COMP4321 G1 Report

Zhang Zhe, Kong Tsz Yui, Gupta Harsh Vardhan

May 2024

Contents

1 Overall design of the system

This search engine is written with Python 3.12.0, while we are using HTML and CSS for the user interface. The system contains the following 4 major components, which are crawler, indexer, retriever (Search engine) and also the front-end part. The function also comes with a main program for users to run. Users should not run the individual components (other than main.py) although some can be executed solely for debugging use only.

1.1 Crawler

The crawler will crawl the designated number of pages and website, by using the algorithm breadth-first search. To crawl different data from the website, we have used the package `beautifulSoup` to parse the crawled HTML documents. After that, the data crawled from the website are then stored into different tables.

1.2 Indexer

The indexer will read every word collected from the crawler, perform stemming with removing stopwords. Then, these words are converted into word_id. Followed by that, these words are then inserted into different tables. Then from the inverted_idx we created, we will then create forward index. During this process, pagerank for each page is also being calculated in the indexer.

1.3 Retriever

In the retriever file, we will first convert users queries into `word_id`, by removing every stopwords in the query, then for each word in the query, we will perform stemming. By using regex, we can also separate words that are quoted in the queries. After that, both queries and documents are converted into vectors, and by using cosine similarity, we can calculate a score for each query result. Results with highest 50 scores are being returned.

1.4 Frontend

In the frontend file, we have decided to use the `flask` library to communicate with the backend search engine and display results to users. The UI is built using HTML and CSS.

1.5 Main

Every time the user runs main program, we will first delete the old database to ensure all website crawled are up-to-date. Then we will call the function `create_file_from_db`, where the database will first get initialized. Then it will recursively crawl all the files, then perform indexing. Finally it starts the web server with Flask.

A more detailed algorithm will be explained in the later part.

2 File structures used in the index database

The database totally contains 13 tables, these tables are:

1. page_info
   Contains 4 columns which are used to store basic info of the website crawled including website (page_id), size, last modification date, and also their title.

2. id_url
   Contains 2 columns, used to store the relation between a page_id and url.

3. word_id_word
   Contains 2 columns, used to store the relation between a word_id and word.

4. relation
   Contains 2 columns, used to store the relation between the child and their parents.

