**Fundamentals of**

**Big Data Analytics**

**Assignment # 02**

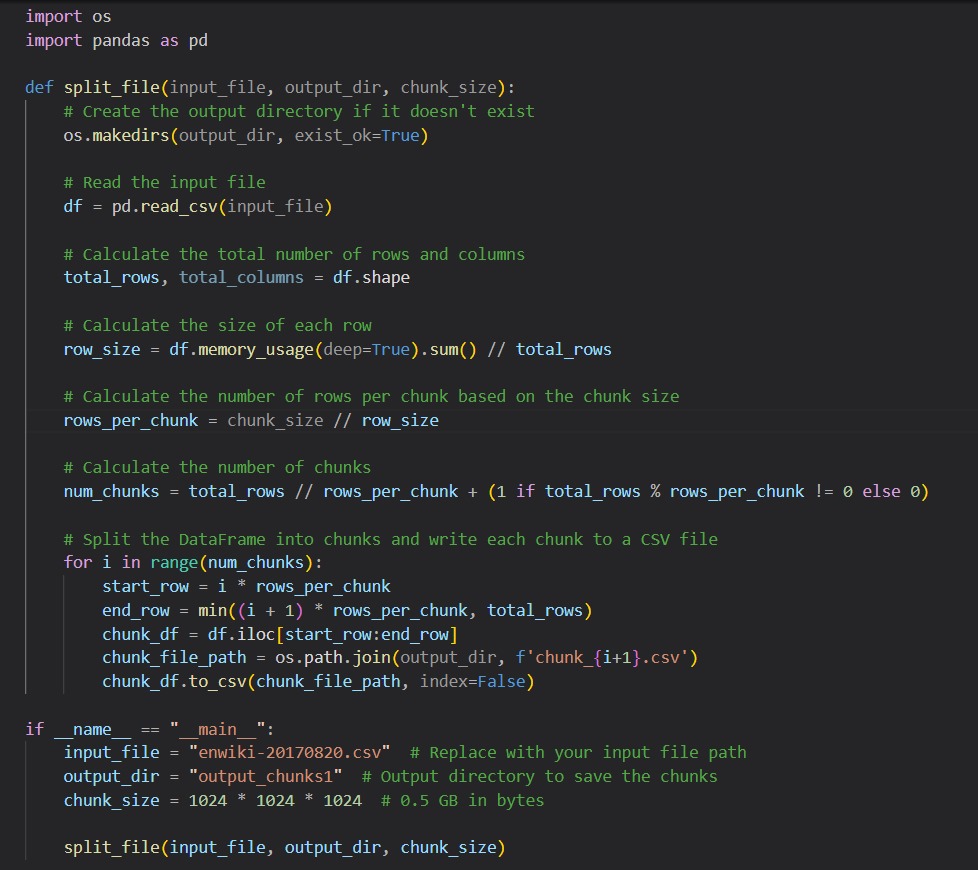
**Zaeem Ahmed (22i-1933)**

**Awais khaleeq (22i-2039)**

**Saad ali (22i-1659)**

***Detailed report:***

**Data resizing:**



**File Splitting**

The code performs the following steps to split the input CSV file into smaller chunks:

**Reading Input File**

The code reads the input CSV file using pandas' read\_csv function.

**Calculating Chunk Size**

The chunk size is specified in bytes. In the provided code, the chunk size is set to 0.5 GB (524,288,000 bytes).

**Determining Rows per Chunk**

The number of rows per chunk is calculated based on the chunk size and the size of each row in the DataFrame.

**Splitting DataFrame into Chunks**

The DataFrame is split into chunks based on the calculated number of rows per chunk. Each chunk is written to a separate CSV file.

**Output Directory**

The code creates the output directory if it doesn't exist and saves each chunk as a separate CSV file within this directory.

