Assignment 2 - Simple Search Engine using Hadoop MapReduce

Rufina Gafiiatullina, r.gafiiatullina@innopolis.university
Innopolis University

1 Methodology

This search engine performs indexing, ranking and retrieval of top 10 text documents for query. The system is designed using distributed technologies to work with large amounts of data.

Full code you can find in GitHub: big-data-assignment2-2025

1.1 Overall Architecture

The overall pipeline of the search engine is following:

1. **Data Collection:** A collection of text documents is loaded into the HDFS for indexing.
2. **Indexing (MapReduce):** Words are extracted from documents and normalized (e.g., lowercased). MapReduce is used to compute the components for BM25 score.
3. **Storage (Cassandra):** The results from previous step are stored in Cassandra.
4. **Query Processing (Spark):** Using a query, Spark loads from Cassandra needed information for query terms, calculates BM25 score for documents, and retrieves 10 documents with high score.

1.2 Cassandra

The schema in Cassandra is effective for searching of term frequencies, document statistics, and metadata of the whole document collection.

To have more precise and categorized searches in the future, it is useful to specify the type of collection to which a document belongs (e.g., scientific articles, news reports, or fiction texts). For this purpose, each document is associated with a **corpus**, allowing for faster and more targeted searches. However, for now all documents are stored in a default corpus labeled **"whole_corpus"**.

1.2.1 Keyspace

```
CREATE KEYSPACE IF NOT EXISTS index_keyspace
WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1};
```

This defines the keyspace used for indexing process.

1.2.2 doc_frequency_of_term table

This table stores the number of documents containing the specific term within a specific corpus.

```
CREATE TABLE IF NOT EXISTS doc_frequency_of_term (
    term TEXT,
    corpus_name TEXT,
    doc_frequency INT,
    PRIMARY KEY (corpus_name, term)
);
```

The data is partitioned by `corpus_name`, allowing you to quickly find data for a particular corpus.

Within each partition, data is clustered by `term`, allowing efficient querying of term frequency within a corpus.

1.2.3 term_frequency_in_doc table

This table stores the frequency of each term in a particular document that has the document ID and title.

```
CREATE TABLE IF NOT EXISTS term_frequency_in_doc (
    term TEXT,
    corpus_name TEXT,
    doc_id INT,
    doc_title TEXT,
    term_frequency INT,
    PRIMARY KEY ((term, corpus_name), doc_id, doc_title)
);
```

Due to partitioning by `(term, corpus_name)`, term frequencies for the same term within a specific corpus are stored together in the same partition, which optimizes reads and writes for specific terms in a corpus.

Also data is clustering by `doc_id`, `doc_title`. This allows efficient retrieval of term frequencies for a given document.

1.2.4 doc_info table

This table stores metadata about each document including its length, which is used for calculating BM25 score.

```
CREATE TABLE IF NOT EXISTS doc_info (
    doc_id INT,
    doc_title TEXT,
    doc_length INT,
    PRIMARY KEY (doc_id, doc_title)
);
```

1.2.5 corpus_info table

This table stores metadata about about each corpus, including total number of documents and total length. Now this table contains only one corpus information for `whole_corpus`.

```
CREATE TABLE IF NOT EXISTS corpus_info (
    corpus_name TEXT PRIMARY KEY,
    doc_n INT,
    total_doc_length INT
);
```

1.3 Document Indexing with Hadoop MapReduce

The indexing process contains a two-stage MapReduce pipeline. By default, documents in `/index/data` in HDFS are indexed. But it is possible to give as an argument a local document, which is first converted to the required format, then copied to the HDFS `/local_files`, and indexed.

1.3.1 MapReduce1

- **Mapper 1:** The mapper takes the format (<doc_id> <doc_title> <doc_text>) separated by tab. It takes the text of the document, extract just the words (alphanumeric and apostrophes), obtain unique words for document, and maps 1 to each word in the set.
- **Reducer 1:** The reducer aggregates same words to obtain number of documents that have this word. After that it works with Cassandra and inserts (if term does not exists in table) or updates (if we index new document and tables in Cassandra already store information about previous indexing) values to_doc_frequency_of_term table.

1.3.2 MapReduce2

- **Mapper 2:** The mapper takes the format (<doc_id> <doc_title> <doc_text>) separated by tab. It takes the text of the document, extract just the words (alphanumeric and apostrophes), and prints doc_id, doc_title, and word with assigned 1.
- **Reducer 2:** The reducer computes term frequency per document, document metadata, and corpus information. After that it works with Cassandra and inserts (if term does not exists in table) or updates (if we index new document and tables in Cassandra already store information about previous indexing) values to term_frequency_in_doc, doc_info, and corpus_info tables.

1.3.3 Initializing Cassandra Schema

Before indexing, the Cassandra schema is initialized.

```
cqlsh cassandra-server -f /app/cassandra/schema.cql
```

This ensures that all necessary tables present in Cassandra before inserting index data.

1.4 Query Processing and Ranking with Spark RDD

1.4.1 Query Preprocessing

Each input query is tokenized into unique lowercase terms:

```
query_terms = list(set([word.lower() for word in query[1].split()]))
```

1.4.2 Retrieving Needed Info for Cassandra

Using Spark, I load information for each unique term in a given query from the `doc_frequency_of_term` table from Cassandra. If none of the terms exist in the corpus, the program immediately writes an empty result.

For terms that exist, I collect information about their frequency of appearance in documents. The corresponding rows from the `term_frequency_in_doc` table, which contains the frequencies of terms for each document, are also loaded. Then using the id and title of the documents, I extract information about each document from the table `doc_info`. For these documents, I will count the score. In addition, the corpus information is extracted from the `corpus_info` table in Cassandra.

1.4.3 BM25 Score Calculation

The BM25 formula is following:

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

Where:

- N is the total number of documents,
- $\text{df}(t)$ is the document frequency of query term t,
- $\text{tf}(t,d)$ is the frequency of term t in document d,
- dl is the length of document d,
- dl_{avg} is the average document length across the corpus,
- k_1 and b are hyperparameters (set to 1 and 0.75 respectively).

1.4.4 Ranking and Output

After computing BM25 scores for each term-document pair, scores are aggregated per document using summation and sorted in descending order. The top 10 results go to a CSV file:

```
ranked_docs = joined_df.groupBy("doc_id", "doc_title").sum("bm25")
ranked_docs.orderBy(col("sum(bm25)").desc()).limit(10)
```

1.5 Limitations

This system has following limitations:

- **Simple Tokenization:** The current system uses simple whitespace-based tokenization without support for stemming or stop word removal.
- **No query tokenization:** The system does not remove punctuation marks and other non-alphabetic characters in the query, which may affect the results.
- **Limitation to a single enclosure:** Although the corpus concept is implemented, there is no input mechanism to easily specify or switch between corpora during searching or indexing.

1.6 Summary

Pipeline Step	**Script Name**	**Input**	**Output**
Data collection and preparation	`prepare_data.sh`	`a.parquet`	`/index/data` and `/data` in HDFS
Indexing tasks	`index.sh`	Raw `.txt` file in local system or `/index/data` directory in HDFS	Records in Cassandra tables
Ranking with BM25	`search.sh`	User query, Cassandra tables	Top 10 ranked documents written to CSV in /bm25_output in HDFS

2 Demonstration

2.1 Instructions to Run

2.1.1 Main Pipeline

To run search engine pipeline, clone the repository and run Docker:

```
git clone https://github.com/Rufina2323/big-data-assignment2-2025.git
cd big-data-assignment2-2025
docker compose up
```

This will find documents for "football game" query searching in /index/data documents.

2.1.2 Indexing your File

If you want to index your file, before you run docker, uncomment the last line in app.sh (it is needed so that the master node does not exit after finishing executing the script) and comment index.sh and search.sh executing:

```
# Run the indexer
# bash index.sh

# Run the ranker
# bash search.sh "football game"

tail -f /dev/null
```

Then write in master node following command, where you should specify path to your file that you want to index (for example, /data/sample.txt):

```
bash index.sh /data/sample.txt
```

2.1.3 Searching your Query

If you want to search your query, before you run docker, uncomment the last line in app.sh (it is needed so that the master node does not exit after finishing executing the script) and comment search.sh executing:

```
# Run the ranker
# bash search.sh "football game"

tail -f /dev/null
```

Then write in master node following command, where you should specify your query:

```
bash search.sh "query"
```

2.2 Examples

I run only this part of app.sh to show then that code for indexing and searching work correctly:

```
#!/bin/bash
# Start ssh server
service ssh restart

# Starting the services
bash start-services.sh

# Creating a virtual environment
python3 -m venv .venv
source .venv/bin/activate

# Install any packages
pip install -r requirements.txt

# Package the virtual env.
```

```
venv-pack -o .venv.tar.gz

# Collect data
bash prepare_data.sh
```

2.2.1 Indexing /index/data

Fig.1 shows that Cassandra does not have specified keyspace "index_keyspace".

