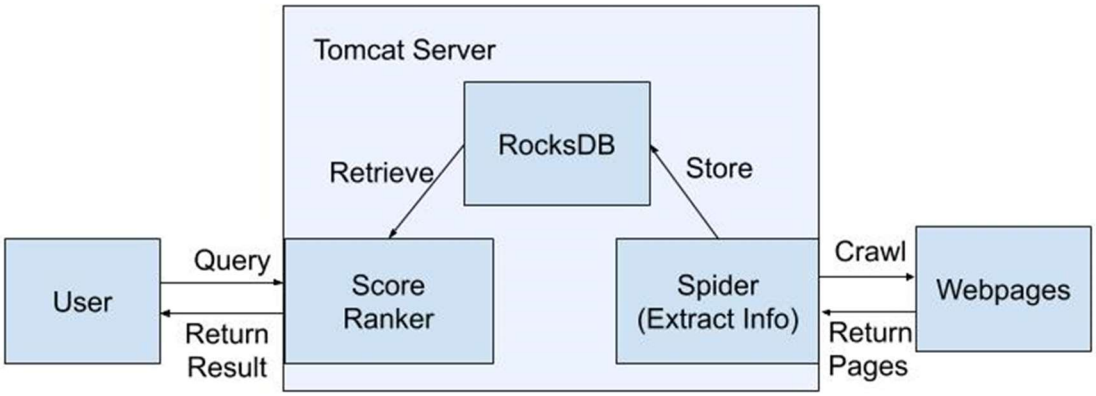
2019

COMP4321 - Project

GROUP 25

AU, HO LEONG (HLAUAC@CONNECT.UST.HK) 20260060
KIM, HYUN GYU (HGKIM@CONNECT.UST.HK) 20375138
CHEUNG, KA HO (KHCHEUNGAP@CONNECT.UST.HK) 20465294

OVERALL DESIGN



SPIDER (CRAWLER)

The crawler fetches the pages in www.cse.ust.hk recursively until there are no unvisited websites under it. 13 RocksDBs are created when the program is running. Stop-word Removal and Stemming are processing towards the words in a webpage.

SCORE RANKER

After getting the indexed pages from the database and the queries from web interface, the page scores of queries are calculated according to cosine similarity, PageRank and title score. The sorted results will be sent to the web interface.

DATABASE

RocksDB made by Facebook is used as our database for storing all the indexes from the Spider. It does not just store the web page itself, but there is many other information that are extracted from the web page for the ranker to compute ranking fast.

WEB INTERFACE

There are two web pages written in html which serves as a user interface, which is ‘index’ and ‘result’. Index page allows users to input their queries. The result page shows the search result dynamically to the user.

File Structure

The source code of the whole environment are:

src/main/	kotlin/	main/	SpiderMain.kt TfidfMain.kt
		spring/	Application.kt Web.kt WebController.kt
		util/	CSVParser.kt HTMLParser.kt Porter.kt Ranker.kt RocksDB.kt
	resources/	templates/	index.html result.html
		favicon.ico	
		stopwords.txt	

DataBase Structure

There are total of 13 different databases for the Ranker.

DB	Key	datatype	Content	datatype
URL_DB	Web URL	String	Web ID	Int
REVERSE_URL_DB	Web ID	Int	Web URL	String
URL_INFO_DB	Web ID	Int	Triple(Title, Date-Modified, Size)	(String, Long, Int)
URL_CHILD_DB	Web ID	Int	Child Web ID	List(Int)
URL_PARENT_DB	Web ID	Int	Parent Web ID	List(Int)
URL_WORDS_DB	Web ID	Int	List(Word ID)	List(Int)
PAGE_RANK_DB	Web ID	Int	PageRank	Double
URL_WORD_COUNT_DB	Web ID	Int	List(word ID, count)	List(Int, Int)
TF_IDF_DB	Web ID	Int	TF-IDF	Double
URL_LENGTH_DB	Web ID	Int	Length	Long
WORD_DB	Word	String	Word ID	Int
REVERSE_WORD_DB	Word ID	Int	Word	String
SPIDER_DB	Word ID	Int	List(Web ID)	List(Int)

Deployment

We deployed in two systems, one from AWS (8 CPUs + 16 GB RAM) and one from university (2CPUs + 4GB RAM). All the ports used are 80.

