BigData Assignment 2. Search Engine

Anastasia Pichugina DS-02

Repository: https://github.com/caxapb/BD_assignment2

Methodology

The search engine system is built with several technologies:

- Hadoop and Yarn: distributed data processing
- Cassandra: used database
- PySpark: efficient data processing and scores calculation

Application Workflow

The workflow is exactly the same as described in the assignment description:

1) .parquet file is processed to separate .txt files,
2) all data is moved to HDFS,
3) MapReduce pipeline computes required statistics, and saves it
4) app.py connects to the cassandra-server, creates the keyspace and tables, and inserts data from the reducer's output,
5) query.py reads the input query, takes data from Cassandra, calculates the BM25 score and prints the resulting queries.

Some implementation details:

- docker-compose up starts the application. The service cluster-master has the entrypoint app.sh that runs all scripts in the correct order: start-services.sh, prepare_data.sh, index.sh, and search.sh.
- start-services.sh:
    - starts all services required for Hadoop components running other scripts: HDFS, Yarn, web UI for MapReduce, and prepares Spark.
    - Sets the virtual environment and installs dependencies from the requirements.txt
- prepare_data.sh: loads a.paquet to the HDFS, prepares data using util .py files:
    - prepare_data1.py: goes through the main files and creates new separate .txt files. All files are loaded to the HDFS /data folder as <doc_id>_<doc_title>.txt.
    - prepare_data2.py creates the RDD object from those files and creates {doc_id}\t{title}\t{content} which will be saved to the /index/data/ in HDFS (part-00000 and _SUCCESS) using saveAsTextFile().

Note! index.sh doesn't accept any arguments. To change the amount of collected documents change the n parameter in the prepare_data1.py.

- MapReduce pipeline and data loading to Cassandra is conducted in the index.sh script. The script runs mapper1.py, reducer1.py, and app.py:

- mapper1.py: takes the input from the "/index/data" in HDFS, processes terms for each document and prints to the stdout info about terms and documents. The prints have 'tags': 'TF' and 'DOC'. Each row starts with one of them so that the reducer could distinguish document and terms data.
- reducer1.py: processes mapper1.py output. Based on tags, it collects all required statistics: term frequencies - how many times document x meets the term y, document frequencies - how many documents contain word x, document data (doc_id - doc_length - doc_title), and global statistics so that query.py doesn't compute them every time: document total count and average document length. The reducer's output is collected in the /tmp/index_output folder (part-00000 and _SUCCESS files).
- app.py connects to the Cassandra-server, creates the keyspace "search_engine" and tables, takes the /tmp/index_output/part-00000 content and inserts it into tables.

- Then search.sh is running. This script must be run with an argument - input query. SparkSession is created and connected to the cassandra server. The input query is tokenized using regex. Cassandra tables are loaded and converted to the RDD objects:
  - terms_rdd: RDDwith tuples of "term" and "document frequency".
  - term_frequencies_rdd: RDD where each tuple is (term, (doc_id, term frequency value)). Since term frequencies are defined as "amount of term appearance id doc", each tuple is unique,
  - documents_rdd: RDD with tuples of (doc_id, (length, title))
  - stats (global stats from Cassandra) isn't converted to RDD but all needed values are extracted.
  - Based on obtained RDD objects, a new RDD is created: (doc_id, title, length, term frequency value, inverse document frequency value). Such tuples exist for each unique pair: (term from the query) - (document containing at least 1 word from the query).
  - The search.sh output is written to the console and output.txt

Cassandra schema and data storage:

Keyspace: "search_engine"

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

Tables:

terms:

```
CREATE TABLE IF NOT EXISTS search_engine.terms (
            term text PRIMARY KEY,
            df int
        );
```

Since the BM25 formula requires document frequency for each term, this table is created.

term_frequencies:

```
CREATE TABLE IF NOT EXISTS search_engine.term_frequencies (
            term text,
            doc_id int,
            tf int,
            PRIMARY KEY (term, doc_id)
        );
```

Here the primary key is purple, because Term Frequency is defined by term and doc_id

documents:

```
CREATE TABLE IF NOT EXISTS search_engine.documents (
            doc_id int PRIMARY KEY,
            length int,
            title text
        );
```

For each document we need to know its length, thus this table keeps these values.

global_stats:

```
CREATE TABLE IF NOT EXISTS search_engine.stats (
            key text PRIMARY KEY,
            value float
        );
```

Number of documents and average document length are constant, so we don't need to compute them every time processing input query, so this table keeps 2 of these values. The table has only 2 rows: ("key":"doc_total", "value": …), ("key":"avg_length", "value": …).

BM25 Search using RDD objects.

