# 1 Introduction

1.1 Overview of the Project

The primary goal of this project is to develop a comprehensive system capable of creating an inverted index from a large set of text documents and processing queries using this index. This project stands at the intersection of data processing, search engine technology, and advanced programming, showcasing the application of complex algorithms and data structures to facilitate efficient and accurate information retrieval.

1.2 Significance of Inverted Index in Information Retrieval

An inverted index is a pivotal component in modern search engines, serving as the backbone for efficient query processing. It allows for rapid retrieval of information by storing a mapping from terms to their locations within a set of documents. This structure is crucial in handling large datasets, like the MSMARCO Passages dataset used in this project, enabling quick searches even in the context of extensive and varied data. The implementation of an inverted index is a complex task, involving considerations of data format, indexing speed, search efficiency, and storage requirements.

1.3 Objectives and Scope of the Project

The project's objectives are multifaceted, focusing on both the creation of a robust inverted index and the development of an effective query processing system. The scope includes:

* Data Handling and Processing: Efficiently managing and processing a dataset comprising 8.8 million documents, ensuring compatibility with various languages and character sets.
* Inverted Index Creation: Building an inverted index that accurately and quickly maps terms to their document occurrences.
* Query Processing: Developing a system to execute ranked queries using algorithms like TFIDF and supporting both conjunctive and disjunctive queries.
* User Interaction: Implementing a user-friendly command-line interface for query input and results presentation.
* Testing and Validation: Ensuring the reliability and correctness of the system through comprehensive JUnit testing.

# 2. Indexing

Indexing is a pivotal process in any search engine, as it determines how effectively the system can store, retrieve, and process information. This chapter delves into the indexing module of the search engine, detailing its sub-processes and the roles of different classes involved.

2.1 Preprocessing of the Documents

The first stage in indexing is the preprocessing of documents. This is where raw text data is cleaned and transformed into a more structured format, making it suitable for further processing and indexing.

2.1.1 Text Cleaning

Text cleaning is the initial step in the preprocessing phase, managed by the Preprocess class. This process involves removing irrelevant characters, correcting formatting issues, and eliminating any data that could hinder the accuracy of the indexing process. Examples of text cleaning include stripping HTML tags, removing special characters, and correcting encoding issues.

2.1.2 Tokenization and Text Normalisation

After cleaning the text, the next step is tokenization and normalization, also handled by the Preprocess class. Tokenization involves breaking down the text into individual words or terms. Normalization then processes these tokens to ensure consistency; for instance, by converting all characters to lowercase, thereby reducing redundancy in the index. This step is crucial for preparing the text for efficient indexing and retrieval.

2.1.3 Stopword Removal and Stemming

The final step in preprocessing is stopword removal and stemming. Stopwords (common words like “the”, “is”, “at”, etc., which are frequent but usually irrelevant to the search queries) are removed to improve the efficiency of the index. Stemming is performed in this project using the Porter Stemmer and reduces words to their base or root form. This process helps in consolidating different forms of a word into a single index entry, enhancing the search engine’s performance.

2.2 Indexing

After preprocessing, the text data is ready to be indexed. Indexing involves creating data structures that allow quick and efficient retrieval of information.

2.2.1 SPIMI

The SPIMI class implements the Single-Pass In-Memory Indexing (SPIMI) algorithm. SPIMI is an efficient way to index large text collections, particularly when memory constraints are a concern. It processes the data in a single pass and creates intermediate indices, which are then merged into a final index. This approach is advantageous for its low memory footprint and scalability.

2.2.2 Merging

The merging process, likely handled by the SPIMIMerger class, is a crucial step in the SPIMI indexing process. It involves combining the multiple intermediate indices created during the SPIMI process into a comprehensive, final index. This step is essential for ensuring that the final index is complete and includes all processed documents.

2.3 Compression

An important aspect of indexing is the compression of index files, which is crucial for efficient storage and retrieval. We implemented Variable Byte Encoding and Unary compression.