon ec2 -- more specifically, a cluster with one master and two workers. The process starts with extracting urls and their neighbours from the database provided by the crawler, which are stored on EBS volumes. After some basic data cleaning, these url relationships are put into a HDFS to be used as input for the PageRank Spark task.

A practical concern is the size of the output, from which the search engine frequently reads. Caching them all in memory is a desirable solution but only if the size is small enough. Two major steps are done to reduce the size of the PageRank result: 1) remove entries that are not indexed by indexer (their pageranks will never be retrieved), this reduces the size by an order of magnitude; and 2) remove the "http[s]://" prefixes from the urls, which further reduces the size by more than 10%. In the end, for the half million page corpus, only 26 MB is needed to store the pagerank results[4].

Search Engine

The search engine is a servlet based webapp running on jetty. It is combined with indexer component on the same master node. User can either enter text (with auto-complete) or uploading image to search. The request parameters are sent in the type of multipart/form-data. Then the servlet parses out parameters and retrieves a QueryResultItems object from indexer per query. There are two methods handling parameters. If the parameter is

4 If the corpus size gets even larger, we may use document id in lieu of url as the primary key for pages.

text, the server straightforwardly matches entries in indexer and return results. If the parameter is an image, search engine consult to a deep learning program running on a remote server to get the classification of the image, which is a query in text. Afterwards, the search engines handles the query as normal. The QueryResultItems includes the query, a list of top scored result pages we crawled and a knowledge entity if the query residents in our wiki index. If the data being sent is a image, the search engine firstly transforms it into a query which best describes it and then go through the same process as plaintext queries.

Finally the servlet wraps up the QueryResultItems and dispatch to JSP which organizes all the information to be displayed in our result page. It also handles corner cases such as no results were found for entered query or the image format user uploaded was illegal.

VIII. Evaluation

Crawler

11 * m4.large EC2 instance were used for workers and 150 threads for each in our latest crawled. Within about one hour, we sent 2239044 HTTP requests totally and about 75.00% of them return OK. 1177276 web pages were fetched (rests are robots.txt) and only 565353 pages were stored in db since others were removed by components mentioned before. About ⅓ of fetched pages are hot pages and 13628 urls were filtered by URL Filter. Fetch speed could be up to over 100 pages/s (about 100 KB/page) per instance but gradually reduce to about 35 pages/s. Some potential reasons we found would be: the consumption of Host Heap is much faster than finishing a new url and threads would be blocked when heap is empty. One solution is to tune the multiple between active hosts and thread numbers. Currently we use a value of 30 (i.e. 4500 active hosts) to make threads busy for a longer time, pretty big comparing to Mercator[5]. Another reason is some unhandled exceptions and crawl-delay. More detailed stat could be found here below project's Bitbucket repository: ./crawler/logs/.

Indexer

The precomputing for hierarchical champion list and wiki posting list runs on the cluster of t2.large instances.

Table1, Indexer Performance Metric

Input size	Result size	Number of workers	Time
240k web pages + 2 million wiki entities	12G	8	46 min
240k web pages + 2 million wiki entities	12G	16	24 min
507k web pages + 2 million wiki entities	23G	16	51 min
2 million web pages + 2 million wiki entities (attempt)	> 100G	16	6.7 h

PageRank

The PageRank module is implemented on top of run on m3.large instances (1 master + 2 workers)

5 Heydon, Allan, and Marc Najork. "Mercator: A scalable, extensible web crawler." World Wide Web 2.4 (1999): 219-229.

Table2, PageRank Performance Metric

Input size(number of lines)	Number of Iterations	Number of Cores	Time
8 million	10	4	1.6 min
8 millions	20	4	2.4 min

Search Engine

Search engine is combined with indexer component running on the master node of t2.large cluster(1 master + 16 workers)

Table3, Search Engine Concurrency Performance Metric

Total Number of Requests	Concurrency Level	Number of Completed Requests
10000	1000	9810
10000	2000	9950
10000	4000	9978

IX. Conclusions

For initial seeds of crawling, websites like shopping and mirco-blog (i.e. tumblr) should be removed since out-links are tremendous but not informative. Within a worker, we also sample the performance and found that returning responses from hosts is the most time-consumed part (normally over 80%). A further research on DNS resolve would be a great help.

It is crucial to tune the ranking function so that the server returns most precise results. On the tf-idf side, documents whose "anchor information", as known as title and url text, hits the keywords in query gains more weight than documents whose content hits query keywords. Referring to equation(1), α, β, γ represent the importance of terms occurring in different parts of the doc and contribute to the precision of displayed results. On the other side, how tf-idf and PageRank combined to influence the final score is a significant factor resulting to the quality of search results as well. After days of experiments, we found that web pages filtered by equation (4) will provide the

$$total\ score = \sum_{x_j \in content(i)} tfidf(x_j,\ doc(i)) + 2 \sum_{x_j \in title(i)} tfidf(x_j,\ doc(i)) + 2 \sum_{x_j \in url(i)} tfidf(x_j,\ doc(i)) + 2\ pagerank(x_j)$$

(4)

most satisfying result to our users.

Apart from the content we present to users, there are other details lying in the user interface can enhance user experience effectively. Auto-completion function helps predict what users are typing as well as giving search suggestions, making the searching process more efficient from human's perspective. The layout of the result displaying page is user-friendly as well. The search bar on the top retains the query people just typed. Keywords are highlighted in the text of results. Transformed query from the uploaded image is also shown on top of the results. The right side of the page provides a fast track gateway by showing user wiki information and links that match query keywords.