Medline Search Engine (MSE)

CIS 612 – Big Data & Analytics

Final Project Report

Submitted By:

Dinky Mishra – 2864923
Alim Khan Abdul – 2882808
Sushma Reddy Avala – 2885387
Aditya Sairam Pullabhatla – 2863159

Table of Contents

1. Data Description, Data Size, Data Collection Method

Data Description

The Medline Search Engine project aimed to create a comprehensive search platform for medical information by leveraging data from the MedlinePlus Encyclopedia. The dataset contains detailed medical articles, which include:

- **Article Titles**: Capturing the main topic of each article.
- **URLs**: Providing links to the original articles on the MedlinePlus platform.
- **Content**: Textual content covering medical topics, including diseases, symptoms, procedures, medications, and medical conditions. The content was extracted to facilitate efficient indexing and querying.

Data Size

- **Total Articles Scraped**: 4,500 approx. articles from MedlinePlus.
- **Number of Terms**: Over 1.4 million terms after preprocessing, including specialized medical terms and general content terms.

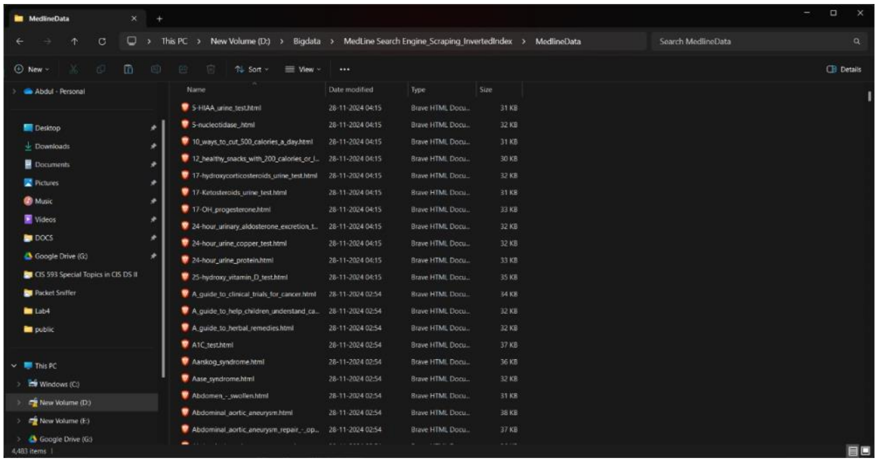
Data Collection Method

Data collection was accomplished through a custom web scraper developed using Node.js and Puppeteer. This scraper was designed to efficiently navigate and extract content from the MedlinePlus website.

- **Tool Used**: webScraper.js

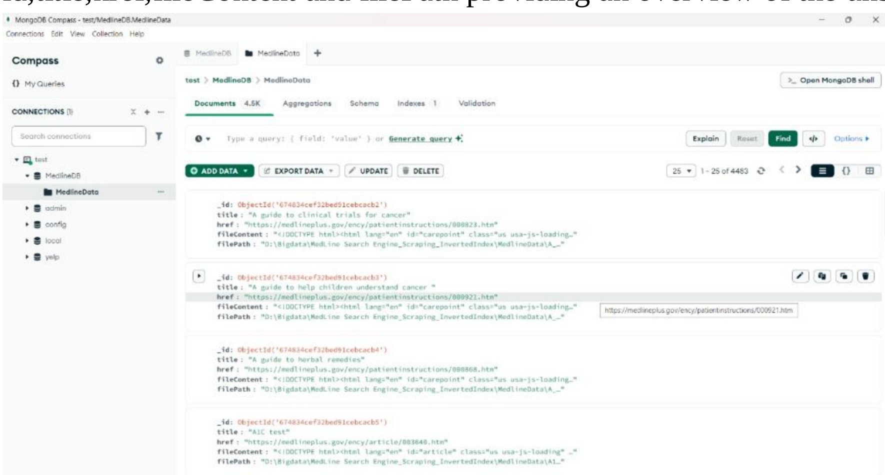
```
import puppeteer from "puppeteer";
import fs from "fs/promises";

const main = async () => {
   const browser = await puppeteer.launch();
   const page = await browser.newPage();
   await page.goto("https://medlineplus.gov/encyclopedia.html");
   // Scrape data logic here
   await browser.close();
};

await main();
```

- **Process**:
  - **Automated Scraping**: The script navigated the MedlinePlus Encyclopedia website, accessed the articles, and extracted key details such as titles, URLs, and HTML content.
  - **Data Storage**: Scraped data was stored initially as raw HTML files and then imported into MongoDB for efficient retrieval and backup purposes.
  - **Data Validation**: Data accuracy and completeness were validated by cross-referencing the number of scraped articles with the total number of articles available on the MedlinePlus website. Duplicate articles were removed, and inconsistencies were handled to maintain data integrity.



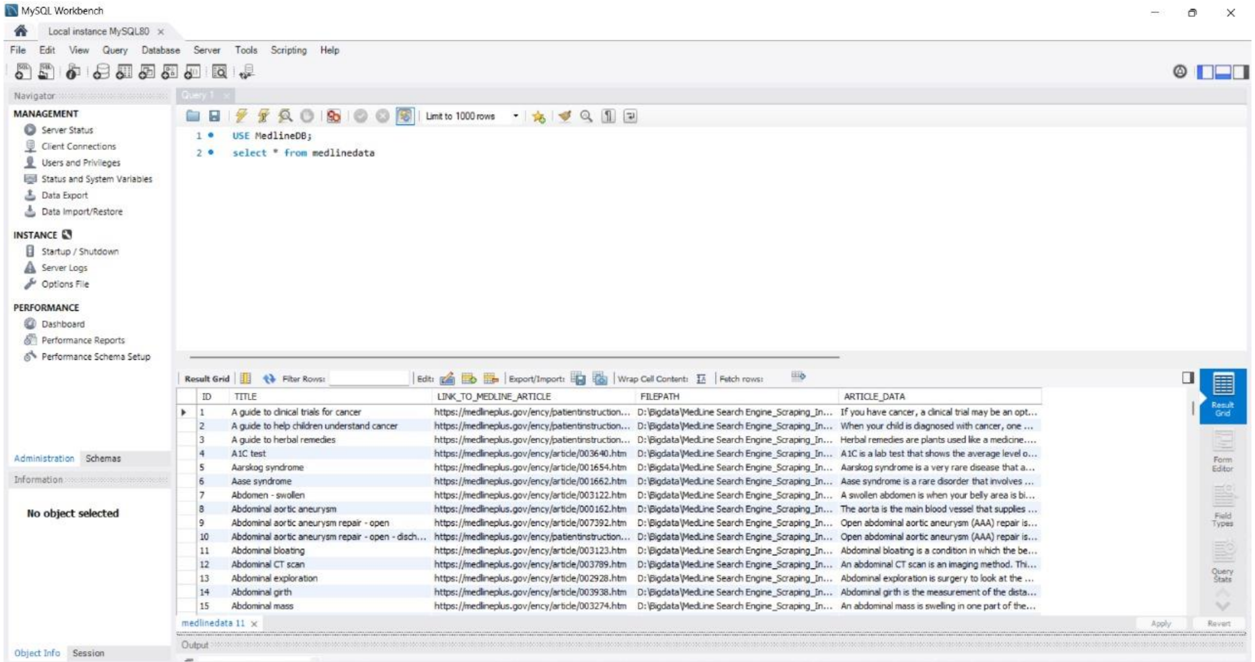
Medline Data Snapshot in MongoDB:

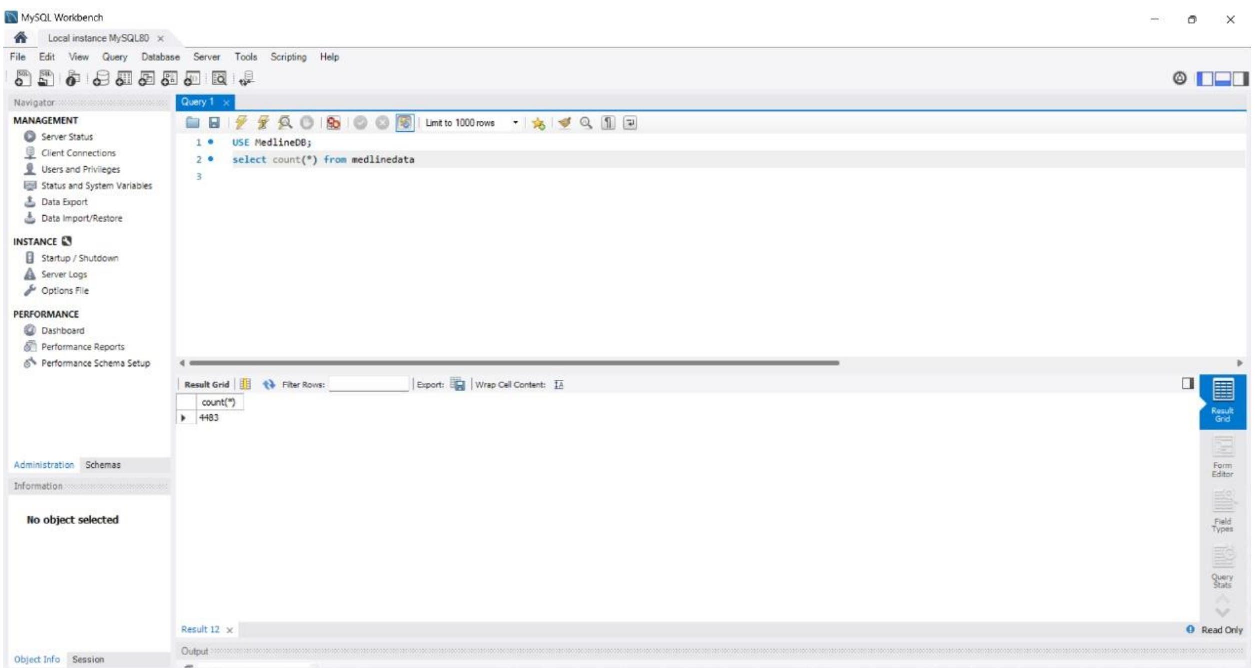
- **Description**: This screenshot shows a preview of the data stored in the medlinedata table in MongoDB. It displays multiple records, including fields such as id,title,href,fileContent and filePath providing an overview of the unstructured data.



Medline Data Snapshot in MySQL

- **Description**: This screenshot shows a preview of the data stored in the medlinedata table in MySQL. It displays multiple records, including fields such as ID, TITLE, LINK_TO_MEDLINE_ARTICLE, FILEPATH, and ARTICLE_DATA, providing an overview of the structured data available for querying.





2. Goal of Your Intelligent Big Data Analytic Application (AI)

Project Goal

The goal of the Medline Search Engine (MSE) project was to develop a sophisticated search platform that could provide efficient and accurate medical information to users. Specifically, the project sought to:

- **Develop a Data Collection Strategy**: Extract a large volume of medical articles from MedlinePlus.
- **Create a Data Processing Workflow**: Preprocess and clean data to make it suitable for analysis and efficient querying.
- **Build an Indexing System**: Create an inverted index to facilitate fast search queries.
- **Implementation of NLP Techniques**: Use Natural Language Processing techniques to enhance the accuracy of search results.
- **Develop a User-Friendly Search Interface**: Provide a responsive web application interface for users to enter queries and retrieve real-time search results.

AI Components

- **Natural Language Processing (NLP)**: The project leveraged various NLP techniques, such as tokenization, stemming and lemmatization to process and normalize the tokens.
- **Machine Learning Algorithms**:
    - **TF-IDF (Term Frequency-Inverse Document Frequency)**: Used to measure the importance of each term in each document.
    - **Cosine Similarity**: Used to rank documents based on their relevance to user queries, providing a similarity score to determine the most relevant articles.

3. Platform Setting/System Configuration Procedures

Development Environment

- **Operating System**: Windows 11
- **Programming Language**: Node.js (v20.9.0) for backend operations, JavaScript for frontend development.
- **IDE/Editor**: Visual Studio Code (VS Code) for writing and debugging code.

System Configuration Procedures

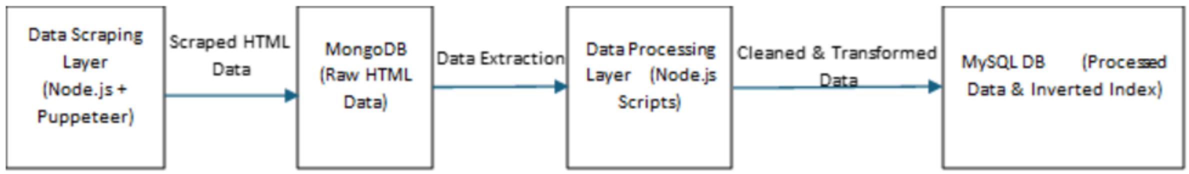
1. **Node.js Environment Setup**:
    a. Installed Node.js and npm (Node Package Manager).
    b. Verified installations using node -v and npm -v commands.
    c. Used npm(v 10.1.0) to handle project packages.
2. **Required Libraries Installation**:
    a. Installed necessary Node.js libraries for scraping, processing, and database interactions:
        i. **Puppeteer**: Used for web scraping (npm install puppeteer).
        ii. **Lemmatizer,Pluralize and Stem-porter**: Libraries for NLP preprocessing (npm install lemmatizer , npm install pluralize and npm install stem-porter).
        iii. **MongoDB and MySQL**: Drivers for database operations (npm install mongodb and npm install mysql).
3. **MongoDB Setup**:
    a. Installed MongoDB(v2.3.2) Community Edition.
    b. Created a database named MedlineDB to store raw HTML data.
4. **MySQL Setup**:
    a. Installed MySQL Server Community version (v8.0.39) for storing processed data.
    b. Created a database named MedLineDB and used MySQL Workbench for management.
5. **Frontend Setup**:
    a. Installed the Next.js framework for building a responsive web application (npx create-next-app@latest medline-search-engine).
    b. Configured necessary dependencies for creating an interactive UI.
6. **Server Configuration**:
    a. Configured backend server using Node.js to handle incoming requests and process queries.Used Next.js built-in routing system.
    b. Set up local development servers for testing and debugging.

4. System Design (Architecture) of Your AI Application in Detail

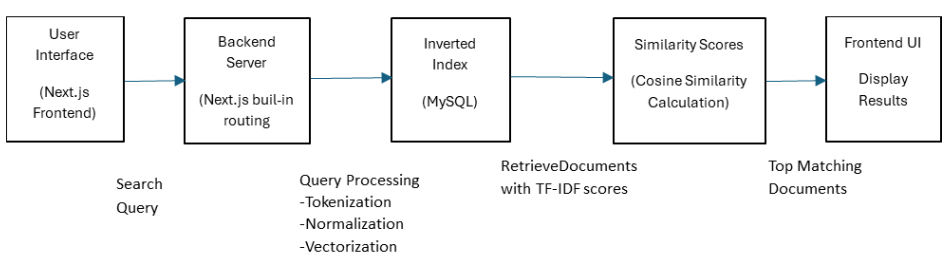
Overall Architecture

The system architecture of the Medline Search Engine application comprises multiple layers:

Data Extraction and Ingestion Flow:



1. **Data Collection Layer**:
    a. The webScraper.js script used Puppeteer to scrape medical articles from MedlinePlus.
    b. Data was stored as raw HTML files and then transferred to MongoDB for backup and retrieval.
2. **Data Processing Layer**:
    a. The cleanData.js script retrieved the raw HTML content, cleaned it by removing unnecessary tags and content, and stored the processed text in articleDetails.json.
    b. The buildInvertIndex.js script constructed the inverted index by tokenizing articles, calculating term frequencies, and storing them in MySQL.
3. **Data Storage Layer**:
    a. **MongoDB** was used for storing raw data scraped from the website.
    b. **MySQL** was used for storing processed data, including document metadata and the inverted index.
4. **Application Layer**:
    a. **Backend Server**: Developed with Node.js, responsible for handling requests, processing user queries, and interacting with the MySQL database.
    b. **Frontend Interface**: Created with Next.js, providing users with a search bar to enter queries and displaying results interactively.



5. **NLP Components**:
    a. Implemented several NLP techniques, such as tokenization, stemming, and lemmatization.
    b. Used TF-IDF weighting and Cosine similarity for ranking search results and improving relevance.