Figure 1: Checking Cassandra keyspaces

Fig.2 shows that HDFS has needed directory /index/data. Then I execute index.sh without arguments. All index.sh outputs and logs you can see in fig.2, fig.3, fig.4, fig.5.

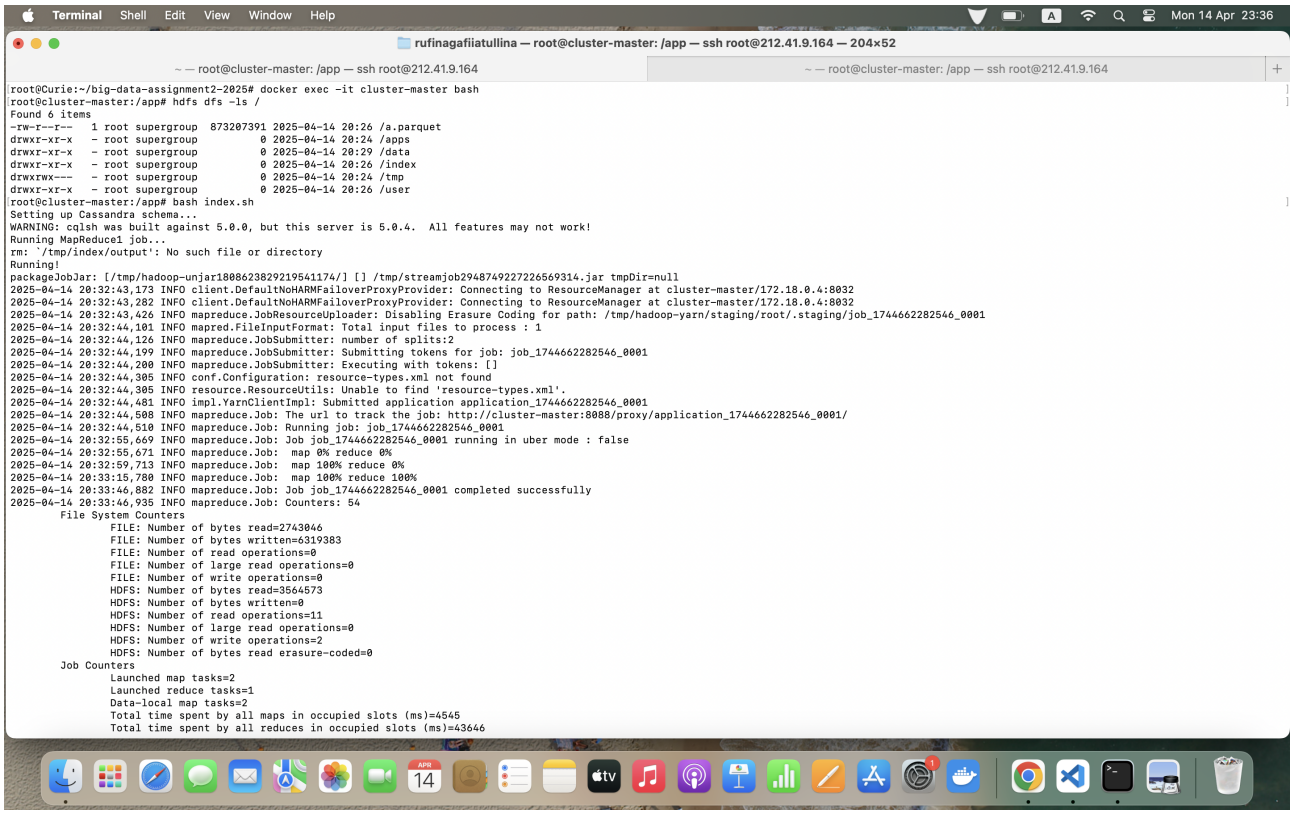

Figure 2: HDFS files

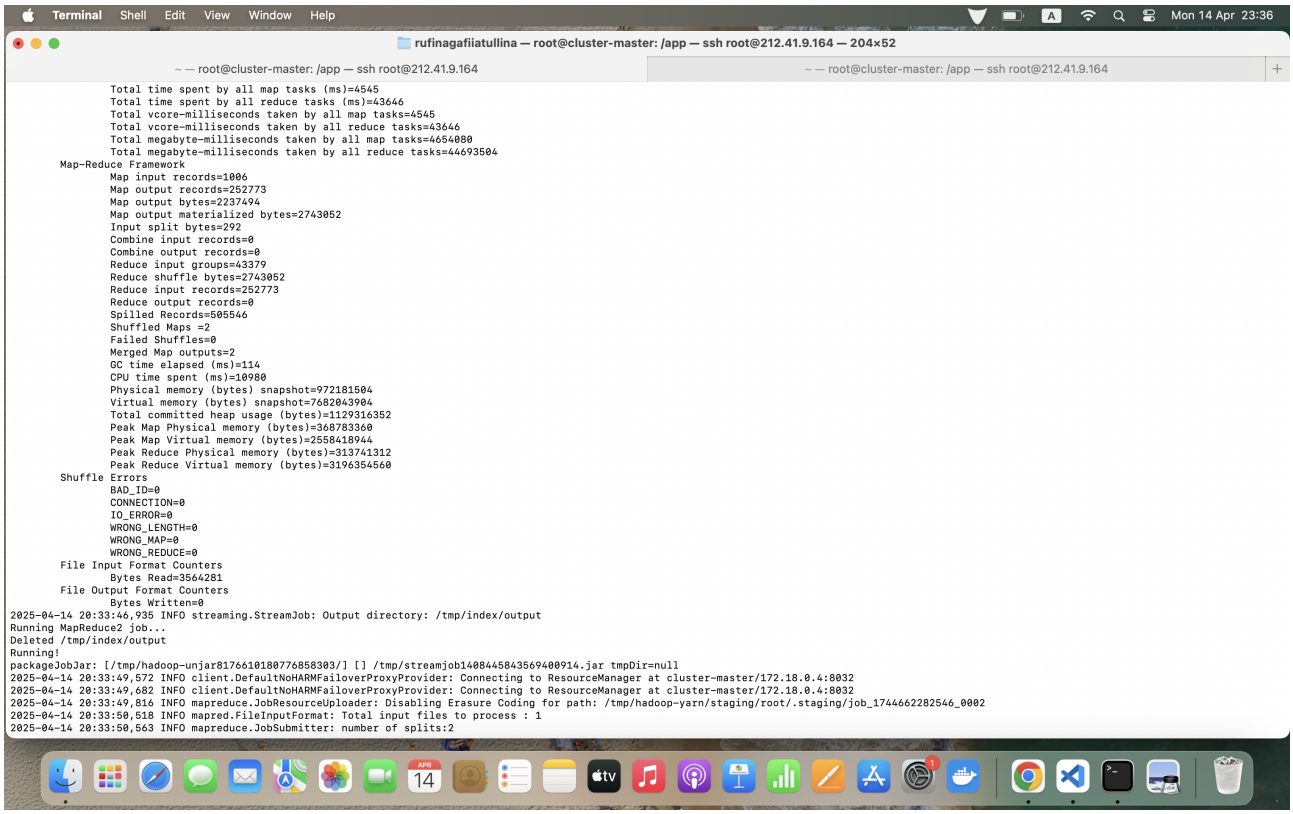

Figure 3: Indexing files

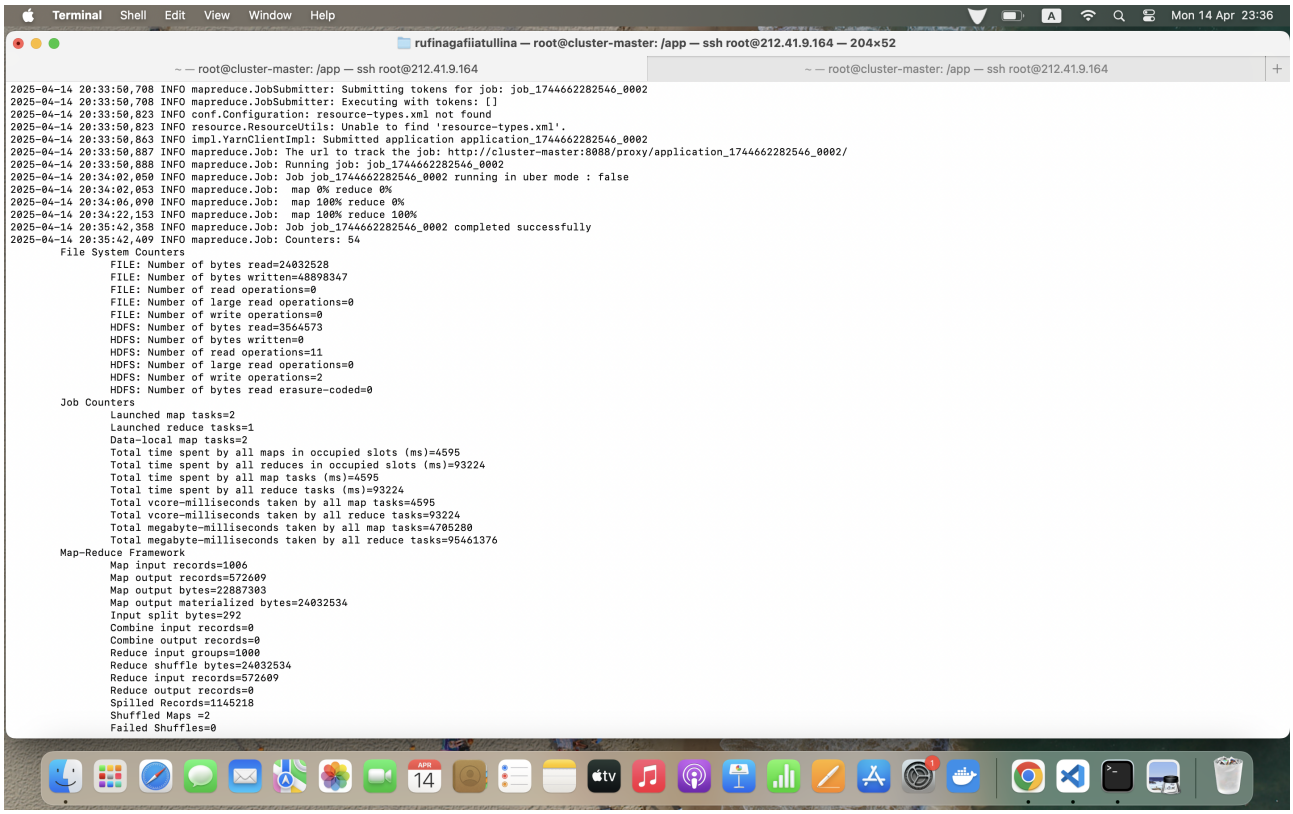

Figure 4: Indexing files

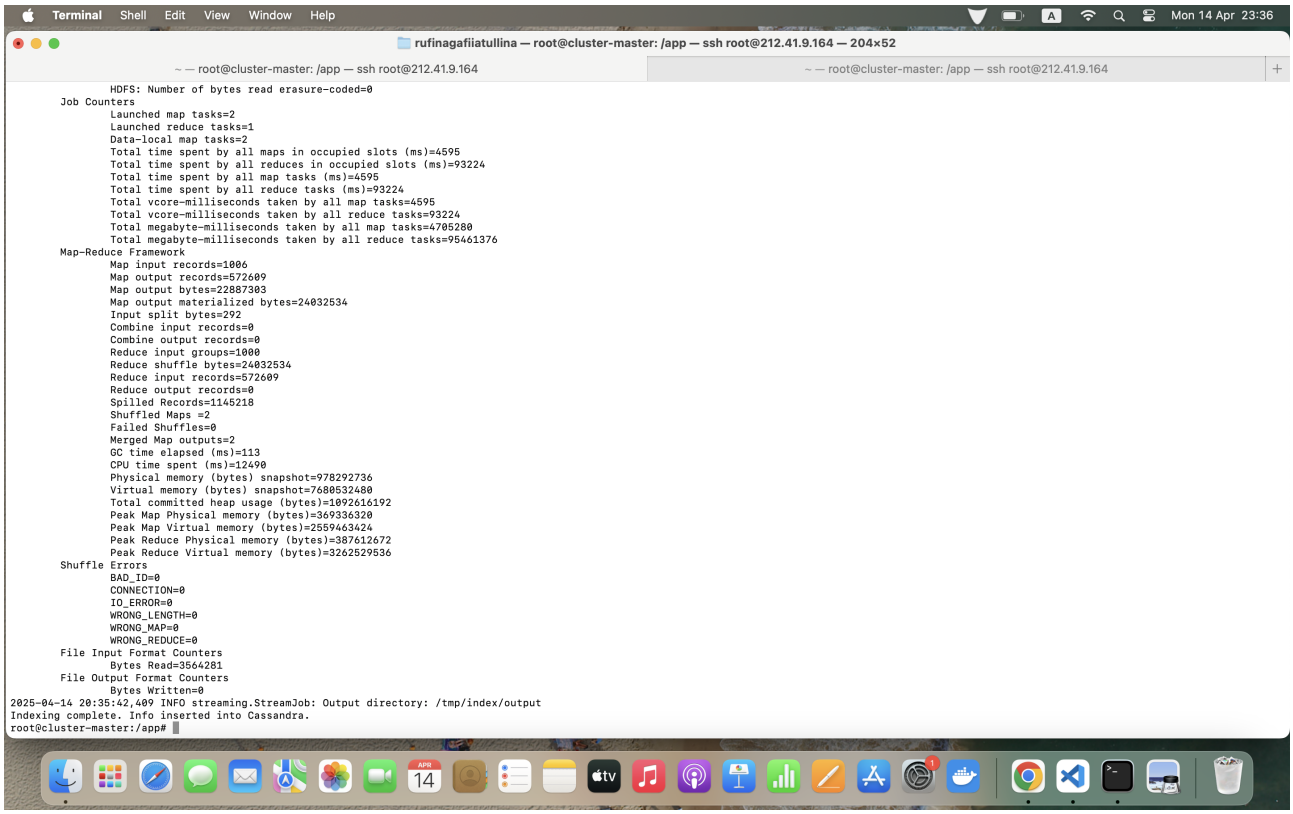

Figure 5: Indexing files

After that I check Cassandra again. Fig.6 shows that now I have `index_keyspace` with four tables. Output from tables of some examples you can see in picture. In total I have 1000 indexed documents.

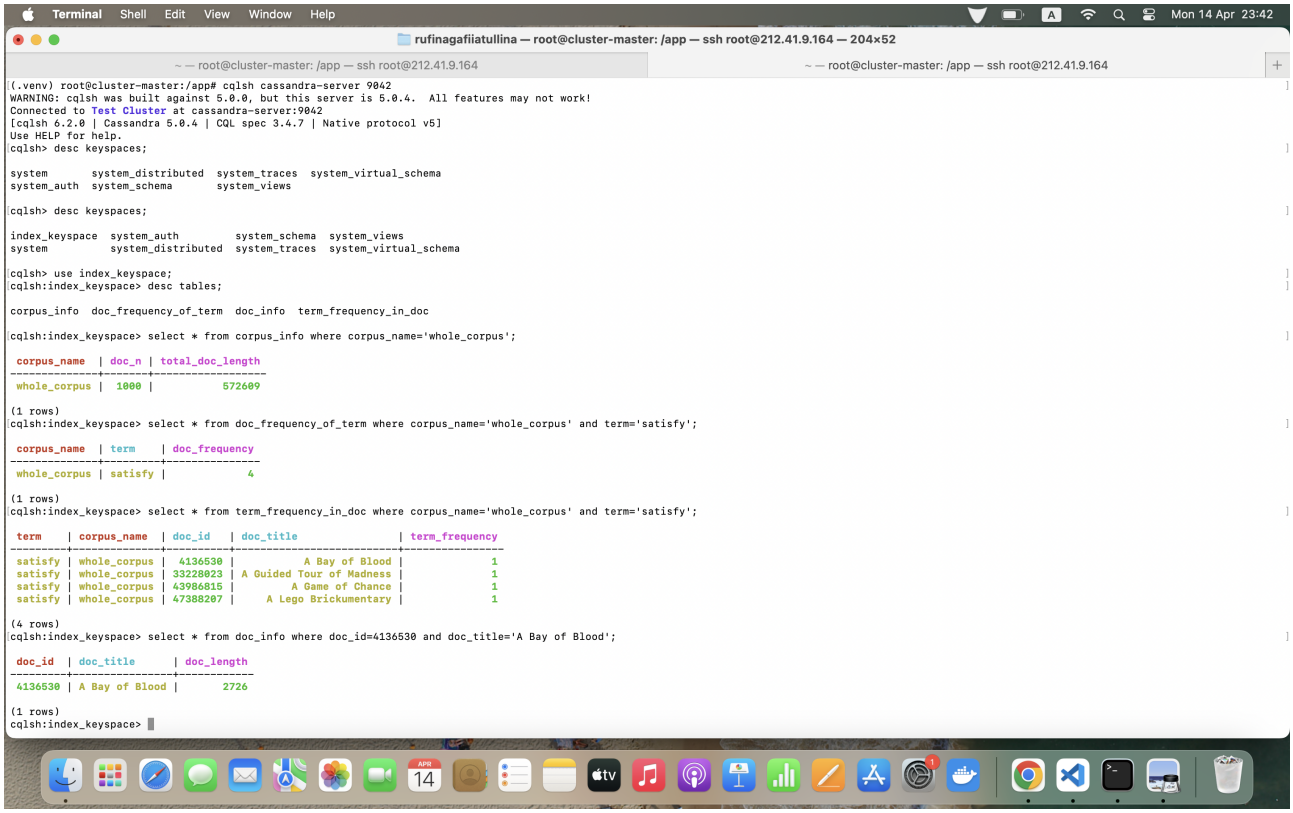

Figure 6: Checking Cassandra after indexing

2.2.2 Indexing Local File

I want to index my small sample.txt file. The content of this file is following:

```
1 Hello everyone! Nice to meet you!
2 It is my sample text file.
```

Since I want to track that my script update information in Cassnadra properly, then I output information about word 'hello' from Cassandra tables. Result you can see in fig.7. These results I will use to compare with results after indexing sample.txt file. I execute index.sh with /data/sample.txt argument. The logs and outputs you can see in fig.8, fig.9. After that I check in HDSF /local_files directory where index.sh stores local files that should be indexed. The results you can see in fig.10. The file not only appeared on the HDFS, but also received a random 6-digit ID (doc_id), the original file name (doc_title), and its contents (doc_text).