0) As it was mentioned before, Cassandra tables are loaded to RDD object and we have the following (*df* - documents frequency, *idf* - inverse document frequency, *tf* - term frequency):

N - total documents count
avg_length - average document length
terms_rdd - RDD with tuples: (term, df)
term_frequencies_rdd - RDD with tuples: (term, (doc_id, tf))
documents_rdd - RDD with tuples: (doc_id, (length, title))

1) Convert df values in terms_rdd to idf values. terms_rdd becomes (term, idf).
2) Join term_frequencies_rdd with it to get (term, ((doc_id, tf), idf)) and simplify it / transform to: (doc_id, (term, tf, idf)).
3) Join it with documents_rdd and transform to get: (doc_id, title, length, tf, idf). These tuples are unique for *doc_id - term* pairs, and this is exactly what we need. Since the formula iterates over all terms in a query and computes scores for each document for this term, this format allows map operation for each record.

4) Apply bm25_scores computing function (add a new field "score" to tuples) and reduce them by key summing scores for each document.
5) Sort documents by scores and extract top 10. Write them to the output.txt and the console.

To compute BM25 score I used the formula provided in the Assignment description:

$$BM25(q, d) = \sum_{t \in q} \log\left[\frac{N}{df(t)}\right] \cdot \frac{(k_1 + 1) \cdot tf(t, d)}{k_1 \cdot \left[(1 - b) + b \cdot \frac{dl(d)}{dl_{avg}}\right] + tf(t, d)}$$

The only 1 change I made: added 0.0001 to N/df(t) to avoid log(0).

Demonstration.

To run this project you need to:

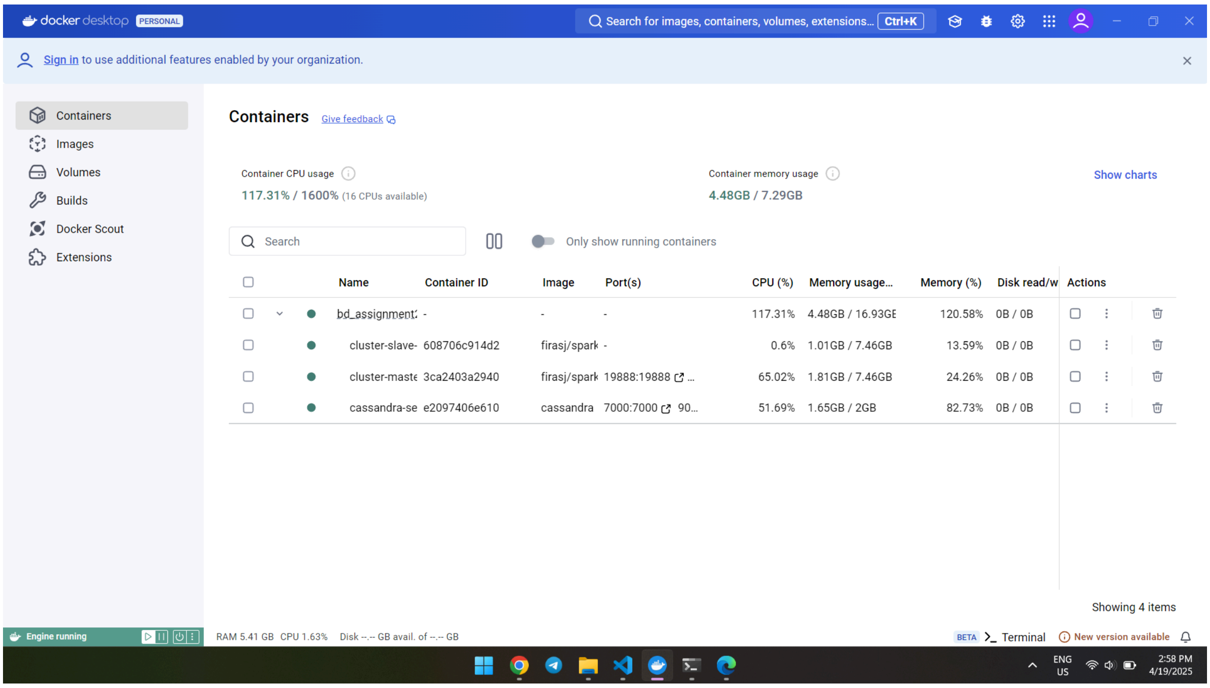
- Clone the git repository:

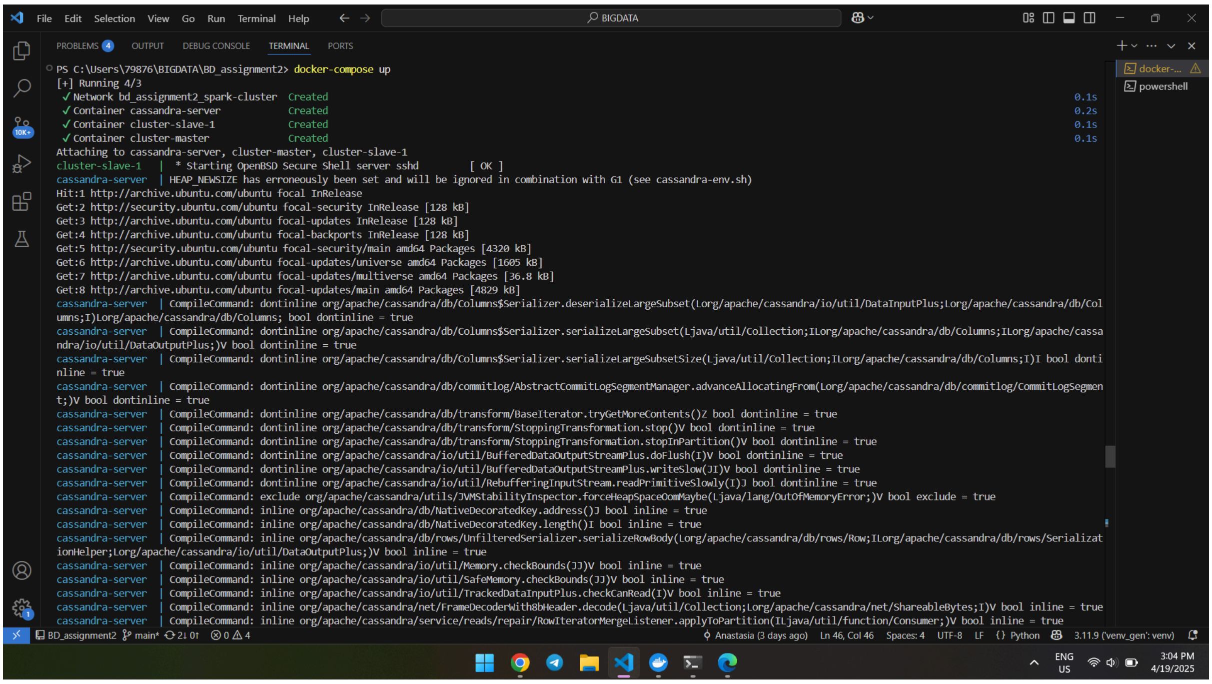
```
git clone git@github.com:caxapb/BD_assignment2.git
```

- Make sure you have a valid ".parquet" file in the app/data/ folder. If not, create the data folder and copy the a.parquet file inside.
- Run the docker compose:

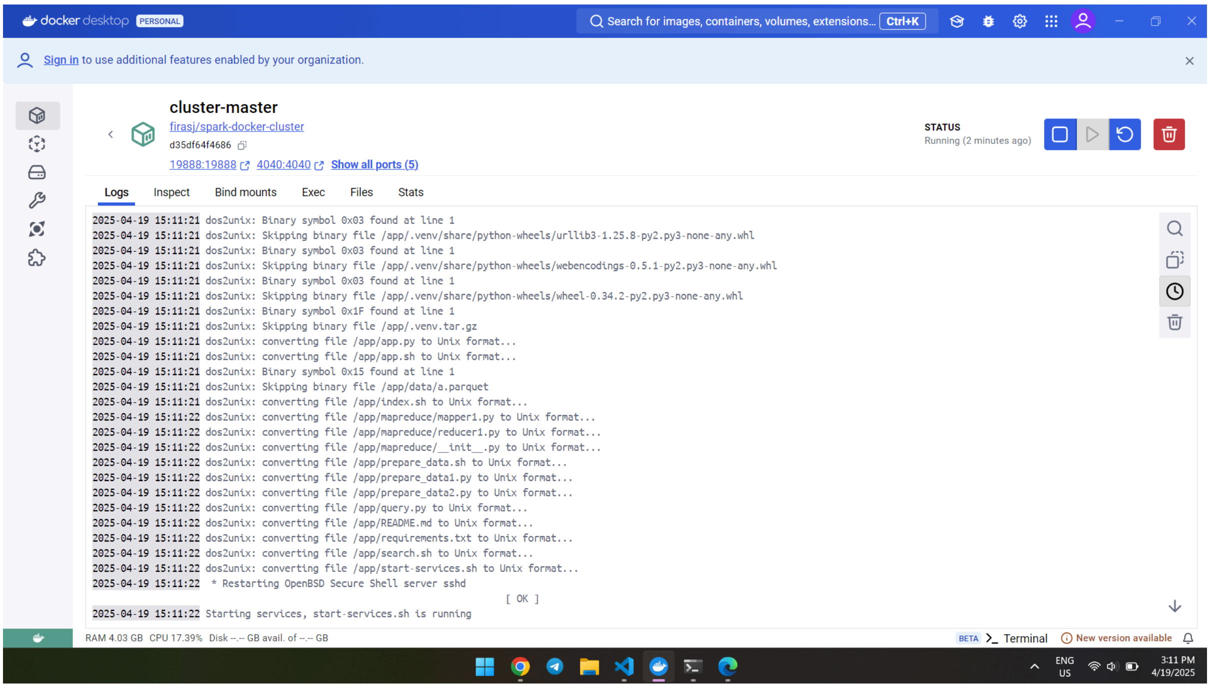
```
docker-compose up
```

This command will run app.sh that runs 3 containers (cluster-master, cassandra-server, and cluster-slave-1) and starts all other scripts.