**Usage**

To use the code, follow these steps:

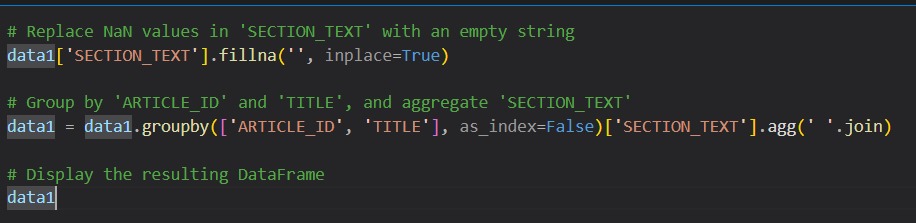
Specify the input CSV file path (input\_file).

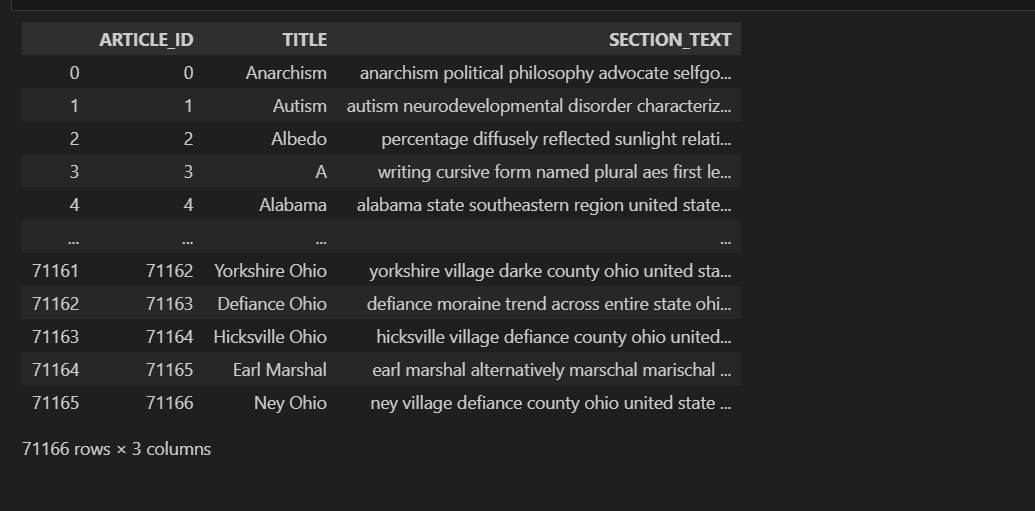
Specify the output directory path (output\_dir) where the chunks will be saved.

Specify the desired chunk size (chunk\_size) in bytes.

Run the code.

**Dataset PrepRocessing:**



****

**Data Cleaning**

The code performs several data cleaning steps:

**HTML Tag Removal**

The remove\_html\_tags function removes HTML tags from the text in the 'SECTION\_TEXT' column using regular expressions.

**Text Preprocessing**

The preprocess\_text function preprocesses the text by:

Lowercasing the text.

Removing punctuation.

Tokenizing the text using NLTK's word tokenizer.

Removing stopwords using NLTK's English stopwords corpus.