Figure 7: Term 'hello'

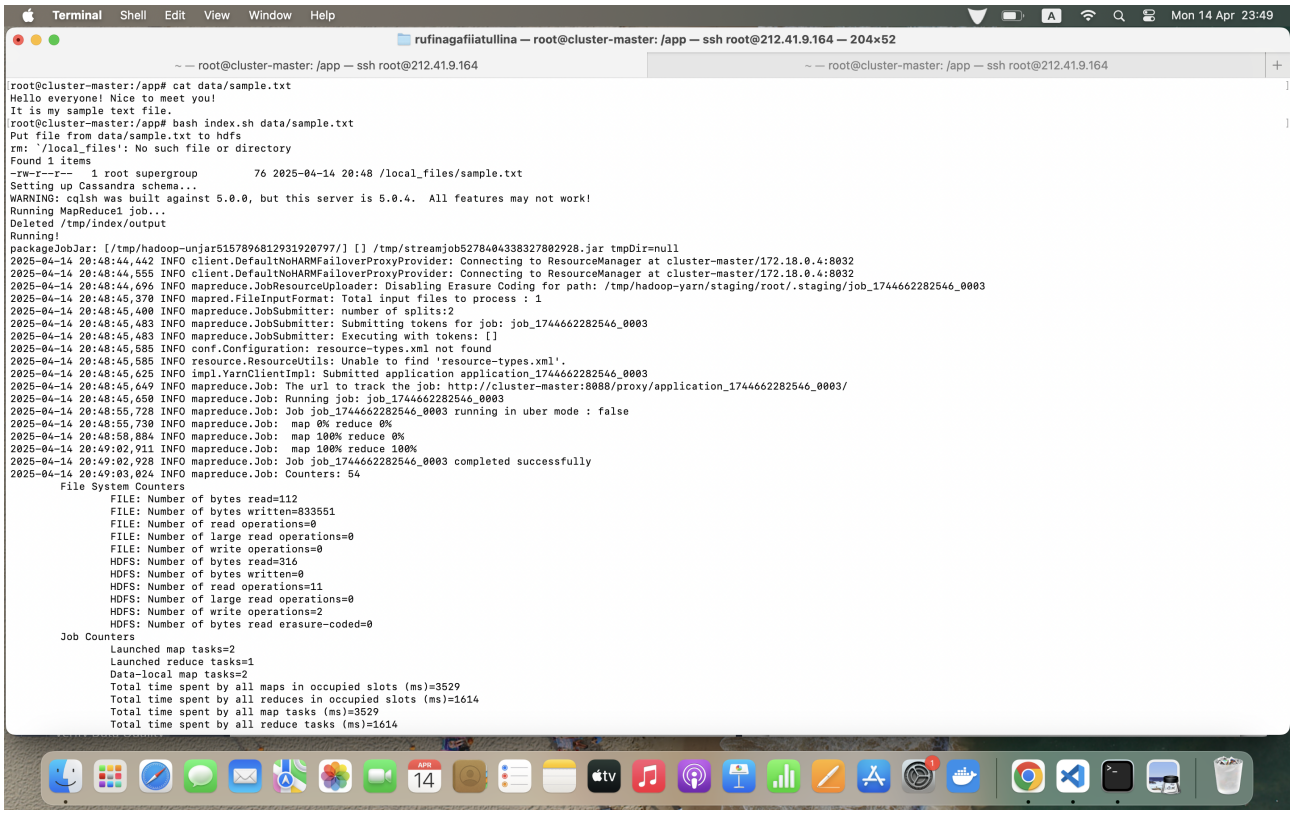

Figure 8: Indexing local file

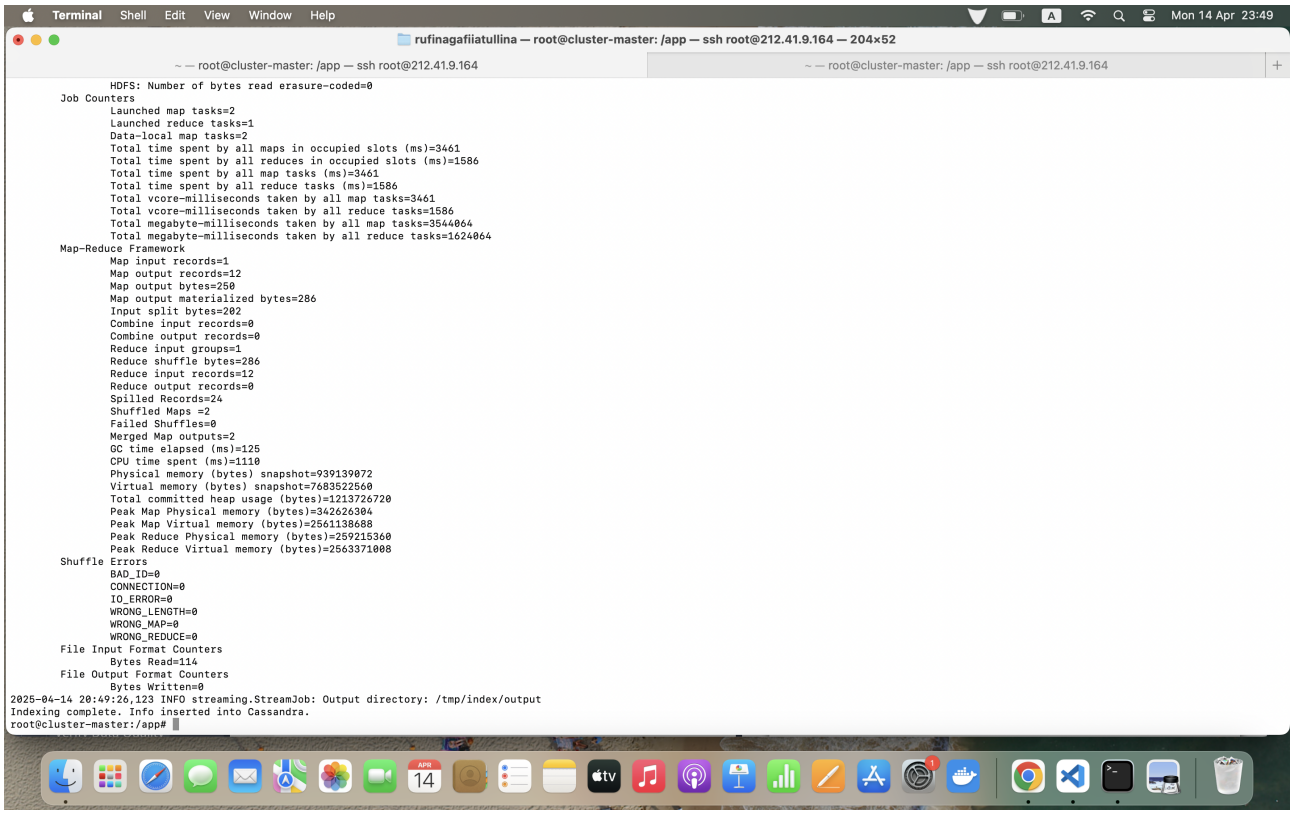

Figure 9: Indexing local file

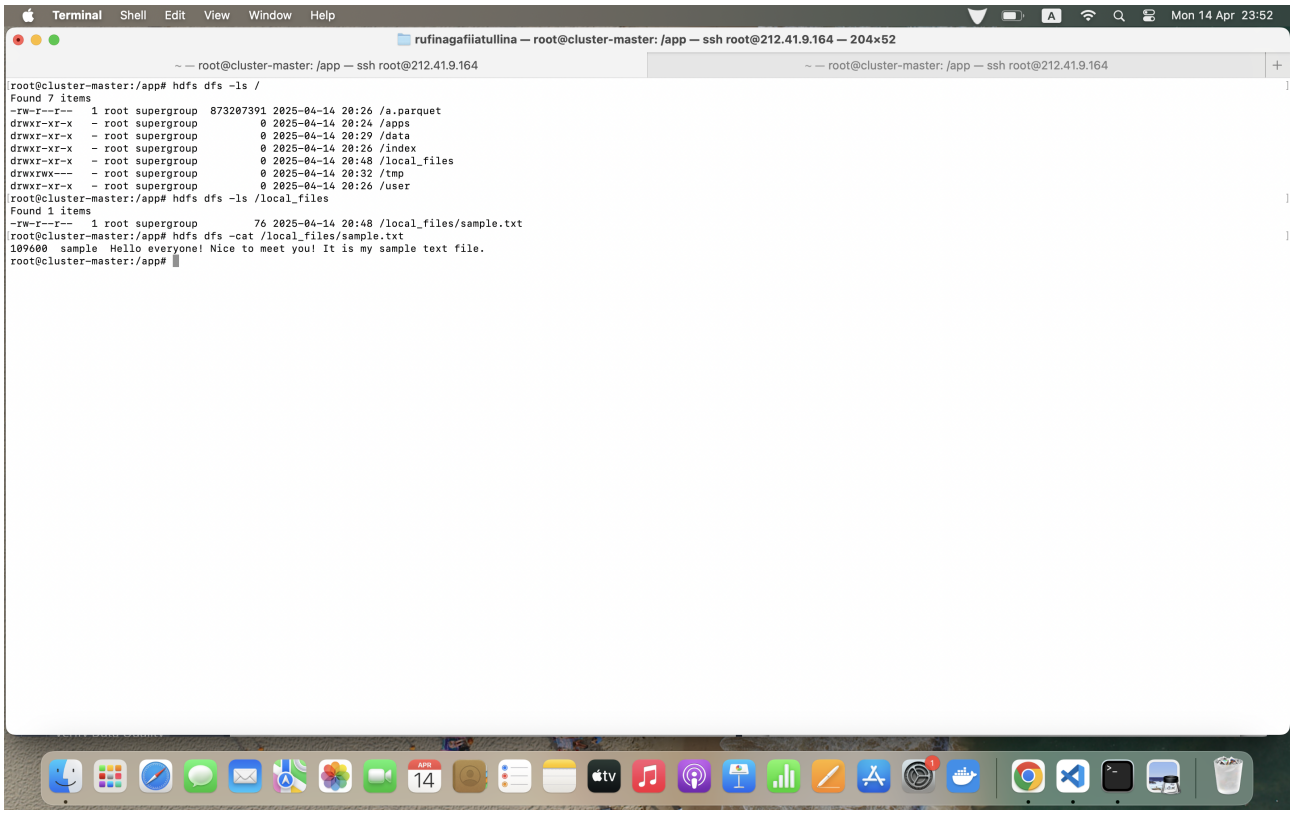

Figure 10: Check HDFS

After that I again check Cassandra tables for term 'hello'. As the result, values changed. Moreover, using doc_id from HDFS result, I find sample.txt in `doc_info` table. It means that index.sh index local file properly.

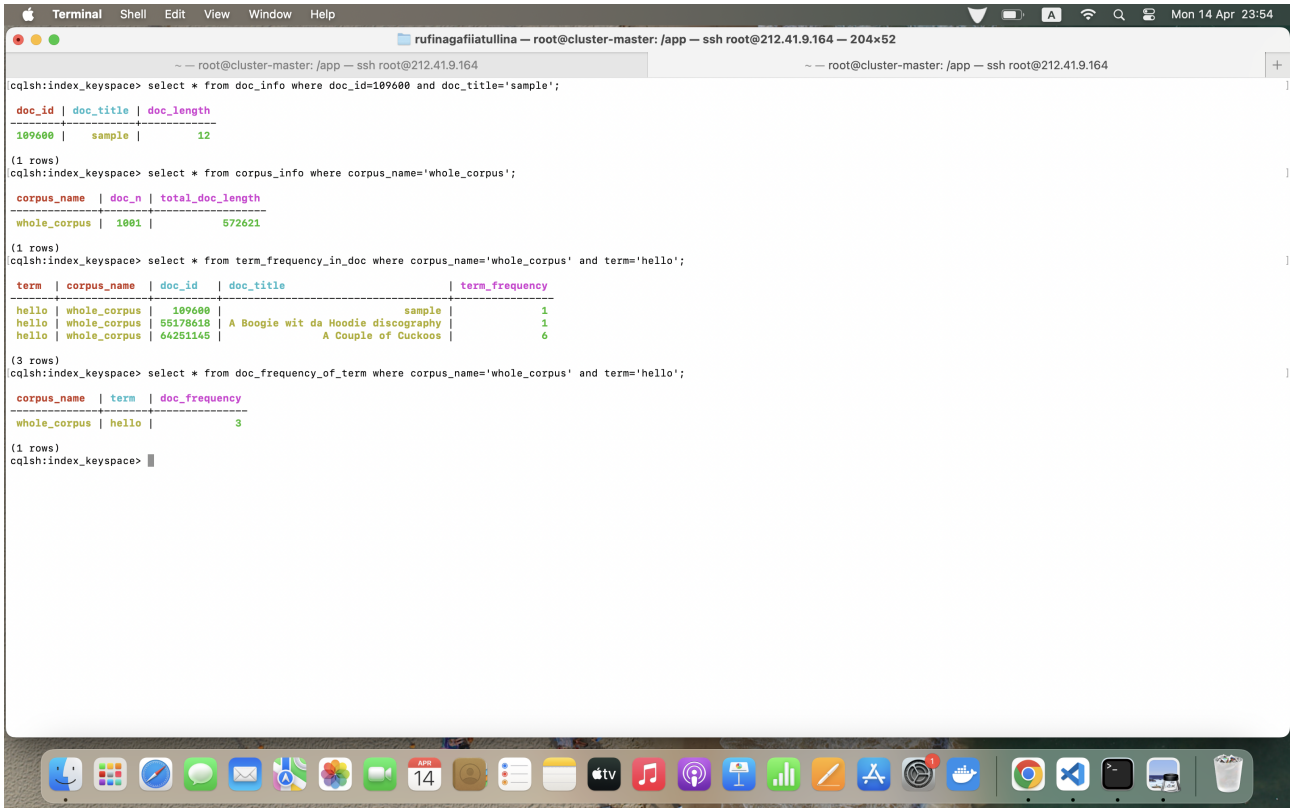

Figure 11: Checking sample.txt in Cassandra

2.2.3 Searching

I search documents for query "football game" that shown in fig.12. The results present in fig.13. Here we can notice that our search engine gives us relevant documents (e.g. "A Football Life", "A Fan's Notes (film)").

I also search documents for non-existing term in query that shown in fig.14. The result is empty file that shown in fig.15 (nothing is printed to output).

I search documents for query "hello" that shown in fig.16. In the results you can see our local file that we indexed in previous examples that shown in fig.17. This further proves that the file has been indexed.

I search documents for query "create innovation" that shown in fig.18. The results present in fig.19. Here we can notice that our search engine gives us relevant documents.

