COMP4321 Final Report

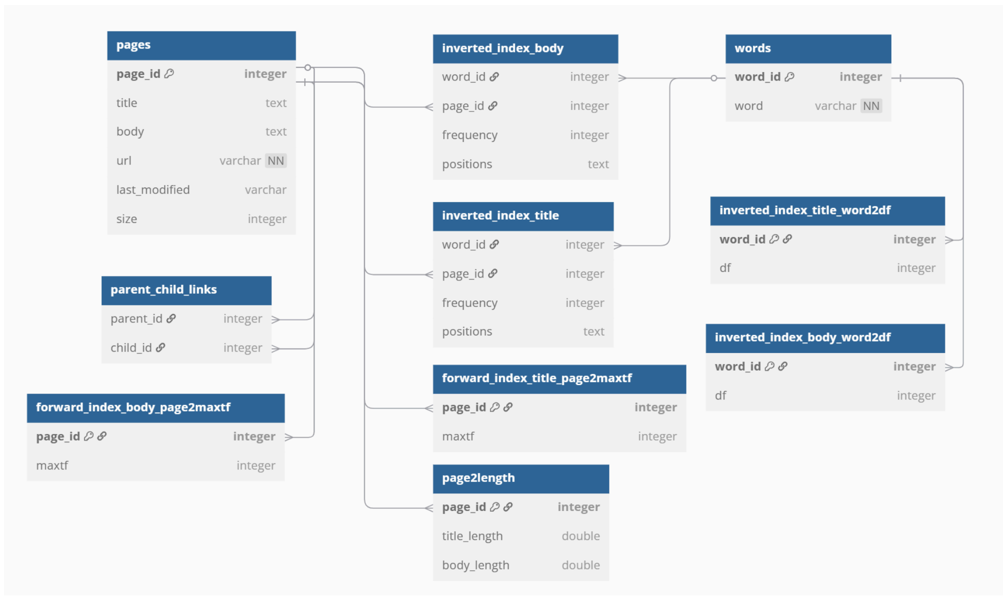
Member:
ABDULLAH Imtiaz
SI TOU Tin Nok Abraham
Li Xinwei Tina

Overall design of the system

This project implements a modular web search engine with four main components: a crawler, a database, a search engine, and a web interface. The goal is to allow users to search for information across crawled web pages efficiently and effectively.

1. **Crawler**
    - **Purpose**: Automatically visits web pages starting from a specified URL, following links to discover new pages.
    - **How it works**: Uses a *breadth-first search (BFS)* strategy to traverse the web, fetches and processes HTML content, extracts important words (removing common "stopwords"), and records their positions for later search.
    - **Output**: Saves all collected data (content, metadata, and links between pages) into a local database.
2. **Database**
    - **Purpose**: Serves as the central storage for all crawled data.
    - **Design**: Uses SQLite with carefully designed tables for storing pages, indexed words, relationships (links), and additional statistical data needed for search and ranking.
    - **Functionality**: Supports fast lookup of word occurrences, document relationships, and enables efficient search queries.
3. **Search Engine**
    - **Purpose**: Processes user queries to find and rank relevant pages.
    - **Functionality**:
        - Supports both single-word and phrase searches.
        - Uses stemming (to handle word variations) and stopword removal for better search relevance.
        - Calculates *TF-IDF* vectors and uses *cosine similarity* to rank results.
        - Boosts matches found in page titles for improved accuracy.
    - **Output**: Returns a ranked list of search results, with relevant metadata and highlights.
4. **Web Application**
    - **Purpose**: Provides a user-friendly interface for searching and browsing results.
    - **Implementation**: Built with Flask, it handles search requests, displays paginated results, and supports exploration of related pages based on keywords.
    - **Stateless Design**: Ensures reliability by initializing a fresh crawler instance for each request.

The file structures used in the index database



Database Schema Diagram

SQL is used to build our index database. We create 10 tables in total, including 6 forward indexes and 4 inverted indexes.

