A logo with a magnifying glass and a magnifying glass

Description automatically generated

**PROJECT TITLE: AXION (Search Engine)**

**Class: BSCS-13 (C)**

|  |  |
| --- | --- |
| **Members** | **CMS ID** |
| Furqan Ahmad Basra | 462974 |
| Muhammad Saad Akhtar | 458102 |
| Hanzla Kalim | 479152 |
| Aqib Ramzan | 459729 |
| Zakriya Mansoor | 458368 |

**Project overview**

**1.1 Problem Definition:**

In today's digital age, the exponential growth of information presents a significant challenge in accessing relevant data efficiently. Users often face difficulties in locating accurate and precise information from vast collections of documents or datasets. Existing search solutions may be either overly generic, slow in response, or inadequate in providing contextually relevant results for specific queries.

This project addresses the need for a robust and efficient search engine that can:

1. **Index** large volumes of data for rapid retrieval.
2. **Process Queries** to extract the most relevant results accurately.
3. **Rank Results** based on their relevance to the user's intent.

The primary objective is to implement a search engine using advanced data structures and algorithms, ensuring optimal performance, scalability, and user satisfaction in diverse use cases.

**1.2 Design Goals:**

**Efficiency**

* Design the search engine to process and retrieve results quickly, minimizing response time for user queries.
* Optimize the indexing and search algorithms for low time and space complexity.

**Relevance**

* Ensure the search engine provides accurate and contextually relevant results by ranking documents based on keyword occurrence, proximity, and importance.

**User-Friendly Interface**

* Provide a simple and intuitive interface for users to enter queries and view results.
* Incorporate features like autocomplete, suggestions, and spell-check for a seamless user experience.

**Modularity**

* Develop the search engine in a modular fashion, allowing independent development and testing of indexing, query processing, and ranking modules.
* Enable easy integration of additional features or algorithms in the future.

**Customizability**

* Allow flexibility to adjust ranking algorithms or indexing methods to meet specific needs or domains.

1.3 **Methodology to be Followed**

1. **Collecting Data**

The first step in the development process was to acquire a dataset that could be processed and indexed for use in the search engine. For this, we obtained a dataset from the Kaggle website in **CSV format**.

**Challenges with Raw Data**

The raw dataset contained:

* Numerous **null values** in various columns.
* **Unnecessary columns** that were irrelevant to the search engine functionality.
* **Errors in data format**, including invalid or incomplete JSON objects.

**Data Cleaning and Preprocessing**

To transform the raw dataset into a reliable and specific format, we utilized the **Pandas library** for Python. The following steps were undertaken:

1. **Column Selection**: Irrelevant columns were identified and removed.
2. **Handling Null Values**: Rows containing null values were filtered out to ensure data consistency.
3. **Error Correction**: JSON objects within the dataset were validated and corrected.
4. **Format Conversion**: The cleaned data was converted into a structured **JSON format**, which is both lightweight and suitable for indexing and processing.

**About our data:**

The dataset comprised **articles and research papers**, making it ideal for testing the search engine's capabilities in retrieving information relevant to user queries.

2. **Lexicon Formation**

The second step involved creating a **lexicon**, a comprehensive repository of words that the search engine could use to perform searches. This process required extracting meaningful data from the JSON dataset and preparing it for efficient indexing and retrieval.

**Data Extraction**

We focused on specific fields within the JSON objects, such as:

* **Keywords**
* **Title**
* **Abstract**
* **Year**
* **Author Names**

These fields provided the core data for building the lexicon, as they represent the most relevant searchable content.

**Preprocessing Steps for Lexicon Formation**

1. **Tokenization**:
   * The extracted strings were broken down into individual tokens (words) and we remove the stop words(words like is, are, am ) from it
2. **Case Normalization**:
   * All tokens were converted to **lowercase** to ensure case-insensitive search functionality.
3. **Lemmatization**:
   * Words were converted to their **base forms** using a **rule-based English lemmatization** approach. For example:
     + "**Running" → "Run"**
     + **"Studies" → "Study"**
4. **Word Cleaning**:
   * Non-alphabetic words and special characters were filtered out.
   * Strings containing only valid alphabetic characters were retained to ensure lexicon quality.

**Storage**

* To ensure **uniqueness** and **efficient lookup**, the processed words were stored in a **HashSet**.
* This data structure helped eliminate duplicates automatically and provided rapid access during query processing.
* Finally, the data is stored in the json format.