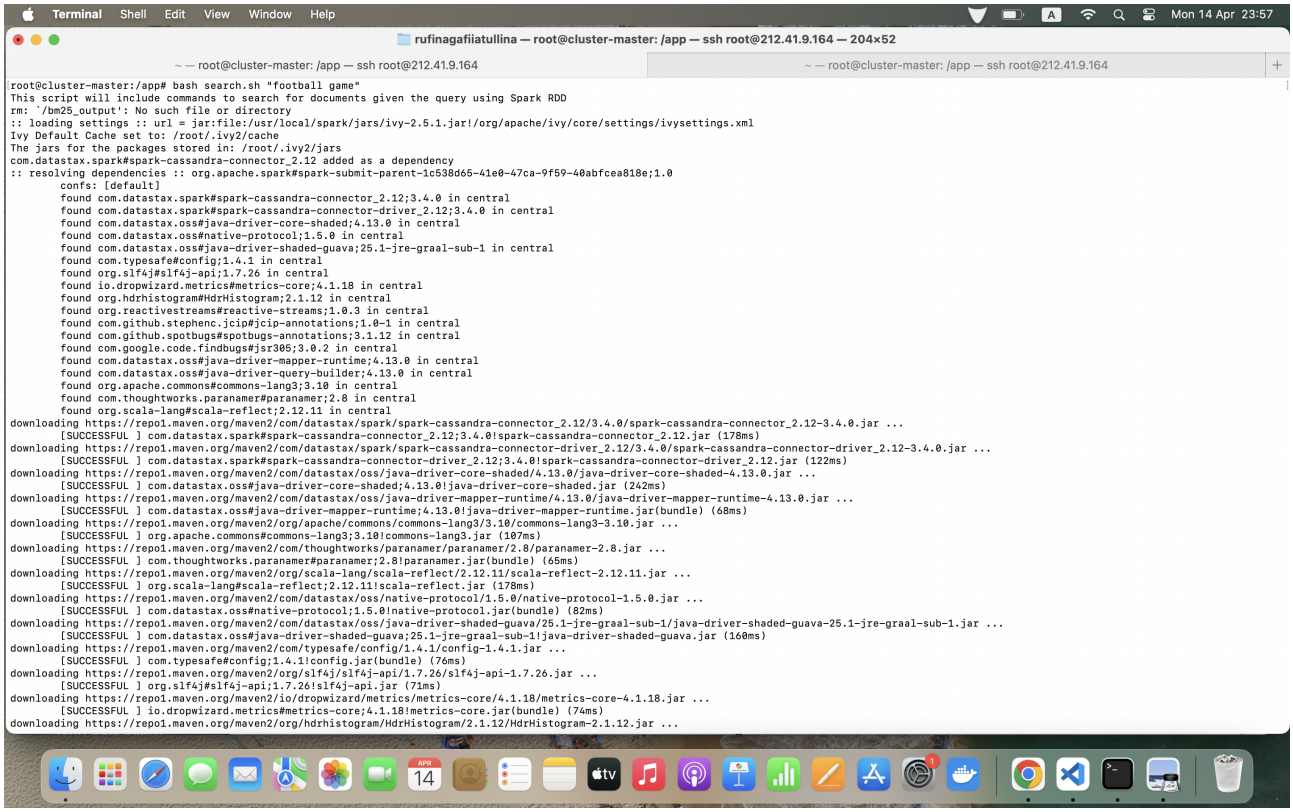

Figure 12: search.sh "football game"

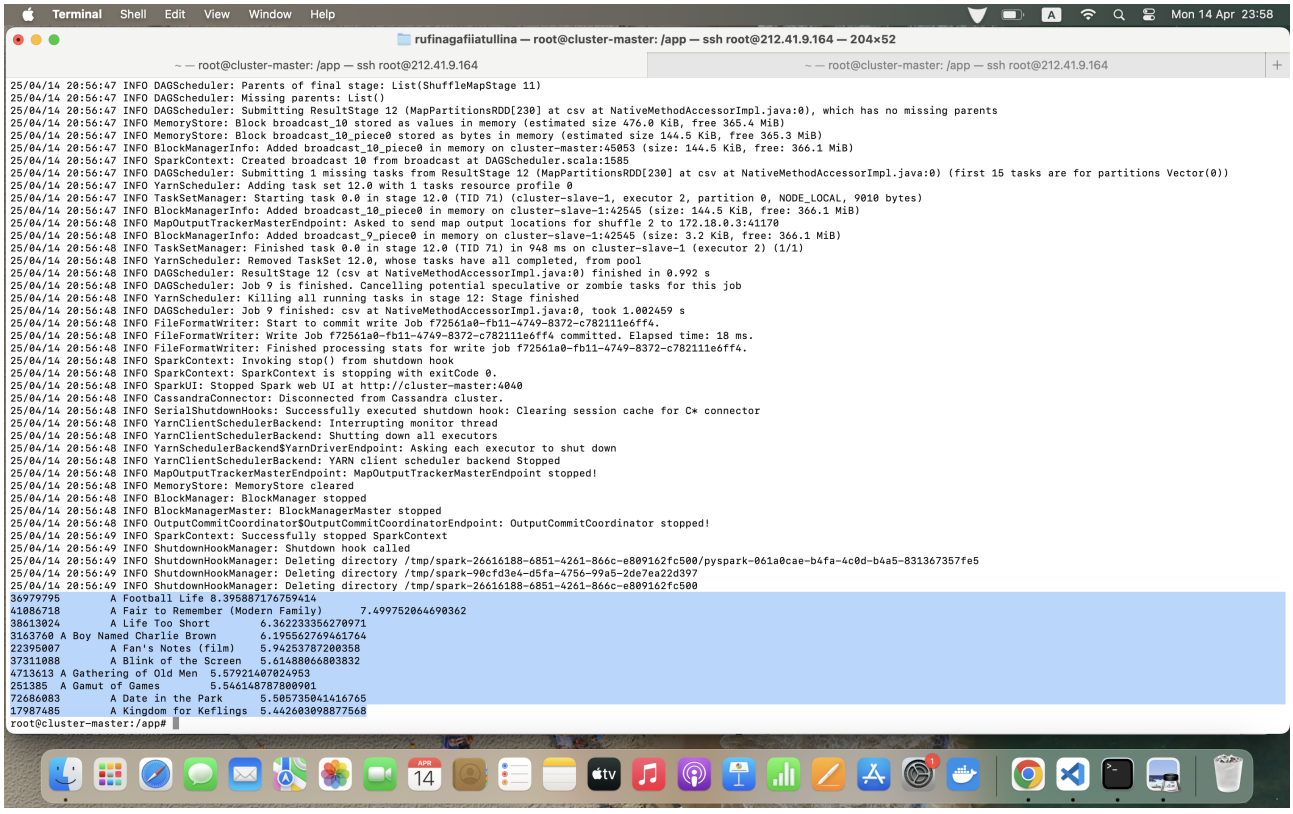

Figure 13: Results for "football game"

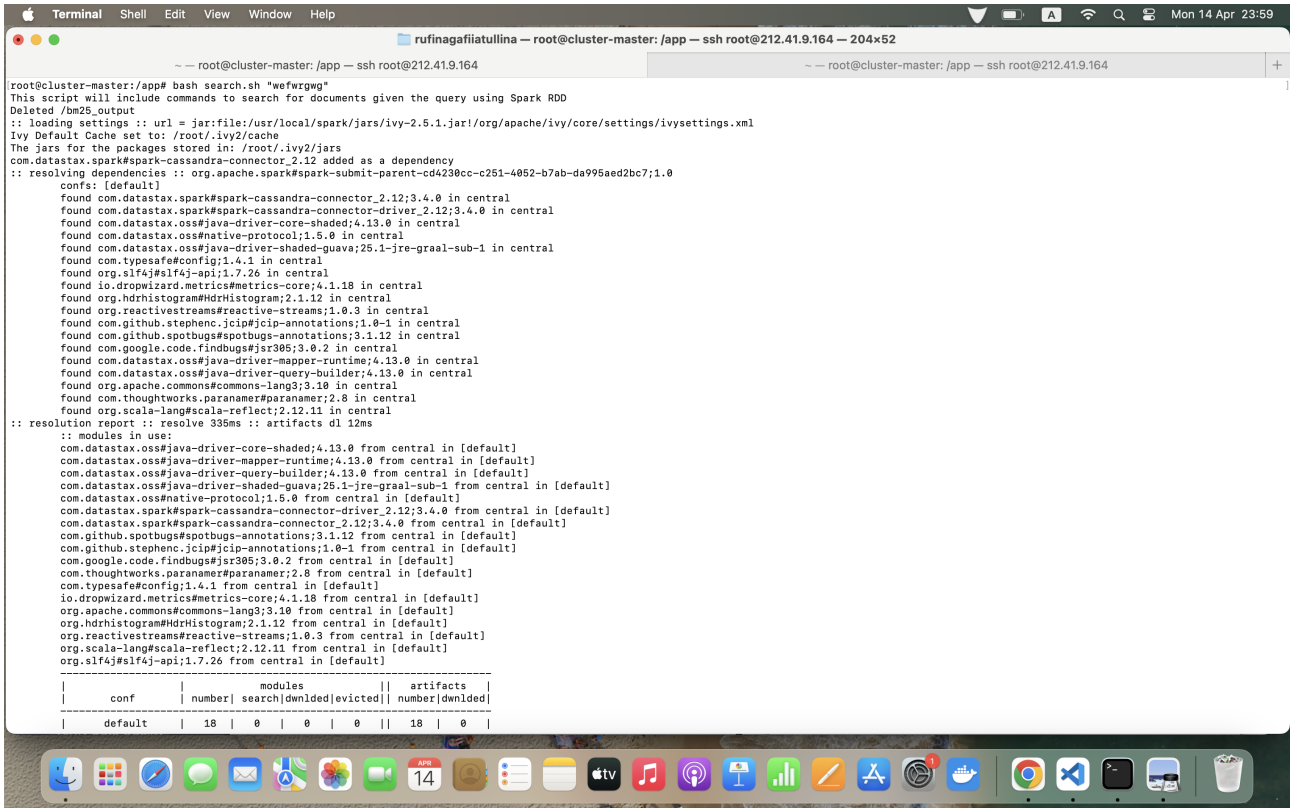

Figure 14: search.sh "wefwrgwg"