Forward indexes

1. **Pages Table**
   - **Purpose:** Stores metadata for each crawled page.
   - **Columns:**
     - `page_id`: A unique identifier for each page.
     - `title`: The page title (extracted from the HTML).
     - `body`: Text content of the page (used for potential full-document display or re-indexing).
     - `url`: The unique URL of the page; this is critical for preventing duplicate fetches.
     - `last_modified`: The last modification date as obtained from the HTTP headers.
     - `size`: The size of the page in bytes.
2. **Words Table**
   - **Purpose:** Maintains a unique list of words (keywords) extracted from both page titles and bodies.
   - **Columns:**
     - `word_id`: A unique identifier for the word.
     - `word`: The keyword.
3. **Parent-Child Links Table**
   - **Purpose:**
     - Captures the hierarchical relationship among pages by linking parents with their child pages.

        - This structure is utilized both for constructing the navigation graph (or sitemap) and later for search result displays.
    - **Columns:**
        - `parent_id`: Corresponds to the parent page identifier.
        - `child_id`: Corresponds to the child page identifier.
4. **Forward Index page2maxtf Table (Body and Title)**
    - **Purpose:**
        - Record each page's highest term frequency
    - **Columns:**
        - `page_id`: A unique identifier for the page.
        - `maxtf`: The maximum frequency of any single term in this page.
5. **page2length Table**
    - **Purpose:**
        - Record each page's document length (sqrt(sigma(dik^2))) as one of the dividends for calculating cosine similarity fast.
    - **Columns:**
        - `Page_id`: A unique identifier for the page.
        - `title_length`: The page's title's length (norm)
        - `body_length`: The page's body's length (norm)

Inverted indexes

1. **Inverted Index Table (Body and Title)**
    - **Purpose:**
        - Record the pages where each word appears, along with the corresponding number of appearances and their positions.
    - **Columns:**
        - `word_id`: References the word in the Words table.
        - `page_id`: References the page where the word appears.
        - `frequency`: Count of how many times the word appears in the page.
        - `positions`: A list of positions where the word occurs in the page(for phrase queries, but doesn't implement eventually)
2. **Inverted Index word2df Table (Body and Title)**
    - **Purpose:**
        - Stores how many pages contain each word (document frequency).
    - **Columns:**
        - `word_id`: References the word in the Words table.
        - `df`: Document frequency - count of pages containing this word.

Algorithms used

1. Web Crawling (BFS Mechanism)

The crawler uses a Breadth-First Search (BFS) algorithm to traverse and collect web pages starting from a given seed URL. BFS ensures that pages closer to the starting point are visited first, which helps in capturing the local structure of the website and avoids getting stuck in deep or cyclic links.

The crawler maintains a queue of URLs to visit and a set of visited URLs to prevent revisiting the same page.

Steps:

1. Initialize a queue with the start URL.
2. While the queue is not empty and the maximum page limit is not reached:
3. Dequeue a URL, fetch its content, and extract links.
4. For each extracted link, if it hasn't been visited, enqueue it.
5. Store the page content, title, and metadata in the database.

2. Indexing (Inverted Index)

The system builds inverted indexes for both the body and title of each page. For every word (after stemming), it records:

· The pages (documents) where the word appears.
· The frequency and positions of the word in each document.
· Separate indexes are maintained for body and title to allow for differentiated scoring.

3. Query Processing

When a user submits a query, the following steps are performed:

- Tokenization and Stemming

  The query is split into single terms and phrases (text in quotes).
  All terms are lowercased and stemmed using the Porter Stemmer to improve matching.

- Phrase Matching

  For phrases, the engine checks if all words in the phrase appear consecutively in the body of a document by comparing word positions.

- Document Retrieval

  For each term and phrase, the engine retrieves the set of documents (URLs) where they appear, using the inverted indexes.

4. Ranking (TF-IDF with Title Boost)

The ranking algorithm is based on cosine similarity between the query vector and each document vector, using a TF-IDF (Term Frequency-Inverse Document Frequency) weighting scheme.

- TF-IDF Calculation
  - TF (Term Frequency): Number of times a word appears in the document (body and title).