**3. Forward Index Formation**

The third step in our search engine development process was to create the forward index. The forward index is a data structure that stores information about the terms (words) present in each document, along with their frequency of occurrence. This index is critical for efficiently retrieving relevant documents during search queries.

**Data Processing for Forward Index**

1. **Data Extraction:**  
   We continued using the same fields extracted during lexicon formation (keywords, title, abstract, year, author names) from the JSON data. These fields were processed to construct the forward index.
2. **Tokenization and Preprocessing:**
   * The text data from each document was tokenized into individual words.
   * Lemmatization was applied to reduce words to their base forms (e.g., "running" → "run").
   * Non-alphabetic characters and errors (e.g., encoding issues) were removed to ensure clean tokens.
3. **Data Structuring:**
   * For each document, we created an entry in the hash table.
   * The key of the hash table was the document ID, ensuring that each document is uniquely identifiable.
   * The value associated with each document ID contained the words present in that document, along with their frequency of occurrence.

Finally, the data is stored in the json format. A sample of the forward index structure is provided in picture name forward\_index\_pictiure at the end

1. **Inverted Index Formation**

The fourth step involved the creation of the inverted index, which is a crucial component for efficient document retrieval in a search engine. The inverted index allows for fast lookup of documents that contain a specific word, facilitating quicker search query responses.

**Data Processing for Inverted Index**

1. **Utilizing Lexicon and Forward Index:**
   * We leveraged the lexicon and the forward index to build the inverted index.
   * The lexicon provided the list of unique words, while the forward index contained the document IDs and the frequency of each word in those documents.
2. **Building the Inverted Index:**
   * We iterated through the words in the lexicon.
   * For each word, we checked if it existed in the forward index. If the word was found, we retrieved the associated document IDs (from the forward index) in which the word occurred.
3. **Inverted Index Structure:**
   * The structure of the inverted index was designed as a hash table with the word as the key and the document IDs as the value.
   * The value contains the list of document IDs in which the word is present. This allows the search engine to quickly retrieve all documents containing a specific word.
4. **Storage**

* To ensure **uniqueness** and **efficient lookup**, the processed words were stored in a **HashSet**.
* This data structure helped eliminate duplicates automatically and provided rapid access during query processing.

1. **Dividing into Barrels**

* Once the lexicon was created, we moved to the next step of optimizing data storage.
* We calculated the **hash value** for each document ID and took its **modulo 50**. This operation ensured that the document IDs were distributed into **50 separate barrels**, with index values between **0 to 49**.
* Dividing the data into these barrels helps in efficient data loading, as not all data needs to be searched into memory at once, it reduces the search space, making retrieval operations more efficient.

1. **Storing the documents into barrels**
   * We store the documents in **barrels**, where each barrel corresponds to a **hash value** of the document ID.
   * The **document ID** is the **key** in the hash table, and the **values** associated with the key include all the relevant data for the document (such as title, abstract, keywords, etc.).
   * To determine the barrel where a document should be stored, we calculate the **hash value** of the document ID and then take the **modulo 50** of the hash value. This assigns the document to one of the **50 barrels** (indexed from 0 to 49).
   * This method ensures that we can limit the search to specific barrels during retrieval, reducing the number of documents to be searched and improving the efficiency of the system.
2. **Ranking and Retrieval**

The final step in the search engine process involves **ranking** and **retrieving** the most relevant documents based on the user's query. This step ensures that the most pertinent articles are presented first, offering users a streamlined search experience.

**Query Processing**

1. **Query Input**:
   * The user enters a **search query** in the form of a string.
   * The query is **lemmatized** to convert words into their base form, ensuring consistency in the search process.
   * The string is then **tokenized**, breaking it down into individual words (tokens).
2. **Search in Inverted Index**:
   * For each token in the query, we calculate its **hash value** and take the **modulo 50** to determine the **barrel number** in which to search.
   * This division into barrels (from 0 to 49) ensures efficient searching, limiting the scope to a specific set of documents.
   * We then search the **inverted index** to retrieve the **document IDs** associated with the words in the query. Since the inverted index is stored as a hash table, the search operation is **constant time** (O (1)).
3. **TF-IDF Calculation (Ranking Process)**:
   * Once the document IDs are found, we calculate the **TF-IDF** (Term Frequency-Inverse Document Frequency) score for each document, with the use of inverted and forward index.
   * The **TF** represents how frequently a term appears in a document, while the **IDF** measures the term's importance across all documents in the collection.
   * If a search query contains multiple words, the **TF-IDF** score for each document is incremented for every word found in that document.
   * This ensures that documents containing more relevant words from the query receive higher rankings.
4. **Sorting by Relevance**:
   * After calculating the **TF-IDF** scores for each document, we sort the documents based on their relevance to the query.
   * **Insertion sort** is used for sorting the documents according to their TF-IDF scores. The document with the highest **TF-IDF** score is considered the most relevant and is placed at the top of the list.
5. **Displaying the Results**:
   * After sorting, we retrieve the most relevant documents form the document barrels, as we find the hash of the ids , and find them in their relevant barrels , reducing the search space and present them to the user.
   * The documents are stored in **JSON format**, which provides an easy-to-use, structured output.
   * The result is a ranked list of documents, with the most relevant document at the top, ensuring that the user gets the most accurate search results based on their query.