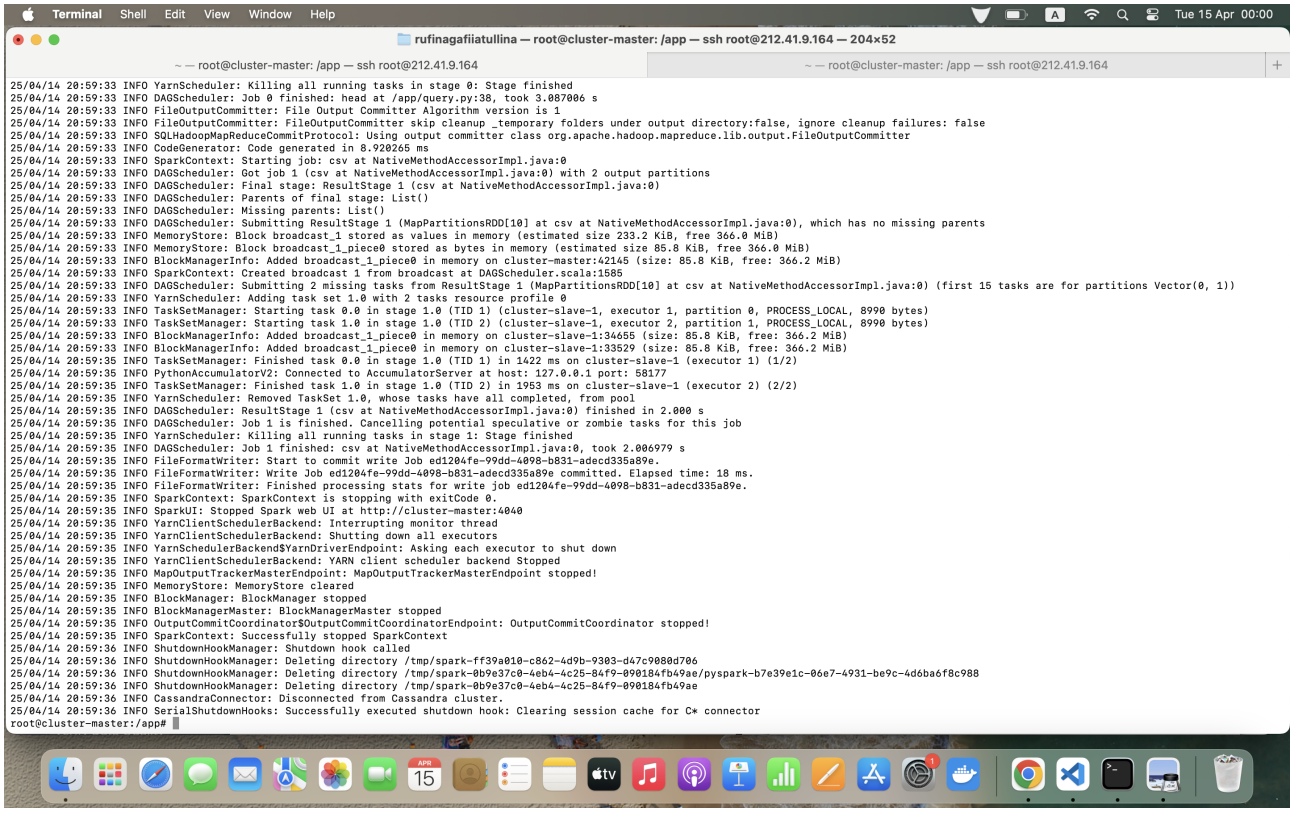

Figure 15: Results for "wefwrgwg"

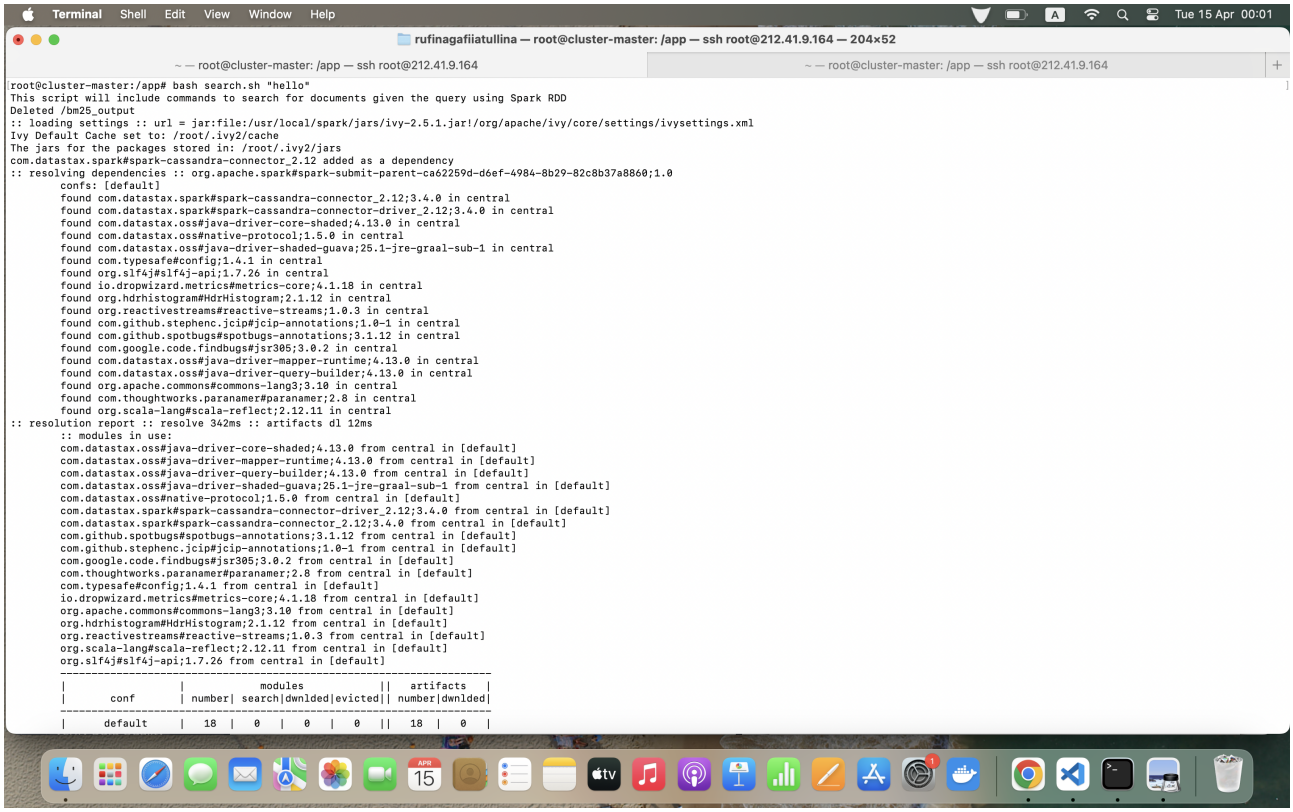

Figure 16: search.sh "hello"