    - IDF (Inverse Document Frequency): Calculated as log(N / df), where N is the total number of documents and df is the number of documents containing the term.
- Favoring Title Matches
    - Title matches are favored by **applying a weight multiplier (e.g., 2.0)** to term frequencies found in the title. This means that if a query term appears in the title, it contributes more to the document's score than if it appears only in the body.
- Cosine Similarity
    - Both the query and each document are represented as vectors in the term space.
    - The similarity score is the cosine of the angle between these vectors, computed as the dot product divided by the product of their norms.

$$\mathrm{CosSim}(D_i, Q) = \frac{D \circ Q}{|D||Q|}$$

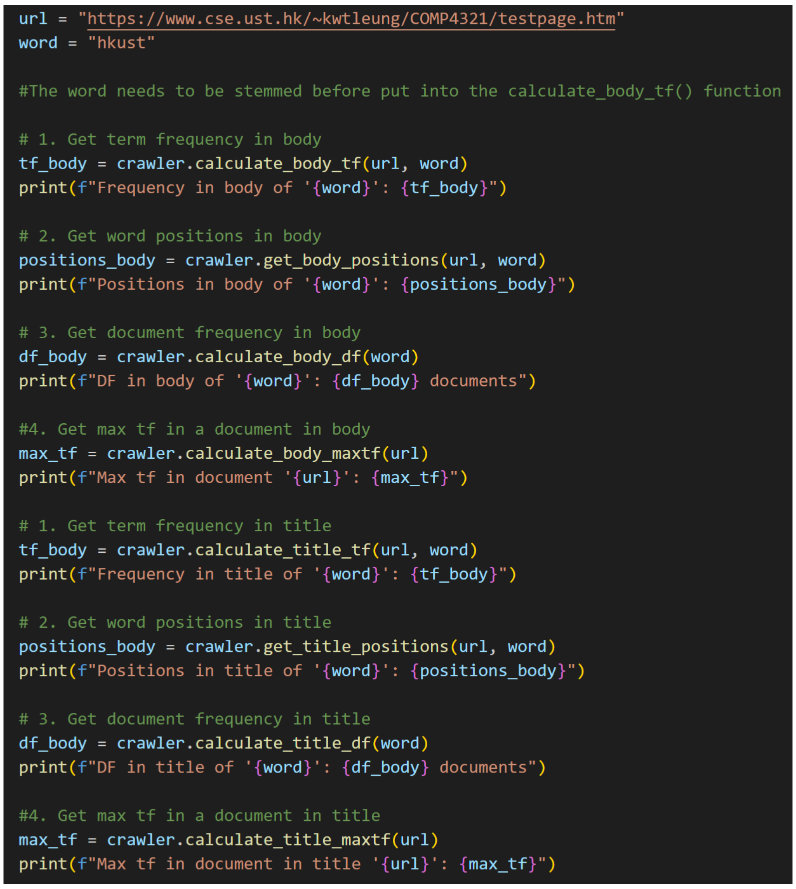
- Result Sorting
    - Documents are sorted by their similarity scores, and the top results are returned.

Installation procedure

1. “python -m venv .venv” to make a virtual environment.
2. “.venv\Scripts\Activate”
3. “pip install -r requirements.txt”
4. “python main.py” to crawl the pages, store them in the database and create spider_result.txt
5. “python app.py” . Copy the link appeared and start searching.

