HomeWork-3 Report Template

1. **Crawling [Note down steps implemented for each of the below]**
    a. **URL Canonicalization**
        i. **Parse the URL**: Break down the full URL into its constituent parts using the urlparse function. This includes the scheme, netloc (which includes domain and port), path, parameters, query, and fragment.
        ii. **Lowercase scheme and netloc**: Convert the URL's scheme (e.g., 'HTTP', 'HTTPS') and netloc (the domain and optionally the port) to lowercase to eliminate discrepancies due to case sensitivity.
        iii. **Remove default ports**: If the URL's scheme is 'http' and includes port 80, or if it's 'https' and includes port 443, remove the port number. These are the standard ports for their respective protocols and are therefore redundant.
        iv. **Handle relative URLs**: Ensure the path of the URL begins with a forward slash to convert any relative URLs to absolute ones, establishing a clear and complete path.
        v. **Eliminate fragments:** Remove the fragment portion of the URL, which starts with a '#' symbol, as it usually points to a specific part of a page and is not necessary for the base URL.
        vi. **Consolidate duplicate slashes:** Replace any instance of double slashes ('//') found in the URL's path with a single slash to avoid treating similar URLs as different due to mere formatting variations.
        vii. **Reconstruct the URL:** Reassemble the URL into its whole form using the urlunparse function, piecing together the processed components into a standardized, canonical URL

    b. **Frontier Management**
        i. **Frontier Initialization:** The Frontier class initializes the frontier using a priority queue (queue.PriorityQueue). It also maintains a dictionary (waves) to keep track of URLs based on their wave number.
        ii. **Adding Seeds:** When the frontier is initialized, it adds the seed URLs provided to it. Each seed URL is wrapped in a FrontierEntity object, which encapsulates information such as URL, in-links, and score.
            1. **Score Calculation**:
                a. The FrontierEntity class calculates the score for each URL based on its relevance to the crawling task. The score is determined by factors such as keyword match and in-link count.
                b. Keywords such as "pandemic", "flu", "h1n1", and "swine" contribute to the score if they are present in the URL.

c. The score is calculated as a combination of keyword match score and in-link score, where the keyword match score is calculated using the exponential function to penalize URLs with fewer keyword matches, and the in-link score is calculated based on the number of in-links.
d. The score reflects the quality and relevance of the URL, guiding the frontier in prioritizing URLs for crawling based on their importance to the crawling task.

iii. **Empty Check:** The empty() method checks whether the frontier is empty or not.
iv. **Pop URL:** The pop() method retrieves the next URL from the frontier based on its priority (wave number and score). It returns the URL with the highest priority.
v. **Push URL:** The push() method adds a new URL to the frontier. It updates the waves dictionary and inserts the URL into the priority queue with its wave number and score.
vi. **Update In-links:** The update_in_links() method updates the in-links of a URL in the frontier. This is used when a URL is crawled, and it's in-links need to be updated with new information.

c. Politeness Policy

i. **Initialization of Robots Information:** When processing a URL from a domain for the first time, the crawler initializes the Robots object for that domain. This object reads the robots.txt file of the domain and extracts information such as crawl delay and allowed/disallowed URLs.
ii. Check Robot Rules Before Fetching: Before fetching a URL, the crawler checks the robots.txt rules of the domain to determine whether it's allowed to crawl the given URL.
1. If the robots.txt rules disallow crawling the URL, the crawler skips processing that URL.
iii. **Respect Crawl Delay:** The crawler respects the crawl delay specified in the robots.txt file of each domain. It ensures that the crawling rate complies with the specified delay to avoid overloading the server.
1. If the current time since the last request to the domain is less than the crawl delay, the crawler waits for the remaining time to comply with the politeness policy before making the next request.
iv. **Multithreading with Domain-Level Executor:** To handle multiple domains efficiently, the crawler maintains a separate thread pool (ThreadPoolExecutor) for each domain it encounters.
1. When processing a URL, the crawler checks if there's an existing thread pool for the domain. If yes, it submits the URL processing task to that domain's thread pool. If not, it creates a new thread pool for that domain.

