**COMP4321 Search Engine**

**Project Report**

Chan Tony Yuen Yeung, 20864385

Cheng Yan Hei, 20865872

WU, Man Yui, 20769755

Date: 03/05/2024

COMP4321 Search Engine

[Overall Design of the System 3](#_Toc166088154)

[Folder Structure 3](#_Toc166088155)

[Backend: Web Crawler and Indexer (Java) 3](#_Toc166088156)

[Frontend: Search Engine (Apache Tomcat) 3](#_Toc166088157)

[File Structures Used in the Index Database 4](#_Toc166088158)

[Database Design 4](#_Toc166088159)

[Algorithms Used 6](#_Toc166088160)

[Result Ranking 6](#_Toc166088161)

[Breadth-First Search (BFS) 6](#_Toc166088162)

[Text Processing (Stop Stem and N-gram) 6](#_Toc166088163)

[Installation Procedure 7](#_Toc166088164)

[Prerequisites 7](#_Toc166088165)

[Building the Project 7](#_Toc166088166)

[Highlights of Features Beyond the Required Specification 8](#_Toc166088167)

[Additional Functionalities 8](#_Toc166088168)

[User Experience Enhancements 9](#_Toc166088169)

[Testing of the Functions Implemented 10](#_Toc166088170)

[Conclusion 11](#_Toc166088171)

[Strengths 11](#_Toc166088172)

[Weaknesses 11](#_Toc166088173)

[Re-Implementation 12](#_Toc166088174)

[Interesting Features 12](#_Toc166088175)

[Contribution 12](#_Toc166088176)

# Overall Design of the System

## Folder Structure

The project is organized into the following directories:

1. **data**: Contains the indexed database stored in the "database.db" file, utilizing the Java JDBM library. Additionally, it houses project documentation such as the Database Design document and the "stopwords.txt" file.
2. **docs**: Holds project documentation, including design documents and guides.
3. **lib**: Stores third-party libraries used in the project.
4. **src**: Divided into backend and frontend components, where backend code handles web crawling and indexing, while frontend code encompasses the search engine interface implemented using Apache Tomcat.

## Backend: Web Crawler and Indexer (Java)

The backend of the system is implemented in Java, utilizing the following components:

1. **IRUtilities**: Package containing the Porter algorithm used for word stemming.
2. **StopStem**: Class responsible for stop word removal and word stemming.
3. **Spider**: Class used for crawling web pages and extracting basic information such as headers and body content.
4. **Indexer**: Class responsible for handling words, including calculating word frequency, and managing the indexed database.
5. **CrawlandIndex**: Central function that combines the web crawler and indexer to process page information.
6. **TestProgram**: Main testing program that controls the "CrawlandIndex" function and manages the database location, including copying it to the frontend directory. (You can set the maximum number of pages and the starting website here)

## Frontend: Search Engine (Apache Tomcat)

The frontend of the system is implemented using Apache Tomcat, with the following components:

1. **apis**: Contains the "getData.jsp" file, responsible for handling input from the search engine, cleaning, stopping, and stemming the input. It retrieves relevant indexed words from the database, calculates final frequencies, and returns a list of pages with detailed information and sorted page IDs.
2. **html**: Holds frontend HTML elements used in the search engine page.
3. **images**: Stores the rectangular logo of the search engine.
4. **utils**: Contains JavaScript code responsible for handling button click events, tab navigation, and generating query results.
5. **WEB-INF**: Includes static materials used in the search engine result:
   1. **classes**: Holds the Porter algorithm for stopping and stemming the input.
   2. **databases**: Contains the "database.db" file for storing indexed results, which is the same as the one used in the backend.
   3. **lib**: Stores libraries used in the frontend, mirroring those used in the backend.
6. **index.jsp**: The main page of the frontend, containing code for the search engine interface.
7. **styles.css**: CSS styling code used to style the frontend interface.

# File Structures Used in the Index Database

The index database utilizes the jdbm1.0 storage mechanism, storing data in a (Key, Value) format, with "|" and "," used as separators within values. A single database file is used, as all eight databases are included as objects within a single "Record Manager."