**Summary of the Ranking and Retrieval Process**

* **Input**: User query
* **Preprocessing**: Lemmatization, Tokenization
* **Search**: Inverted index lookup, barrel division for optimized search
* **Ranking**: **TF-IDF** scoring, query-based weightage
* **Sorting**: Insertion sort for relevance
* **Output**: Display ranked documents in **JSON** format

**Overall structure:**

The project is divided into two main components: Frontend and Backend.

1. **Frontend (React Application)**

* Framework: Built with React for the user interface.
* Function: Users enter search queries, which are sent to the backend.
* Port: Hosted locally on a specific port (e.g., localhost:3000).
* Request: Sends HTTP requests (e.g., POST /search) to the backend.
* Response: Displays the search results returned in JSON format.

1. **Backend (Search Engine Processing)**

We made the **backend server using the crow library**

* **Function:** Processes the search query, searches the inverted index, ranks results, and generates the response.
* **Port:** Runs on a different port **18080**.
* **Search Process:** Preprocesses the query, searches the inverted index, calculates TF-IDF values, and returns the results.

1. **Communication Between Frontend and Backend**

* The Frontend sends a query via HTTP request to the Backend, which processes it and sends back a JSON response with the relevant search results.

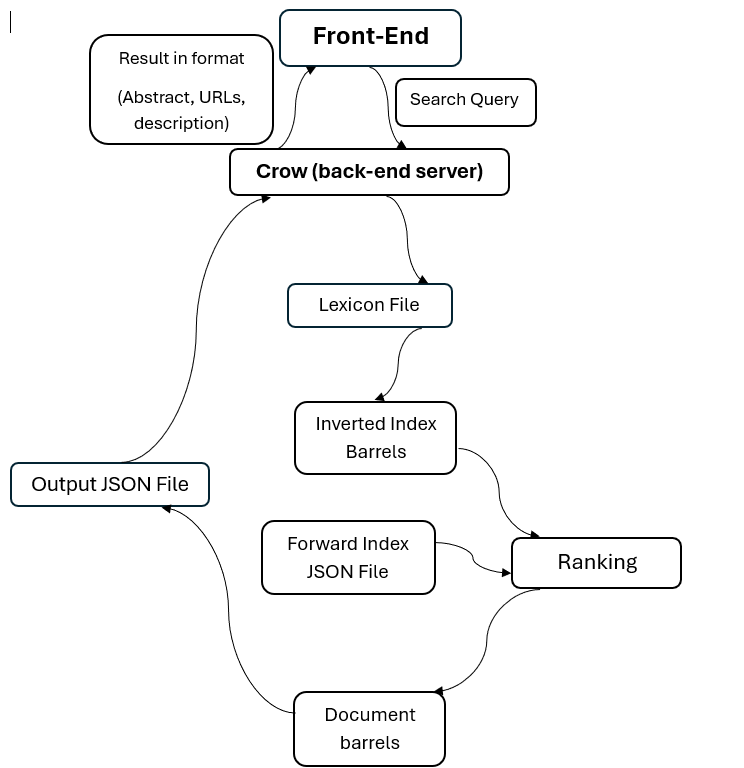
**Expected outcome:**

* Efficient Search and Retrieval: The search engine will efficiently process user queries and return the most relevant documents based on the input keywords. By using techniques like tokenization, lemmatization, and the inverted index, the system will ensure accurate and fast retrieval of relevant research articles or papers.
* Scalable Indexing: The use of forward and inverted indices, along with the division of data into barrels, will optimize the storage and search operations, enabling scalability as more data is added. The indexing strategy ensures minimal search space and faster query processing.
* Improved User Experience: The frontend built in React will provide a user-friendly interface, allowing users to easily input their queries and view results in a structured and readable format.
* Accurate Document Ranking: By implementing the TF-IDF ranking mechanism, the search engine will accurately rank documents according to their relevance to the search query, ensuring that the most pertinent documents appear first.
* Modular and Maintainable Architecture: The separation of frontend and backend components ensures that the system is modular, making it easier to maintain, scale, and update without affecting the entire system.
* Real-time Query Processing: With the system designed to run on local servers, users can expect real-time search query processing, providing quick and accurate results.
* JSON-Based Response: The backend will return search results in JSON format, which can be easily processed by the frontend for dynamic and efficient display.

**TECHNICAL DETAILS:**

1. **Architecture diagram:**

Diagram is provided outside it represent the final running how the input is coming to backend and how the final output is begin made.



1. **Workflow of the project:**
2. **User Input:**
   * The user enters a search query on the frontend, which is built using React. This could be a keyword, or a phrase related to research articles or papers.
3. **Request to Backend:**
   * The frontend sends the user’s query as an HTTP request to the backend server. The query is usually sent as a JSON object through a POST request to a specific API endpoint.
4. **Preprocessing the Query:**
   * On the backend, the query is received, and preprocessing steps are applied:
     + **Tokenization:** The query is broken down into individual tokens (words).
     + **Lemmatization:** The tokens are converted to their base form (e.g., "running" becomes "run").
     + **Lowercasing:** The query is converted to lowercase to ensure case insensitivity.
5. **Searching the Inverted Index:**
