Report on BM25 Document Search Engine Implementation

1 Methodology

2 Design Overview

The search engine implemented in this repository follows a classic document indexing and retrieval architecture based on the BM25 ranking algorithm. The system is designed to work in a distributed environment using Hadoop MapReduce for indexing and PySpark for retrieval operations. Cassandra is used as the underlying database to store index information.

3 System Components and Architecture

3.1 Data Preparation Pipeline

- Downloads a Parquet file containing document data (id, title, text)
- Extracts at least 1000 documents with the required fields
- Creates individual document files following the naming convention `<doc_id>_<doc_title>.txt`
- Uses PySpark to normalize and transform the data
- Stores the processed documents in HDFS under the `/data` directory
- Prepares the data for indexing by creating an RDD with appropriate format

3.2 Indexing Pipeline (Hadoop MapReduce)

Mapper (mapper1.py):

- Reads document data in the format: `<doc_id><doc_title><doc_text>`
- Tokenizes and normalizes the text using NLTK for:
  - Tokenization
  - Lemmatization

  - Stopword removal
- Counts term frequencies for each document
- Outputs data in the format: `<doc_id>\t<term>\t<term_frequency>\t<doc_length>`

Reducer (reducer1.py):

- Groups data by doc_id
- Calculates document frequency (df) for each term
- Computes inverse document frequency (IDF) for terms
- Calculates BM25 scores for all term-document pairs
- Stores results in Cassandra tables

3.3 Cassandra Schema Design

- **terms** table: Stores vocabulary information
  - `term (text, primary key)`
  - `doc_frequency (int)`
  - `idf (double)`
- **documents** table: Stores document metadata
  - `doc_id (text, primary key)`
  - `doc_length (int)`
- **term_docs** table: Stores term-document relationships with BM25 scores
  - `term (text, part of composite primary key)`
  - `doc_id (text, part of composite primary key)`
  - `term_frequency (int)`
  - `bm25_score (double)`
- **global_stats** table: Stores corpus-wide statistics
  - `stat_name (text, primary key)`
  - `stat_value (double)`

3.4 BM25 Score Calculation

The system uses the BM25 ranking formula to score documents:

$$\text{IDF}(t) = \log\left(\frac{N}{df(t)}\right)$$

$$\text{BM25}(t,d) = \text{IDF}(t) \cdot \frac{tf(t,d) \cdot (k_1 + 1)}{tf(t,d) + k_1 \cdot \left(1 - b + b \cdot \frac{dl(d)}{dl_{avg}}\right)}$$

Where:

- $tf(t,d)$ = Term Frequency of term t in document d
- $dl(d)$ = Length of document d
- dl_{avg} = Average document length in the corpus
- $k_1 = 1.0$, $b = 0.75$

3.5 Query Processing Pipeline

query.py (PySpark):

- Reads and processes user queries
- Tokenizes and normalizes query terms using the same approach as in indexing
- Retrieves BM25 scores for each term from Cassandra
- Aggregates scores for each document
- Returns the top 10 relevant documents with their titles

search.sh:

- Shell script to run the query processing PySpark application
- Handles the configuration of the Spark environment
- Passes user queries to the PySpark application

4 Implementation Details

4.1 Text Processing

The system uses NLTK for natural language processing:

- Tokenization with `nltk.word_tokenize()`
- Lemmatization with `WordNetLemmatizer()`
- Stopword removal using NLTK's English stopwords list
- Text normalization includes lowercasing and removal of special characters

4.2 Optimization Techniques

- Batch processing in Cassandra operations to improve database write performance
- Spark configuration optimization for Parquet file processing
- Dynamic allocation of Spark executors for better resource utilization
- Use of PySpark's RDD operations for parallel processing

5 Demonstration

5.1 Project Execution Status

While the implementation of the system components and architecture is complete, the project could not be successfully executed in a fully running state. The Docker containers and indexing scripts were set up and prepared, but the full integration of all services and the final query execution could not be completed due to unresolved issues during runtime.

5.2 Repository Setup

Setup Instructions

- Clone the repository
- Install Docker and Docker Compose
- Run `docker compose up` to start the containers (Hadoop master, worker, and Cassandra)

Indexing Pipeline Execution

- The system is designed to automatically run `app.sh` which executes:
  - `prepare_data.sh` to extract and process documents
  - `index.sh` to initiate the Hadoop MapReduce indexing pipeline
- Indexing logic was developed and tested in isolated components, though end-to-end execution remains pending

Querying Documents

- The querying pipeline (`search.sh` and `query.py`) was implemented but not tested with live search results due to integration challenges

5.3 Screenshots and Explanations

Screenshots were not included as the full pipeline could not be executed. Once integration is completed and queries are successfully executed, relevant screenshots (e.g., successful indexing, query outputs) can be added to demonstrate results.