NATIONAL UNIVERSITY
of Computer & Emerging Sciences

Fundamental of Big Data Analytics (DS2004)

Assignment #2

Deadline: 31-03-2024

Course Team		
Dr. Muhammad Ishtiaq	Course Coordinator	m.ishtiaq@nu.edu.pk
Ms. Kainat Iqbal	Course Instructor	kainat.iqbal@isb.nu.edu.pk
Ibrahim Bin Umair	Teaching Assistant	i200567@nu.edu.pk
Muhammad Huzaifa Khan	Teaching Assistant	i212689@nu.edu.pk
Hashim Muhammad Nadeem	Teaching Assistant	i211675@nu.edu.pk

Assignment Guidelines:

- This assignment is meant to be completed as a team.
- Only **one team member** should submit the assignment. Multiple or duplicate submissions will lead to a deduction of marks.
- To submit the assignment, please create a GitHub repository and upload your source codes along with your result file. **Ensure the repository is public.** Share **only** the repository link on Google Classroom for submission. Do *not* include any additional materials on Google Classroom, as marks will be deducted for doing so.
- Additionally, you must provide a comprehensive report detailing your work and findings. This report should be written within the **README.md** file located in your GitHub repository **(Example)**. Avoid submitting any separate document files.
- You have the flexibility to utilise any programming language that is compatible with Apache Hadoop, such as Python, Java, or C/C++.
- You may refer to online sources such as websites and/or ChatGPT, but make you can explain the code you have written.
- To earn bonus marks, please use comments and adhere to correct **PEP 8** coding conventions.

Plagiarism Policy:

Plagiarism is a grave academic offense that can severely undermine your academic integrity and reputation. Any instance of a student found to have plagiarised their assignment, whether from a peer or external source, will be subject to strict consequences. This may result in a zero score for the current or all assignments, or in severe cases, even a failure in the course. Furthermore, all instances of plagiarism will be promptly reported to the Disciplinary Committee for further action.

Dataset Description:

For this assignment, we'll be using a portion of the English Wikipedia dump provided by Wikimedia, which includes around 5 million Wikipedia articles.

- **ARTICLE_ID:** `int`, serves as the identifier for each unique title.
- **TITLE:** `str`, denotes the titles of articles.
- **SECTION_TITLE:** `str`, represents subsection titles from each article.
- **SECTION_TEXT:** `str`, contains text from each subsection.

You can obtain the subset of the dataset in comma-separated values (CSV) format from the following link: https://drive.google.com/file/d/1lGVGqzF5CNWaoV-zoz8_mlThvHwMgcsP/view?usp=sharing

Developing:

The first step is to thoroughly review Apache's **MapReduce Tutorial**. It elucidates the capabilities of MapReduce and includes examples with configurations. For this task, focus solely on the **ARTICLE_ID** and **SECTION_TEXT** columns, ensuring that basic cleaning of the text is conducted to achieve precise and accurate results.

Organising the Project:

Teams should organise before starting the project, discuss specifics, assign modules, and responsibilities. While no strict guidelines exist, we recommend following best practices. A **public GitHub repository should be created**, with incremental commits made to track progress. Avoid last-minute uploads and utilise version control for collaborative development benefits.

Testing:

The processing time can be substantial depending on the dataset size. To accelerate the development process, it's advisable to conduct local testing on a smaller dataset.

Using Your Search Application:

You need to offer a method to run the indexer on a specified set of inputs through provided arguments, and another method to execute a query. The query, along with the desired length of the output list of relevant documents, should be specified in the list of arguments. For instance:

```
hadoop jar /path/to/jar Indexer /path/to/input/files arg1 arg2 arg3
```

```
hadoop jar /path/to/jar Query arg1 arg2 arg3 10 "query text"
```

The results of the query job should be saved onto the disk.