## Database Design

We use true data types in the database design but implement keys and values as String types for simplicity. Here's the database design:

**PageInfo**  
Description: Contains information about each page.

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Description** |
| PageID | Integer | Unique key ID for pages |
| PageTitle | String | Title of the page |
| URL | String | URL of the page |
| LastModificationDate | String | Date of the last modification |
| SizeofPage | Integer | Size of the page |
| MaxFreq | Int | Maximum Term Frequency |
| pageMag | Double | Page Magnitude |

Data:

|  |  |
| --- | --- |
| 1 | COMP4321 Project Website|[www.testing.net](http://www.testing.net)|11/03/2024|96|20|32.5 |

**PageURLMapping**  
Description: Maps URL to their respective PageID.

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Description** |
| URL | String | URL of the page |
| PageID | Integer | Unique key ID for pages |

Data:

|  |  |
| --- | --- |
| [www.testing.net](http://www.testing.net) | 1 |

**PageParentMapping**Description: Maps child links to their respective parent pages.

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Description** |
| PageID | Integer | ID of the parent page |
| ParentID | Integer | ID of the parent pages |

Data:

|  |  |
| --- | --- |
| 3 | 1, 2 |

**PageChildMapping**  
Description: Maps child links to their respective parent pages.

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Description** |
| PageID | Integer | ID of the parent page |
| ChildID | Integer | ID of the child pages |

Data:

|  |  |
| --- | --- |
| 1 | 2, 3 |

**InvertedBodyWord**  
Description: Stores inverted index for words found in the body of pages.

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Description** |
| WordID | Integer | Unique identifier for body words |
| PageID | Integer | ID of the page containing the word |
| Frequency | Integer | Frequency of the word in the body |
| TFIDF/Max | Double | TF-IDF/Max value of the word |

Data:

|  |  |
| --- | --- |
| 1 | 1|5|2.1, 2|4|1.8 |

**BodyWordMapping**  
Description: Maps body words to the pages they appear in.

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Description** |
| PageID | Integer | ID of the page |
| WordID | Integer | ID of the body word |
| Frequncy | Integer | Frequency of the word |
| Word | String | The Word |

Data:

|  |  |
| --- | --- |
| 1 | 1|3|apple, 3|4|egg, 6|5|test, |

**InvertedTitleWord**  
Description: Stores inverted index for words found in the title of pages.

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Description** |
| TitleWordID | Integer | Unique identifier for title words |
| PageID | Integer | ID of the page containing the word |
| Frequency | Integer | Frequency of the word in the title |
| TFIDF | Double | TF-IDF value of the word |

Data:

|  |  |
| --- | --- |
| 3 | 1|5|2.1, 2|4|1.8 |

**WordMapping**  
Description: Maps body word IDs to their respective words.

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Description** |
| Word | String | Body word |
| WordID | Integer | ID of the body word |

Data:

|  |  |
| --- | --- |
| Apple | 1 |

# Algorithms Used

## Result Ranking

To rank search results, we employ a weighted sum approach. We calculate the cosine similarity of both the heading and the body of each page, assigning a weight of 25 to the heading and 10 to the body. This weighted sum helps prioritize pages that closely match the search query in both their heading and body content.

## Breadth-First Search (BFS)

For web crawling and indexing, we utilize a Breadth-First Search algorithm. Starting from the parent node, we systematically discover new child nodes and add them to a queue if they haven't been visited before. We maintain a separate array to track visited nodes. Once all information from the current node is indexed, we visit the first node in the queue, repeating this process until there are no more nodes in the queue or the maximum number of visits is reached (in our project, set to 300 pages).

## Text Processing (Stop Stem and N-gram)

In text processing, we implement stop word removal, word stemming, and extraction of n-grams. Using the HTML parser introduced in our lab, we extract words from the body of each page. The “extractWords” function in Spider.java facilitates this process. We utilize a combination of “String Bean” and “String Tokenizer” to create a moving window of three words (i.e. this, next and next\_2). By applying “nextToken” and “hasMoreTokens”, we generate combinations of stemmed words to be indexed, including 2-grams and 3-grams. Additionally, we leverage “prev” and “prev\_2” to capture all possible positions of the words within the text.