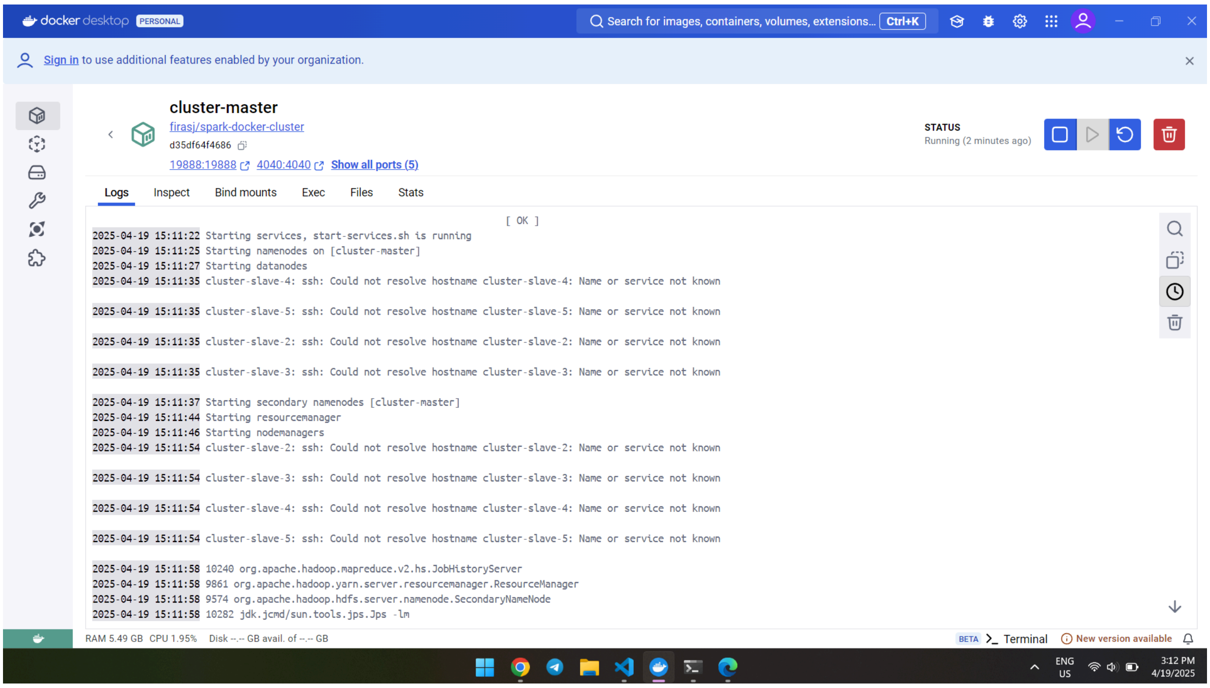
While loading you can see logs about converting files:

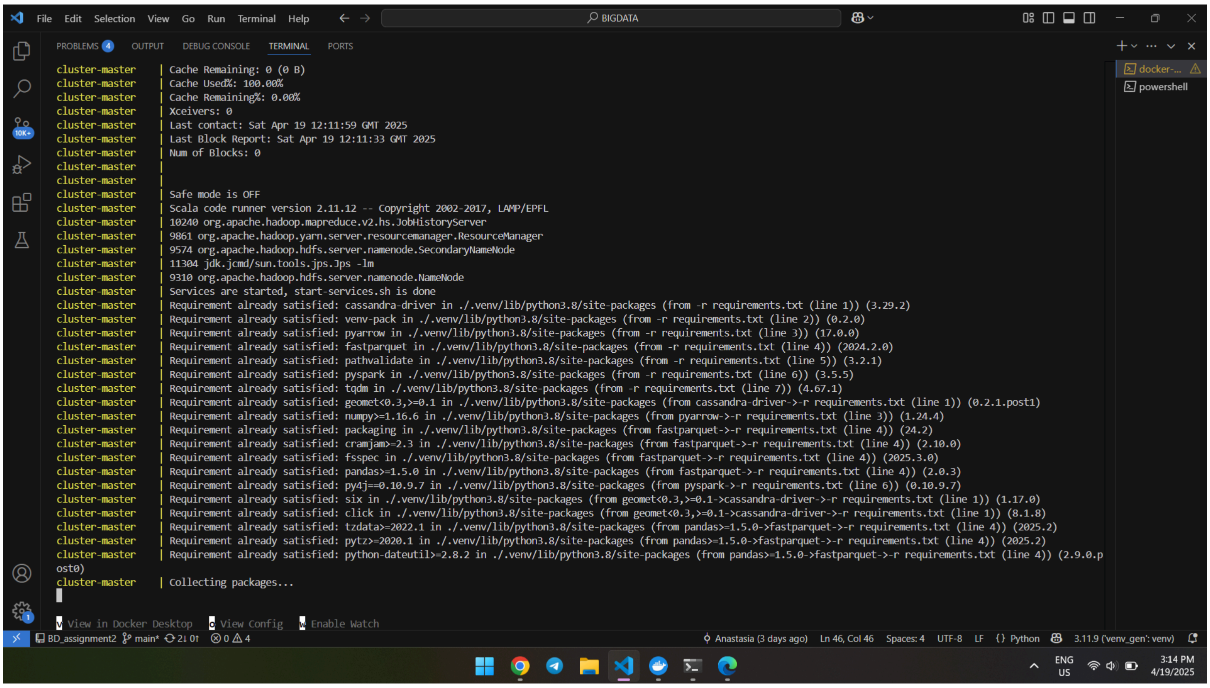


This is done to avoid errors in Windows-written files which have ‘\r\n’ at the ends of lines and took about 6 minutes for me. Since I already have the .venv folder, all files from .venv are also checked. However when cloning the repository, .venv doesn’t exist and will be created later. So, the conversion time will be reduced after cloning.

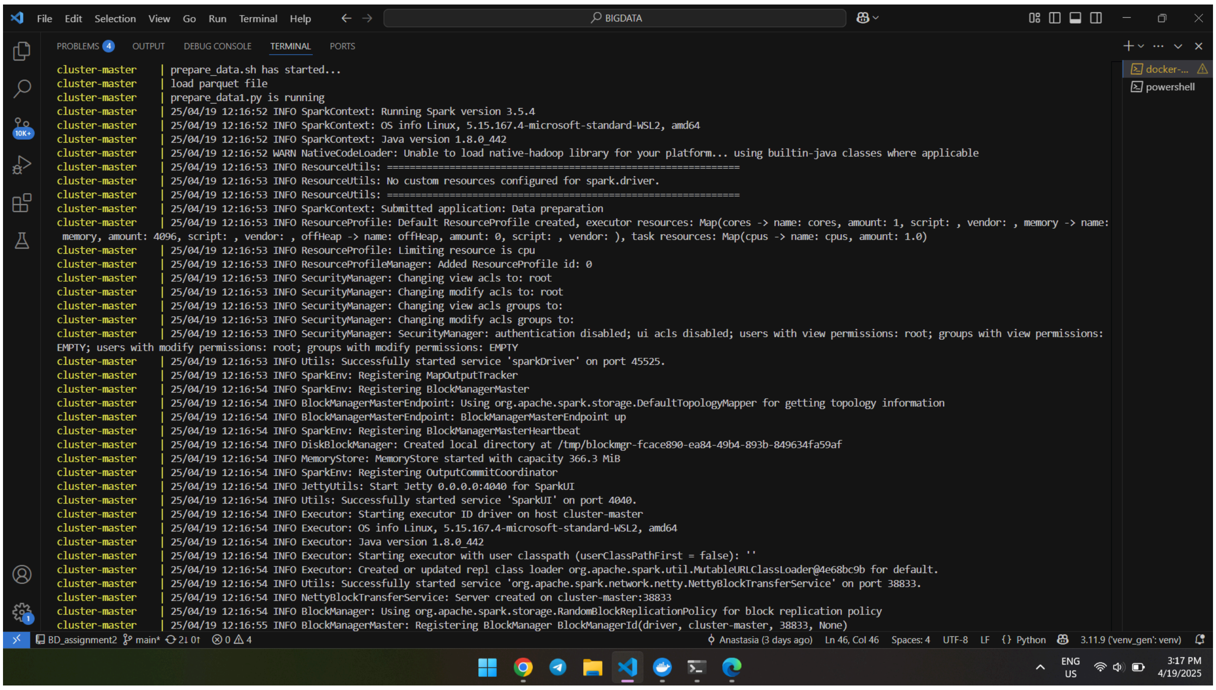
app.sh runs start-services.sh:

- starts Hadoop services
- sets the virtual environment and installs all packages from requirements.txt. This will take about 10 minutes.