   * The backend uses the inverted index to search for the relevant words in the dataset. For each token in the query, the backend performs a hash operation to find the appropriate "barrel" (subdivision of the index).
   * The inverted index helps locate the document IDs where the relevant words appear.
6. **Ranking the Results:**
   * After finding the document IDs, the backend calculates the TF-IDF (Term Frequency-Inverse Document Frequency) for each document to rank their relevance to the search query.
   * **Sorting:** The documents are sorted based on their TF-IDF values, with the most relevant documents placed at the top.
7. **Returning Results:**
   * Once the results are ranked, the backend sends the list of document IDs and their corresponding details back to the frontend as a JSON response.
8. **Displaying Results:**
   * The frontend receives the JSON data and displays the search results to the user in a structured format, showing relevant details like article titles, abstracts, and authors, ordered by their relevance.
9. **Data structures used and libraries:**
10. **Lexicon (HashSet)**:
    * A **HashSet** is used to store the unique words (tokens) derived from the dataset (articles, research papers). This structure ensures fast look-up and avoids duplicates, as only unique words are kept.
    * **Purpose**: This is used for constructing the **inverted index** and for tokenizing the query during the search process.
11. **Forward Index (HashTable)**:
    * The **Forward Index** is implemented using a **hash table**  where the keys are the **document IDs** and the values are the words present in the document along with their frequencies.
    * **Purpose**: It helps in quick access to the words present in any document for building the **inverted index**.
12. **Inverted Index (HashMap)**:
    * The **Inverted Index** is another key data structure, where the **key** is a **word** (token), and the **value** is a list of **document IDs** in which the word appears.
    * **Purpose**: This is used to quickly retrieve documents that contain a specific word, allowing for efficient query processing.
13. **Document Storage (HashTable)**:
    * Documents are stored in a **hash table** where the key is the **document ID**, and the value is the actual document data (title, abstract, authors, etc.).
    * **Purpose**: This allows for quick retrieval of document details during search result display.

**Components of the project:**

1. **Crow library:** for running the backend on the local host
2. **React:** for running the front end
3. **Nlohmann JSON:** for dealing with the json files.
4. **Regex:** for dealing with the string data, and data cleaning
5. **Implementation:**

* **Lexicon:**

input (original data) output (lexicon json)

main functions and classes :

// CustomSet implementation

VectorOfString split\_into\_words(const std::string &str)

string clean\_word(const string &word)

string lemmatize\_word(const string &word)

int main()

* **forward index : input (original json data ) output( foward index json file )**

split\_into\_words(const string& str)

string clean\_word(const string& word)

string lemmatize\_word(const string& word)

load\_stop\_words(const string& stopword\_file)

void generate\_forward\_index(const string& stopword\_file, const string& data\_file, const string& output\_file)

* **inverted index : input( lexicon json , foward index json ) output is inverted index json in barrels**

class Document

class WordFrequency

class HashMap

class InvertedIndex

* **idlink5**

class DataEntry

class Node

class HashMap

string limitTo100Words(const string& input) {

tf\_idf2.h : input (user string ) output json responce

#include "C:\Users\Hanzla Kalim\source\repos\dsa\loading.h"

struct Node

class LinkedListOfIds

vector<string> split\_into\_words(const string& str)

class DataEntry1

int hashfunction\_for\_doc(string key)

void TF\_IDF(string input\_by\_user) {

* **loading.h**

input all the json files , adn output all data loaded , at the time ot running the server.

