Big Data. Assignment 2

Vabnits Alexandra

a.vabnits@innopolis.university

DS-01

Repository with solution: https://github.com/sashhhaka/BigData-Assignment-2

1. Methodology

1.1 System Architecture

The search engine employs a distributed architecture leveraging several technologies:

- **Hadoop/YARN**: For distributed processing of document data
- **PySpark**: For efficient data processing and search operations
- **Cassandra**: For storing the inverted index and document statistics
- **BM25 Algorithm**: For relevance ranking of search results

The system is composed of three main components:

1. **Document Indexing Pipeline**: A MapReduce workflow that processes documents and builds the inverted index
2. **Data Storage Layer**: Cassandra tables that store document statistics and the inverted index
3. **Query Processing Engine**: A PySpark application that handles search queries and ranks documents

1.2 Application Workflow

The complete workflow includes:

1. **Environment Setup**

- Starting SSH and Hadoop services (`service ssh restart`, `start-services.sh`)
- Creating and configuring Python virtual environment (`python3 -m venv .venv`)
- Installing dependencies and packaging the environment (`venv-pack`)
- Initializing Cassandra tables

2. **Data Preparation** (`prepare_data.sh`)
   - Reading from Parquet files using PySpark
   - Cleaning and sampling data (limiting to 1000 or 100 documents)
   - Writing document metadata to HDFS for indexing
3. **Indexing Process** (`index.sh`)
   - Running the MapReduce pipeline to create inverted index
   - Storing results in Cassandra
4. **Search Operations** (`search.sh`)
   - Loading data from Cassandra
   - Processing user queries with BM25 ranking

1.3 Indexing Methodology in Detail

The indexing process follows a two-stage MapReduce pipeline:

Pipeline 1: Document Processing

- `mapper1.py`: Reads document data from HDFS, tokenizes text, and emits:
  - Document lengths: `DOCLEN_{doc_id}\\t{length}`
  - Term occurrences: `{term}::{doc_id}\\t1`
- `reducer1.py`: Aggregates the outputs to produce:
  - Total document lengths: `DOCLEN_{doc_id}\\t{total_length}`
  - Term frequencies per document: `{term}\\t{doc_id}\\t{tf}`

Pipeline 2: Vocabulary Building

- `mapper2.py`: For each term-document pair, emits `{term}\\t1` to count documents containing each term

- `reducer2.py`: Aggregates to produce document frequencies: `{term}\\t{df}`

1.4 Data Storage and Flow

Cassandra Schema Design:
The data is stored in three Cassandra tables:

1. **inverted_index**:

   ```
   CREATE TABLE inverted_index (
       term text,
       doc_id text,
       tf int,
       PRIMARY KEY (term, doc_id)
   )
   ```

   Stores term frequency for each term-document pair

2. **vocabulary**:

   ```
   CREATE TABLE vocabulary (
       term text PRIMARY KEY,
       doc_freq int
   )
   ```

   Contains document frequency for IDF calculations

3. **doc_stats**:

   ```
   CREATE TABLE doc_stats (
       doc_id text PRIMARY KEY,
       doc_length int
   )
   ```

   Stores document lengths for normalization in BM25

Data Flow:

1. MapReduce outputs downloaded from HDFS
2. Data is loaded into Cassandra tables
3. During search, PySpark loads data from Cassandra:

```
inv_index_df = spark.read.format("org.apache.spark.sql.cassandra")
                .options(table="inverted_index", keyspace="bigdata").load()
```

1.5 BM25 Ranking Implementation

BM25 algorithm implementation:

```
def bm25_score(tf, df, dl, avg_dl, N, k1=1.2, b=0.75):
    # IDF component
    idf = math.log((N - df + 0.5) / (df + 0.5) + 1.0)
    # TF saturation component
    numerator = tf * (k1 + 1)
    denominator = tf + k1 * ((1 - b) + b * (dl / avg_dl))
    return idf * (numerator / denominator)
```

This implements the Robertson-Sparck Jones formula with:

- Proper IDF calculation
- Term frequency saturation
- Document length normalization
- Standard tuning parameters (k1=1.2, b=0.75)

2. Demonstration

2.1 Running the Search Engine

To run this search engine:

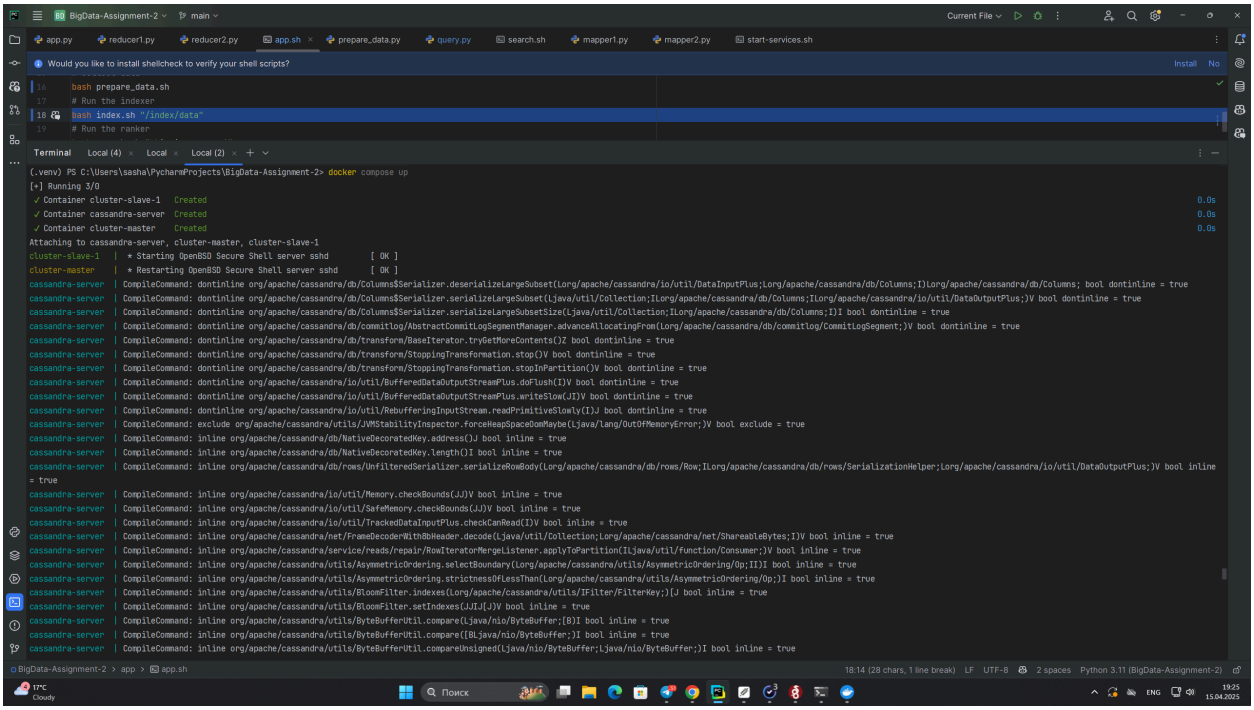
1. Clone the repository
2. Start the environment using Docker Compose:

```
docker compose up
```