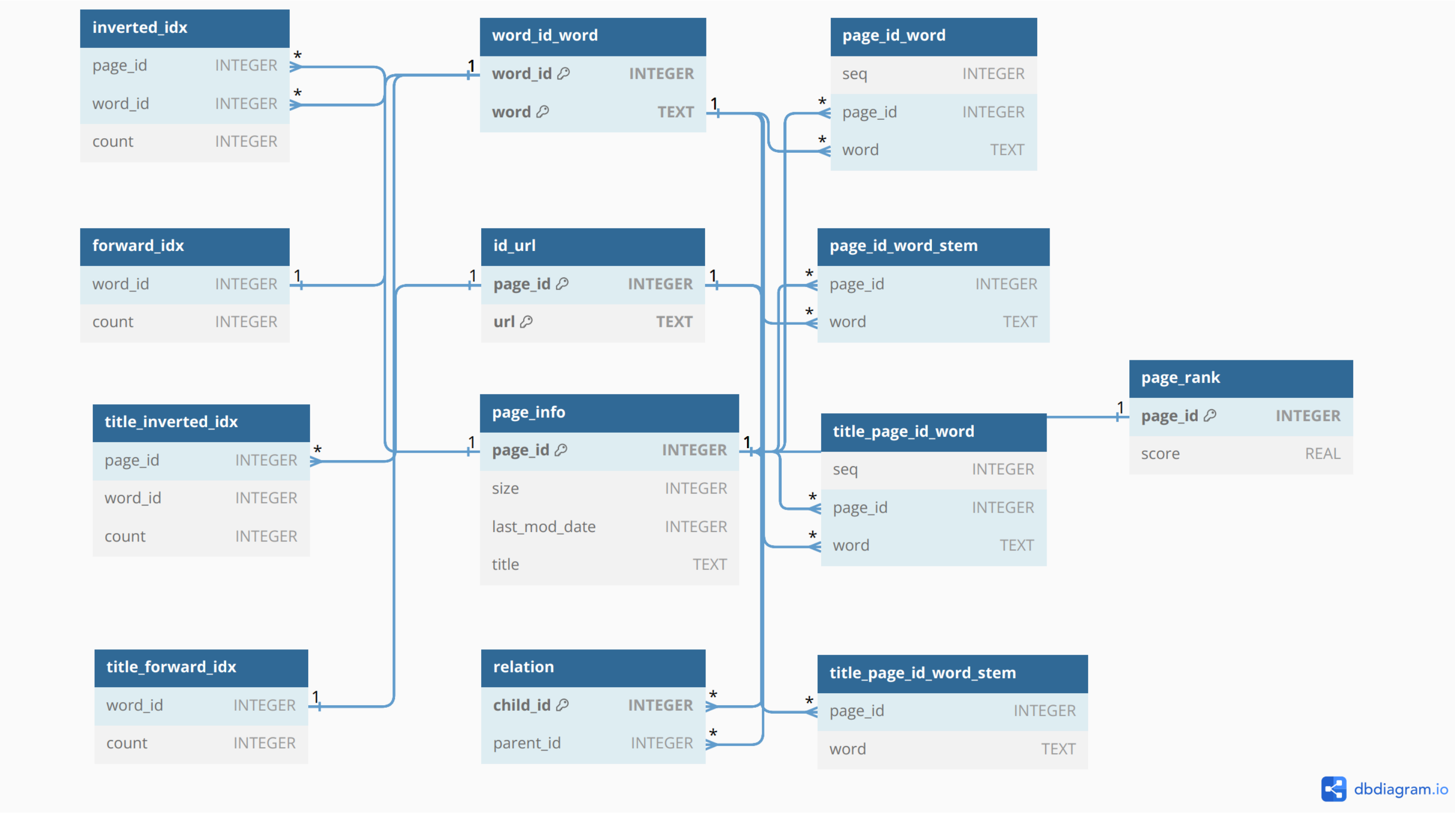
5. 4 tables of (title_)page_id_word(_stem)
   Contains 2 columns, used to store all the words in each page_id. Those tables with (title_) only contains words in the title, otherwise only contains words in the body. Those with (_stem) contains the word that have been stemmed.

6. 2 tables of (title_)inverted_idx
   Contains 3 columns, used for backward indexing table. All the word stored in the table are stemmed. Those tables with (title_) only contains words in the title, otherwise only contains words in the body.

7. 2 tables of (title_)forward_idx
   Contains 2 columns, used for forward indexing table. All the word stored in the table are stemmed. Those tables with (title_) only contains words in the title, otherwise only contains words in the body.

8. page_rank
   Contains 2 columns, used for storing the pageRank for each website.

For more details about the table, one may refer to following table.



inverted_idx	
page_id	INTEGER
word_id	INTEGER
count	INTEGER

forward_idx	
word_id	INTEGER
count	INTEGER

title_inverted_idx	
page_id	INTEGER
word_id	INTEGER
count	INTEGER

title_forward_idx	
word_id	INTEGER
count	INTEGER

word_id_word	
word_id	INTEGER
word	TEXT

id_url	
page_id	INTEGER
url	TEXT

page_info	
page_id	INTEGER
size	INTEGER
last_mod_date	INTEGER
title	TEXT

relation	
child_id	INTEGER
parent_id	INTEGER

page_id_word	
seq	INTEGER
page_id	INTEGER
word	TEXT

page_id_word_stem	
page_id	INTEGER
word	TEXT

title_page_id_word	
seq	INTEGER
page_id	INTEGER
word	TEXT

title_page_id_word_stem	
page_id	INTEGER
word	TEXT

page_rank	
page_id	INTEGER
score	REAL

3 Algorithms used

3.1 Crawler

In the crawler part, the algorithm Breadth-First Search (BFS) is being used to crawl the website to ensure the websites are visited in the order of depth. In this part of program, a list (Python array) `queue` is used as a queue of BFS. Furthermore, another list called `visited` is created to store all the visited sub-pages to prevent crawling into cyclic links repeatedly. The crawler begins, by putting the root website inside the tree. Then it pops out