	IPv4 (connected domain)	Pros	Cons
AWS	52.79.247.238 (www.kimhyungyu.com)	Fast calculation	Slow transmission
UNI	143.89.130.56	Fast transmission	Slow calculation

ALGORITHMS USED

LINKS

To get links as much as possible, the algorithm is recursively crawling the child links in BFS way which the idea is not to visit a website more than once. This is done by using multi-threads of parallel stream method. The time required is close to linear on the number of websites crawled since it is BFS. It also depends on the internet speed since it is done in parallel way requiring high internet speed. In total, we needed about 100 seconds to fetch all the links. There are many websites with links just to direct within its own using headers, which we had to filter all of them out as well. We have found a total of 4900 unique links in cse websites which some require authorization for the access.

KEYWORDS

We first processed all the text using our own filter which filters out any non-English letters since the focus of the search engine is for English words with length more than 2 characters. Then we applied Tokenization using Java built-in library before we remove stopwords using the given stopwords list. Finally, stemming was carried out for each word using Porter's algorithm. This way, we can fetch about 600,000 unique keywords from all 4900 websites.

QUERIES + RANKING

Ranking Formula: Normalized Cosine Similarity + Normalized PageRank / 3 + Title Match * Mean of Cos Sim / 3

After a user submits his queries, the queries go through the same step as keywords such as stop-word removal and stemming. The only difference is that, it detects any quotation marks to retrieve any phrases in the query. There are databases for every word in every document, TfIdf score which is pre-calculated for faster search retrieval and the document length which is also pre-calculated using the TfIdf scores of every word. Using these two databases with the query length, cosine similarity score is calculated. The difference for phrase search is that, since we did not store any n-gram structure of words, we would look up for a matching sequence of words in another database for storing all word IDs in the document. Then it would use phrase appearance count multiplied by the sum of TfIdf of the terms in searching term for evaluation for cosine similarity score. This will further be explained in the bottom section. The cosine similarity score is finally normalized by the highest frequency word in each document.

In addition to cosine similarity score, we also have PageRank scores for each page pre-calculated. It is a link-based score so only if there is a non-zero cosine similarity score, we add the PageRank value. However, since PageRank values are usually much higher than cosine similarity score, we normalized it by the highest PageRank in all documents and still divided by 3 to make the results more precise. The number 3 is derived from the experiment of our own after our personal feedback on the results.

Also, since title matching should be given a higher score overall, if we found any query term in the title, we would add up the mean value of unnormalized cosine similarity divided by 3. The reason why we used flexible value for the title is to give a title score proportional to other scores like cosine similarity since the cosine similarity depends too much on the common terms in the query. Also, the number 3 is again derived from experiment to make some title scores less strong.

WEB APPLICATION

We use Spring Boot and WebJars. Sprint Boot provides a good platform for Java/Kotlin developers to develop a stand-alone and production-grade spring application that you can just run. Many of the steps found on the Spring Guides for creating a RESTful service can be followed verbatim for Kotlin.

WebJars are client-side web libraries packaged into JAR files. It can explicitly and easily manage the client-side dependencies in JVM-based web applications.

We used bootstrap for web design since bootstrap allows simple UI while manipulation of objects gets easier. It also allows some mobile-responsiveness as a default.

PHRASE SEARCH

In order to achieve phrase search, query terms are first to be translated into word IDs and then parsed as a List<List<String>>. If the query does not contain any phrases, the resulting list will simply be a list of List<String> of all size 1. Phrases are denoted by enclosing quotation marks, e.g. “Hong Kong”, in the query. If a phrase is detected, it will be parsed as a List<String> containing the phrase, with each word as a String within the list.