Developing a Naïve Search Engine Utilising MapReduce Technology

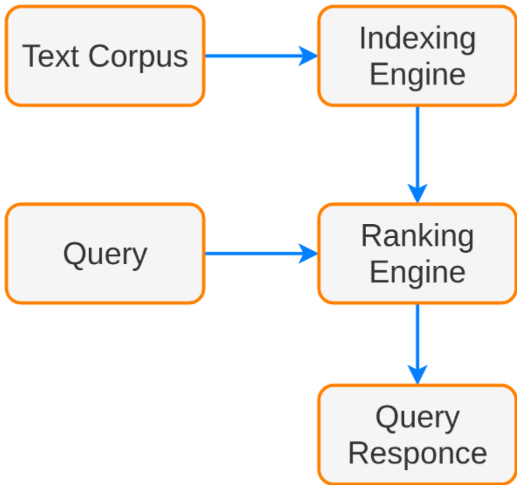
The objective this time is to develop a basic search engine. Search engines were pioneers in addressing the challenge of handling vast amounts of data while maintaining low-latency responses. Consider an average search engine with millions of indexed documents. It must swiftly process hundreds to thousands of queries per second and deliver a list of the most relevant documents in under a millisecond.

The challenge of locating relevant information stands as a critical issue within the realm of information retrieval, comprising two interconnected tasks:

- Document Indexing
- Query Processing

While the latter task prioritises swift response times, indexing can be accomplished offline and aligns more closely with the concept of batch processing. Hadoop's MapReduce framework offers a viable solution for indexing extensive text corpora that exceed the capacity of a single machine. For this assignment, you will undertake a basic implementation of both tasks using the MapReduce paradigm.

Refer to the diagram below for the schematic representation of such a search engine:



Before delving into the intricacies of MapReduce programming, it's beneficial to first understand the fundamental approach to information retrieval.

Basics of Information Retrieval for Text:

The most common task in information retrieval involves retrieving textual information. When a user submits a query, the ranking engine is responsible for calculating the most relevant documents within its collection. To accomplish this, the engineer must establish the format for representing both documents and queries, and specify the measure of relevance between a query and a given document. One of the simplest information retrieval models is the TF/IDF Vector Space model.

Vector Space Model for Information Retrieval:

To help you understand the vector space model better, let’s create a small collection of three documents:

ID	Content
1	I wonder how many miles I’ve fallen by this time?
2	According to the latest census, the population of Moscow is more than two million.
3	It was a warm, bright day at the end of August.
4	To be, or not to be?
5	The population, the population, the population

To explain the vector space model, we’ll introduce three key ideas: vocabulary, Term Frequency (TF), and Inverse Document Frequency (IDF).

Term:

A term refers to a distinct word.

Vocabulary:

The vocabulary consists of all the unique terms found in the corpus. In the example provided, the vocabulary can be described as:

```
{'a', 'according', 'at', 'august', 'bright', 'by', 'census', 'day', 'end',
'fallen', 'how', 'i', 'is', 'it', 'i’ve', 'latest', 'many', 'miles',
'million', 'more', 'moscow', 'of', 'population', 'than', 'the', 'this',
'time', 'to', 'two', 'warm', 'was', 'wonder', 'or', 'not', 'be'}
```

For easier reference in the following description, assign a unique ID to each term in the vocabulary:

```
(0, 'a'), (1, 'according'), (2, 'at'), (3, 'august'), (4, 'bright'), (5,
'by'), (6, 'census'), (7, 'day'), (8, 'end'), (9, 'fallen'), (10, 'how'),
(11, 'i'), (12, 'is'), (13, 'it'), (14, 'i’ve'), (15, 'latest'), (16,
'many'), (17, 'miles'), (18, 'million'), (19, 'more'), (20, 'moscow'), (21,
'of'), (22, 'population'), (23, 'than'), (24, 'the'), (25, 'this'), (26,
'time'), (27, 'to'), (28, 'two'), (29, 'warm'), (30, 'was'), (31, 'wonder'),
(32, 'or'), (33, 'not'), (34, 'be')
```

Term Frequency (TF):

Term Frequency (TF) represents how often a term ***t*** appears in a document ***d***. The documents mentioned earlier can be depicted using Term Frequency (TF), where each term is formatted as `(ID, frequency)`.