# Installation Procedure

## Prerequisites

OpenJDK version: 21.0.2

## Building the Project

To compile each Java file, follow the commands below (Please make sure you are in the src directory):

1. **StopStem**:

|  |
| --- |
| javac "StopStem.java" |

1. **Indexer**:

|  |
| --- |
| javac -cp "../lib/jdbm-1.0.jar;." "Indexer.java" |

1. **Spider**:

|  |
| --- |
| javac -cp "../lib/htmlparser.jar;." "Spider.java" |

1. **CrawlandIndex**:

|  |
| --- |
| javac -cp "../lib/htmlparser.jar;../lib/jdbm-1.0.jar;." "CrawlandIndex.java" |

1. **TestProgram**:

|  |
| --- |
| javac "TestProgram.java" |

**Executing the Program:**  
After compiling the Java files, execute the TestProgram to initiate the web crawling and indexing process using the following command:

|  |
| --- |
| java -cp "../lib/htmlparser.jar;../lib/jdbm-1.0.jar;." "TestProgram" |

After executing the TestProgram, the following files will be generated:

|  |
| --- |
| "database.db" in the backend folder, "../data/" and in the frontend folder, "./apache-tomcat-10.1.20/webapps/comp4321/WEB-INF/database/". |

(The database.db should be copied to the frontend folder automatically by the TestProgram. If not, please copy it manually for the frontend to work properly.)

**Running the Frontend:**  
Before running the frontend, set up the system environment variables for Apache Tomcat as follows:

|  |  |
| --- | --- |
| **KEY** | **VALUE** |
| CATALINA\_HOME | {path to this project}\COMP4321-Crawler\src\apache-tomcat-10.1.20 |
| JAVA\_HOME | {path to your JDK}  (e.g. "C:\Program Files\Eclipse Adoptium\jdk-21.0.2.13-hotspot") |

After setting up the system environment variables, start the Apache Tomcat server by running the following command (Please make sure you are in the src directory):

|  |
| --- |
| apache-tomcat-10.1.20/bin/startup.bat |

Then, open a web browser and navigate to the following URL to access the frontend: <http://localhost:8080/comp4321/>

# Highlights of Features Beyond the Required Specification

## Additional Functionalities

1. Operation on Search Query Result   
   A group of rectangular boxes with text

   Description automatically generated

Query History

Remove Specific Item

The user's search results can be conveniently stored in local storage, which remain in the browser after the session ends, allowing them to preview and perform multiple AND/OR operations to filter the results.

The resulting content is visually labelled with different colors to indicate which operation they belong to.

Additionally, users have the ability to search within the filtered content.

A search sequence is also shown to aid users in keeping track of those operations applied.

Note: Click “Remove All” to remove those query results from your local storage.

A screenshot of a computer screen

Description automatically generated 

1. Keywords Table

A screenshot of a phone

Description automatically generatedThe keywords table provides a list of stemmed keywords with the highest frequency or TFIDF scores and users then submit these keywords directly into the query.

Besides, users can also modify the number of keywords displayed according to their specific needs.

1. Get Similar Pages Button

The “Get Similar Pages” button is attached to each of the resulted page, it automatically submits the most frequent word found within that specific page as a new query.

## User Experience Enhancements

Building upon the strengths discussed in the Conclusion section, our system offers several user experience enhancements beyond the required specifications:

### 1. Smooth Searching Experience

**Efficient Query Processing**: Users benefit from rapid query processing, ensuring quick retrieval of relevant search results and a seamless searching experience.

### 2. Short Scrolling Area

**Tabbed Navigation**: Organizing search results into tabs with a limited number of results per tab minimizes scrolling, improving user efficiency and satisfaction.

**Optimized Display**: Each tab presents a concise list of results, enabling users to quickly locate desired information without excessive scrolling.

### 3. Simplistic Style

**Clean and Organized Design**: The simplistic and organized design promotes focus and clarity, with streamlined elements and minimal distractions enhancing user comprehension.