Data Flow Diagram

```
[User Interface (Next.js)]
        |
        V
[Backend Server (Node.js)]
        |
        V
[MySQL Database (Processed Data)]
        ^
        |
[Data Processing Scripts]
        ^
        |
[MongoDB (Raw Data)]
        ^
        |
[Web Scraper (Puppeteer)]
```

5. Raw Big Data Preprocessing Methods and Intermediate Results

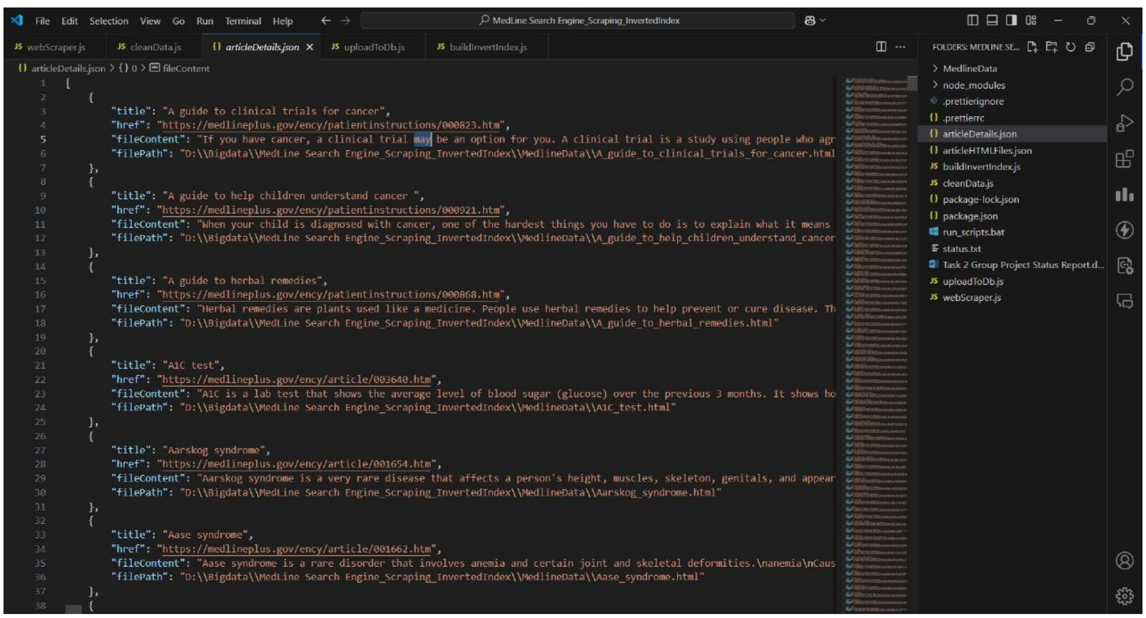
Data Cleaning (cleanData.js)

- **Extraction of Relevant Content**:
  - Identified and extracted relevant text from key HTML tags such as <p>, <h1>, <h2>, etc.
  - Ignored irrelevant content, including advertisements, navigation links, and redundant text.

```
import { createConnection } from "mysql2/promise";
import fs from "fs/promises";

const main = async () => {
   const conn = await createConnection({ host: "localhost", user: "root", password: "password123" });
   await createDatabase(conn, "MedLineDB");
   await createTable(conn, "MedLineData");
   // Data cleaning logic here
   conn.close();
```

```
};

await main();
```

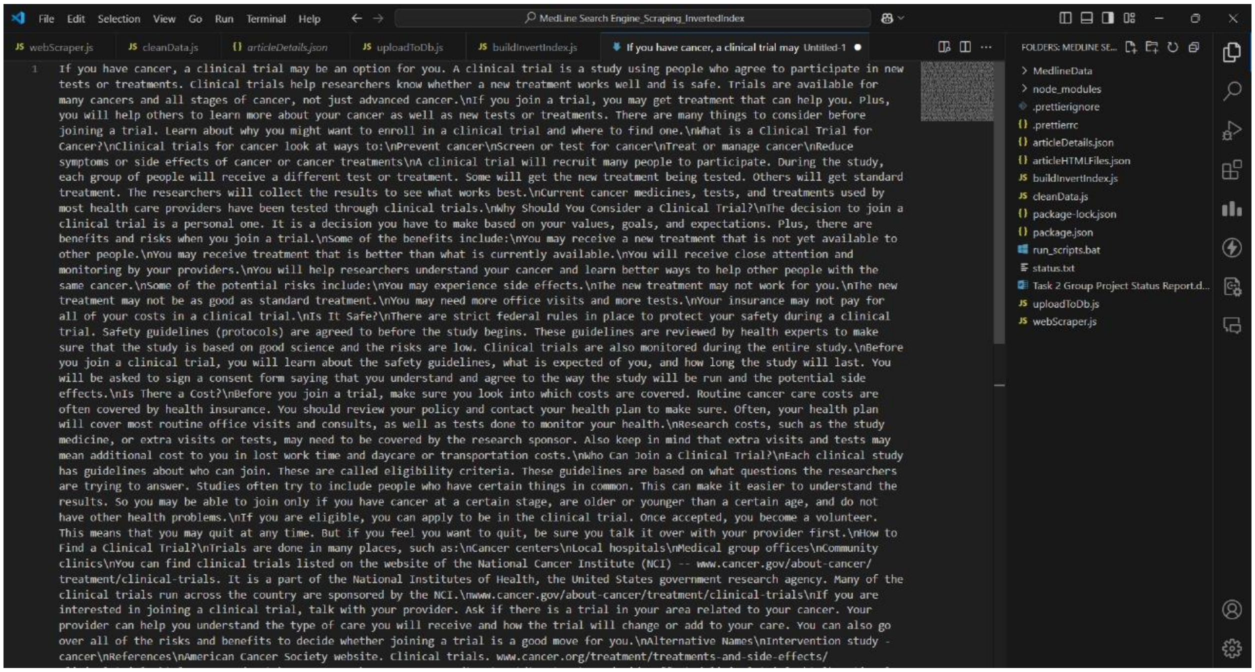
- **Text Normalization**:
  - Converted text to lowercase to ensure uniformity.
  - Removed special characters, punctuation, and unwanted symbols.
  - Addressed encoding issues and ensured the text was properly formatted for further processing.
- **Intermediate Output**:
  - Cleaned articles were stored in a JSON file, articleDetails.json, containing fields like title, href, fileContent (cleaned text), and filePath.
  - This screenshot shows the articleDetails.json file containing structured data after the data cleaning process. Each record includes the article's title, href, fileContent and filePath, ready for further processing and indexing.



- Data in this format was well-structured and ready for feature extraction and indexing.

Sample Cleaned Data Snapshot

- **Description**: This screenshot shows a sample of the cleaned content for an article. It highlights how the original HTML has been processed to extract only the meaningful text data, ensuring consistency and readability.



6. Design of Big Data Processing Pipeline, Data Transformation Methods

Processing Pipeline Steps

1. **Data Ingestion**:
    a. Loaded cleaned data from articleDetails.json using the uploadToDb.js script to import the content into the MySQL database.

```
import { createConnection } from "mysql2/promise";
import fs from "fs/promises";

const main = async () => {
   const conn = await createConnection({ host: "localhost", user: "root", password:
"password123" });
   await createDatabase(conn, "MedLineDB");
   await createTable(conn, "MedLineData");
   // Load data into database logic here
   conn.close();
};
```

await main();