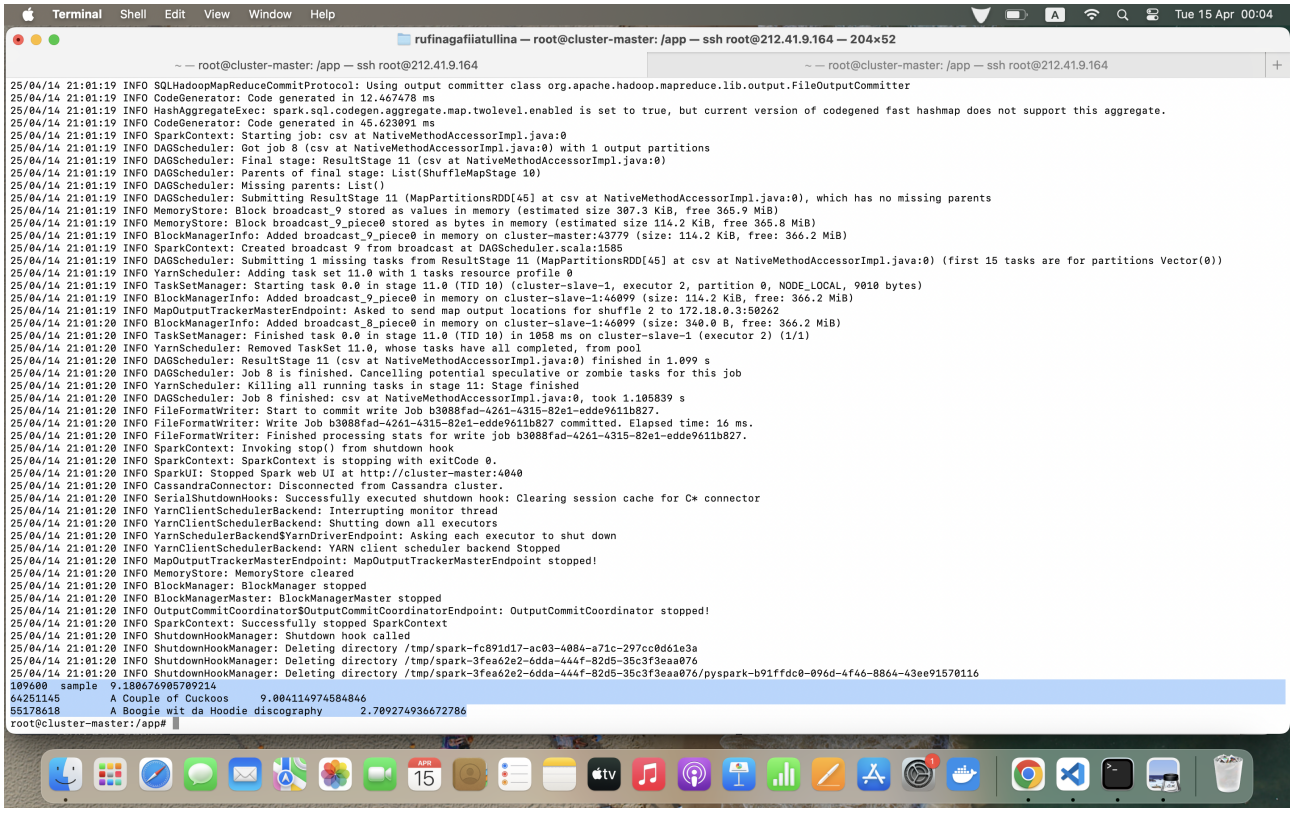

Figure 17: Results for "hello"

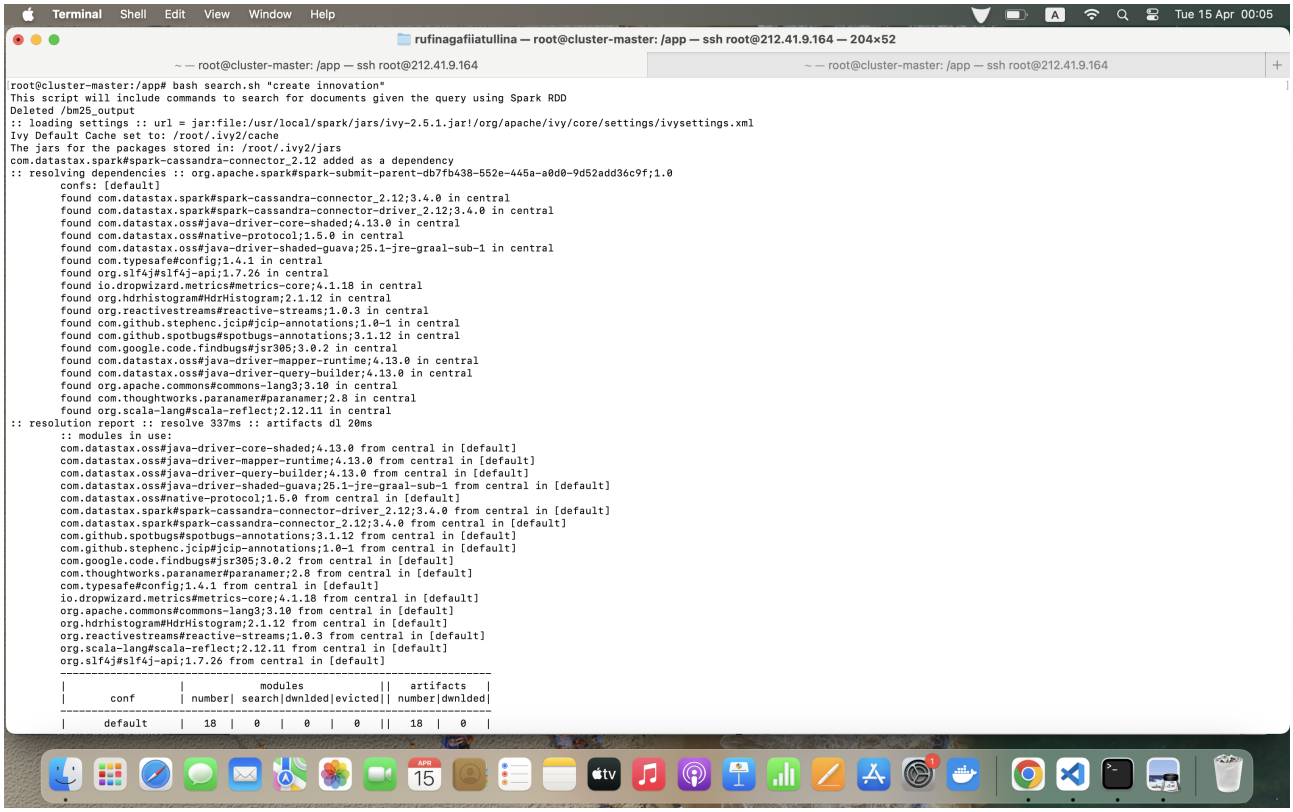

Figure 18: search.sh "create innovation"

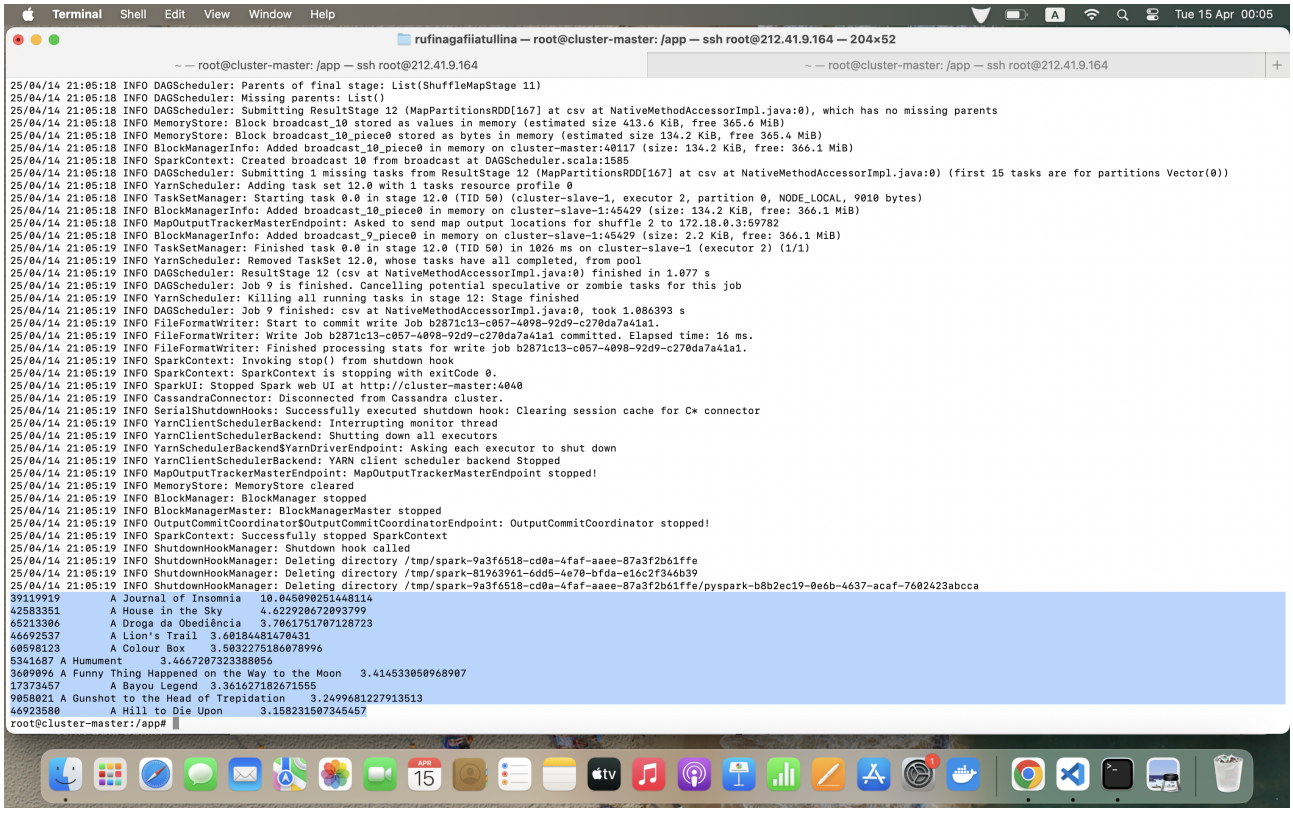

Figure 19: Results for "create innovation"