For example, let’s say the word IDs of Hong, Kong and Computer are 1, 2 and 3 respectively. If the query is *Hong Kong Computer* without any phrases (i.e. no quotation marks), the query will be parsed as [[1], [2], [3]]. However, if the query is *“Hong Kong” Computer* with Hong Kong as a phrase, the query will instead be parsed as [[1, 2], [3]].

Once the query is parsed to List<List<String>>, we go through this list and look for phrases, i.e. the inner list has size > 1. Once a phrase is detected, we check whether this phrase appears in the document, if so, how many times does it appear in the document. This can be achieved since we can get the sequence of word IDs in a document from urlWordsDB given a document ID, and we can simply do a string compare to see whether the sequence of wordIds contains the sequence of query term IDs, using the .contains() function in the String class. To count how many times the phrase appears in the document, we used the countMatches() function in StringUtils class in apache.

To get the tfidf score for the phrase, we simply sum up the two words’ individual tfidf scores and multiply it by the number of times the phrase appears in the document. In other words, we are essentially treating the phrase as a single entity and checking whether the entire entity appears in the document.

For example, let’s say a document is made up of the wordIds [1, 2, 3, 4, 5, 2, 3, 7, 2, 9] and the query is [[1], [2, 3]]. Our algorithm will parse the query terms one by one, check whether the wordID(s) appear in the query, and get its tfidf score. Since 1 appears in the doc, its tfidf will be added to the doc for calculation of cosine similarity later. Then for [2, 3], since it spears in the document twice, 2 x the sum of the tfidf scores of 2 and 3 will be added. Although 2 appears 3 times in the document, since we are using contains() and countMatches() functions, and that “2, 3” only appears twice in the wordIds, it will correctly calculate the score

INSTALLATION PROCEDURE

(it could be as simple as “Type make in the project directory”)

Installation from scratch,

1. Install Java 8 or above.
2. To compile and run, use gradle 4.9 and above to the project in any IDE
3. Compile the project and run main/SpiderMain.kt
4. SpiderMain.kt will run crawling, TfIdf calculation then finally open the Tomcat server on port 80.
5. However, if you do not wish to crawl and use existing databases, just run spring/Application.kt

Installation using pre-built jar,

1. Install Java 8 or above
2. Download jar file from http://bit.ly/2WcaRsp (The default file name is SearchEngine.jar)
3. run java -jar path/to/jar <args>
4. If you want to run from crawler to server, pass “spider <max number of websites to fetch>” as arguments
5. If you want to run only the server, pass “server” as arguments