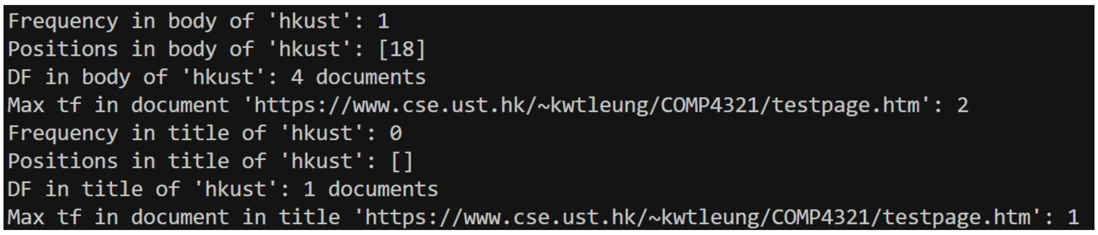
Testing of the functions implemented

We evaluate eight functions that calculate the term frequency (TF) of a word on a page, the document frequency (DF) of a word, the maximum term frequency (MaxTF) on a page, and the positions of a word within both the title and body of the page.



```python
url = "https://www.cse.ust.hk/~kwtleung/COMP4321/testpage.htm"
word = "hkust"

#The word needs to be stemmed before put into the calculate_body_tf() function

# 1. Get term frequency in body
tf_body = crawler.calculate_body_tf(url, word)
print(f"Frequency in body of '{word}': {tf_body}")

# 2. Get word positions in body
positions_body = crawler.get_body_positions(url, word)
print(f"Positions in body of '{word}': {positions_body}")

# 3. Get document frequency in body
df_body = crawler.calculate_body_df(word)
print(f"DF in body of '{word}': {df_body} documents")

#4. Get max tf in a document in body
max_tf = crawler.calculate_body_maxtf(url)
print(f"Max tf in document '{url}': {max_tf}")

# 1. Get term frequency in title
tf_body = crawler.calculate_title_tf(url, word)
print(f"Frequency in title of '{word}': {tf_body}")

# 2. Get word positions in title
positions_body = crawler.get_title_positions(url, word)
print(f"Positions in title of '{word}': {positions_body}")

# 3. Get document frequency in title
df_body = crawler.calculate_title_df(word)
print(f"DF in title of '{word}': {df_body} documents")

#4. Get max tf in a document in title
max_tf = crawler.calculate_title_maxtf(url)
print(f"Max tf in document in title '{url}': {max_tf}")
```

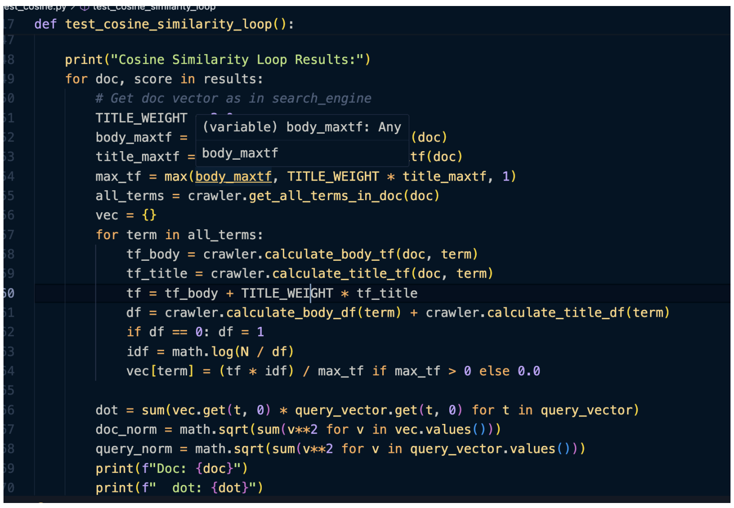
Test functions



```
Frequency in body of 'hkust': 1
Positions in body of 'hkust': [18]
DF in body of 'hkust': 4 documents
Max tf in document 'https://www.cse.ust.hk/~kwtleung/COMP4321/testpage.htm': 2
Frequency in title of 'hkust': 0
Positions in title of 'hkust': []
DF in title of 'hkust': 1 documents
Max tf in document in title 'https://www.cse.ust.hk/~kwtleung/COMP4321/testpage.htm': 1
```

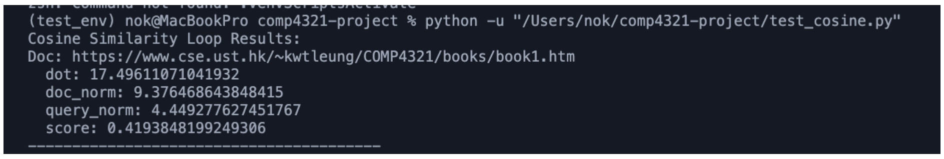
Test results

We also evaluate the cosine similarity using test_cosine.py.



```python
def test_cosine_similarity_loop():

    print("Cosine Similarity Loop Results:")
    for doc, score in results:
        # Get doc vector as in search_engine
        TITLE_WEIGHT   (variable) body_maxtf: Any
        body_maxtf =                                (doc)
        title_maxtf =  body_maxtf                   tf(doc)
        max_tf = max(body_maxtf, TITLE_WEIGHT * title_maxtf, 1)
        all_terms = crawler.get_all_terms_in_doc(doc)
        vec = {}
        for term in all_terms:
            tf_body = crawler.calculate_body_tf(doc, term)
            tf_title = crawler.calculate_title_tf(doc, term)
            tf = tf_body + TITLE_WEIGHT * tf_title
            df = crawler.calculate_body_df(term) + crawler.calculate_title_df(term)
            if df == 0: df = 1
            idf = math.log(N / df)
            vec[term] = (tf * idf) / max_tf if max_tf > 0 else 0.0

        dot = sum(vec.get(t, 0) * query_vector.get(t, 0) for t in query_vector)
        doc_norm = math.sqrt(sum(v**2 for v in vec.values()))
        query_norm = math.sqrt(sum(v**2 for v in query_vector.values()))
        print(f"Doc: {doc}")
        print(f"  dot: {dot}")
```