### 4. Contrast on Important Features

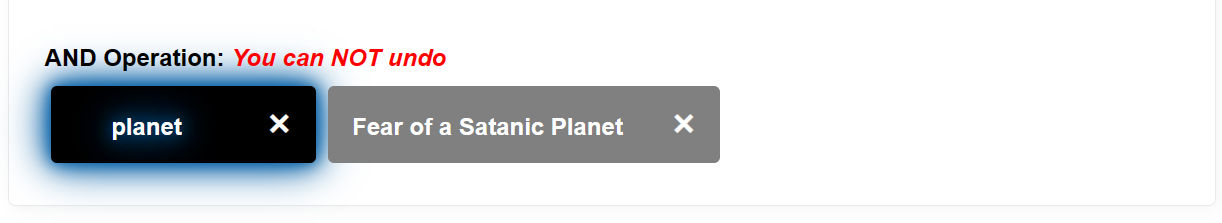
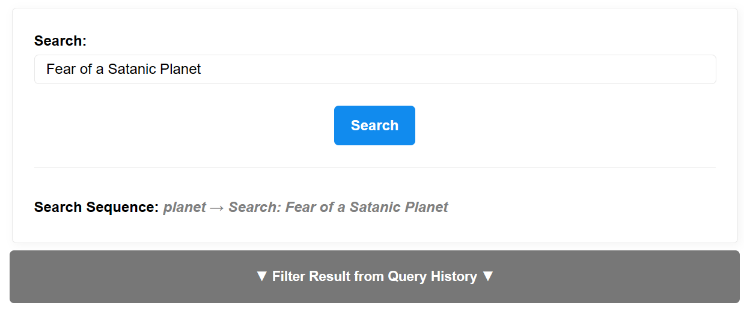
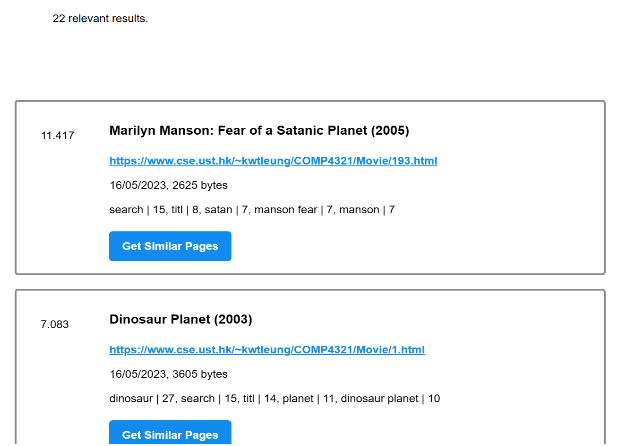
**Effective Highlighting**: Key features and relevant information are highlighted effectively, drawing users' attention to critical elements and improving usability.

### 5. Enhanced Query Handling

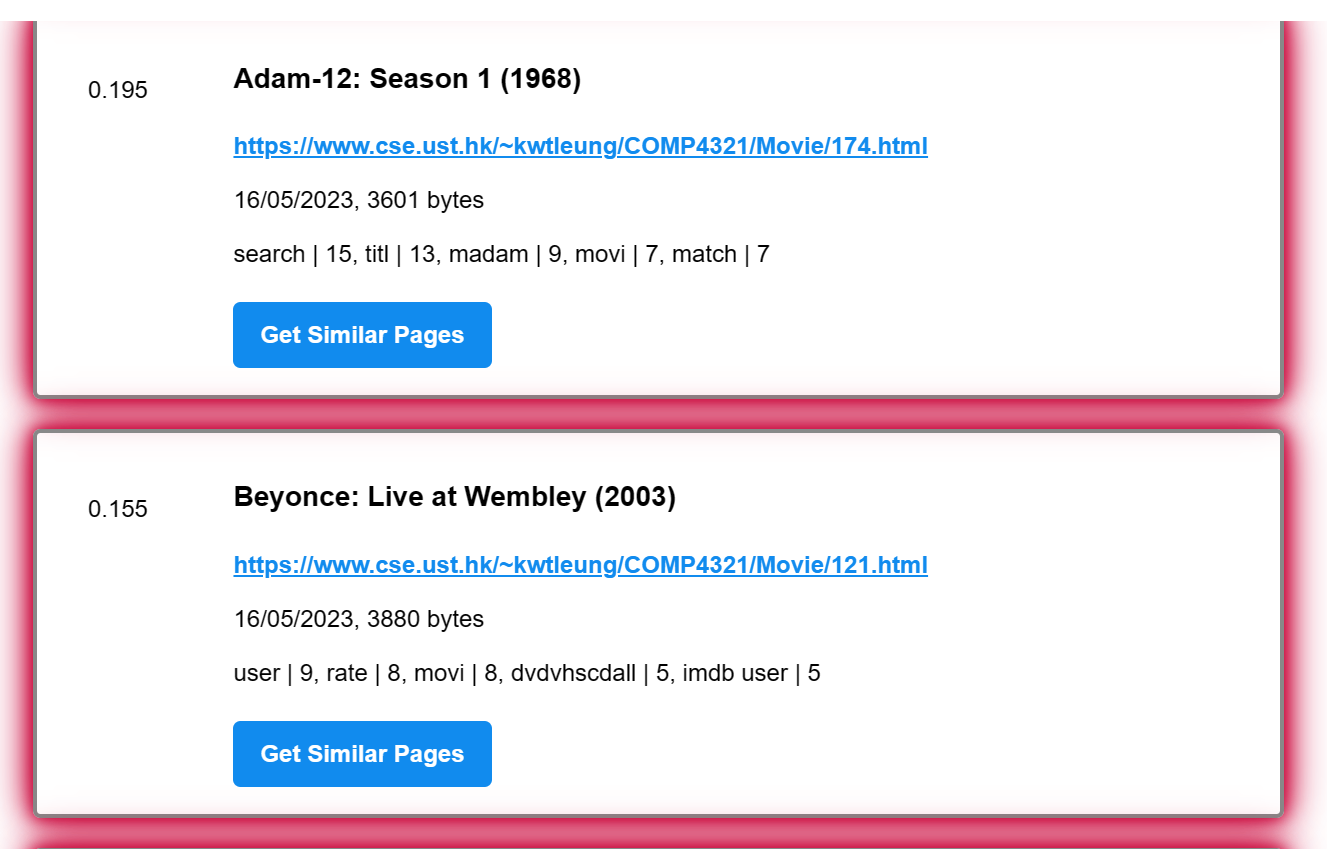
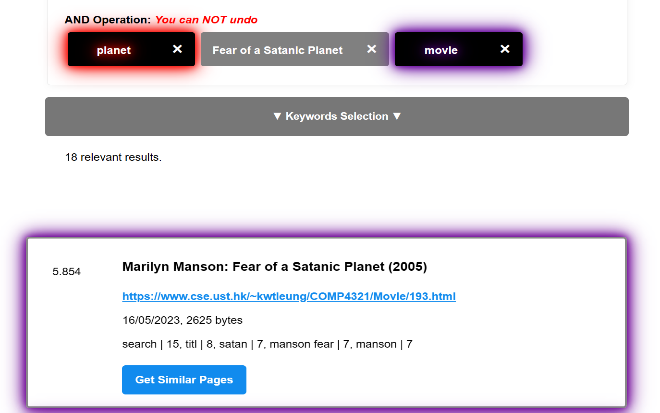
**Multi-Phrase Support**: Users can input multiple phrases and keywords, with advanced processing mechanisms such as stop-word removal and word stemming, resulting in more accurate and relevant search results.