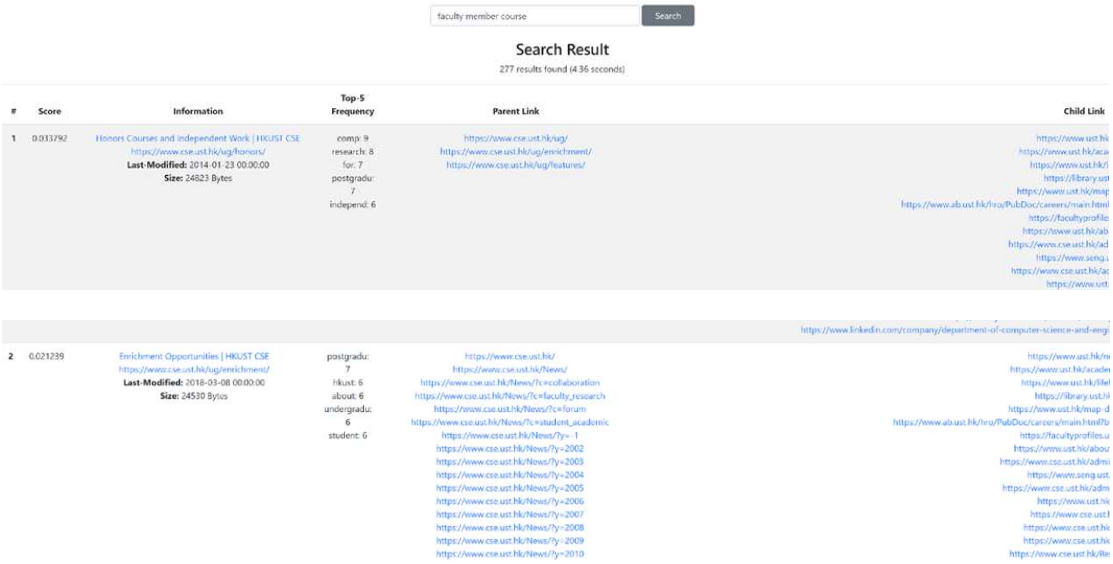
FEATURES BEYOND THE REQUIRED SPECIFICATION

PageRank Algorithm - It uses all the child link matrix to calculate the PageRank of each page until convergence with formula of PR = 0.15 + 0.85(sum of PageRank from incoming edges)

TESTING

Testing 1:

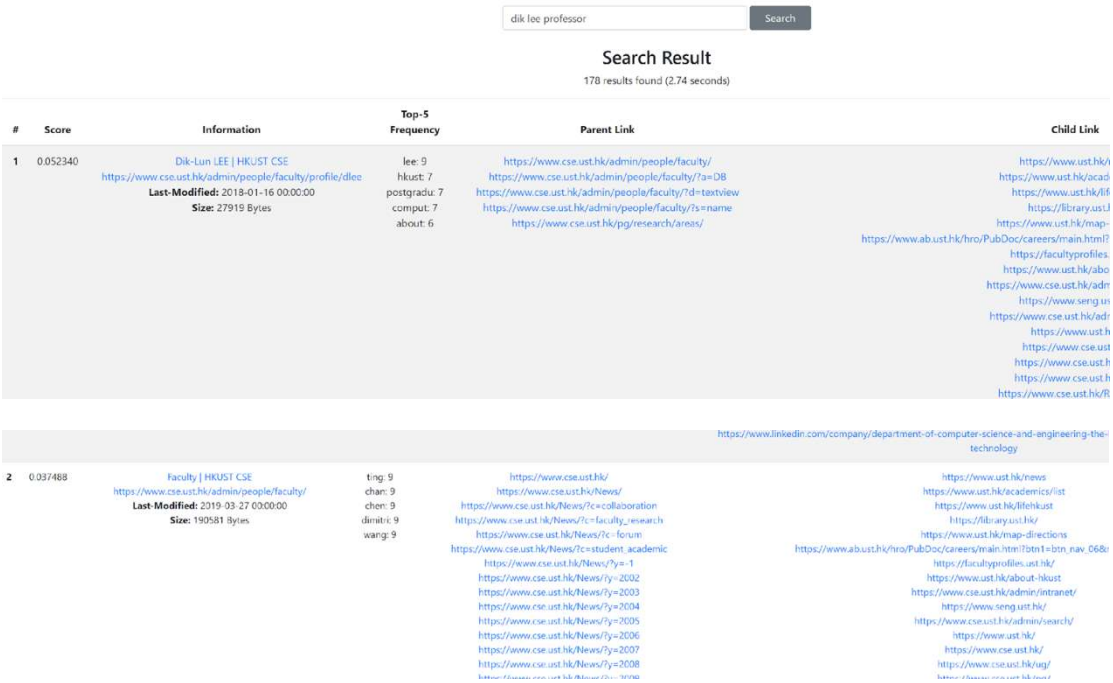
Query		faculty member course
#1	Score: 0.033792	Honors Courses and Independent Work | HKUST CSE https://www.cse.ust.hk/ug/honors/ Last-Modified: 2014-01-23 00:00:00 Size: 24823 Bytes
#2	Score: 0.021239	Enrichment Opportunities | HKUST CSE https://www.cse.ust.hk/ug/enrichment/ Last-Modified: 2018-03-08 00:00:00 Size: 24530 Bytes
Time taken		4.36 seconds for 277 results



At this point of testing, it was nearly a first draft and hence, the score values or websites crawled may not be accurate while the time taken for querying is slow. It only included a simple cosine similarity which we found some bugs in the calculation.