the root website, fetch all the info of the site, and most importantly, crawl all the sub-links in such website. Then the crawler will put all the sub-links into the `queue`, while the root website is being put in `visited`. Then followed by the First-In-First-Out (FIFO) Principle, the first link being put into the queue is being popped out. The process is being repeated with a while loop, until the queue is eventually empty, or we have reached the required number of pages.

We have handled different kinds of errors when crawling the website, such as when the website is not accessible, or accessing such website requires more than 15 seconds, we will skip crawling such website. However, such website will still be included in parent-children relationship to accurately show their relationship.

To increase the efficiency for crawling the pages, asynchronous I/O have been applied so that the crawler can be run without the need of waiting for previous website finishes crawling.

3.2 Retrieval Function

For the retrieval function, we will first convert the document into vector. For each word j in the document i, we convert them into term weight w_{ij} by the following formula:

$$w_{ij} = \frac{\mathrm{tf}_{ij} \times \mathrm{idf}_j}{\max(\mathrm{tf})_{ij}}$$

and idf can be found by the following formula:

$$\mathrm{idf}_j = \log_2\left(\frac{\mathrm{N}}{\mathrm{df}_j}\right)$$

Here, tf_{ij} is the frequency of term j in document i. The frequency is collected from the inverted index table that we have created in the earlier time, while $\max(\mathrm{tf})_{ij}$ is by selecting the maximum frequency from the inverted index table. N is the number of documents in collection. It can be calculated by counting the unique documents i stored in the table `id_url`. Finally, df_j is being calculated by counting number of documents i that contains the word j with the help of the forward index table.

Note that the process is done for both title and body for each document. Also for user query, we will also convert them into vector. However, preprocessing have to be done. This includes split the query into 2 parts, which are single words and phrases, then for each part, we will remove all the stop words, then stem the remaining words.

Since we want to treat each word which is not a stop-word with the same weight in the query, we create a vector using the count of the query to show importance of those word (ie. the more a non-stop word is repeated, the more important it is compared to the other non-stop words in the query.). The (cosine) similarity for the document vector D_i and query vector Q is calculated with the following formula:

$$\mathrm{CosSim}(D_i, Q) = \frac{D_i \circ Q}{|D_i||Q|} = \frac{\sum_{k=1}^{t} d_{ik} q_k}{\sqrt{\sum_{k=1}^{t} d_{ik}^2}\sqrt{\sum_{k=1}^{t} q_{k_{t_2}}^2}}$$

To increase the efficiency of the program, we will perform reduction to the vector, where all the terms where both vectors does not exists were being removed.

Also, to ensure that phrases search are supported, we have implemented an algorithm, which will search for the phrase from the table `(title_)page_id_word_stem`. If such phrase required is not found, then such result will not be further considered.

The score for each document vector d, where (d_t, d_b) are title and body, of a given query vector Q, is calculated with the following formula:

$$(\alpha \times \text{CosSim}(d_t, Q) + \beta \times \text{CosSim}(d_b, Q)) \times \text{PageRank}$$

we will introduce α, β and PageRank variable in the later time.

3.3 Algorithm favouring title match

To favour the title match, we have given a higher weighting to the title matches. Define $\alpha, \beta \in \mathbb{R}$, where:

$$\alpha + \beta = 1$$

such that α is the weighting given to the title, while β is the weighting given to the body text. Such weight is being multiplied to the cosine similarity that is being calculated before.

In this project, we have set $\alpha = 0.75$, and hence $\beta = 0.25$. This implies every query, the title will be given a much higher weighting, thus favouring the title match.