**SELF REFLECTION:**

During the development of the search engine project, various challenges were encountered, ranging from data preprocessing to optimizing the search query performance.

During the development of the search engine project, several challenges were encountered:

1. **Data Collection and Preprocessing**:
   * **Challenge**: The dataset had errors, null values, and irrelevant columns, taking **two full days** to process.
   * **Solution**: Used **Pandas** to clean and convert data into a usable **JSON format**.
2. **Library Setup**:
   * **Challenge**: Installing **Nlohmann JSON library** was difficult due to compatibility issues.
   * **Solution**: Troubleshot and successfully set up the environment using online resources.
3. **Corw library setup:**

Crow wasn’t able to setup in visual studio, it need visual studio for its setup , so we had to shift to the visual studio.

1. **Error Handling**:
   * **Challenge**: Missing or incomplete fields like abstracts caused issues.
   * **Solution**: Implemented **if-else checks** to handle incomplete data.
2. **Unicode Handling**:
   * **Challenge**: **Unicode characters** caused processing errors.
   * **Solution**: Used **regex** to clean unwanted characters.
3. **Efficient Search and Retrieval**:
   * **Challenge**: Searching large datasets quickly was difficult.
   * **Solution**: Used **hash sets** and **hash maps** for fast **O (1)** search times.
4. **Handling Large Data**:
   * **Challenge**: Loading all data at once could affect performance.
   * **Solution**: Divided the **inverted index** into **barrels** for more efficient searching.

**Main issue at the end:**

Our whole project was based on hashing, and the hash functions in Visual Studio and Visual Studio Code used different methods to calculate the hash. In the end, our whole project wasn't working because of this. We had to switch to Visual Studio on the final night of the project.

**Learning outcomes:**

Throughout the development of this search engine project, we gained valuable insights into the architecture of large-scale systems, particularly by studying **Google’s search engine structure**. This high-level understanding influenced the design of our project and helped us make decisions related to data storage, indexing, and retrieval. We learned to work extensively with **HashSets** and **HashTables**, which are critical for efficient search operations. Dealing with large datasets allowed us to explore these data structures in depth, especially focusing on edge cases to ensure the system could handle various data anomalies while maintaining performance. We optimized the search engine for efficiency by using techniques like indexing and barrel-based data storage to reduce load times and enhance retrieval speed. This experience not only deepened our technical knowledge of search engine functionality but also honed our problem-solving skills in creating scalable and high-performance systems.

**Data Preprocessing Skills:**

* Gained hands-on experience in cleaning and processing real-world data, handling issues such as null values, irrelevant columns, and encoding errors, using tools like Pandas and regex.

**Library Integration:**

* Developed a deeper understanding of integrating external libraries, specifically the Nlohmann JSON library, and overcoming compatibility and setup challenges.

**Search Engine Design:**

* Learned how to design and optimize the search functionality of a search engine by implementing efficient data structures like hash maps and hash sets to ensure fast search and retrieval times.

**Error Handling and Data Validation:**

* Gained experience in implementing proper error handling and data validation checks to ensure the integrity of the search engine, even with incomplete or inconsistent data.

**Efficient Data Storage:**

* Learned how to divide large datasets into manageable parts (barrels) to optimize memory usage and improve search performance, ensuring scalability and efficiency.

**Query Ranking and Optimization:**

* Developed an understanding of ranking mechanisms such as TF-IDF for evaluating the relevance of documents and improving search results accuracy.

**Full-Stack Development:**

* Gained practical experience in building both the backend (handling data processing and search logic) and frontend (user interface) using React, learning how to integrate them effectively.

**Project Management:**

* Improved time management and problem-solving skills by overcoming various challenges, from data preprocessing to optimizing search performance, ensuring that all parts of the project were completed efficiently.

**Images:**

**Forward index visualization:**

A screen shot of a computer code

Description automatically generated

**Inverted index visualization:**