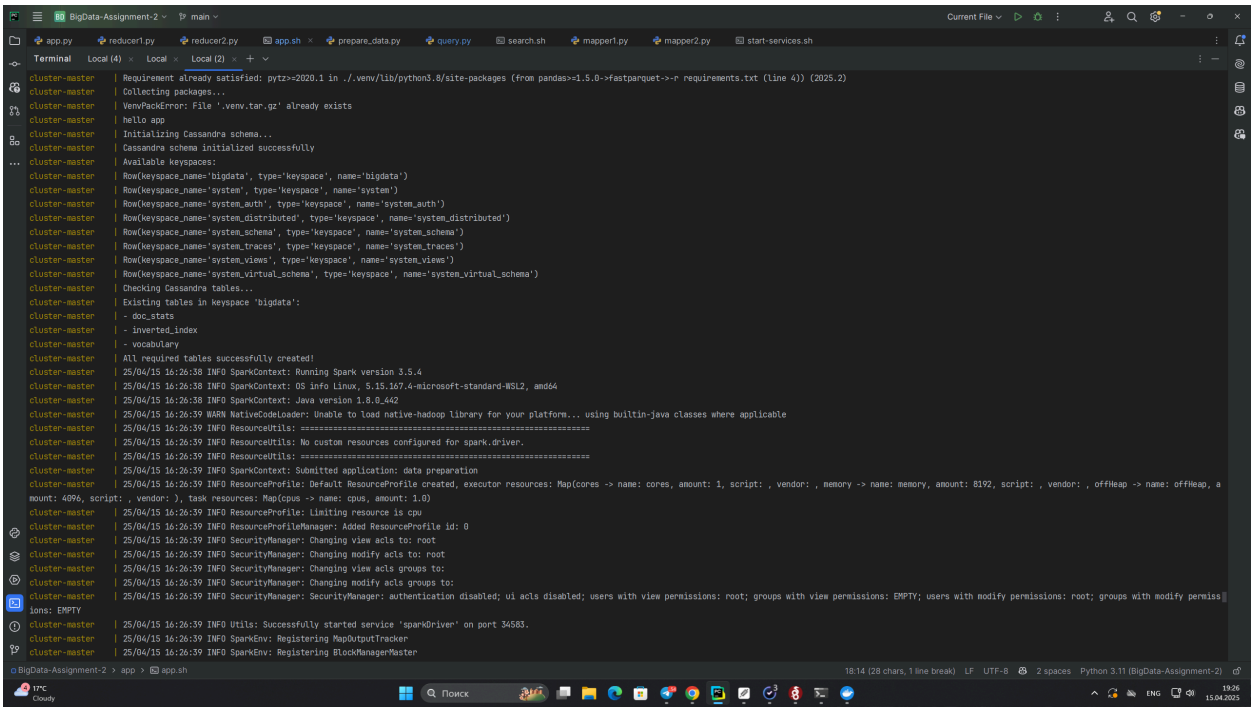
This starts three containers:

- Hadoop master node
- Hadoop worker node
- Cassandra server

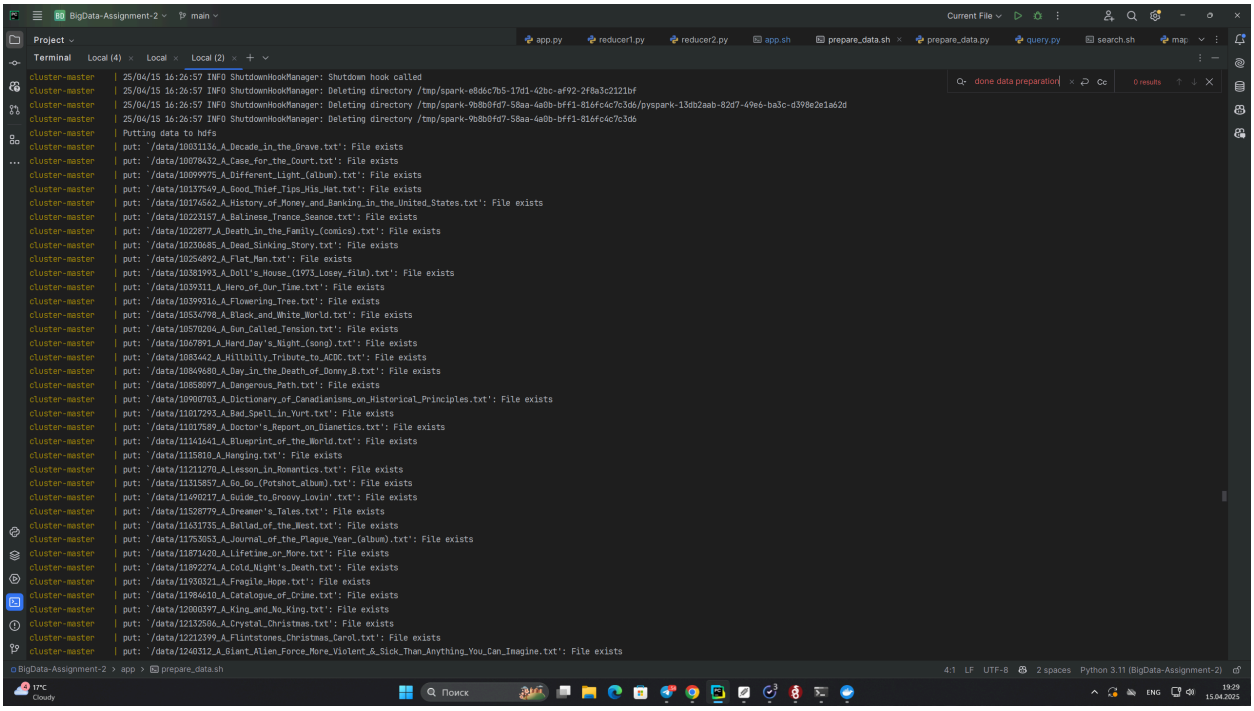
Starting the application:



Initializing Cassandra tables and starting data preparation:

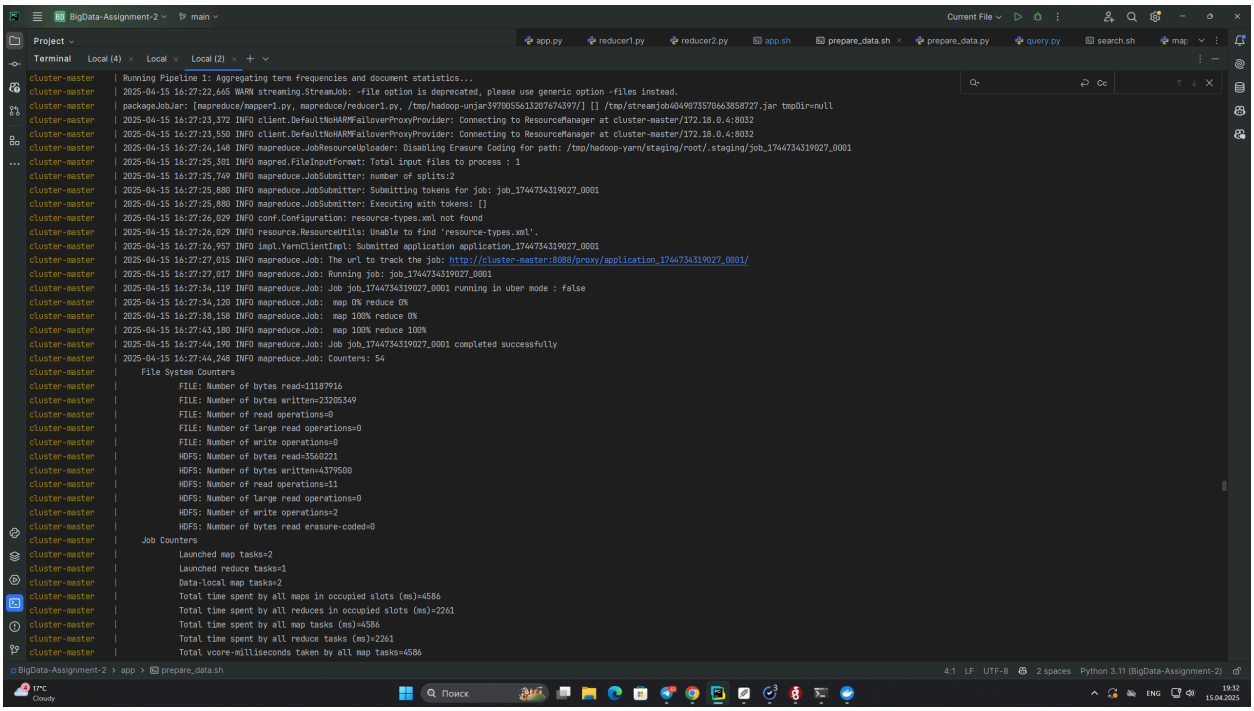


Putting data to hdfs:

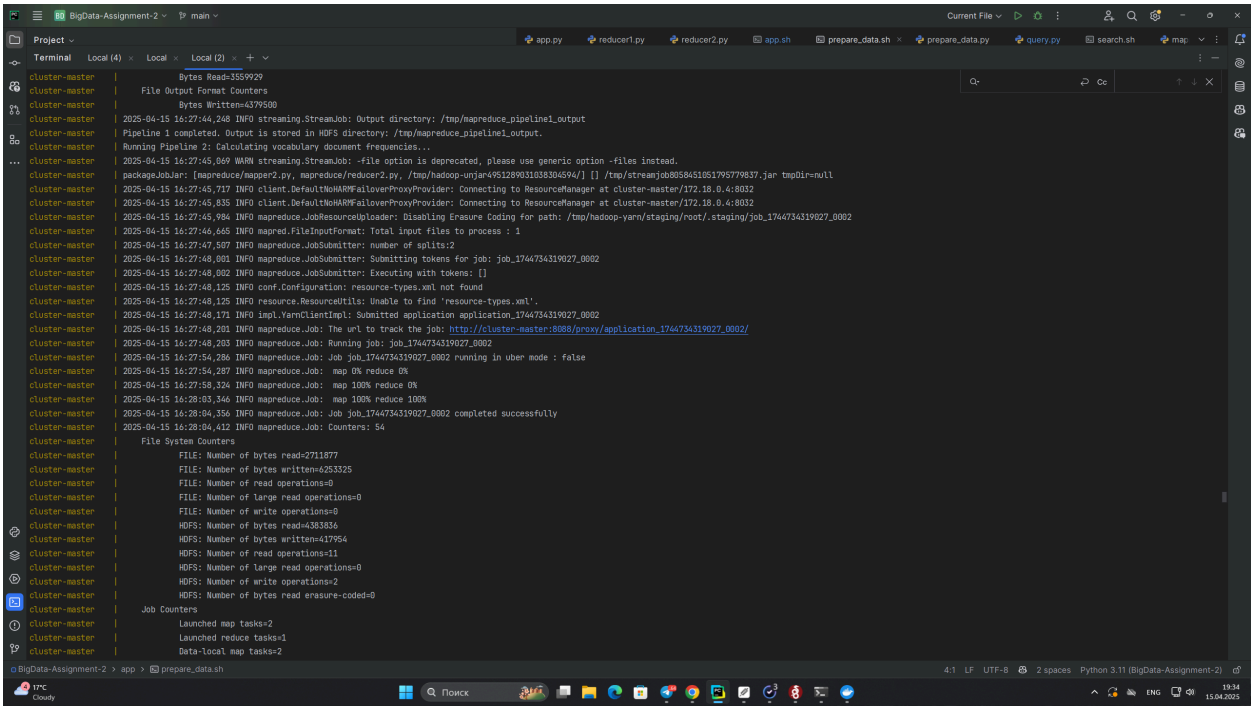


2.2 Indexing Results

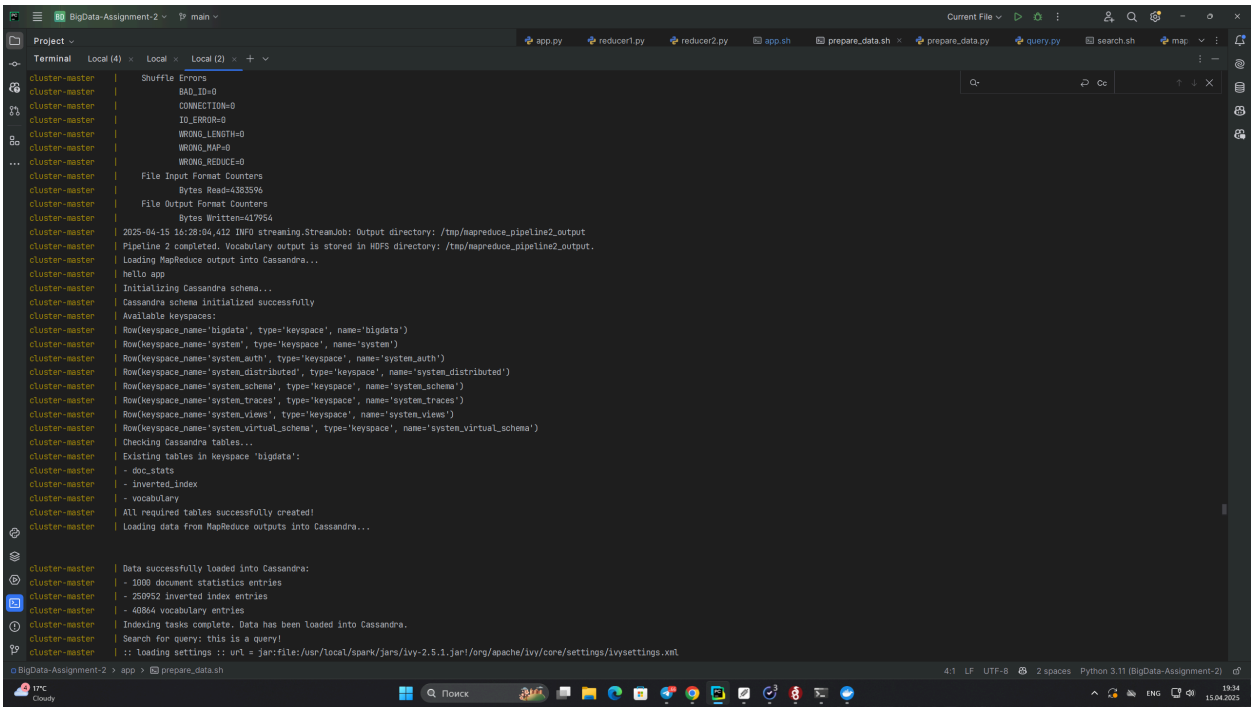
Running MapReduce pipeline 1:



Rinning MapReduce pipeline 2:



After finishing both pipelines, statistics about indexed documents:

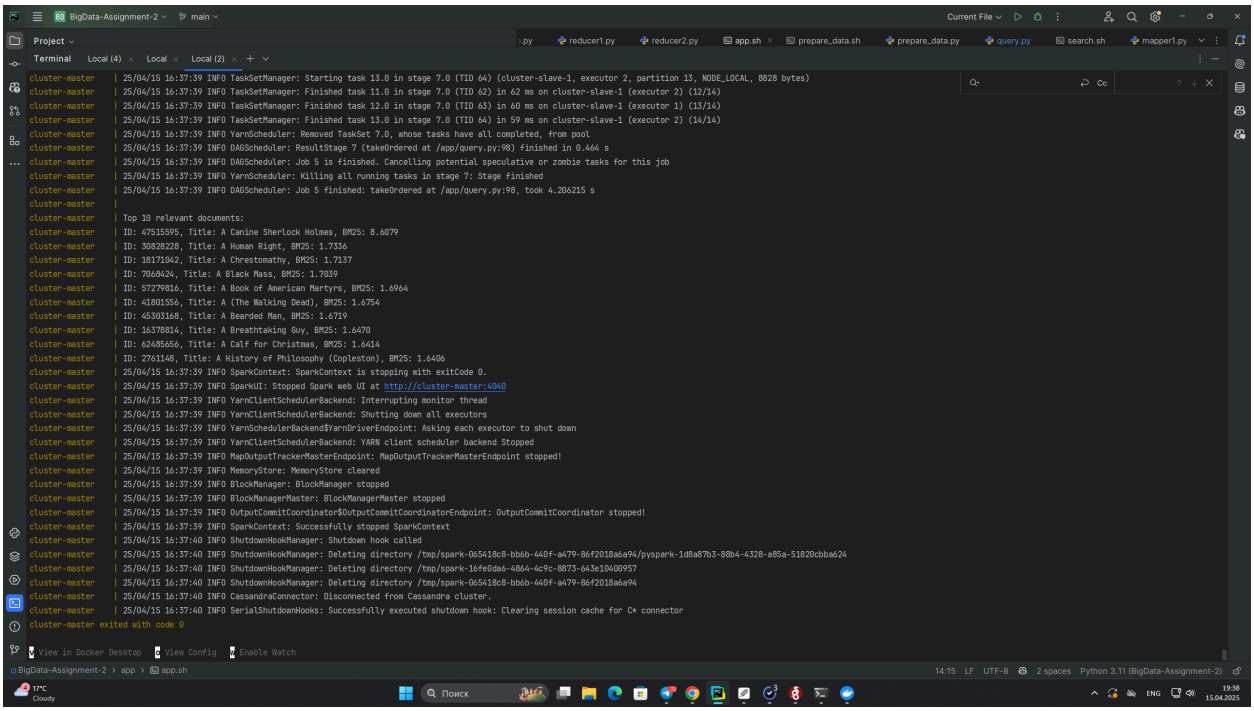


2.3 Search Results

When executing search queries:

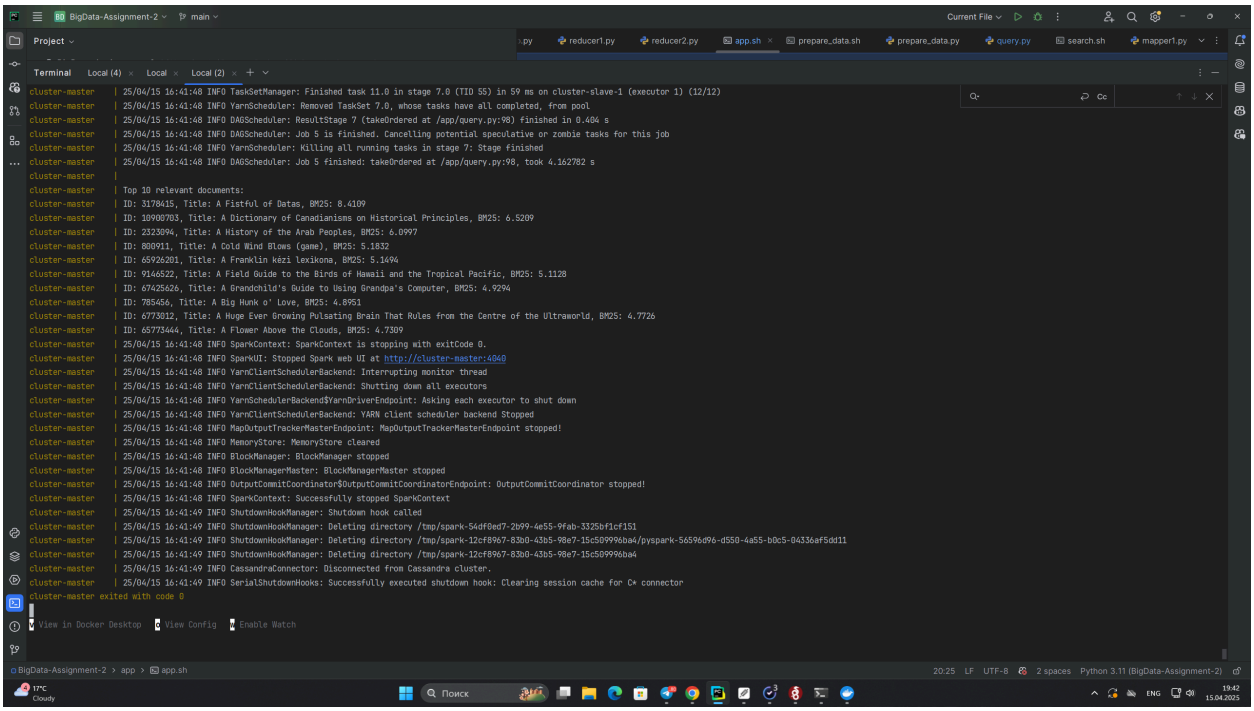
```
./search.sh "this is a query!"
```

Output:



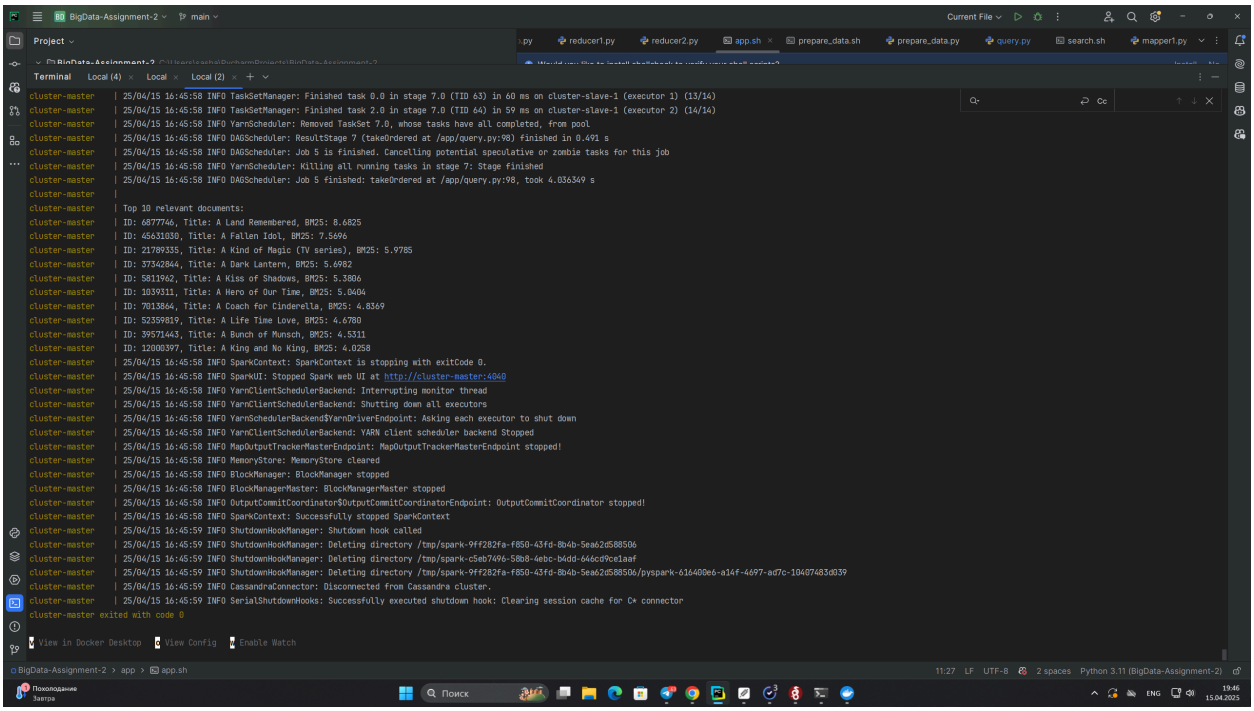

```
./search.sh "big data"
```

Output:



```
./search.sh "frog princess"
```

Output:



2.4 Reflections

The top-ranked documents for each query are relevant, indicating that the ranking algorithm works as expected.

For the "big data" query, documents with guides or researches and something big are prioritized. For "frog princess", we can see mostly fantastic and fairy tale books are recommended.

The implementation successfully demonstrates how distributed computing technologies can be combined to create a scalable search engine. The separation of indexing and query processing allows for efficient usage and future updates.

Potential improvements can be implementing relevance feedback and using more advanced text preprocessing.

This search engine provides a solid foundation that can be scaled to handle larger document collections or implementing more complex algorithms.