3.4 Pagerank algorithm

The page rank algorithm applied in this project is the version taught in the lectures. We choose to implement the "synchronous iteration" version of it because the information sent to the page rank function will be the latest version. Therefore there is no need to over-complicate the calculation of page rank function. Our implementation

requires these specific values:

- "startingPageRank" matrix, this is just to give the flexibility if in case we do not want to give all the documents the page rank an initial score of 1
- "adjacencyMatrix" matrix is used to show how the documents are linked to each other.

  The following matrix is a $n \times n$ matrix, where n is the number of documents in the database. The matrix's row represent the child document and the matrix's column represent the parent document.

  each entry in the matrix has two possible values, 0 or 1, where 0 represents no link between the two documents and 1 represents a child-parent relationship between the document in the row and the document in the column

  The values of these are calculated using the `relation` table in our database. When looping through all the documents in the database, we get all of their child pages and set the column values to 1 for that particular row. All of the tracking of parent and child pages are done through a 2D-python dictionary, and when the values are calculated, only the values of dictionary is returned in the form of a 2D-numpy array
- the `teleportation_probability` (equivalent to d), to calculate the page rank
- `max_iterations` to set the limit for the number of iterations

The calculation of the page rank is done by loop through these calculation until there is a convergence(we used `numpy.allclose(array1, array2)` for this) or the `max_iterations` has been reached

Here is the pseduo-code for the page rank calculation algorithm

```
while there is convergence or max iterations has been reached:
    for each page:
        pageParents <- childParent[page]
        value <- pageParent.dot(pageRankScores)
        newPageRank[page] <- d + (1-d)*value
```

while there is convergence or max iterations has been reached:

- for each page
    - we get the array of child-parent for that particular array.
    - then there is a dot product between the array of child-parent and the old page rank scores.
    - new page rank score for that document is then calculated to be $d + (1 - d) \cdot (\text{value calculated for that document})$

After all of these have been executed, the value is stored in a dictionary with the document as the key, and then is committed to the database for later use.

4 Installation Procedure

Users are suggested to run the program in macOS or Linux, with `bash` or `zsh` as the shell since the coding and tests are performed on Unix-based platform. However, they can also run the program on Windows.

The steps to run the test program are as follows.

1. Confirm that the Python version is correct.

2. Confirm that the working directory is correct. The correct directory should contains `main.py` file.

3. Create a virtual environment (venv).

```
python -m venv .venv
```

4. Activate the virtual environment using the following command.

```
(Unix-based OS)
source .venv/bin/activate
(Windows) (Run in Command Prompt instead of PowerShell)
.\.venv\bin\activate
```

5. Install the required packages.

```
pip install -r requirements.txt
```

6. Run the program. The program shall do everything, including initializing databases.

```
python main.py
```

7. Now you may see a line that looks like

```
* Running on http://127.0.0.1:5000
```

Simply go to http://127.0.0.1:5000, and you can see the search engine there.

5 Bonus features

In this section, we will describe our project's features that are beyond the required specification.

5.1 User-friendly interface

The frontend utilizes the `flask` library to communicate user query to the search engine and dynamically load search results into the user interface.

The user interface is designed with HTML and CSS. It neatly arranges the most important functions and information front and center, like the search bar. Parent and child links are collapsed to avoid any result burying others below it. Each most frequent stemmed keyword and its frequency is displayed individually in a box.

5.2 Keeping track of query history

The frontend makes use of browser cookies to store a user's search history. Users can open the search history drop-down menu to quickly access recent queries. They can select a query in the history to view its results.

Note: Extremely long queries (e.g. 25165 words) will be included in the search history.

5.3 Considering links in result ranking

The search engine makes use of the PageRank algorithm to sort results. Its implementation is detailed in section 3.4.