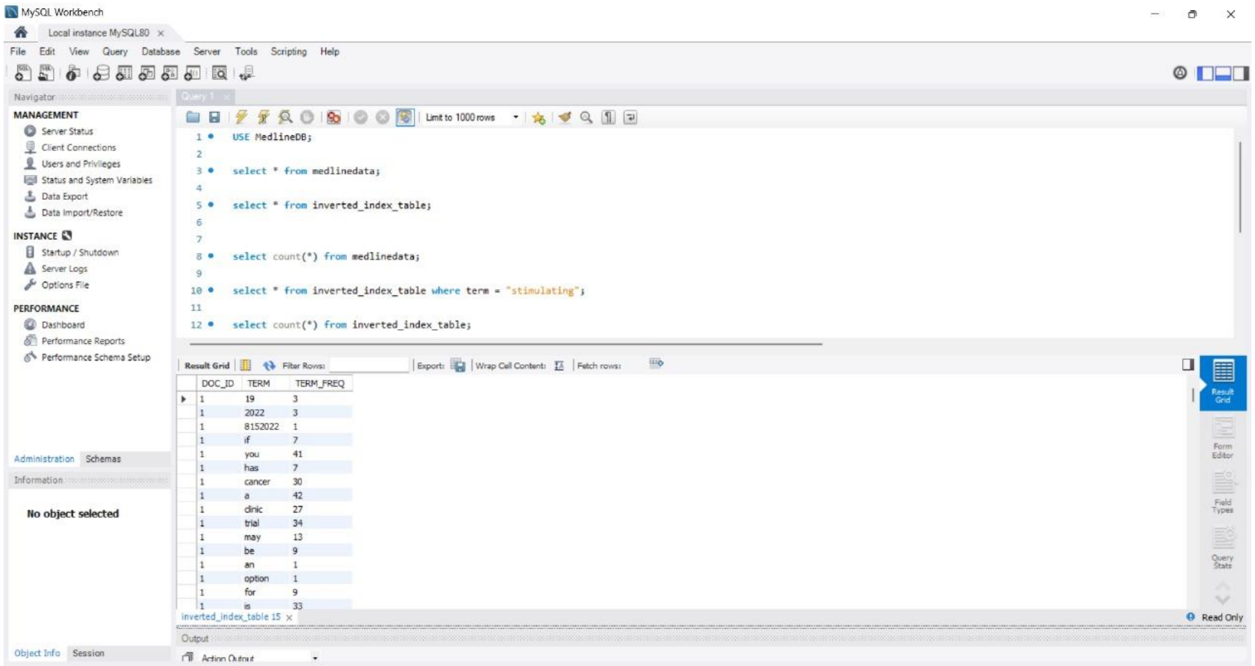
2. **Tokenization**:
    a. Split the cleaned text into individual tokens (words) using the tokenize() function to facilitate NLP processing.
3. **Normalization**:
    a. **Lowercasing**: Converted tokens to lowercase to ensure uniform representation.
    b. **Pluralization**: Reduced words to their singular form using the Pluralize library.
    c. **Stemming and Lemmatization**: Reduced words to their root form using the stem-porter and lemmatizer libraries.
4. **Feature Extraction**:
    a. Calculated **Term Frequency (TF)** to determine the number of times each term appeared in a document.
    b. Computed **Inverse Document Frequency (IDF)** to evaluate the importance of each term in the corpus.
5. **Building the Inverted Index**:
    a. Built an inverted index by mapping each term to the documents in which it appeared along with the TF-IDF scores.
    b. Stored the inverted index in MySQL for efficient retrieval and querying.

Data Transformation Methods

- **NLP Techniques**:
    - Enhanced understanding of the text through tokenization, lemmatization and stemming.
- **Vectorization**:
    - Represented documents and queries as vectors using TF-IDF scores.
- **Indexing**:
    - Created an inverted index using MySQL to facilitate fast and efficient search operations.

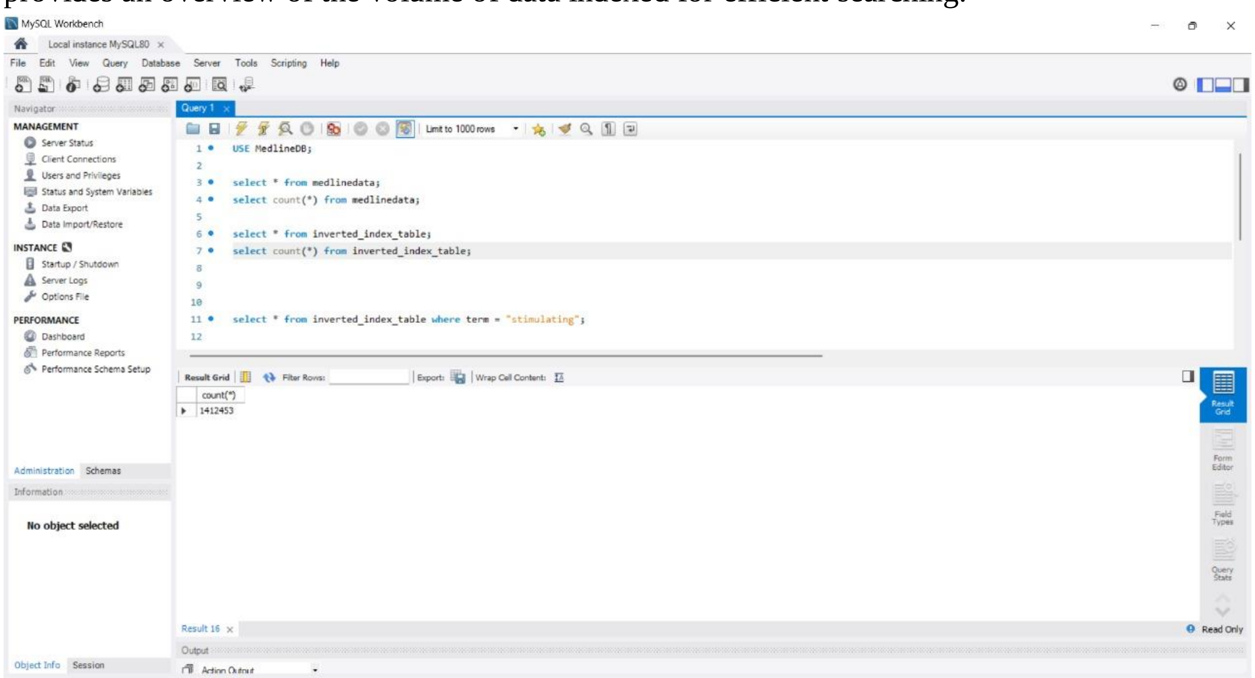
Inverted Index Table Snapshot

- **Description**: This screenshot shows a portion of the inverted_index_table in MySQL. It includes fields like DOC_ID, TERM, and TERM_FREQ, showcasing the tokenized words, their respective documents, and term frequencies.



Inverted Index Table Count Snapshot

- **Description**: This screenshot displays the total count of entries in the inverted_index_table in MySQL, indicating a total of 1,421,453 indexed terms. This provides an overview of the volume of data indexed for efficient searching.



7. Description of Your Knowledge Base Structure/Database Design for Information Retrieval Application

Database Schema Design

The Medline Search Engine application involved multiple databases to store different types of data, including scraped content, processed terms, and indexing tables. Here is the detailed structure:

1. **MongoDB (Raw Data)**:
   a. **Collection: RawHTML**:
      i. **Fields**:
         1. document_id: Unique identifier for each document.
         2. title: Title of the article.
         3. url: URL of the Medline article.
         4. html_content: Original HTML content of the article.

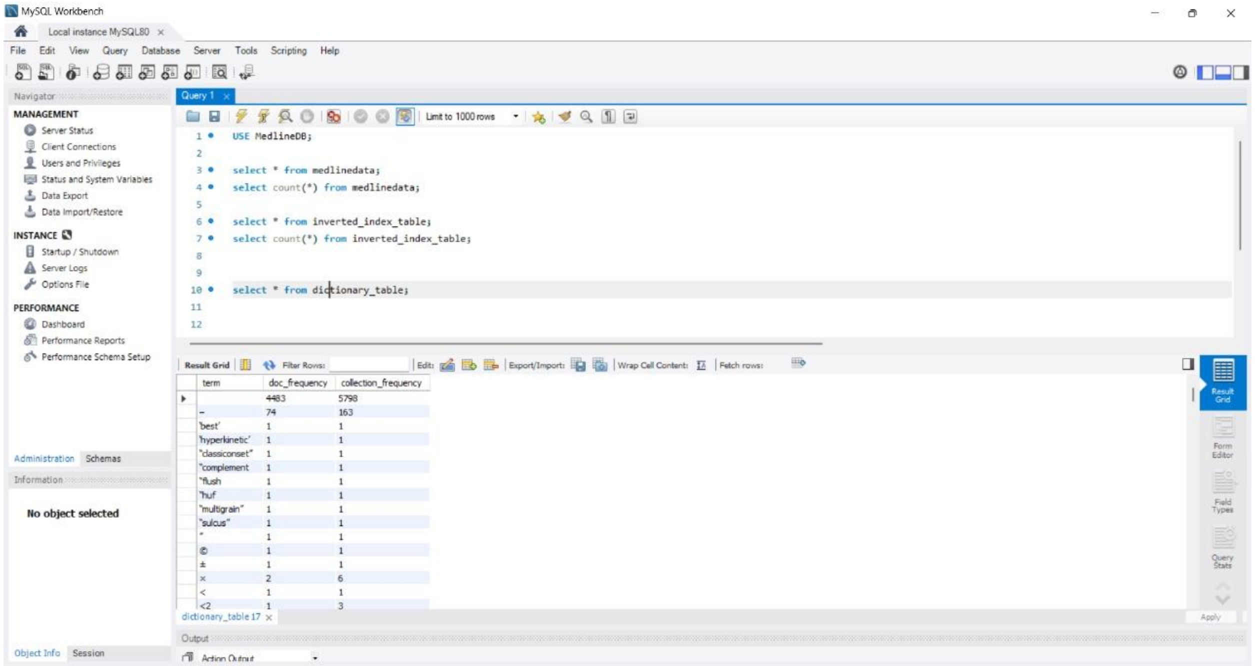
2. **MySQL (Processed Data and Inverted Index)**:
   a. **Table: Documents**:
      i. **Fields**:
         1. ID: Unique identifier.
         2. TITLE: Title of the document.
         3. LINK_TO_MEDLINE: URL link.
         4. ARTICLE_DATA: Processed and cleaned content.
         5. FILEPATH: Metadata information.
   b. **Table: INVERTED_INDEX_TABLE**:
      i. **Fields**:
         1. DOC_ID: Document in which term is present.
         2. TERM: The word itself.
         3. TERM_FREQ: Frequency of the term in the particular document.
   c. **Table: DICTIONARY_TABLE**:
      i. **Fields**:
         1. TERM,DOC_FREQUENCY, TERM_FREQUENCY: Relevant information for the mapping of terms and their frequency within documents.