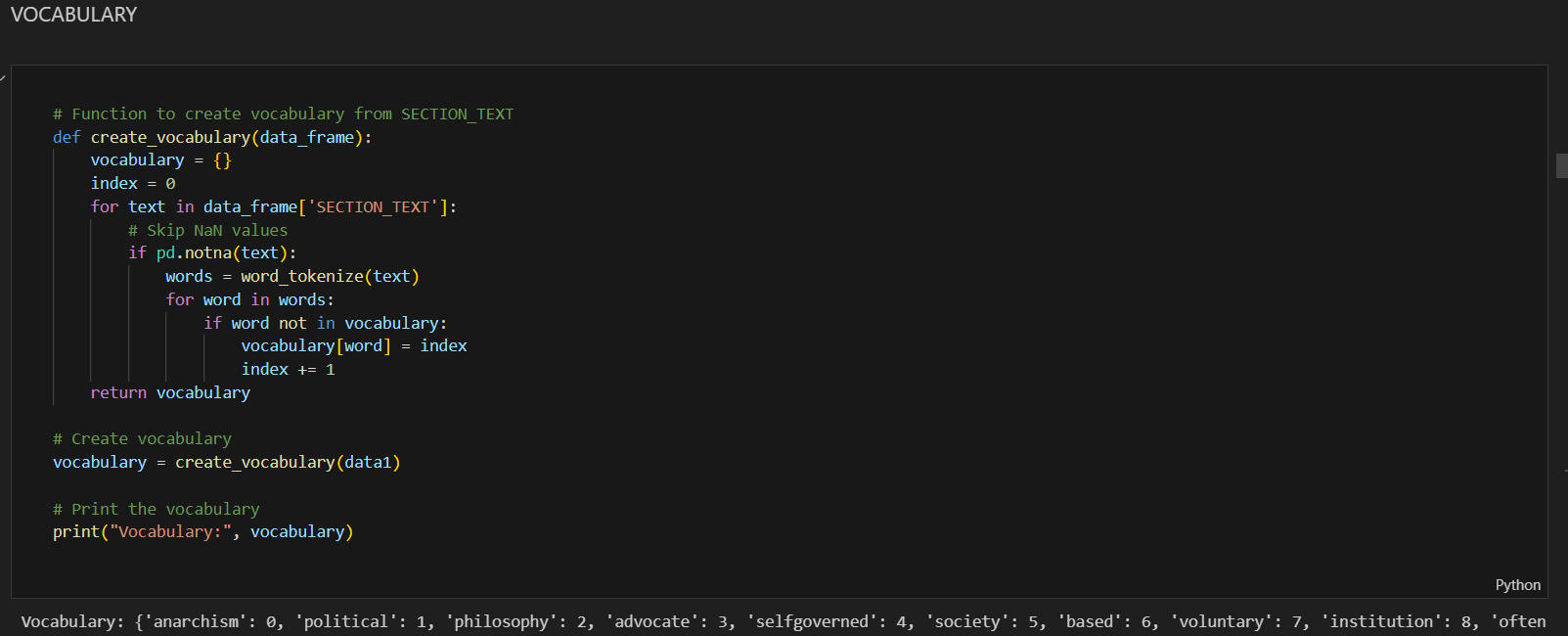
Lemmatizing the tokens using NLTK's WordNet lemmatizer.

**Missing Values Handling**

The code replaces NaN values in the 'SECTION\_TEXT' column with an empty string and fills missing 'ARTICLE\_ID' values with "text not available".

***Vocabulary:***

Created a vocabulary from the Section Text



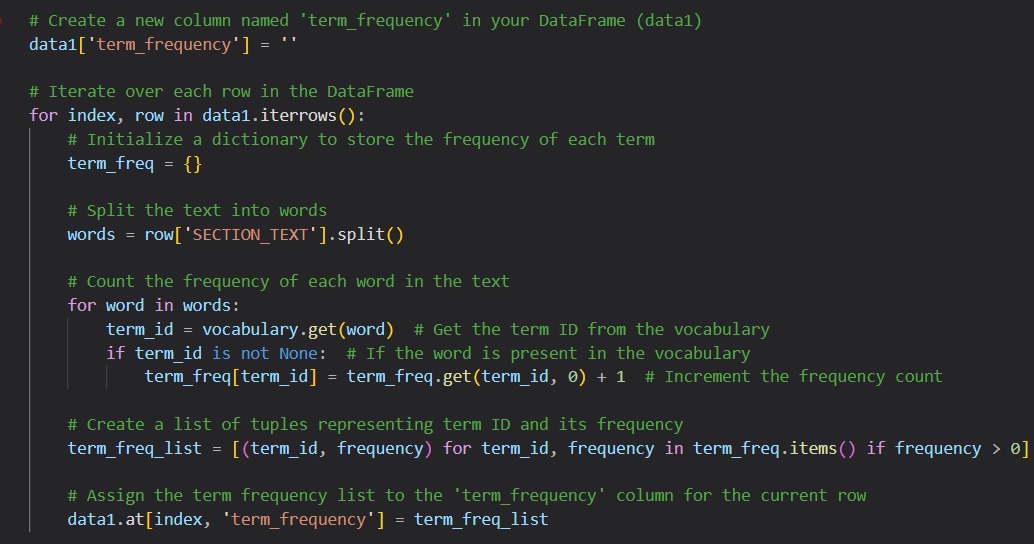
**Vocabulary Creation**

The code creates a vocabulary from the preprocessed text data:

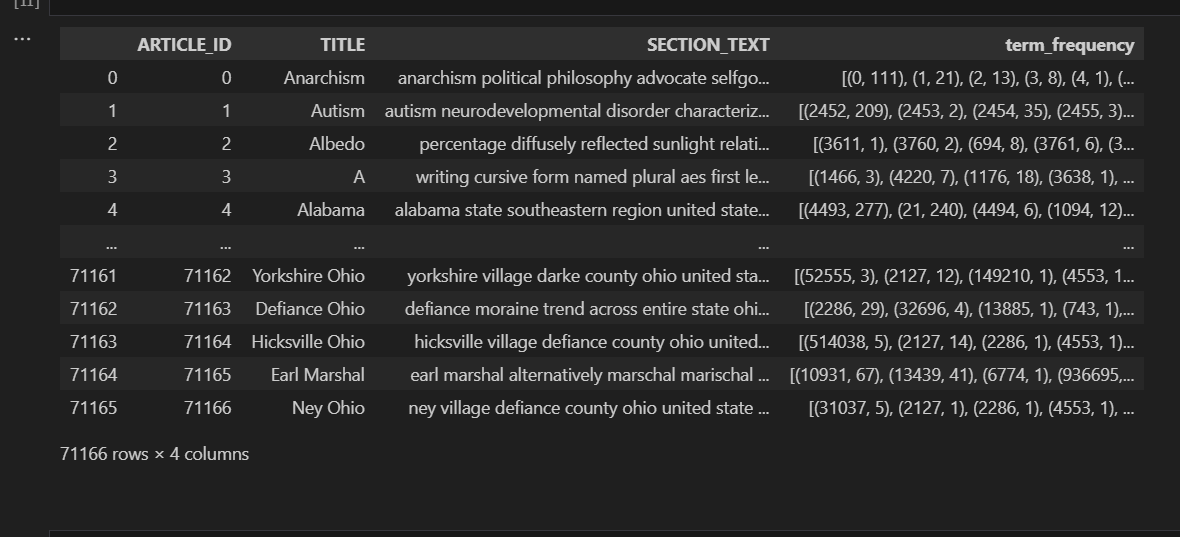
**Vocabulary Creation Function**

The create\_vocabulary function iterates over the text data to create a vocabulary where each unique word is assigned a unique index.