# Testing of the Functions Implemented

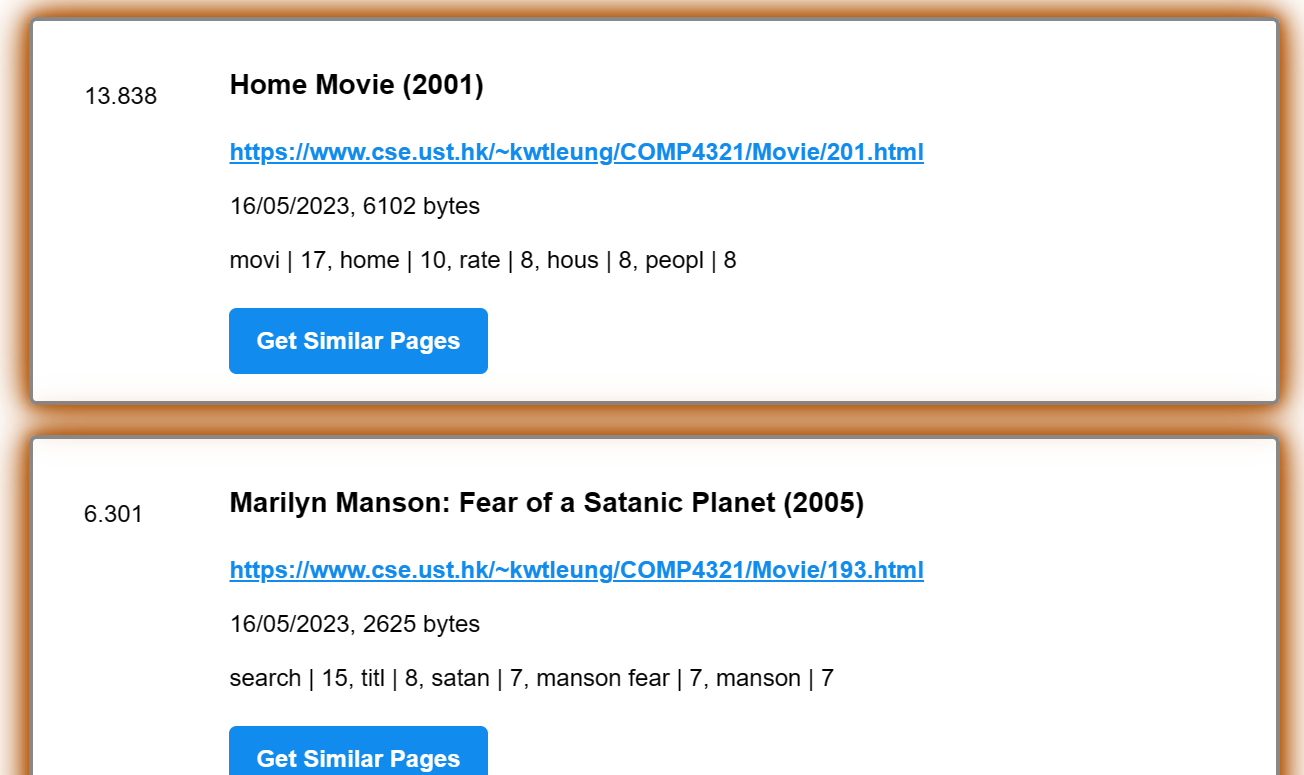
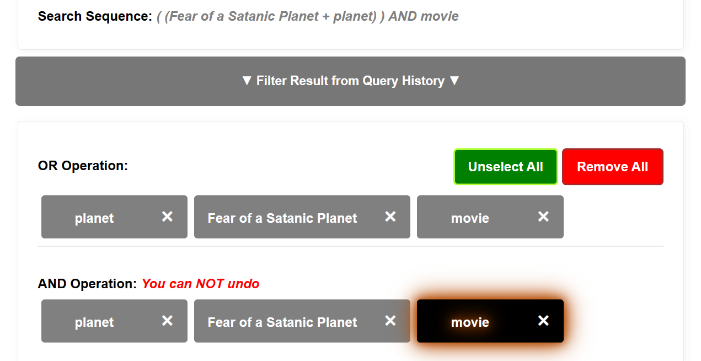
1. Searching on top of previous result using AND operator