Knowledge Base Features

- **Efficient Querying**:
  - Designed with the goal of rapid information retrieval.
- **Normalization**:
  - Data normalization was done to eliminate redundancy.

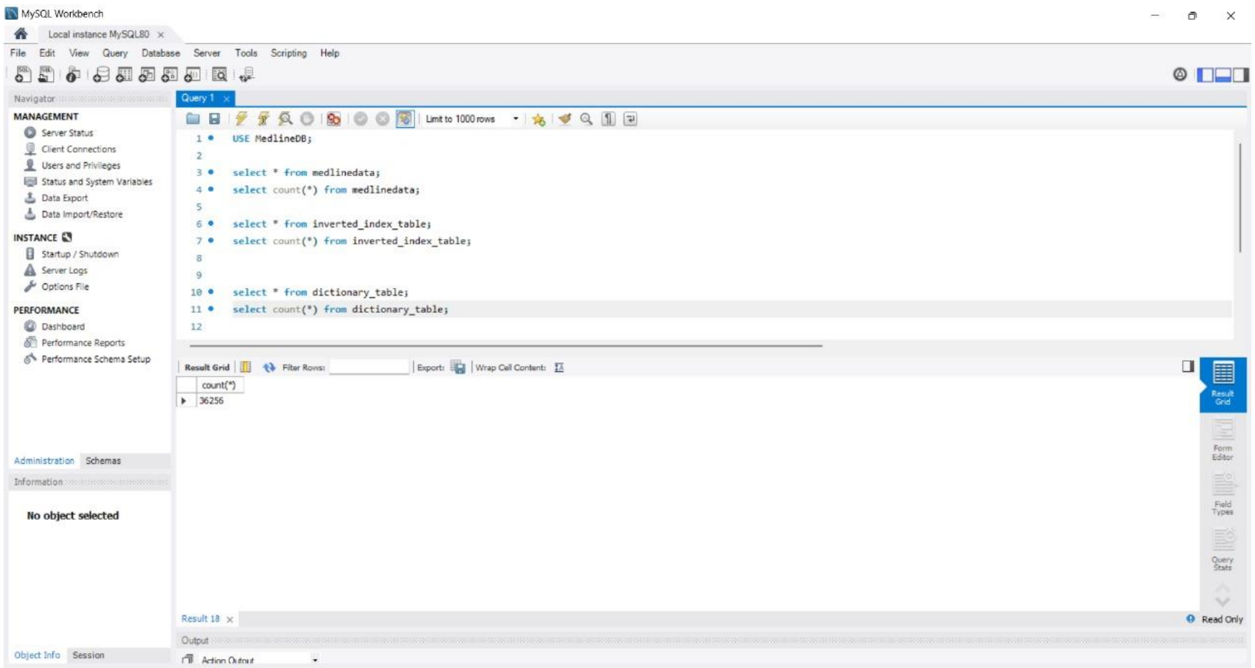
Dictionary Table Snapshot

- **Description**: Shows the DICTIONARY_TABLE in MySQL containing key terms and their associated document frequency and collection frequency.



Dictionary Table Count Snapshot

- **Description**: Displays the total count of terms available in the dictionary_table (36,256), indicating the scale of the data processed.



8. Scoring/Ranking Algorithm

The following Information Retrieval algorithms were implemented in the project:

8.1 Term Frequency-Inverse Document Frequency (TF-IDF)

- **Purpose**: To evaluate the importance of a term in a document within a collection.
- **Components**:
  - **Term Frequency (TF)**: Measures how frequently a term appears in a document.
  - **Inverse Document Frequency (IDF)**: Reflects the importance of a term relative to the corpus size.

8.2 Cosine Similarity

- **Purpose**: Calculate similarity between the user's search query and the document vectors.
- **Implementation**: This was used to determine which documents were most relevant to user queries, based on their TF-IDF vectors.

9. The Problems/Errors Encountered and Your Resolutions

Problems Faced

1. **XPath Issues During Web Scraping:**
   - **Problem:** During the web scraping process, the XPath used initially (articleDataXpath1 and articleDataXpath2) worked for certain elements but resulted in null data for others after an update on the website.
   - **Resolution:** After extensive debugging, it was discovered that the data was located in additional classes. The XPaths were updated accordingly, which resolved the issue and enabled consistent data extraction.
   - **Description**: The screenshot shows the XPaths (articleDataXpath1 and articleDataXpath2) used to identify the main content elements during scraping.
2. **Inefficient Database Query Execution:**
   - **Problem:** The initial system executed approximately 4,000 SQL queries for every user query, where each query corresponded to one document in the knowledge base. This process took around 60-75 minutes to retrieve results.
   - **Resolution:** A more efficient query approach was devised. Instead of querying each document individually, a consolidated query was developed that retrieved all relevant data in a single pass. This significantly reduced the response time and improved system performance.

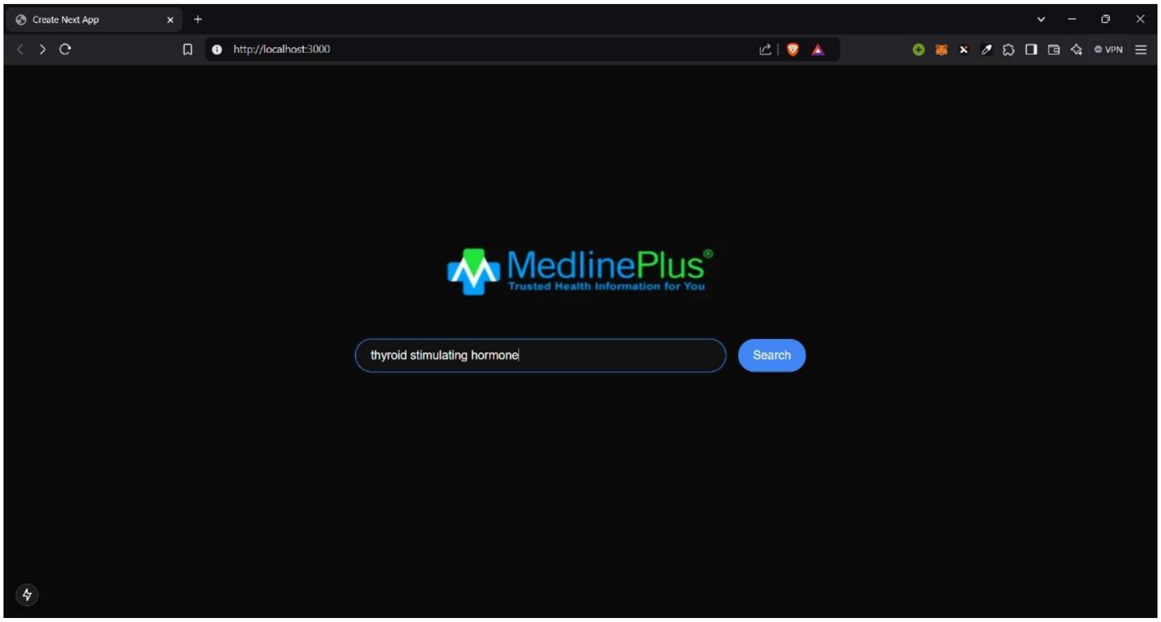
11. System Demo

User Interface

- **Frontend Framework**: Built using Next.js for a responsive and interactive user interface.

Medline UI Snapshot

- **Description**: Shows the initial interface where users can enter search queries.