Test function



```
(test_env) nok@MacBookPro comp4321-project % python -u "/Users/nok/comp4321-project/test_cosine.py"
Cosine Similarity Loop Results:
Doc: https://www.cse.ust.hk/~kwtleung/COMP4321/books/book1.htm
  dot: 17.49611071041932
  doc_norm: 9.376468643848415
  query_norm: 4.449277627451767
  score: 0.4193848199249306
----------------------------------------
```

Test result

Highlight of features beyond the required specification

Beyond the required specification, we also implemented 3 bonus requirements, which are “get similar pages”, “browse and select keywords indexed” and “track of query history”.

Get similar pages

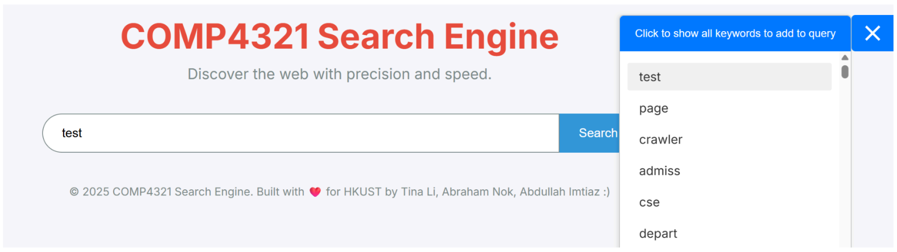
Implementation of the Relevance Feedback Feature: “Get Similar Pages”
The “get similar pages” feature provides a simple form of relevance feedback to help users discover pages related to a result they find interesting. Here’s how it works:

1. User Interaction
   - On the search results page, each result includes a “get similar pages” button.
   - When the user clicks this button for a specific page, the system triggers the relevance feedback mechanism.
2. Keyword Extraction
   - The backend extracts the top 5 most frequent keywords from the selected page.
   - This is done by querying the database for the most common (stemmed) words in both the page’s body and title, excluding stop words.
   - The function get_similar_pages_query(url) in crawler.py handles this logic:
     - It sums the frequencies of each word in the body and title.
     - It filters out stop words.
     - It returns the top 5 keywords as a list.
3. Query Rewriting
   - The original user query is replaced with a new query composed of these top 5 keywords.
   - This new query is constructed by joining the keywords with spaces.
4. Automatic Search Submission
   - The system automatically submits this revised query to the search engine.
   - The user is redirected to a new search results page showing documents similar to the selected page, based on the extracted keywords.

Browse and select keywords indexed

The “browse and select keywords” feature will allow the user to see a list of all (stemmed) keywords indexed in your database, browse through them, and select the keywords he/she is interested in, and then submit them as a vector-space query to your search engine.

1.User Interaction:
   - Browse and click any keywords listed at the right side, it will be passed to the search bar. Click the “Search” button and it will conduct the search.


COMP4321 Search Engine

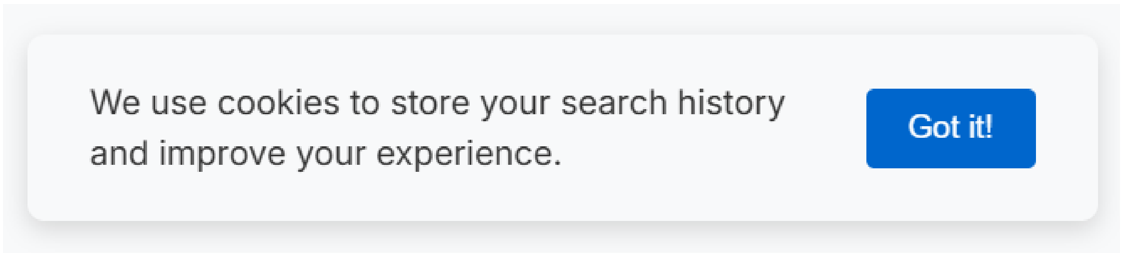
Discover the web with precision and speed.