ID	Content
1	(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 1), (6, 0), (7, 0), (8, 0), (9, 1), (10, 1), (11, 1), (12, 0), (13, 0), (14, 1), (15, 0), (16, 1), (17, 1), (18, 0), (19, 0), (20, 0), (21, 0), (22, 0), (23, 0), (24, 0), (25, 1), (26, 1), (27, 0), (28, 0), (29, 0), (30, 0), (31, 1), (32, 0), (33, 0), (34, 0)
2	(0, 0), (1, 1), (2, 0), (3, 0), (4, 0), (5, 0), (6, 1), (7, 0), (8, 0), (9, 0), (10, 0), (11, 0), (12, 1), (13, 0), (14, 0), (15, 1), (16, 0), (17, 0), (18, 1), (19, 1), (20, 1), (21, 1), (22, 1), (23, 1), (24, 2), (25, 0), (26, 0), (27, 1), (28, 1), (29, 0), (30, 0), (31, 0), (32, 0), (33, 0), (34, 0)
3	(0, 1), (1, 0), (2, 1), (3, 1), (4, 1), (5, 0), (6, 0), (7, 1), (8, 1), (9, 0), (10, 0), (11, 0), (12, 0), (13, 1), (14, 0), (15, 0), (16, 0), (17, 0), (18, 0), (19, 0), (20, 0), (21, 1), (22, 0), (23, 0), (24, 1), (25, 0), (26, 0), (27, 0), (28, 0), (29, 1), (30, 1), (31, 0), (32, 0), (33, 0), (34, 0)
4	(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0), (8, 0), (9, 0), (10, 0), (11, 0), (12, 0), (13, 0), (14, 0), (15, 0), (16, 0), (17, 0), (18, 0), (19, 0), (20, 0), (21, 0), (22, 0), (23, 0), (24, 0), (25, 0), (26, 0), (27, 2), (28, 0), (29, 0), (30, 0), (31, 0), (32, 1), (33, 1), (34, 2)
5	(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0), (8, 0), (9, 0), (10, 0), (11, 0), (12, 0), (13, 0), (14, 0), (15, 0), (16, 0), (17, 0), (18, 0), (19, 0), (20, 0), (21, 0), (22, 3), (23, 0), (24, 3), (25, 0), (26, 0), (27, 0), (28, 0), (29, 0), (30, 0), (31, 0), (32, 0), (33, 0), (34, 0)

Notice how this representation format is quite sparse. We could save space by storing it in a sparse representation, which means removing all zero entries.

ID	Content
1	(5, 1), (9, 1), (10, 1), (11, 1), (14, 1), (16, 1), (17, 1), (25, 1), (26, 1)
2	(1, 1), (6, 1), (12, 1), (15, 1), (18, 1), (19, 1), (20, 1), (21, 1), (22, 1), (23, 1), (24, 2), (27, 1), (28, 1)
3	(0, 1), (2, 1), (3, 1), (4, 1), (7, 1), (8, 1), (13, 1), (21, 1), (24, 1), (29, 1), (30, 1)
4	(27, 2), (32, 1), (33, 1), (34, 2)
5	(22, 3), (24, 3)

Inverse Document Frequency (IDF):

Inverse Document Frequency (IDF) indicates the number of documents in which a term appears, reflecting its commonality. A high Inverse Document Frequency (IDF) suggests that the term is not particularly distinctive across documents. The Inverse Document Frequency (IDF) values for our corpus are as follows:

```
(0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1), (9,
1), (10, 1), (11, 1), (12, 1), (13, 1), (14, 1), (15, 1), (16, 1), (17, 1),
(18, 1), (19, 1), (20, 1), (21, 2), (22, 2), (23, 1), (24, 3), (25, 1), (26,
1), (27, 2), (28, 1), (29, 1), (30, 1), (31, 1), (32, 1), (33, 1), (34, 1)
```

Notice that the term `be` appears twice in the last document, yet its Inverse Document Frequency (IDF) is only `1`. Conversely, the term `the` appears in documents `2` and `3`, resulting in an Inverse Document Frequency (IDF) of `2`.