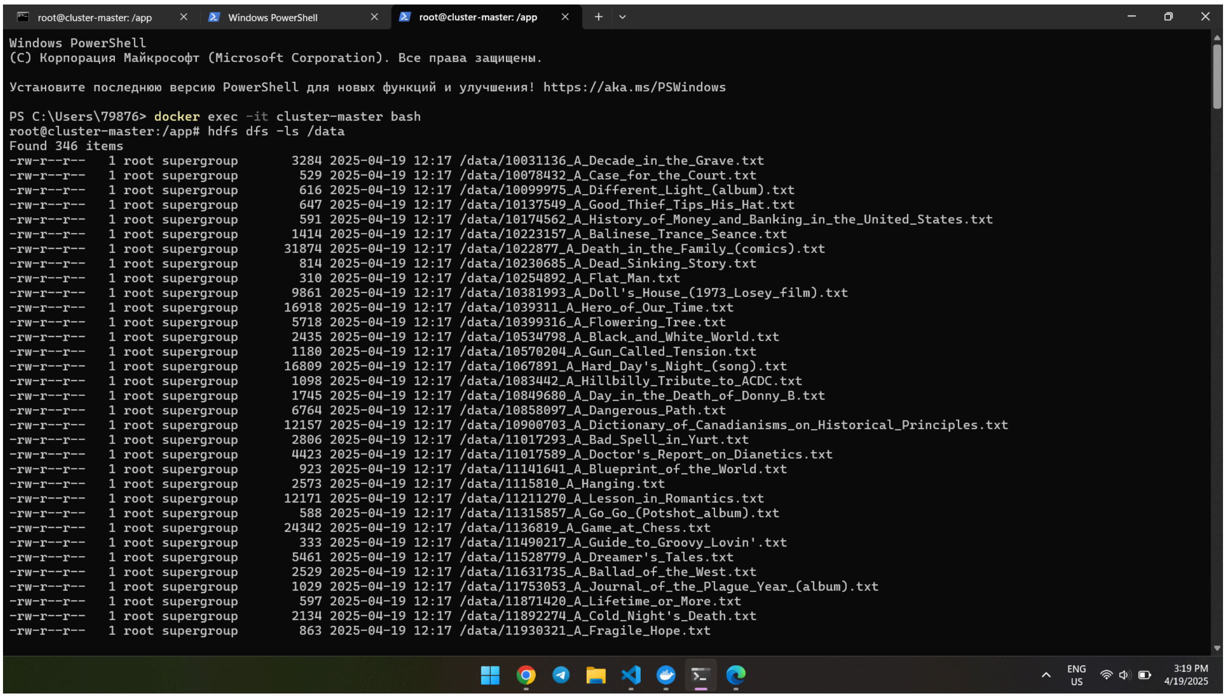




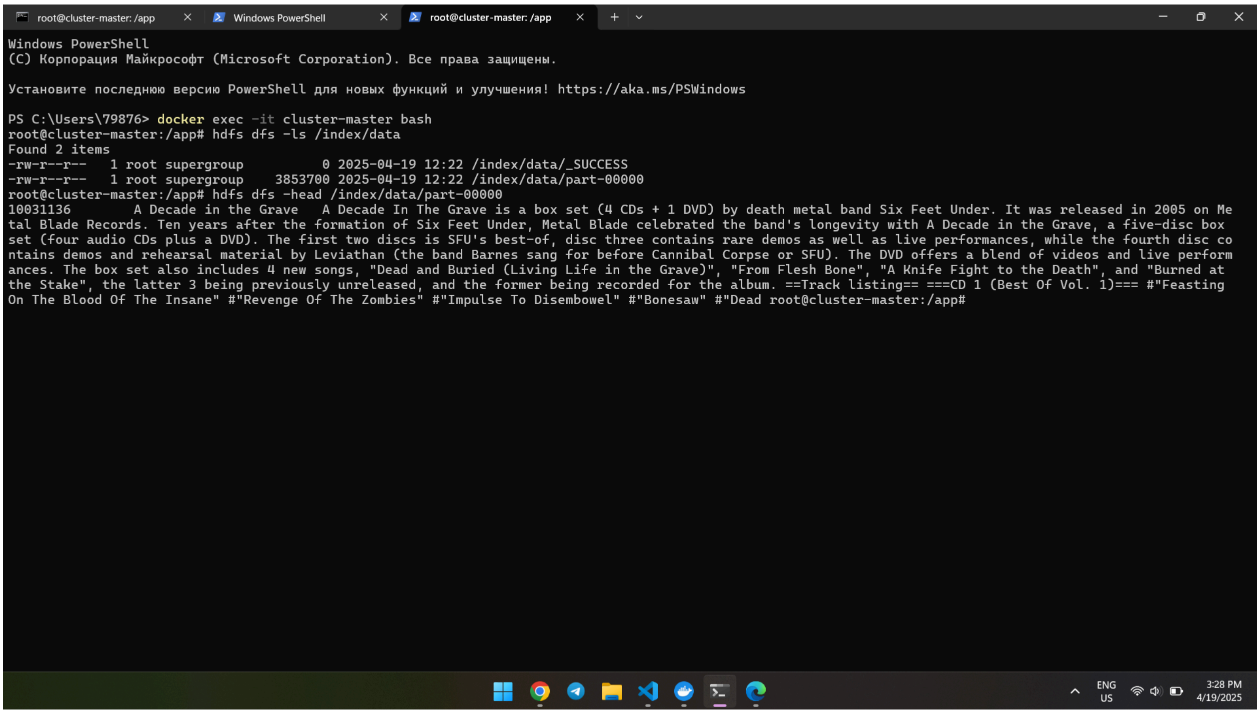
Then app.sh runs prepare_data.sh (here you can see logs like “load parquet file”, “prepare_data1.py is running”, etc.).