Testing 2:

Query		dik lee professor
#1	Score: 0.052340	Dik-Lun LEE | HKUST CSE https://www.cse.ust.hk/admin/people/faculty/profile/dlee Last-Modified: 2018-01-16 00:00:00 Size: 27919 Bytes
#2	Score: 0.037488	Faculty | HKUST CSE https://www.cse.ust.hk/admin/people/faculty/ Last-Modified: 2019-03-27 00:00:00 Size: 190581 Bytes
Time taken		2.74 seconds for finding 178 results



At this point of testing, we did not go through any optimization but just decreased the number of pages fetched for faster testing. Therefore, the fetching time decreased. Also, it did not include any phrase searching yet.

Testing 3

Query		dik lee professor
#1	Score: 0.581151	Faculty | HKUST CSE https://www.cse.ust.hk/admin/people/faculty/ Last-Modified: 2019-03-27 00:00:00 Size: 190581 Bytes
#2	Score: 0.515471	Dik-Lun LEE | HKUST CSE https://www.cse.ust.hk/admin/people/faculty/profile/dlee Last-Modified: 2018-01-16 00:00:00 Size: 27919 Bytes
#3	Score: 0.355507	Professor Samuel Chanson Scholarship and Awards (2007-08) | HKUST CSE https://www.cse.ust.hk/News/SCSA2008/presentation/ Last-Modified: 2008-06-19 00:00:00 Size: 27546 Bytes
Time taken		4.25 seconds for finding 356 results

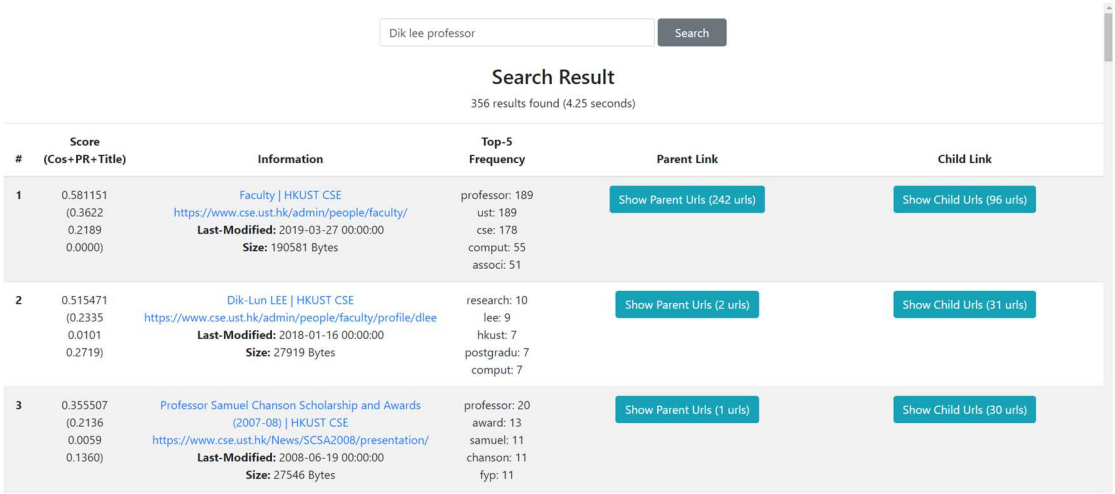


Fig. Searching of “Dik lee professor”

We improved a lot on everything including UI, scoring and optimization. It was still slow for this query since the time taken doubled from test 2, even though other queries usually show the time taken less than 1 seconds after optimization though the number of pages crawled in the database increased to 2200 pages. It also supports phrase searching at this stage using double quotation marks.