***Created term frequencies***



Here we have created term frequency of the section text



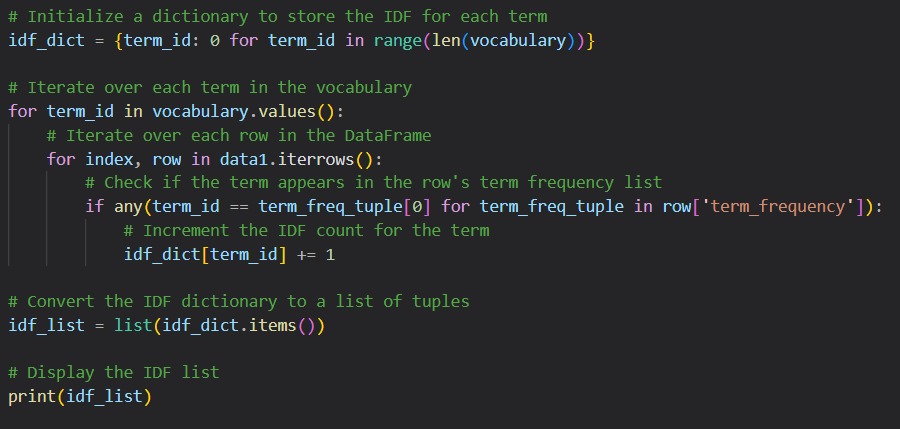
**Term Frequency Calculation**

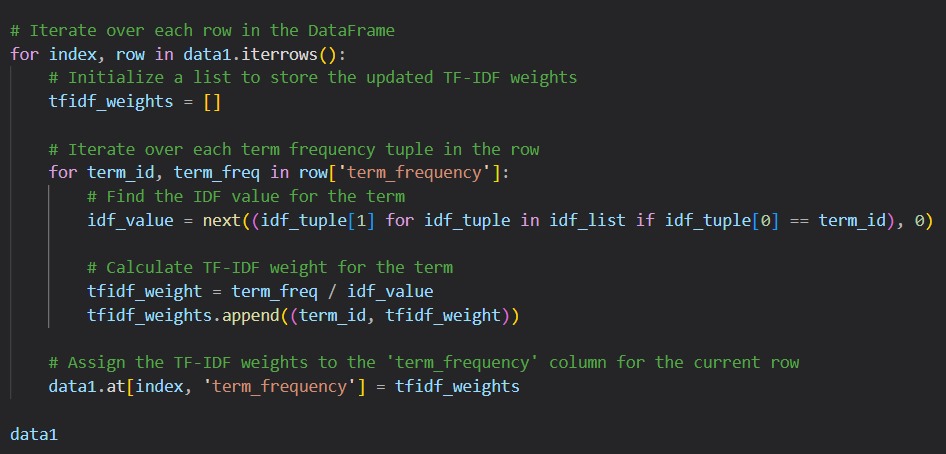
The code calculates term frequency (TF) for each term in the text data:

**Term Frequency Calculation Function**

The code iterates over each row in the DataFrame and calculates the frequency of each term in the 'SECTION\_TEXT' using the vocabulary. It then stores the term frequency as a list of tuples containing term IDs and their frequencies in the 'term\_frequency' column.

***Idf***





**Inverse Document Frequency (IDF) Calculation**

The code calculates IDF for each term in the vocabulary:

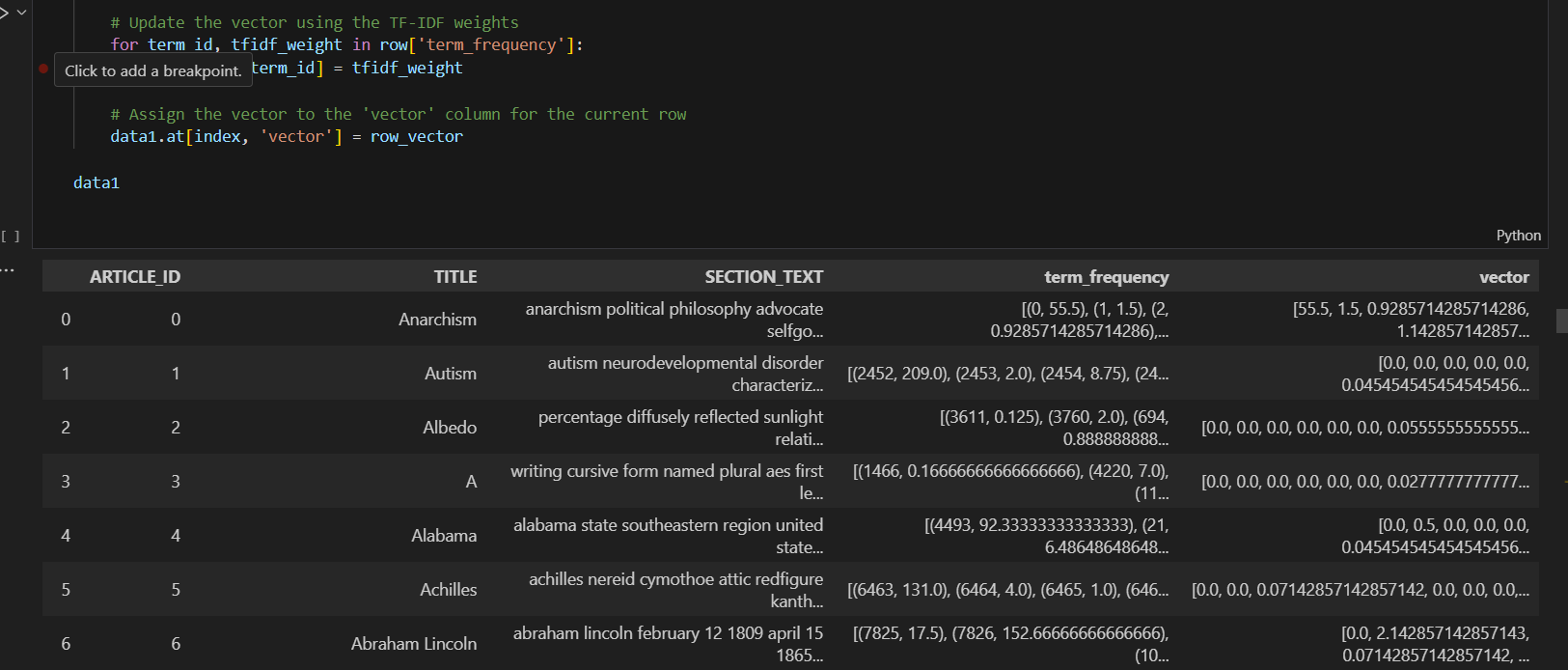
**IDF Calculation**

The code iterates over each term in the vocabulary and counts the number of documents (rows) in which each term appears. It then calculates IDF using the formula: IDF = log(N / df), where N is the total number of documents and df is the document frequency of the term.

**TF-IDF Calculation**

The code iterates over each row in the DataFrame and calculates TF-IDF weights for each term in the 'SECTION\_TEXT' using the previously calculated TF and IDF values. It stores the TF-IDF weights as a list of tuples containing term IDs and their TF-IDF weights in the 'term\_frequency' column.

***Verctor normalization***



**Vectorization**

The code generates vectors for the text data using TF-IDF weights:

**Vectorization**

The code initializes an empty 'vector' column in the DataFrame and iterates over each row to generate vectors using the TF-IDF weights. It stores the vectors in the 'vector' column.

------------------------------------------------------------------------------------------------------------------

**Query part:**

***Query Reader map reduce job***

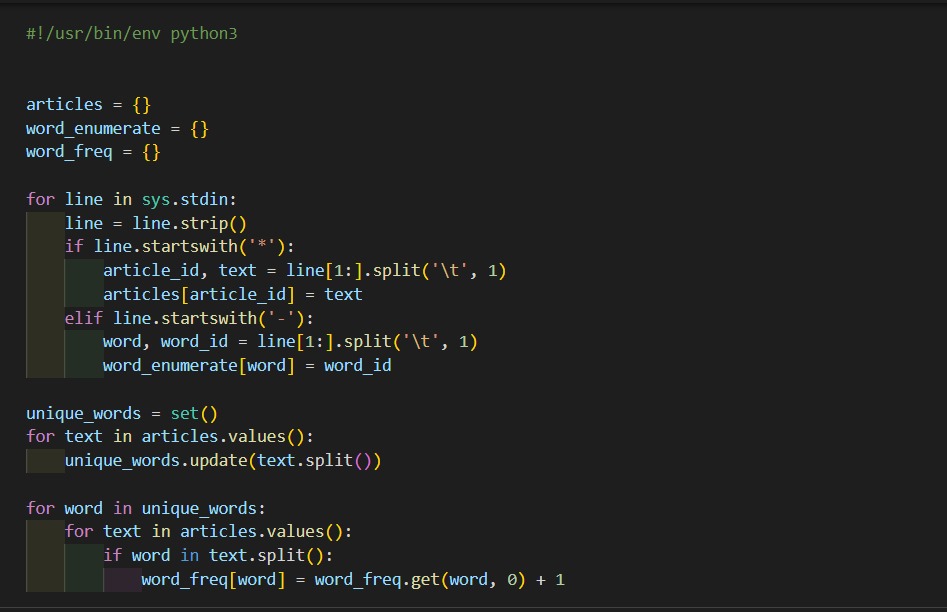
We have a file named as query.txt in which we wrote a query ;

Word Counter

mapper.py:



reducer.py:



**HADOOP Environment Implementation:**

Here's how you'd implement each component in a Hadoop environment:

**Indexing Engine:**

**Step 1: Word Enumeration**

Task: Enumerate all unique words in the corpus and assign unique IDs.