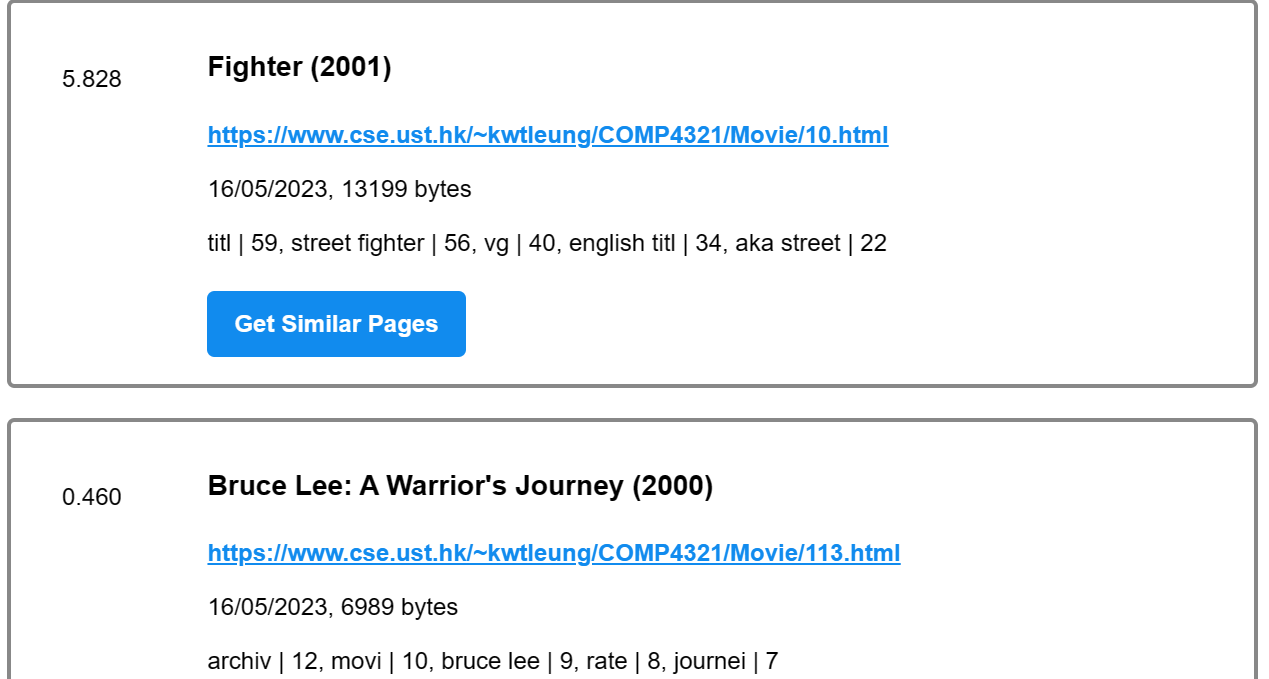
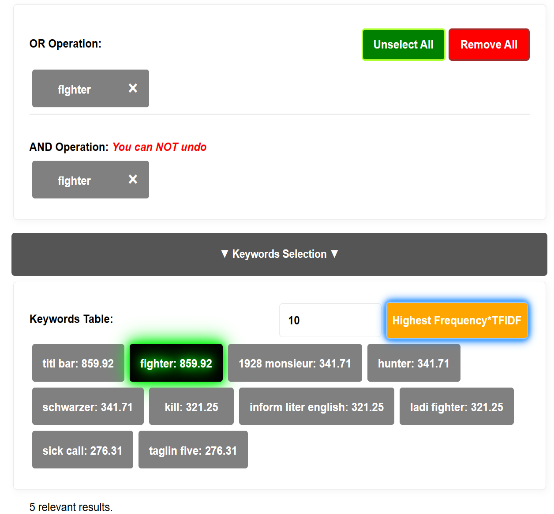
1. Combining the previous two results using AND operator