Test 4 (Final Version)

Query		dik lee professor
#1	Score: 0.600386	Faculty | HKUST CSE https://www.cse.ust.hk/admin/people/faculty/ Last-Modified: 2019-03-27 00:00:00 Size: 190581 Bytes
#2	Score: 0.409438	Dik-Lun LEE | HKUST CSE https://www.cse.ust.hk/admin/people/faculty/profile/dlee Last-Modified: 2018-01-16 00:00:00 Size: 27919 Bytes
#3	Score: 0.308550	Selected PhD Graduates | HKUST CSE https://www.cse.ust.hk/pg/ourgraduates/ Last-Modified: 2019-03-12 00:00:00 Size: 136607 Bytes
Time taken		0.48 seconds for finding 1111 results

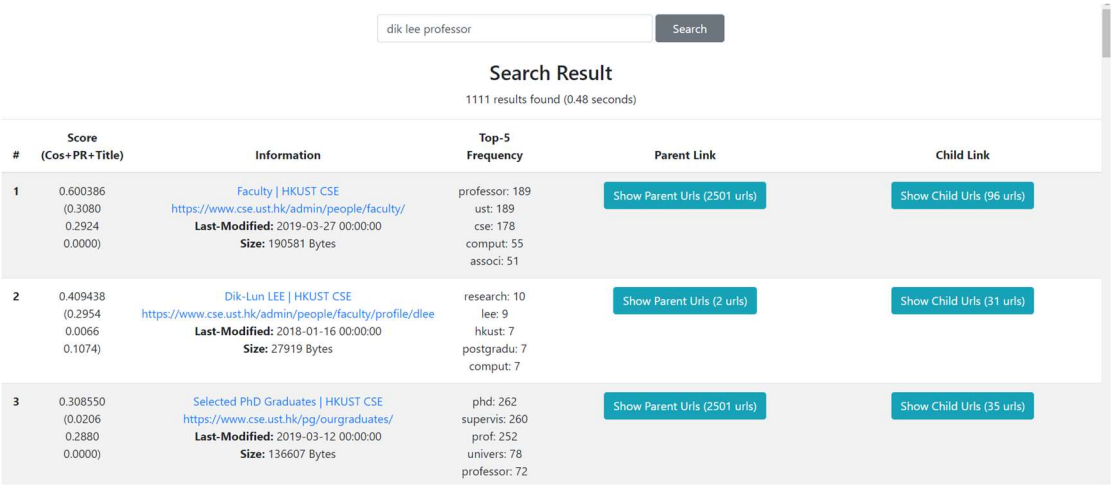


Fig. Searching of “dik lee professor”

We had a drastic improved on optimization by storing all the TfIdf values in memory of the server since it was the reason for slow searching in the previous test. We also indexed all the webpages that we could crawl which adds up to 4900 web pages. The only problem in this is that there is too much information returned in a single page due too many links found in parent links and child links. This led to slow rendering time due to limited download speed. This is due to usage of Amazon EC2 instance for deployment that is in Seoul, Korea. If we deploy in the given school server, the download speed might be faster but since we are only given 2 CPU with 4 GB of RAM, it took about 3 seconds for giving out the same output.

Conclusion

STRENGTHS AND WEAKNESSES

At the time we crawled www.cse.ust.hk , initially the memory gets full and swapping occurs. We figured it out that it was due to RocksDB's frequent read and write which causes memory leakage. This eventually led to an exponential increase in query time for the database. We fixed it by just collecting all things in the Java Collections before writing them at once into the RocksDB which fixed the crawling time complexity in a linear way. Hence, we could crawl all the 4900 websites found under the root link.

Some links that we have crawled are restricted to faculty members, they are encrypted so we cannot access those webpages. By default, those webpages' title, last-modified date and size are 'unauthorized', '1990-01-01 00:00:00' and 0 bytes respectively. They do not contain any keyword; therefore, they are filtered automatically in searching.

INTERESTING FEATURES TO ADD

For ranking function, cosine similarity weighting and PageRank are not enough since it shows same output for every user. We hope we can use more advanced techniques from personalization in which personalization using machine learning algorithms are used a lot for commercial search engines these days.

There are many Chinese words in the webpages, we hope we can handle the Chinese word processing if we had more time. This requires knowledge in Chinese as well.