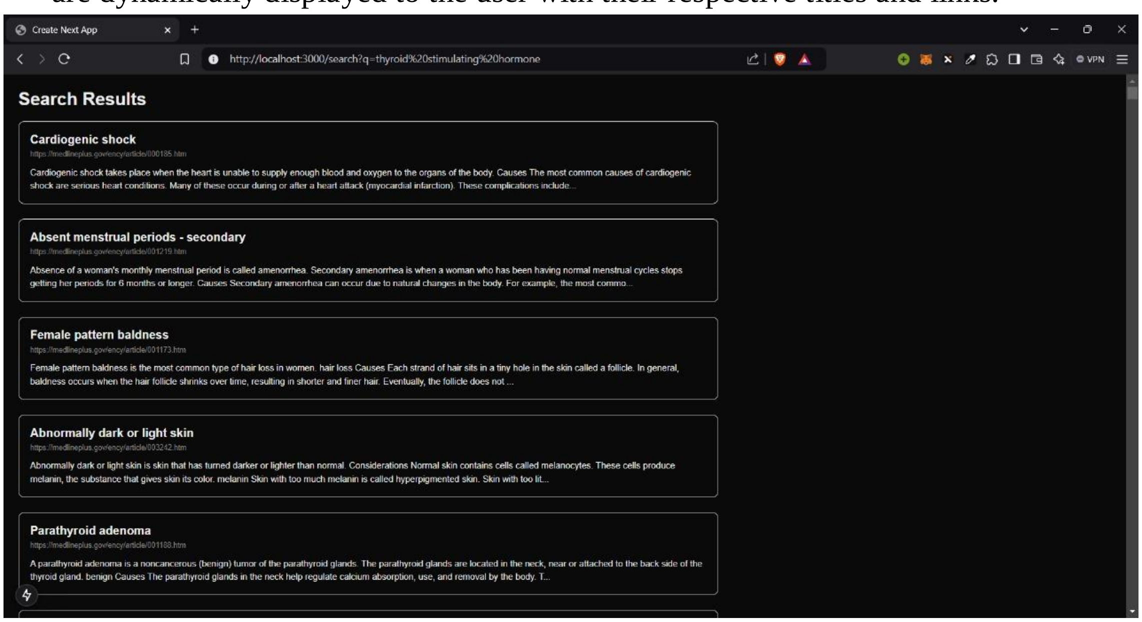


Search Demo

- **Process**:
    - Users entered a search term.
    - The backend processed the query using the inverted index and returned the top relevant results.

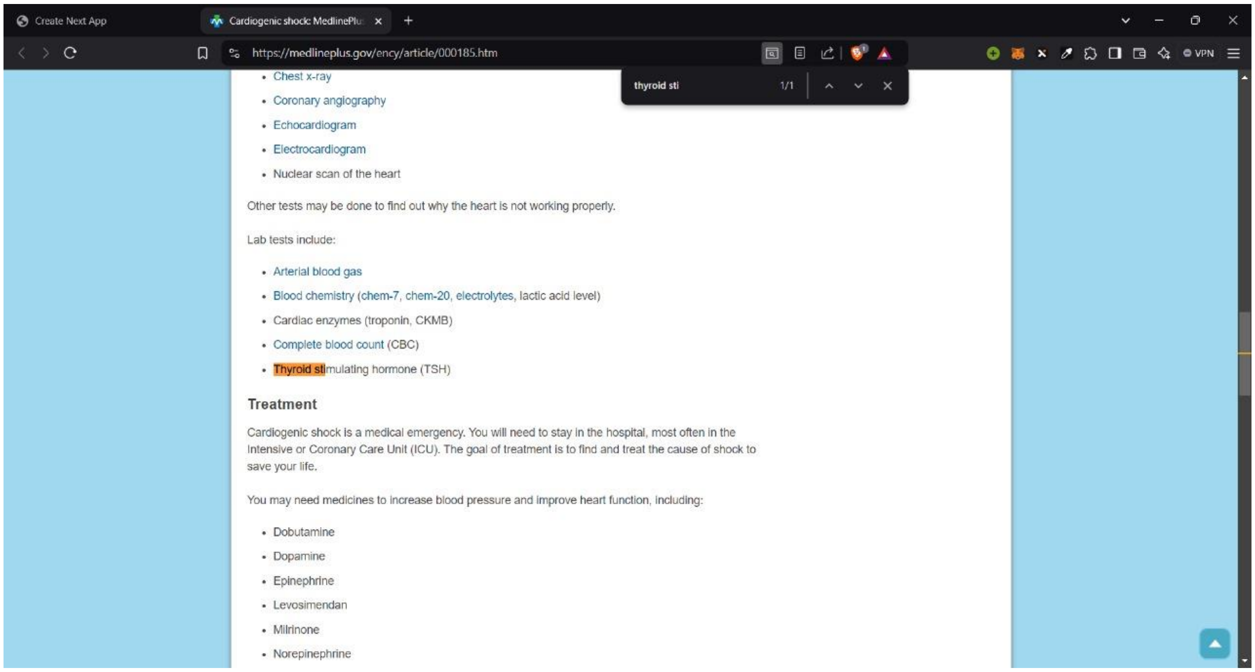
Search Results Snapshot

- **Description**: Demonstrates the output of a search query on the Medline UI. The results are dynamically displayed to the user with their respective titles and links.



Search Result Extended Link Snapshot

- **Description**: This shows additional information from one of the retrieved articles, providing users with detailed medical information.



Backend Search Logic

- **Backend (route.js)**:

```
import express from "express";
import mysql from "mysql2/promise";

const app = express();
const connection = await mysql.createConnection({ host: "localhost", user: "root", database: "MedLineDB" });

app.get("/search", async (req, res) => {
  const query = req.query.q;
  // Query inverted index logic here
  res.send(results);
});

app.listen(3000, () => console.log("Server running on port 3000"));
```

Backend Code Snippet for Processing Search Queries

This code handles incoming user requests, searches the inverted index, and returns relevant documents, efficiently managing user interactions.

Execution Steps:

Part 1: Backend and Data Collection Setup

1. **Unzip MedLine Search Engine Scraping InvertedIndex.zip** and navigate into it:
   - This zip file contains all necessary backend scripts, libraries, and configurations to build the Medline Search Engine.
2. **Run npm install**:
   - This installs all required packages and libraries (e.g., puppeteer, mysql2, mongodb).
3. **Run node webScraper.js**:
   - This script scrapes all data from the MedlinePlus website and uploads it to MongoDB.
   - Make sure MongoDB is installed and the MongoDB server is running on your system.
   - **Code Overview (webScraper.js)**:

```
 import puppeteer from "puppeteer"
import fs from "fs"

import path from "path"
import { fileURLToPath } from "url"
import { MongoClient } from "mongodb"

const url = "https://medlineplus.gov/encyclopedia.html"
const articlesAZXpath = '//*[@id="az-section2"]/ul/li/a'
const pageArticlesXpath = '//*[@id="index"]/li/a'
const folderName = "MedlineData"
const uri = "mongodb://127.0.0.1:27017"
const dbName = "MedlineDB"
const collectionName = "MedlineData"

const writeAllArticleDetailsToFile = (articles) => {
    const filePath = "./articleHTMLFiles.json"
    const jsonData = JSON.stringify(articles, null, 4)
    fs.writeFileSync(filePath, jsonData, "utf-8")
}

const createFolder = async (folderName) => {
    //  Creates a folder to store all speeches.
    try {
        fs.mkdirSync(folderName, { recursive: true }, (err) => {
            if (err) {
                console.error("Error creating folder:", err)
            } else {
                console.log(`Folder created successfully: ${folderName}`)
            }
        })
    } catch (error) {
        console.log(`error in createFolder: ${error}`)
    }
}

const addArticleDatatoFile = (anchorObj, speech) => {
    const fileName = anchorObj.title
        .replace(/ /g, "_")
        .replace(/[<>:"/\\|'?*]/g, "")
```

```
        .concat(".html")
    const filepath = path.join(folderName, fileName)

    fs.writeFileSync(filepath, speech)

    const fullFilePath = path.join(path.dirname(fileURLToPath(import.meta.url)), filepath)

    return fullFilePath
}

const getPageData = async () => {
    const [articalLinks, pageArticleLinks] = [[], []]
    const browser = await puppeteer.launch({
        headless: true,
    })
    const page = await browser.newPage()

    await page.goto(url, { waitUntil: "domcontentloaded" })
    const allArticles = await page.$$(`::-p-xpath(${articlesAZXpath})`)

    for (let articleTag of allArticles) {
        const articleObj = await page.evaluate((el) => ({ title: el.innerHTML, href: el.href }), articleTag)
        articalLinks.push(articleObj)
    }

    for (let eachArticleLink of articalLinks) {
        await page.goto(eachArticleLink.href, { waitUntil: "domcontentloaded" })
        const allPageArticlesTags = await page.$$(`::-p-xpath(${pageArticlesXpath})`)

        for (let pageArticleTag of allPageArticlesTags) {
            const pageArticleObj = await page.evaluate((el) => ({ title: el.innerHTML, href: el.href }), pageArticleTag)
            pageArticleLinks.push(pageArticleObj)
        }
    }

    for (let pageArticle of pageArticleLinks) {
        await page.goto(pageArticle.href, { waitUntil: "domcontentloaded" })
        const pageHtml = await page.content()
        pageArticle.fileContent = pageHtml
        pageArticle.filePath = addArticleDatatoFile(pageArticle, pageHtml)
        console.log(`Scraped data from ${pageArticle.title}`)
    }
    writeAllArticleDetailsToFile(pageArticleLinks)
    await browser.close()
    return pageArticleLinks
}

const insertData = async (pageData) => {
    const client = new MongoClient(uri, { useNewUrlParser: true })

    try {
        await client.connect()
        console.log("Connected to MongoDB")
        const db = client.db(dbName)
        const collection = db.collection(collectionName)
        const result = await collection.insertMany(pageData)
        console.log(`${result.insertedCount} documents were inserted`)
    } catch (error) {
        console.error("Error inserting data:", error)
    } finally {
        await client.close()
    }
}

const main = async () => {
    createFolder(folderName)
    const pageData = await getPageData()
    await insertData(pageData)
}

await main()
```