1. Combining the previous results using OR + AND operator



1. Keywords Table (Highest TFIDF/Highest Frequency\*TFIDF/Highest TFIDF)



# Conclusion

## Strengths

### User Experience (UX)

**Smooth Searching Experience**: The system offers rapid query processing, ensuring users receive search results promptly.

**Short Scrolling Area**: Organizing results into tabs, with five results per tab, minimizes scrolling and aids in locating relevant pages efficiently.

**Easy Tab Navigation**: Users can effortlessly navigate between tabs to access previously important results.

### User Interaction (UI)

**Simplistic Style**: The UI features clean and organized elements, enhancing usability and visual appeal.

**Contrast on Important Features**: Key features are highlighted effectively, guiding users' attention to essential information.

Functionalities

**Allow Input for Multiple Phrases and Keywords**: Users can input multiple phrases and keywords, with the backend employing stop and stem mechanisms for efficient processing.

## Weaknesses

### Crawl and Index

**Resource Intensive**: The crawling and indexing process requires significant resources and time, impacting overall system efficiency.

**Potential Loss of Child Links**: Newly crawled pages may not revisit indexed pages, leading to missing child links in the records (Child links are temporally in the URL format and will be changed to Page ID that are indexed. If the child pages are not indexed in the iteration, they are gone in the parent-child relationship).

Result Ranking

**Omission of Important Features**: Some crucial features, such as metadata and original words, are neglected in the result ranking process.

### Stop and Stem

**Inclusion of Excessive Words and N-gram Phrases**: The stop and stem mechanism may include unnecessary words and n-gram phrases, leading to cluttered results.

## Re-Implementation

### Whole System Setup

**Programming Language**: Transitioning to Python with Django backend and SQLite database, and potentially React for frontend, offers familiarity and access to a broader range of libraries.

**Reasoning**: Python's extensive library ecosystem and streamlined debugging process could enhance development efficiency and performance.

Indexing Functions

**Optimization Requirement**: The indexing functions require optimization to improve runtime efficiency and resource utilization.

## Interesting Features

### Implicit & Explicit Features

**User Feedback Collection**: Implementing implicit and explicit feedback mechanisms to build user profiles for personalized search experiences.

Page Rank based on Category

**Categorization Algorithm**: Introducing a page ranking algorithm based on category tags, akin to Google's approach, to enhance search quality and personalization.

### Input Analysis using LLM Model

**Advanced Input Analysis**: Leveraging Language Model (LLM) to analyze user input strings and adjust weighting dynamically for improved search relevance.

# Contribution

|  |  |
| --- | --- |
| Tony (33.3%) | Backend Crawling, Indexing, Database Design |
| Cathy (33.3%) | Part of the Frontend, Report, Algorithms & Formula |
| York (33.3%) | Part of the Frontend, Video, Additional Features |