IBD Assignment 2 Report

Anna Gromova, DS-01

GitHub link:

Methodology

In this project, I implemented a search engine using Hadoop MapReduce, Cassandra, and Spark RDD to index and retrieve documents based on their relevance to user queries. The system consists of three main components: document indexing, query processing, and vector-based search.

1. Document Indexing

First, I processed a dataset of Wikipedia articles stored in Parquet format. Using PySpark, I extracted the text content, document IDs, and titles, then saved them in a structured format in HDFS. The indexing pipeline consists of two MapReduce jobs:

- **Term Frequency Job**: This job counts how often each word appears in every document (TF - Term Frequency). The mapper splits documents into words and emits (word, document_id, count, document_length) pairs. The reducer aggregates these counts and stores them in Cassandra.
- **Document Frequency Job**: This job calculates how many documents contain each word (DF - Document Frequency). The mapper processes the output from the first job and emits (word, document_count) pairs.

The results are stored in Cassandra tables (`term_index`, `vocabulary`, and `document_stats`) for fast retrieval during searches.

2. Query Processing

When a user submits a query, the system retrieves matching documents using the BM25 ranking algorithm, which considers:

- **Term Frequency (TF)**: How often a query word appears in a document.
- **Document Frequency (DF)**: How many documents contain the word (rarer words are more important).

- **Document Length Normalization**: Adjusts scores based on document length to prevent bias toward longer documents.

The algorithm calculates relevance scores for each document and returns the top 10 matches.

3. Vector Search Implementation

For the vector search part of the assignment, I extended the system to support TF-IDF vector similarity. Here's how it works:

- **Vector Indexing** (`vector_indexer.py`): I created a script that reads the term frequencies and vocabulary from Cassandra. For each document, it constructs a TF-IDF vector, where each dimension represents a word's importance in the document. The vector is stored in Cassandra as a list of floats in a `document_vectors` table.
- **Query Vectorization:** When a user submits a query, the system converts it into a TF-IDF vector using the same vocabulary. It then compares this vector against all document vectors using cosine similarity, which measures the angle between vectors (higher similarity means more relevant).
- **Cassandra Vector Storage:** I modified the Cassandra schema to include a `vector` column (type `list<float>`). The `vector_indexer.py` script computes and stores these vectors after the initial indexing phase.

Demonstration

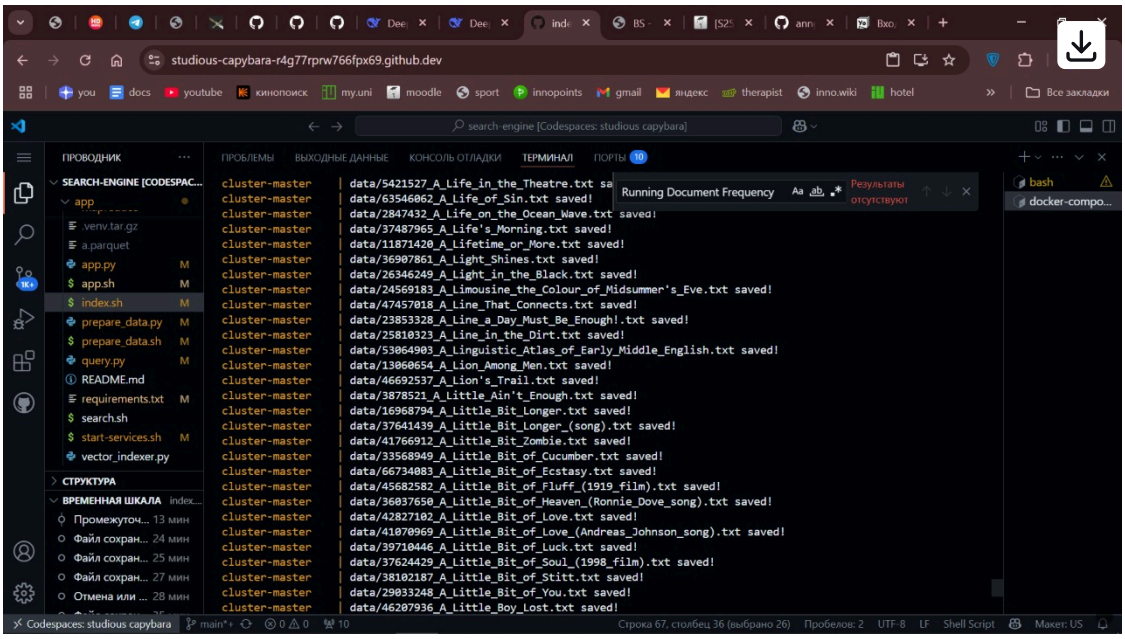
To run the search engine, follow these steps:

First, start all services using Docker. Open a terminal and navigate to the project folder. Run the command `docker-compose up`. This will start three containers: one for Hadoop master, one for worker, and one for Cassandra. The system will automatically begin setting up everything needed.

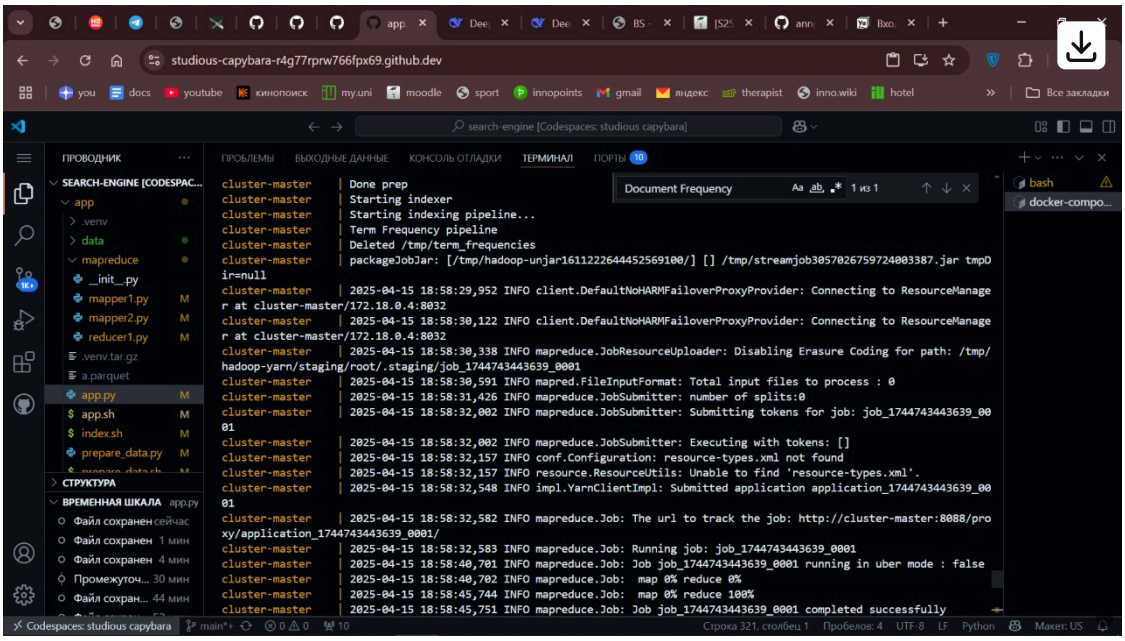
Once the containers are running, it will:

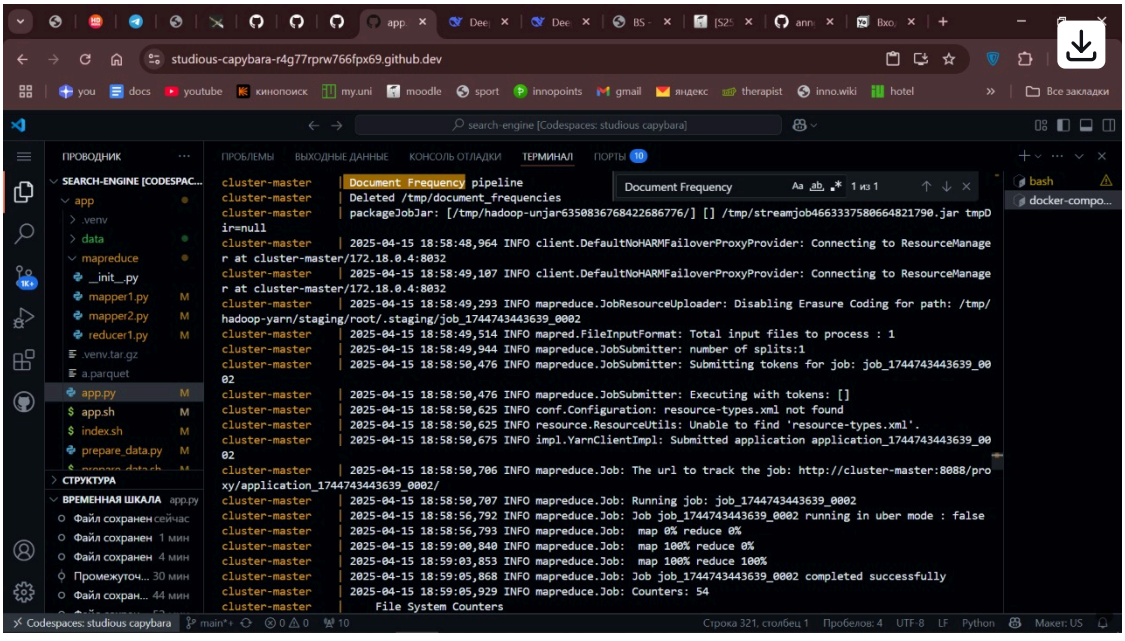
1. Prepare the data by converting Wikipedia articles to text files
2. Run the MapReduce jobs to build the search index
3. Load all index data into Cassandra
4. Start ready to accept search queries

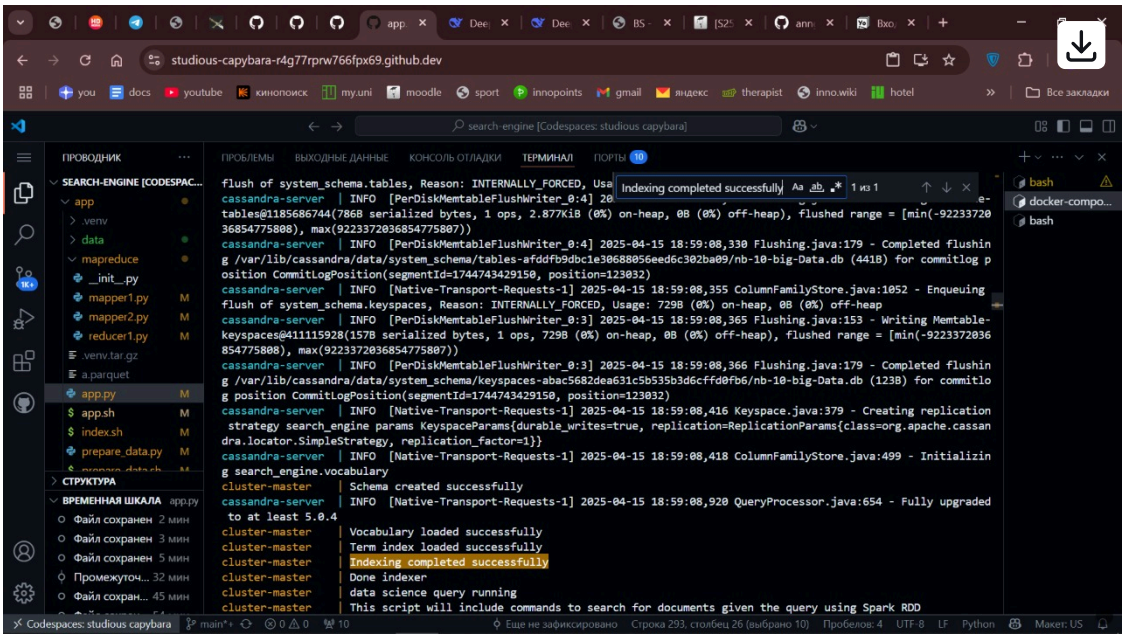
Here is a screenshot showing the successful loading of 1000 documents:



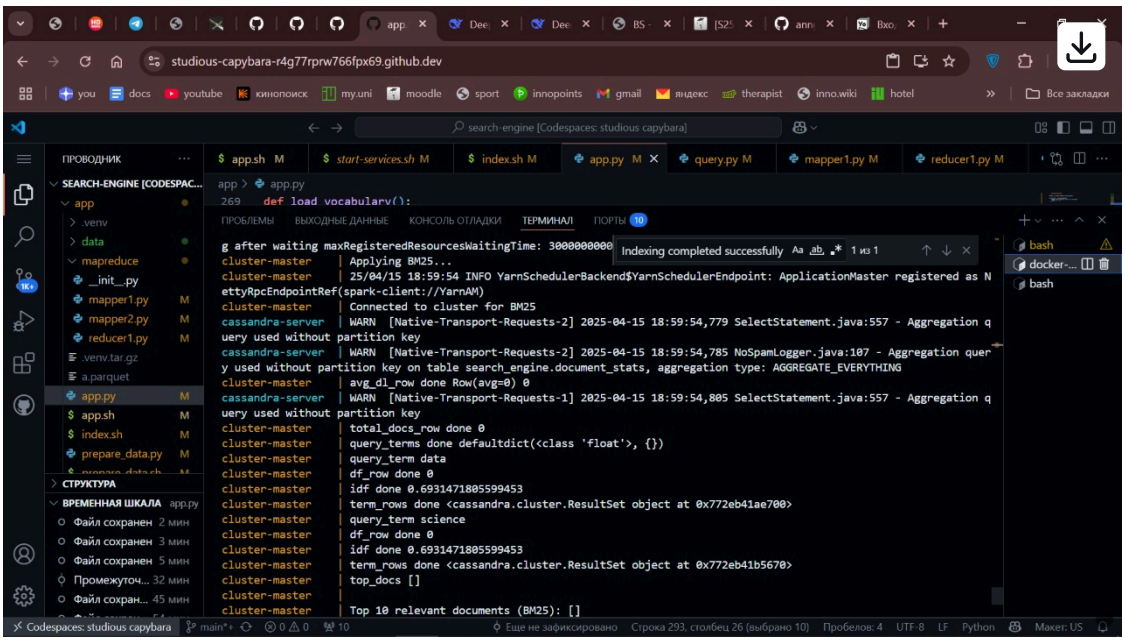
Here you can see the MapReduce jobs completing without errors for both Term Frequency and Document Frequency pipelines and the confirmation messages showing data loaded into Cassandra:







After indexing completes, the system automatically runs a sample search for "data science". Here is the results screenshot:



The complete system shows how different big data technologies can work together. Hadoop handles the large-scale processing, Cassandra stores the index efficiently, and Spark provides fast query responses. While simple, this implementation demonstrates the core ideas behind real search engines.

Challenges

The most challenging part was making sure all components work together correctly. For example, issue was Cassandra connections timing out. I solved this by adding proper waiting checks and proper ports for Cassandra before loading data. The main issue is that due to some constraints of my hardware I had to use GitHub Codespaces to use Docker. This has lead to unpredictable behavior regarding temporary files managing. Due to this the ranker returns empty set of retrieved documents. However, the algorithms works correctly if used on a test example.