- The webScraper.js uses Puppeteer to visit the MedlinePlus website, extract article titles, and save the data in MongoDB.
- writeAllArticleDetailsToFile saves the scraped data to a JSON file for backup.

4. **Run node cleanData.js**:
   - This script cleans the HTML data extracted by webScraper.js.
   - It saves the cleaned data to articleDetails.json.
   - **Code Overview (cleanData.js)**:

```
import puppeteer from "puppeteer"

import fs from "fs/promises"

const articleDataXpath1 = '//*[@class="main"]'
const articleDataXpath2 = '//*[@class="main-single"]'

const writeAllArticleDetailsToFile = async (articles) => {
    const filePath = "./articleDetails.json"
    const jsonData = JSON.stringify(articles, null, 4)
    await fs.writeFile(filePath, jsonData, "utf-8")
}

const main = async () => {
    const browser = await puppeteer.launch()
    const page = await browser.newPage()

    const filePath = "./articleHTMLFiles.json"
    const fileContent = await fs.readFile(filePath, "utf-8")
    const pageData = JSON.parse(fileContent)
    for (let eachJsonObj of pageData) {
        const htmlString = eachJsonObj.fileContent
        await page.setContent(htmlString)
        let [article] = await page.$$(`::-p-xpath(${articleDataXpath1})`)
        if (!article) [article] = await page.$$(`::-p-xpath(${articleDataXpath2})`)

        const articleData = await page.evaluate((element) => {
            const elements = element.querySelectorAll("p, h1, h2, h3, h4, h5, h6, li, a, b, i, u, summary")
            let textContent = ""
            elements.forEach((el) => {
                if (el.textContent !== undefined) textContent += el.textContent.trim() + "\n"
            })
            return textContent
        }, article)
        eachJsonObj.fileContent = articleData
    }
    writeAllArticleDetailsToFile(pageData)

    await browser.close()
}

await main()
```

   - Cleans raw HTML by locating content via XPath (articleDataXpath1 and articleDataXpath2).
   - Stores cleaned content into JSON for structured storage.

5. **Run node uploadToDb.js**:
   - This uploads the cleaned data to the MySQL database (MedLineDB).
   - **Code Overview (uploadToDb.js)**:

```
import { createConnection } from "mysql2/promise"

import fs from "fs/promises"

const host = "localhost"
const user = "root"
const password = "password123"
```

```
const dbName = "MedLineDB"
const tableName = "MedLineData"

const main = async () => {
    const conn = await createConnection({ host, user, password }) // Create a connection to MySQL server

    await createDatabase(conn, dbName)
    await createTable(conn, tableName)
    const filePath = "./articleDetails.json"
    const fileContent = await fs.readFile(filePath, "utf-8")
    const pageData = JSON.parse(fileContent)

    for (let eachJsonObj of pageData) {
        await insertIntoDB(conn, eachJsonObj)
    }

    conn.close()
}

const createDatabase = async (connection, dbName) => {
    // Creates a Database if it does not exist. dbName is already defined at top of the file
    const createDBQuery = `CREATE DATABASE IF NOT EXISTS ${dbName}`
    await connection.query(createDBQuery)
    await connection.changeUser({ database: dbName })
    console.log(`Connected to Database ${dbName}\n`)
}

const createTable = async (connection, tableName) => {
    // Creates a table  if it does not exist. tableName is already defined at top of the file
    const createTableQuery = `CREATE TABLE IF NOT EXISTS ${tableName} (
        ID  INTEGER PRIMARY KEY AUTO_INCREMENT,
        TITLE VARCHAR(100),
        LINK_TO_MEDLINE_ARTICLE VARCHAR(200),
        FILEPATH VARCHAR(250),
        ARTICLE_DATA MEDIUMTEXT);`
    await connection.query(createTableQuery)
    console.log(`Created Table ${tableName}\n`)
}

const insertIntoDB = async (connection, values) => {
    // Inserts the scraped data into the database
    const { title, href, filePath, fileContent } = values
    const insertDBQuery = `INSERT INTO ${tableName} (TITLE, LINK_TO_MEDLINE_ARTICLE, FILEPATH, ARTICLE_DATA) VALUES (?, ?, ?, ?)`

    await connection.execute(insertDBQuery, [title, href, filePath, fileContent])
}

await main()
```

- This script creates a MySQL table called MedLineData and inserts data into it.
- createTable ensures the table is present, while insertIntoDB adds new rows.

6. **Run node buildInvertIndex.js**:
    - This script builds two tables: the dictionary and inverted index.
    - **Code Overview (buildInvertIndex.js)**:

```
import { createConnection } from "mysql2/promise"

import pluralize from "pluralize"
import { lemmatizer } from "lemmatizer"
import stem from "stem-porter"

export const host = "localhost"
export const user = "root"
export const password = "password123"
const createInvertedIndexTable = `CREATE TABLE IF NOT EXISTS INVERTED_INDEX_TABLE(
                            DOC_ID INTEGER,
                            TERM VARCHAR(250),
                            TERM_FREQ INT)`
```

```
const createDictionaryTable = `CREATE TABLE DICTIONARY_TABLE (
                                term VARCHAR(250) PRIMARY KEY,
                                doc_frequency INT,
                                collection_frequency INT
                                ) AS
                                SELECT
                                    term,
                                    COUNT(DISTINCT doc_id) AS doc_frequency,
                                    SUM(term_freq) AS collection_frequency
                                FROM
                                    INVERTED_INDEX_TABLE
                                GROUP BY
                                    term;`
const createPostingTable = `CREATE TABLE POSTING_TABLE (
                                term VARCHAR(250),
                                doc_id INTEGER,
                                term_freq INT
                                ) AS
                                SELECT
                                    term,
                                    doc_id,
                                    term_freq
                                FROM
                                    INVERTED_INDEX_TABLE;`
const updateForeignKey = `ALTER TABLE POSTING_TABLE
                            ADD CONSTRAINT fk_term
                            FOREIGN KEY (term) REFERENCES DICTIONARY_TABLE(term);`

const getMedlineData = async (db) => {
    const result = await db.query(`SELECT ID, ARTICLE_DATA FROM MEDLINEDATA`)
    return result[0]
}

const createTable = async (connection, createTableQuery, tableName) => {
    // Creates a table using createTableQuery. query is  defined at top of the file
    await connection.query(createTableQuery)

    console.log(`Created table ${tableName}`)
}

// Tokenizer for cleaning and processing the text

export const tokenize = (text) => {
    // Step 1: Remove punctuation
    const cleanedText = text.replace(/[.?,:—]/g, " ").replace(/[\/#!$%\^&\*;{}=\-+_`~()"'|\[\]]/g, "")
    // Step 2: Convert text to lowercase
    const tokens = cleanedText.toLowerCase().split(/\s+/)

    // Step 3: Convert plurals to singular using pluralize
    const singularTokens = tokens.map((token) => pluralize.singular(token))
    // Step 4: Convert tokens to their normal form (Stemming)
    const normalizedTokens = singularTokens.map((token) => {
        return stem(token)
    })
    return normalizedTokens
}

export const getTermFrequenciesFromDoc = (tokens) => {
    const frequencies = {}

    for (const item of tokens) {
        // Increment the count if the item exists, or initialize it to 1 if it doesn't
        frequencies[item] = (frequencies[item] || 0) + 1
    }

    return frequencies
}

const insertBatchIntoDB = async (connection, rows) => {
    const insertDBQuery = "INSERT INTO INVERTED_INDEX_TABLE (TERM, DOC_ID, TERM_FREQ) VALUES ?"
    try {
        const formattedRows = rows.map(({ term, docId, frequency }) => [term, docId, frequency])
        await connection.query(insertDBQuery, [formattedRows])
    } catch (err) {
```

```
        console.error("Error inserting batch into DB:", err)
    }
}

const main = async () => {
    // Create a connection to MySQL server
    const medlineDB = await createConnection({ host, user, password, database: "MEDLINEDB" })

    await createTable(medlineDB, createInvertedIndexTable, "INVERTED_INDEX_TABLE")

    const allDocuments = await getMedlineData(medlineDB)

    const tokenizedDocuments = allDocuments.map((eachDocument) => {
        const id = eachDocument.ID

        return {
            id,
            tokens: tokenize(eachDocument.ARTICLE_DATA),
        }
    })

    for (const eachDocument of tokenizedDocuments) {
        const { id, title, tokens } = eachDocument
        const termFrequency = getTermFrequenciesFromDoc(tokens)

        // Prepare term frequencies for batch insertion
        const batchValues = Object.entries(termFrequency).map(([term, frequency]) => ({
            docId: id,
            term,
            frequency,
        }))

        // Insert all rows for the document in a single query
        await insertBatchIntoDB(medlineDB, batchValues)

        console.log(`Inserted all terms for document ID: ${id}`)
    }

    await createTable(medlineDB, createDictionaryTable, "DICTIONARY_TABLE") // Subquery calculates the doc frequency and collection frequency
    await createTable(medlineDB, createPostingTable, "POSTING_TABLE")
    await medlineDB.query(updateForeignKey)
    await medlineDB.close()
}

await main()
```