2. This approach ensures that each domain is processed independently, respecting its crawl delay and politeness policy.

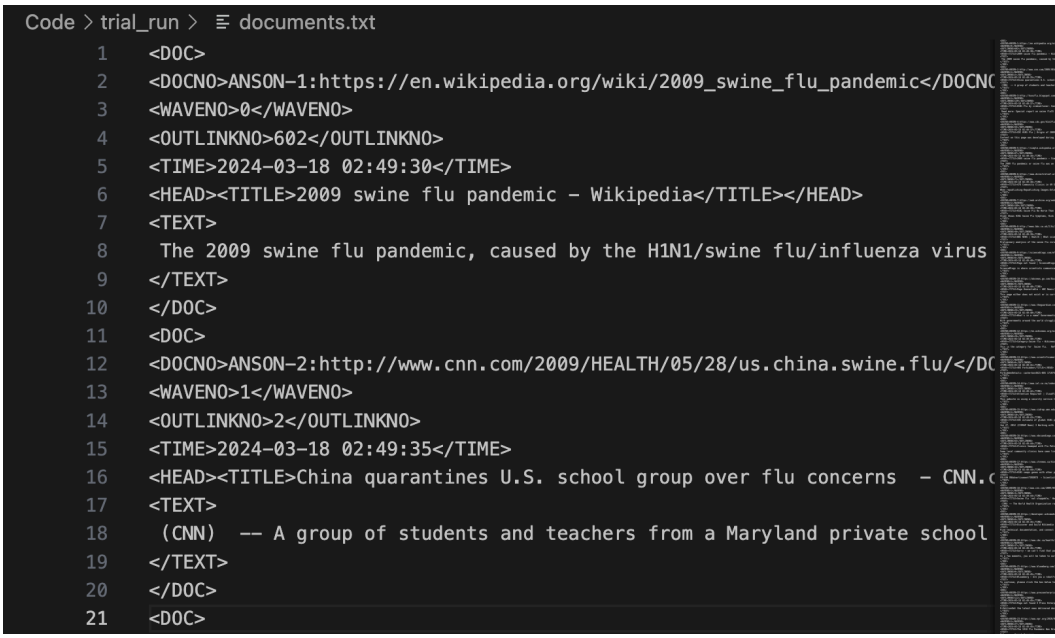
v. **Updating Robots Information:** If the crawler encounters a domain for the first time or if there's a need to refresh the robots.txt rules (e.g., due to timeout or expiry), it updates the robots information for that domain. This ensures that the crawler has the latest information about crawl delay and allowed/disallowed URLs for each domain.

vi. **Handling Errors and Exceptions:** The crawler handles errors gracefully, such as timeouts or exceptions while fetching or parsing the robots.txt file. If an error occurs during the retrieval or parsing of robots.txt, the crawler logs the error and continues processing other URLs.

d. Document Processing

i. **Reading Files:** Read the content of the document.txt, inlinks.json, and outlinks.json files to extract the necessary data for document processing.

ii. Parsing Documents:

1. Split the content of document.txt into individual documents using regular expressions to identify the <DOC> and </DOC> tags.
2. Extract relevant information such as DOCNO, WAVENO, OUTLINKNO, TEXT, and TITLE from each document using regular expressions.
3. 