TF/IDF Weights:

TF/IDF weights are essentially term frequencies adjusted by Inverse Document Frequency (IDF) normalisation.

ID	Content
1	(5, 1), (9, 1), (10, 1), (11, 1), (14, 1), (16, 1), (17, 1), (25, 1), (26, 1)
2	(1, 1), (6, 1), (12, 1), (15, 1), (18, 1), (19, 1), (20, 1), (21, 0.5), (22, 0.5), (23, 1), (24, 0.66), (27, 0.5), (28, 1)
3	(0, 1), (2, 1), (3, 1), (4, 1), (7, 1), (8, 1), (13, 1), (21, 0.5), (24, 0.33), (29, 1), (30, 1)
4	(27, 1), (32, 1), (33, 1), (34, 2)
5	(22, 1.5), (24, 1)

Here, you'll observe that terms with IDs `{21, 22, 24, 27}` `(of, population, the, to)` are reduced in scale.

Basic Vector Space Model:

In the basic vector space model, documents and queries are represented as vectors, reflecting their TF/IDF weights.

The most straightforward method to convert TF/IDF weights into a computer-interpretable vector is to index the array with word IDs and record the TF/IDF value. For example, the vector for document `5` would be:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1.5, 0, 1, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0]
```

The query `the population` will produce a vector:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.5, 0, 0.33,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

The measure that assesses the relevance of a document to the query is calculated by the inner product (scalar product) of the two vectors.

$$r(q,d) = \sum_{i=1}^{|V|} q_i \cdot d_i$$

∴ where $|V|$ is the size of the vocabulary, and q_i is the TF/IDF weight if the i^{th} term in the query.

As observed, the simplistic vector space representation using arrays is highly sparse. Alternatively, you can opt for a sparse vector representation. The distinction lies in the fact that a sparse vector only stores non-zero values. Such a vector can be implemented using a dictionary (referred to as a map in Apache Hadoop terminology).

The sparse vector representation for document `5` is as follows:

```
[(22: 1.5), (24: 1)]
```

and for the query:

```
[(22: 0.5), (24: 0.33)]
```

The relevance function can be easily reformulated as:

$$r(q,d) = \sum_{i:i \in d, i \in q} q_i \cdot d_i$$

∴ where the summation is conducted across all terms that exist in both the query and the document.

If we calculate the relevance of the query `the population` for our toy corpus, the result would be:

```
doc 5: 1.08

doc 2: 0.4678

doc 3: 0.1089

doc 4: 0

doc 1: 0
```

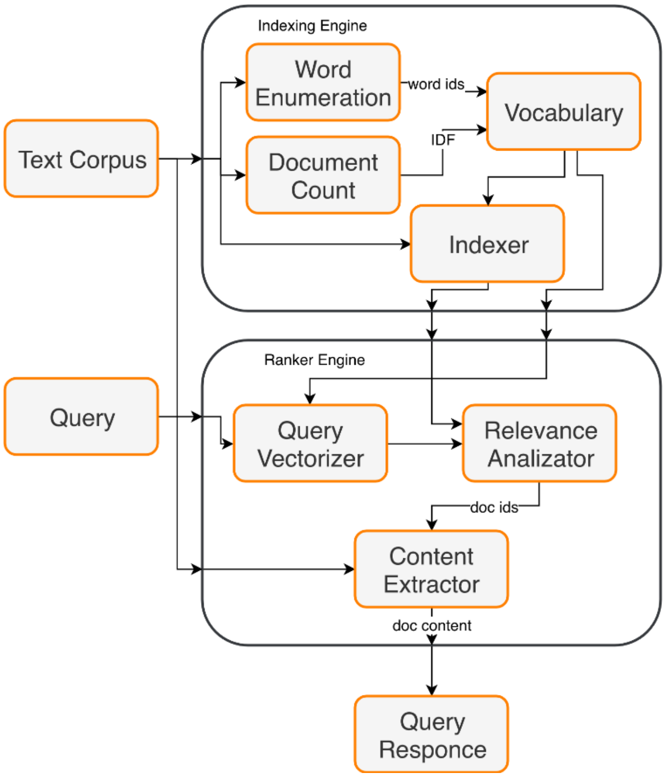
Notice that despite document `5` arguably containing less information, it still ranks first according to our relevance rating.

Room for Speedup:

The indexing and retrieval processes encounter delays partly due to storing documents as arrays and the need to map words to IDs using a `Map` data structure. This slowdown can be partly mitigated by allowing for a certain probability of error. Instead of assigning an ID to every word during preprocessing and repeatedly utilising this mapping, we can dynamically generate an ID using a hash function, thus eliminating the word enumeration step entirely. Depending on the permissible hash range, we can substantially reduce the likelihood of collisions. However, the drawback (or advantage) is increased complexity for the retrieval engine, as it now requires implementing the vector space model exclusively using sparse data structures.

Vector Space Model with MapReduce:

Below is a diagram illustrating one of the potential methods to implement a basic search engine:



Upon receiving the collection of documents, we initiate the indexing engine. This engine initiates several tasks to analyse the entire corpus:

1. Word Enumeration scans the corpus and generates a set of unique words. Subsequently, it assigns a unique ID to each word. This task can be accomplished using MapReduce.
2. Document Count performs a similar function to Word Enumeration, but instead of creating a set of unique terms, it calculates the Inverse Document Frequency (IDF) for each term, or simply the number of documents in which each particular term appears. This task can also be implemented with MapReduce.
3. Vocabulary is a conceptual entity and does not require actual computation. It merely consolidates the results of Word Enumeration and Document Count into a single data structure.

4. The Indexer is responsible for computing a machine-readable representation of the entire document corpus. For each document, the Indexer generates a TF/IDF representation and stores a tuple of `(document ID, vector representation)`. Since each document is processed independently, this task can also be implemented with MapReduce.

Once the index is established, it can be utilised multiple times. The Ranker Engine is tasked with creating a vectorised representation for the query and conducting relevance analysis:

1. The Query Vectorizer function generates the vectorised representation of a query and can be integrated into the Relevance Analizator component.
2. The Relevance Analizator calculates the relevance function between the query and each document, with the implementation potentially utilising MapReduce (performance contingent on available hardware).
3. The index stores the document ID along with its vectorised representation. The Ranker Engine furnishes a sorted list of IDs based on relevance. The Content Extractor retrieves relevant content from the text corpus using the provided relevant IDs.

The Query Response comprises a list of relevant content, such as links accompanied by titles.

The model outlined above demonstrates one potential approach to implementing the system within the framework of the MapReduce paradigm. While you're not obliged to adhere strictly to this architecture, the sole requirement is that the entire system must be implemented using Apache Hadoop's MapReduce.

Note:

This assignment is designed to introduce you to distributed storage and processing in Apache Hadoop, as well as the handling of big data using the MapReduce paradigm. Approach it as an opportunity to gain a solid understanding of big data processing fundamentals. There's flexibility in how you approach the assignment, so feel free to explore innovative methods. While you *can* utilise generative tools like ChatGPT, be prepared to explain your work during the demonstration.

Clustering is a crucial aspect of this assignment, given the substantial size of the dataset, rendering it inefficient to compute on a single computer. While you have the option to implement a local Apache Hadoop cluster, we strongly recommend exploring cloud computing platforms such as **Microsoft Azure Cloud**. They offer $200 free credit for the first month and facilitate clustering controls without the need to download operating systems and dependencies from scratch.