Part 2: Frontend Setup for Search Engine UI

1. **Unzip medline-search-engine.zip** and navigate into the extracted folder:
    - This zip file contains the frontend files needed to create the search engine web interface.
2. **Run npm install**:
    - Installs all required packages and libraries for the Next.js application.
3. **Run npm run dev**:
    - Starts a local development server for the web application.
4. **Navigate to http://localhost:3000**:
    - You will see the Medline Search Engine interface where you can search for medical articles.

TF-IDF/Cosine Similarity Logic: **route.js**:

```
import { createConnection } from "mysql2/promise";

import pluralize from "pluralize";
import stem from "stem-porter";
import { NextResponse } from "next/server";

const host = "localhost";
const user = "root";
const password = "password123";

// Tokenizer for cleaning and processing the text
export const tokenize = (text) => {
    // Step 1: Remove punctuation
    const cleanedText = text.replace(/[.?,:—]/g, " ").replace(/[\/#!$%\^&\*;{}=\-+_`~()"'|\[\]]/g, "");
    // Step 2: Convert text to lowercase
    const tokens = cleanedText.toLowerCase().split(/\s+/);

    // Step 3: Convert plurals to singular using pluralize
    const singularTokens = tokens.map((token) => pluralize.singular(token));
    // Step 4: Convert tokens to their normal form (Stemming)
    const normalizedTokens = singularTokens.map((token) => {
        return stem(token);
    });
    return normalizedTokens;
};

export const getTermFrequenciesFromDoc = (tokens) => {
    const frequencies = {};

    for (const item of tokens) {
        // Increment the count if the item exists, or initialize it to 1 if it doesn't
        frequencies[item] = (frequencies[item] || 0) + 1;
    }

    return frequencies;
};

const cosineNormalize = (vector) => {
    let norm = 0;
    for (let [key, value] of Object.entries(vector)) norm += value * value;
    if (norm === 0) return vector;
    norm = Math.sqrt(norm);
    for (let eachKey of Object.keys(vector)) {
        vector[eachKey] /= norm;
    }
    return vector;
};

const cosineSimilarityScore = (queryVector, docVector) => {
    let score = 0;
    for (let key of Object.keys(queryVector)) {
        score += queryVector[key] * docVector[key];
    }

    return score;
};

const getBestMatchingDocuments = async (userQuery, medlineDB) => {
    const queryTf = getTermFrequenciesFromDoc(tokenize(userQuery));
    const uniqueTokens = Object.keys(queryTf);
    const allDocIds = (await medlineDB.query(`SELECT DISTINCT(DOC_ID) FROM INVERTED_INDEX_TABLE`))[0];
    const docIds = allDocIds.map((eachDoc) => eachDoc.DOC_ID);
    const N = docIds.length;
    const queryVector = {};
    const idf = {};
    for (let token of uniqueTokens) {
        // Calculating query tf
        const frequency_raw = queryTf[token];
        const tf = 1 + Math.log10(frequency_raw);

        // Calculating idf values
```

```
        const query = `SELECT DOC_FREQUENCY FROM DICTIONARY_TABLE WHERE TERM = '${token}'`;
        const result = await medlineDB.query(query);
        let docFrequency;
        if (result[0].length == 0) {
            docFrequency = N;
        } else {
            docFrequency = result[0][0]["DOC_FREQUENCY"];
        }
        idf[token] = Math.log10(N / docFrequency);
        const tfidfWeighting = tf * idf[token];
        queryVector[token] = tfidfWeighting;
    }

    const docVectors = {};

    const queryResult = (
        await medlineDB.query(`SELECT DOC_ID, TERM, TERM_FREQ
                    FROM INVERTED_INDEX_TABLE
                    WHERE TERM in (${uniqueTokens.map((token) => "'" + token + "'")})`)
    )[0];

    for (let eachRecord of queryResult) {
        if (docVectors[eachRecord.DOC_ID] === undefined) {
            docVectors[eachRecord.DOC_ID] = {};
            for (let token of uniqueTokens) docVectors[eachRecord.DOC_ID][token] = 0;
        }
        docVectors[eachRecord.DOC_ID][eachRecord.TERM] = (1 + Math.log10(eachRecord.TERM_FREQ)) * idf[eachRecord.TERM];
    }

    const normalizedQueryVector = cosineNormalize(queryVector);
    const scores = {};
    for (let key of Object.keys(docVectors)) {
        docVectors[key] = cosineNormalize(docVectors[key]);
        scores[key] = cosineSimilarityScore(normalizedQueryVector, docVectors[key]);
    }
    const sortedScores = Object.entries(scores).sort(([, valueA], [, valueB]) => valueB - valueA);
    // const startIndex = p * pagesPerDoc;
    // const endIndex = p * pagesPerDoc + pagesPerDoc;
    const results = sortedScores.map((score) => score[0]);
    return results;
};

export const GET = async (params) => {
    const { url } = params;
    const searchParams = new URL(url).searchParams;
    const queryParams = {};
    for (const [key, value] of searchParams.entries()) {
        queryParams[key] = value;
    }

    const { q } = queryParams;

    try {
        const medlineDB = await createConnection({ host, user, password, database: "MEDLINEDB" });
        const results = await getBestMatchingDocuments(q, medlineDB);

        const links = (await medlineDB.query(`SELECT ID, TITLE, LINK_TO_MEDLINE_ARTICLE, ARTICLE_DATA FROM MEDLINEDATA;`))[0];

        const Objs = links.map((link) => {
            const obj = { ...link };
            obj.ARTICLE_DATA = obj.ARTICLE_DATA.slice(0, 300) + "...";
            return obj;
        });

        const sortedObjects = results.map((pId) => {
            return Objs.find((obj) => obj.ID == pId);
        });

        await medlineDB.close();
        return NextResponse.json(sortedObjects);
    } catch (err) {
        console.log(err.message);
        return NextResponse.json({ error: err.message });
    }
```

```
};
```

By following these execution steps, the Medline Search Engine can be fully set up for data collection, processing, and query interfacing through the web UI. Each step has a corresponding script that has been carefully documented.

Conclusion

The Medline Search Engine project successfully combined various big data, AI, and NLP techniques to build an efficient search platform for medical information. We utilized modern big data tools and machine learning models to build a scalable and interactive solution capable of delivering valuable medical insights to users.