© 2025 COMP4321 Search Engine. Built with ❤ for HKUST by Tina Li, Abraham Nok, Abdullah Imtiaz :)

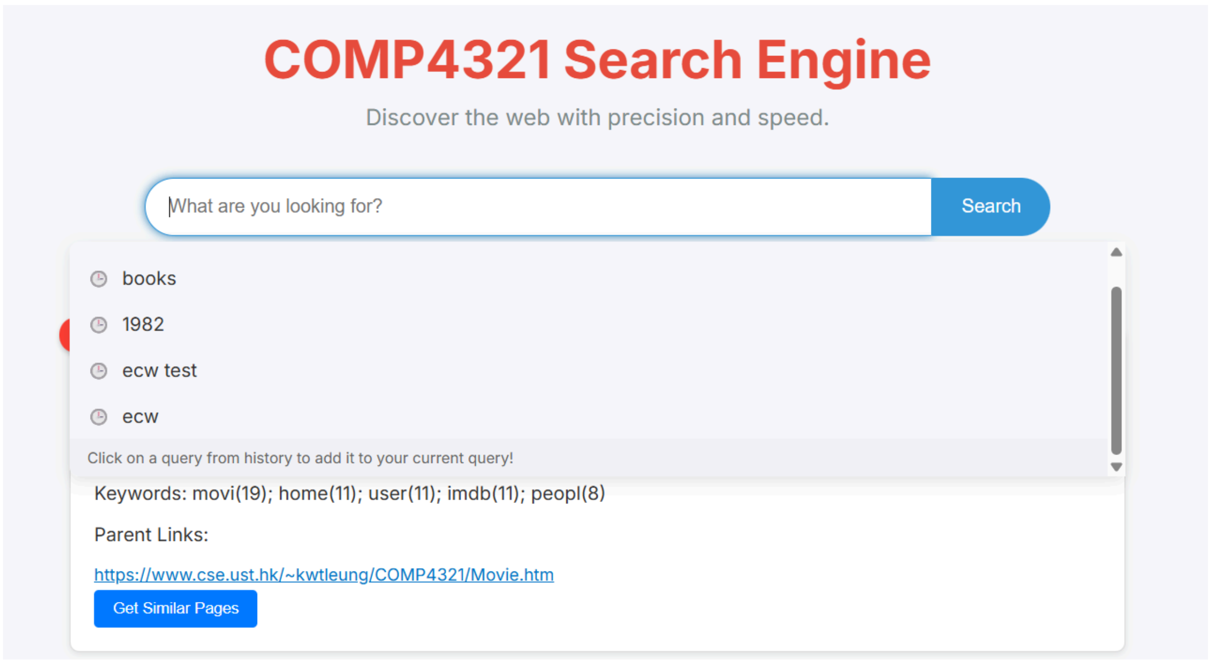
2. List keywords:
   - The “words” table contains all the stemmed words appearing in all pages. The “show_stemmed_keywords()” function will return a list of words indexed in the “words” database and the “get_keywords()” function will display them in the frontend.

Track of query history

We use browser cookies to store a list of previously searched queries in our user’s browser for their future reference. When a user first accesses our web page, they get notified of the usage of our cookies.


We use cookies to store your search history and improve your experience.

Thereafter, whenever users search a query, it is stored and shown to users when they select the search bar to search their query as a dropdown list.


COMP4321 Search Engine

Discover the web with precision and speed.

Click on a query from history to add it to your current query!

Keywords: movi(19); home(11); user(11); imdb(11); peopl(8)