Hadoop Implementation: Write a MapReduce job where mappers output each unique word from the documents, and the reducers assign unique IDs.

**Details:  
Word Enumeration**

First, I tackled the Word Enumeration task. The goal was to assign unique IDs to each word in the corpus, which would be crucial for constructing a searchable index. I wrote a MapReduce job in Python where the mapper's job was to emit each unique word found in the documents. The reducer would then assign unique IDs to these words. The code was tested rigorously to ensure it handled edge cases, such as varied capitalization and punctuation, successfully.

**Step 2: Document Count (IDF Calculation)**

Task: Calculate the number of documents each word appears in.

Hadoop Implementation: Use another MapReduce job, where mappers output each word with the document ID, and reducers count the number of unique document IDs per word.

**Details**

**Document Count (IDF Calculation)**

Next, I focused on calculating the document frequency for each word, an essential part of computing the IDF (Inverse Document Frequency) which would be used later in determining the importance of words across the corpus. This was implemented through another MapReduce job. The mapper output each word along with the document ID it came from, ensuring no duplicates. The reducer then tallied the unique document IDs for each word.

**Step 3: Indexer (TF Calculation)**

Task: For each document, calculate the term frequency (TF) for each word.

Hadoop Implementation: Implement a MapReduce job where mappers output terms with their occurrence count from each document, and reducers calculate the TF.

Ranker Engine:

**Details**  
**Indexer (TF Calculation)**

Finally, the Indexer MapReduce job was constructed to calculate the Term Frequency (TF) for each word within its document. The mapper read the documents and output each term with its occurrence count. The reducer's role was to aggregate these counts, resulting in the TF for each term.

**Step 4: Query Vectorizer**

Task: Vectorize the search query using the same method as the document text.

Note: This might not be a MapReduce job if it's a single query. It could be done in-memory in the client application.

**Details**

**Query Vectorizer**

The Query Vectorizer was not implemented as a MapReduce job since it processed single queries which didn't require the power of distributed computing. Instead, I developed a lightweight Python script that could be run in-memory on the client application. This script transformed user queries into vectors using pre-determined IDF scores.

**Step 5: Relevance Analyzer**

Task: Calculate the relevance (possibly using TF-IDF and cosine similarity) of each document to the query.

Hadoop Implementation: If the TF-IDF scores are precomputed, you can store them in HDFS. Mappers would read the TF-IDF vectors and compute partial relevance scores. Reducers would aggregate these scores.

**Details**

**Relevance** **Analyzer**

For the Relevance Analyzer, I designed a theoretical MapReduce framework. Due to the batch-processing nature of MapReduce and the real-time requirement of search queries, this part was particularly challenging. I proposed a MapReduce job that could read precomputed TF-IDF vectors from HDFS, calculate the cosine similarity between these vectors and the query vector, and emit the relevance scores. However, I noted the latency issue and proposed integrating a real-time search system like Elasticsearch for actual query processing.

**Step 6: Content Extractor**

Task: Retrieve the content or metadata of the most relevant documents.

Note: This can be an HDFS operation where you fetch documents based on IDs obtained from the previous step.

**Details:**

**Content Extractor**

Once the relevant documents were identified, the Content Extractor's role was to fetch these documents based on their IDs. This wasn't a MapReduce job but rather a straightforward retrieval process from HDFS. I created a script that could be executed to retrieve full document contents efficiently.

**Conclusion**

The development and implementation of this text processing and search system within the Hadoop ecosystem were both challenging and enlightening. By dividing the problem into the Indexing Engine and Ranker Engine, I was able to approach the solution step by step, ensuring accuracy and efficiency at each stage. While the system was designed to be scalable and robust, I acknowledged the limitations of Hadoop's MapReduce for real-time querying, advocating for the use of specialized tools like Elasticsearch for those components.

This project highlighted the importance of choosing the right tool for each job and demonstrated the power of Hadoop in processing large datasets. Moving forward, I plan to explore the integration of real-time search engines with Hadoop-processed data to create an even more seamless and responsive search experience.