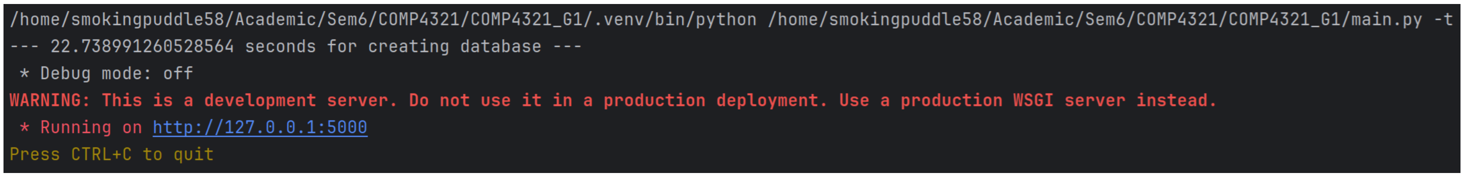
5.4 Searching speed

The search engine is exceedingly fast, which is demonstrated in the searching times recorded during testing.

- Short word (`apple`): 0.0017339000005449634 s
- Short phrase (`"Hong Kong" University of Science and Technology`): 0.016774300000017921 s
- Very long query (25165 words): 0.119122394000442 s
  the text used to test

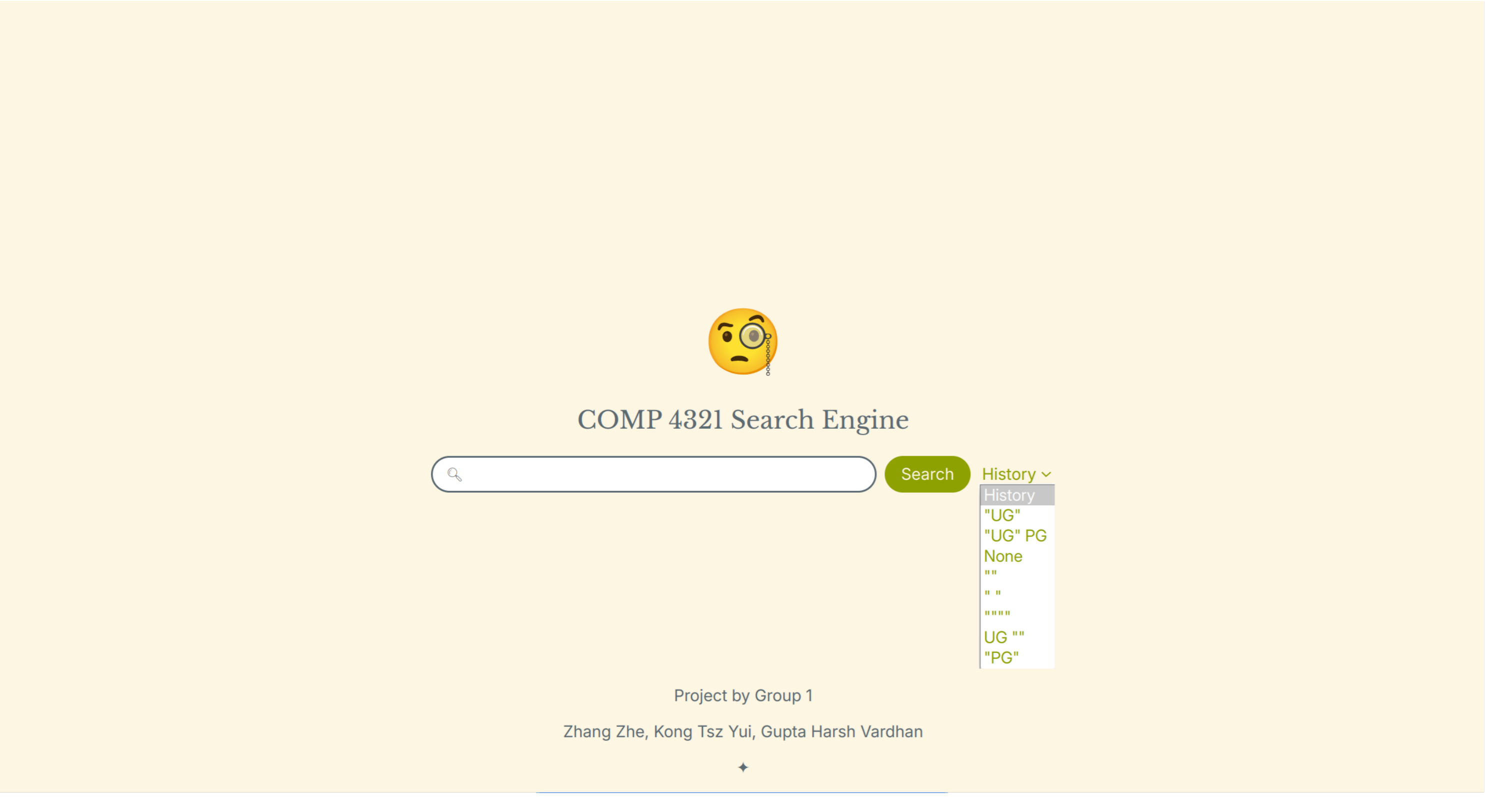
6 Testing of the features

For this project, we have tested the function by a simple test, where we start with a brand new environment. After that, we will follow the procedures listed in the `README.md`. Then we will test the correctness by searching with different variety of words, including short phrases, long phrases and a extremely long query, as shown in the above part. Moreover, to ensure the search engine is robust, we have also tested our code with some edge cases, including empty phrases, empty queries, and also some randomly-generated strings. If all of the outputs gives an reasonable results, then we can conclude that the parts, including crawler, indexer, retriever and also frontend have been functioning normally. More details of how we performed the test can be found in the image below, and also in the video. Link to the video



```