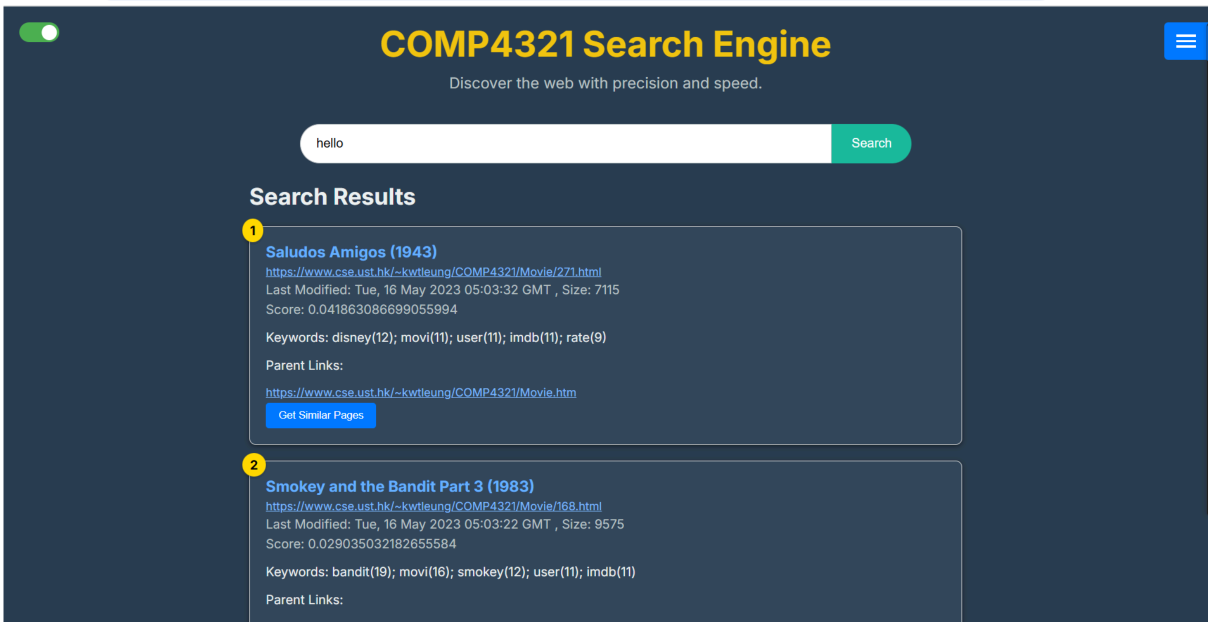
Parent Links:

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

Moreover, users can easily add previously searched queries into their current query by clicking any item stored in the history. We included other user-friendly features such as scrolling within the query history dropdown list and also included a message so users can know the query appension feature.

Light/Dark Mode

We included a switch button in the top-left corner of the webpage to allow users to toggle between light and dark mode. This is to accommodate the modern user interface expectations to cater to people with different preferences of color schemes.



Search Results

1
Saludos Amigos (1943)
https://www.cse.ust.hk/~kwtleung/COMP4321/Movie/271.html
Last Modified: Tue, 16 May 2023 05:03:32 GMT , Size: 7115
Score: 0.041863086699055994

Keywords: disney(12); movi(11); user(11); imdb(11); rate(9)

Parent Links:

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

2
Smokey and the Bandit Part 3 (1983)
https://www.cse.ust.hk/~kwtleung/COMP4321/Movie/168.html
Last Modified: Tue, 16 May 2023 05:03:22 GMT , Size: 9575
Score: 0.029035032182655584

Keywords: bandit(19); movi(16); smokey(12); user(11); imdb(11)

Parent Links:

We allow users to freely toggle it anytime they please. User’s choices are saved throughout their browser session.

Conclusion

Strengths:

- **Efficient Indexing:** By implementing inverted indexes and forward indexes, we can efficiently retrieve the term frequency (TF), document frequency (DF), and maximum term frequency (MaxTF) for each query word without the need to re-scan entire pages, resulting in significant time savings.
- **User Friendly Interface**: The interface allows the users to browse available keywords, find similar pages and store search history. It’s easy and efficient to use these features.

Weaknesses:

- **SQLite Limitations:** Using SQLite simplifies deployment but lacks advanced database optimizations (e.g., distributed indexing) compared to systems like Elasticsearch.

Improvements:

- Add Interesting Features: We stored the positions of the words in each page, if we have more time, we can implement the “phrase search” feature. We can also consider links in result ranking such as the “Pagerank” feature.
- If we can re-implement it, we would like to merge some of the indexes to save the memory overhead.

Contribution:

- ABDULLAH Imtiaz (33.3%, Frontend and cookies, ⅓ of the report)
- SI TOU Tin Nok Abraham (33.3%, Search engine, ⅓ of the report)
- Li Xinwei Tina (33.3%, Index building, ⅓ of the report)