4. Create lists to store the extracted information for each document.

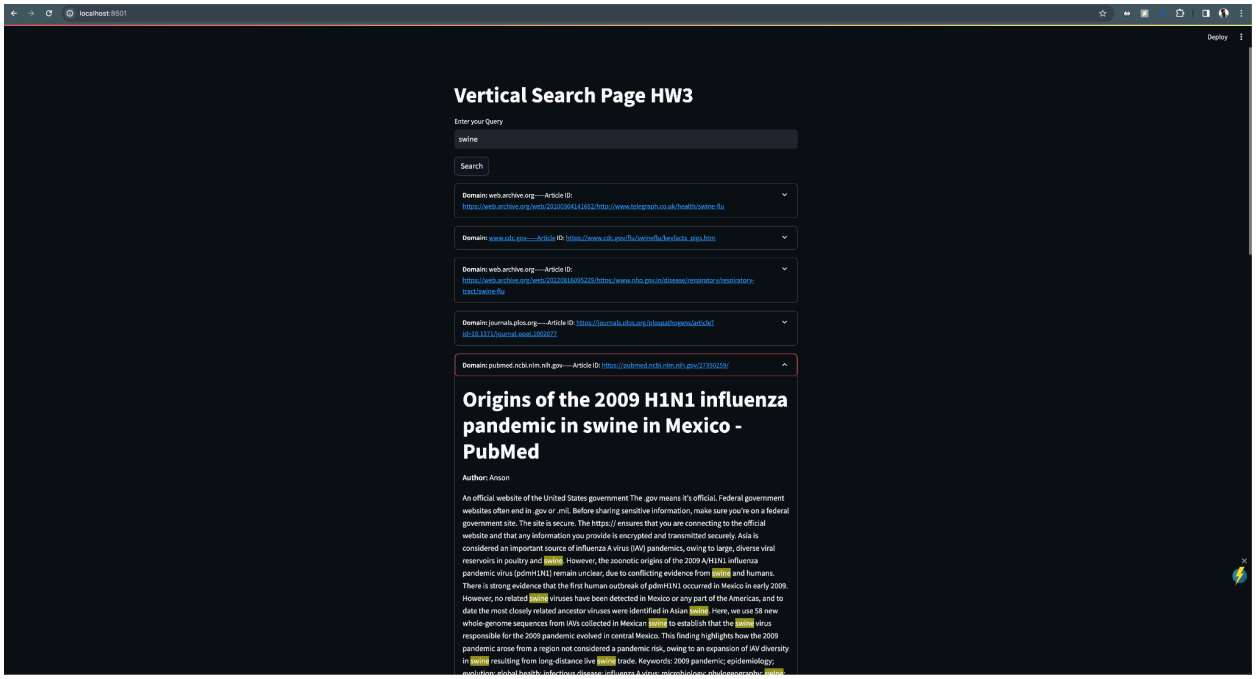
iii. Text Preprocessing:

1. Load stop words from the stoplist.txt file to be used in text preprocessing.

2. Define Elasticsearch index configurations specifying settings for text analysis, including stop words removal, stemming, and lowercasing.

2. Vertical Search

a. Add a Screenshot of your Vertical Search UI



b. Explain briefly how you implemented it.

i. We used the Streamlit app functions as a vertical search interface that queries an Elasticsearch index to display relevant search results for a user's input query. It begins by prompting the user to input their search term. Upon entering a query and initiating a search, the app communicates with an Elasticsearch instance using defined access credentials.

ii. The Elasticsearch search method is invoked to execute a multi_match query against both the title and content fields of documents within the 'crawler' index, aiming to find matches with the user's query and limiting the results to 25 documents.

iii. We used the Streamlit app functions as a vertical search interface that queries an Elasticsearch index to display relevant search results for a user's input query. It begins by prompting the user to input their search term. Upon entering a query and initiating a search, the app communicates with an Elasticsearch instance using defined access credentials.

iv. The Elasticsearch search method is invoked to execute a multi_match query against both the title and content fields of documents within the

'crawler' index, aiming to find matches with the user's query and limiting the results to 25 documents.

3. **Extra Credits Done [Note done what was done for each extra credit]**
    a. **EC1**: Crawled 180,000 documents as a team (120,000 as individual)
    b. **EC4:** Added annotation on top of vertical search functionality