/home/smokingpuddle58/Academic/Sem6/COMP4321/COMP4321_G1/.venv/bin/python /home/smokingpuddle58/Academic/Sem6/COMP4321/COMP4321_G1/main.py -t
--- 22.738991260528564 seconds for creating database ---
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

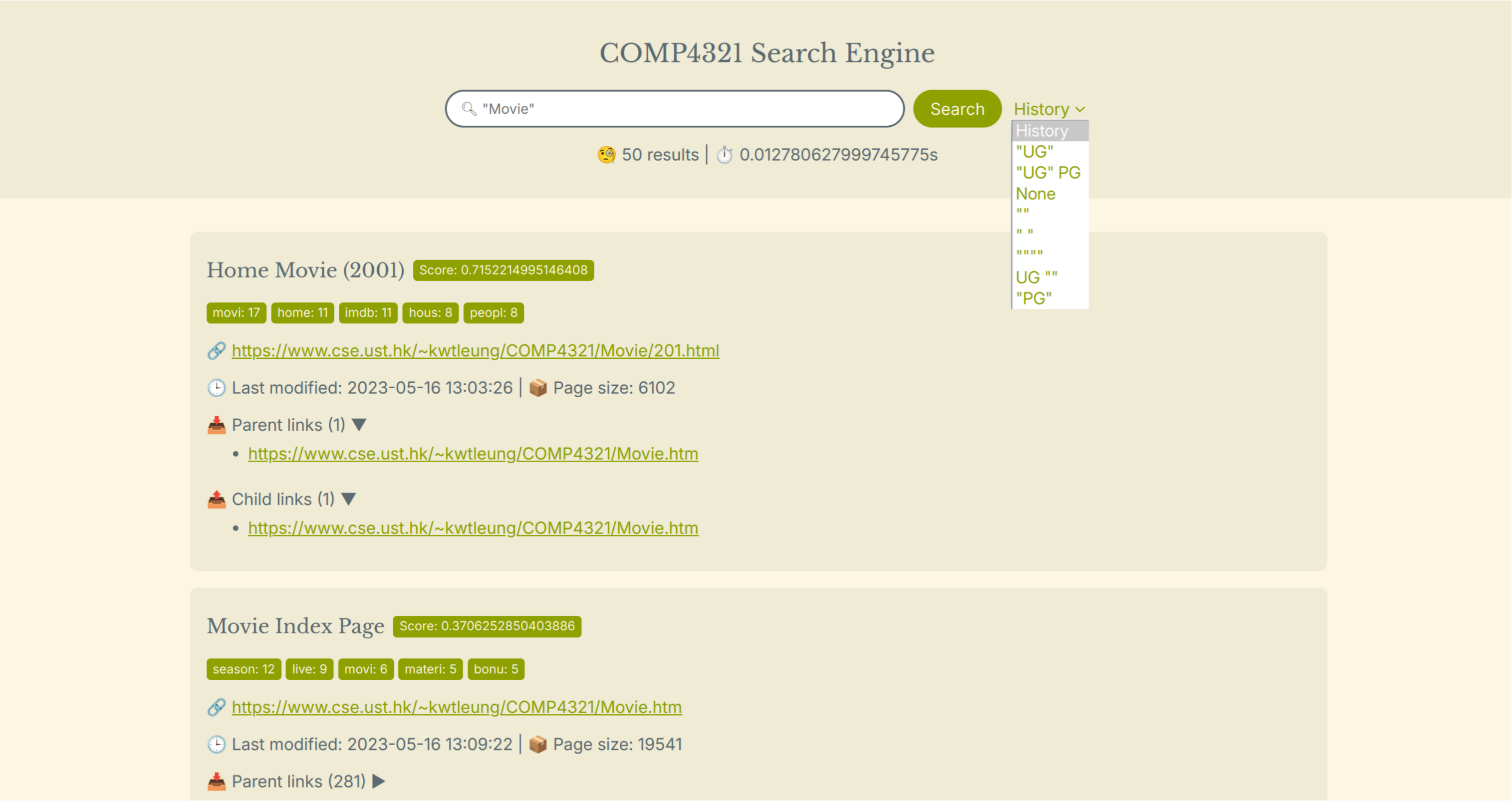
Figure 1: Finished creating database


COMP 4321 Search Engine

Project by Group 1

Zhang Zhe, Kong Tsz Yui, Gupta Harsh Vardhan

Figure 2: UI of the index page


COMP4321 Search Engine

"Movie"

🧐 50 results | ⏱️ 0.012780627999745775s

Home Movie (2001) Score: 0.7152214995146408

movi: 17 home: 11 imdb: 11 hous: 8 peopl: 8

🔗 https://www.cse.ust.hk/~kwtleung/COMP4321/Movie/201.html

🕒 Last modified: 2023-05-16 13:03:26 | 📦 Page size: 6102

📥 Parent links (1) ▼

- https://www.cse.ust.hk/~kwtleung/COMP4321/Movie.htm

📤 Child links (1) ▼

- https://www.cse.ust.hk/~kwtleung/COMP4321/Movie.htm

Movie Index Page Score: 0.3706252850403886

season: 12 live: 9 movi: 6 materi: 5 bonu: 5

🔗 https://www.cse.ust.hk/~kwtleung/COMP4321/Movie.htm

🕒 Last modified: 2023-05-16 13:09:22 | 📦 Page size: 19541

📥 Parent links (281) ▶

Figure 3: UI of the query result and the History feature

7 Conclusion

In conclusion, our search engine runs in a exceedingly fast speed with some algorithms applied that we have stated in the above part, also we have tried to make our search engine more user-friendly, and also more similar to a modern commercial search engines by implementing more features and an improved user interface design, such as keeping the user history. The database has also be designed in a manner such that we only need minimal database querying, and to provide the basics for the fast querying time of the search engine.

However, due to the time limitation, we are unable to implement an algorithm that can read the data from the database, and dynamically update the database. So we need to delete the database everytime `main.py` is being executed. This will incur a 20-40 seconds of waiting time before the search engine can actually be used.

If our group had a chance to re-implement the search engine, we will first re-implement the crawler so that it can dynamically update the data only when the page has a newer modification date. Also we can try to implement more interesting feature such as considering adding the features of "get similar pages", and also to autocorrect the typos in user queries.

8 Contribution

In this project, our groupmates work with each other together. Here is the contribution made by each groupmate.

	Contribution Percentage	Parts Responsible
Zhang Zhe	33.333%	Indexer, Crawler, Report, Video
Kong Tsz Yui	33.333%	Frontend, Report
Gupta Harsh Vardhan	33.333%	Retriever, Report

— END OF COMP4321 GROUP 1 REPORT —