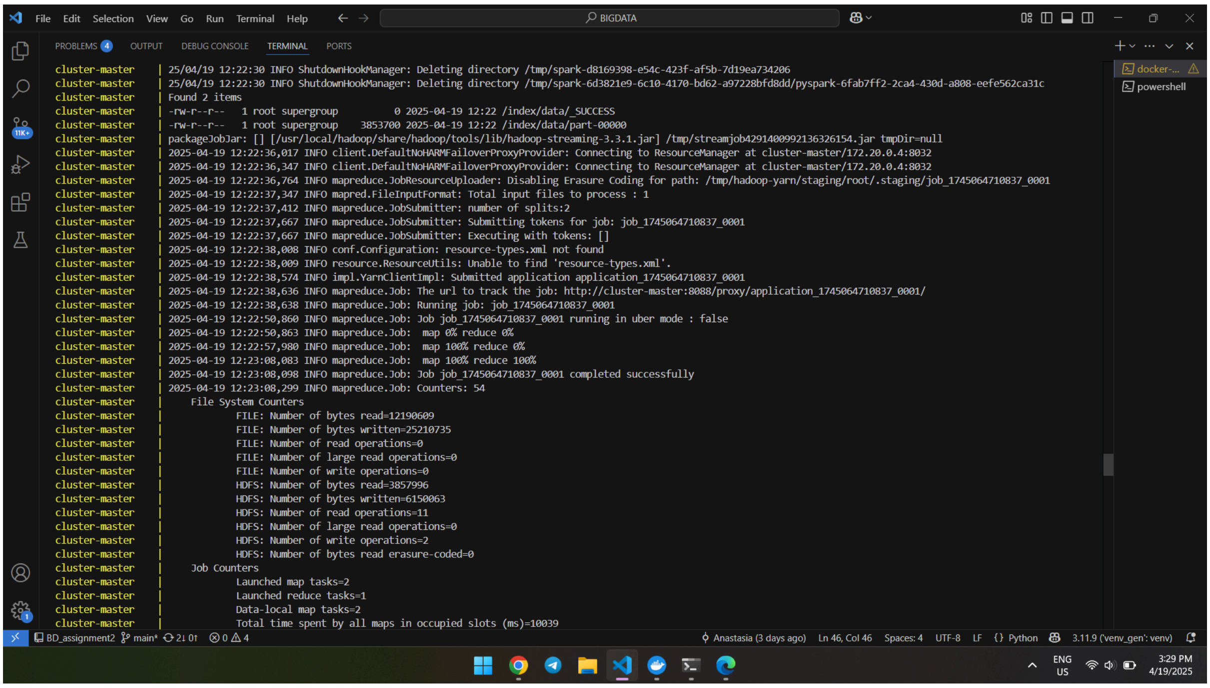
As a result hdfs will collect all needed data ready for MapReduce pipeline in /data folder:



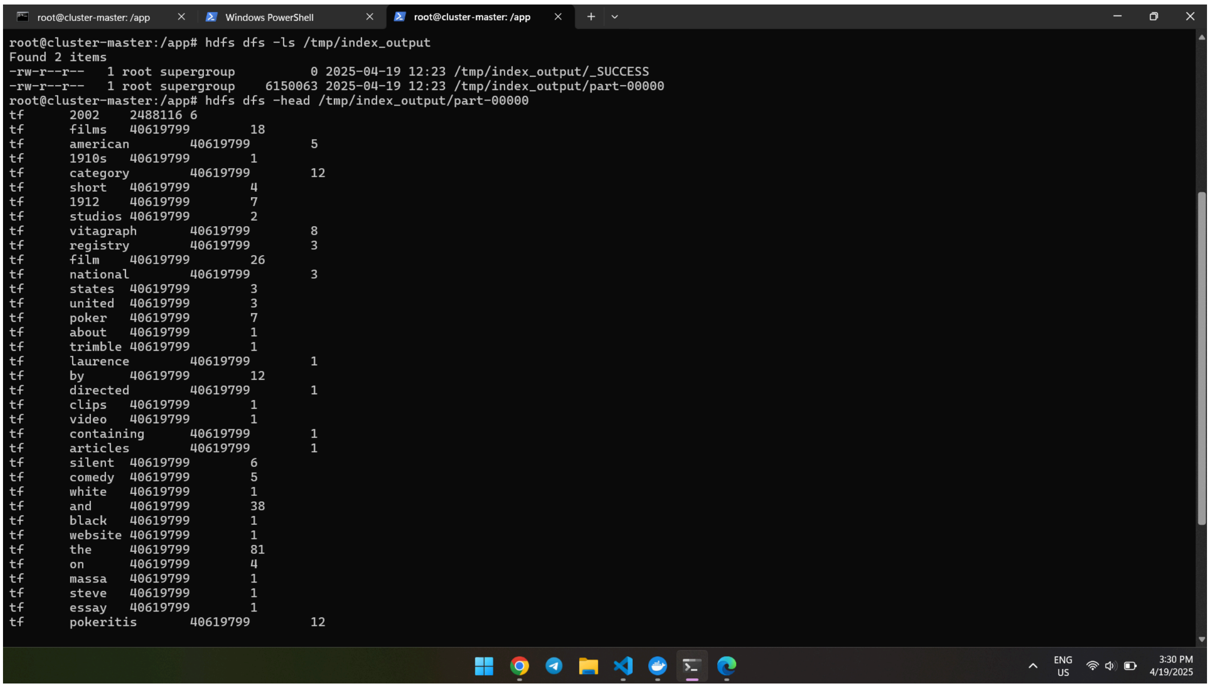
And RDD object after prepare_data2.oy is saved to /index/data:



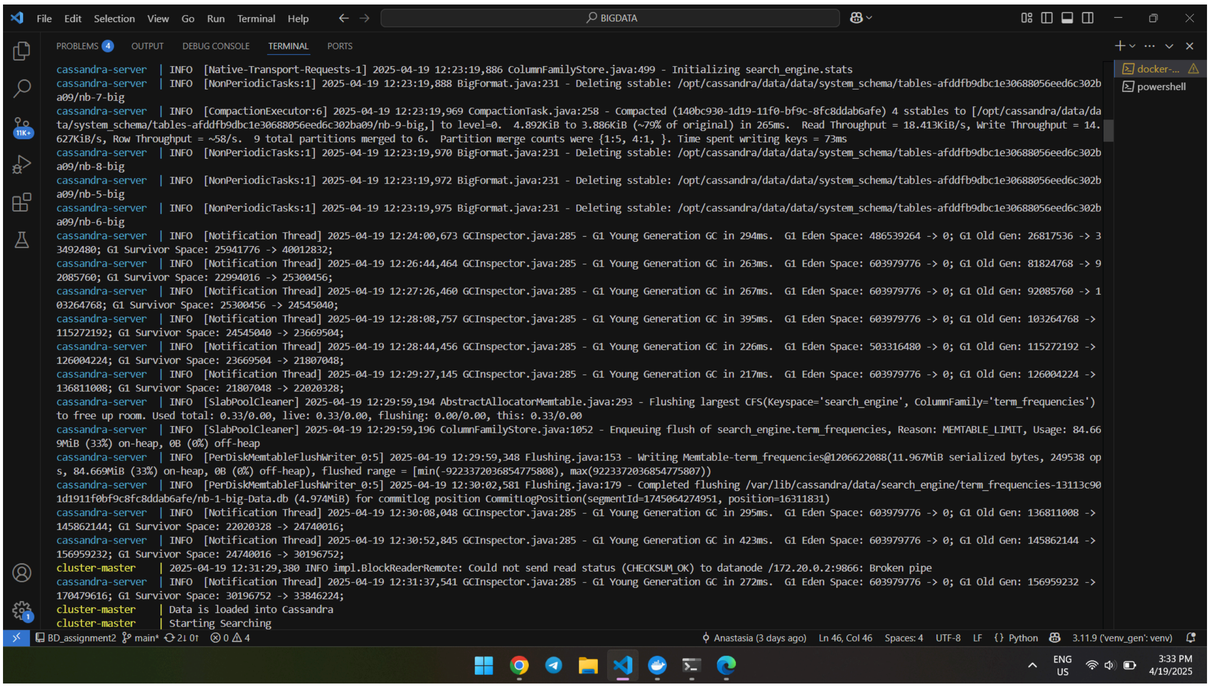
Next, the index.sh script is running:



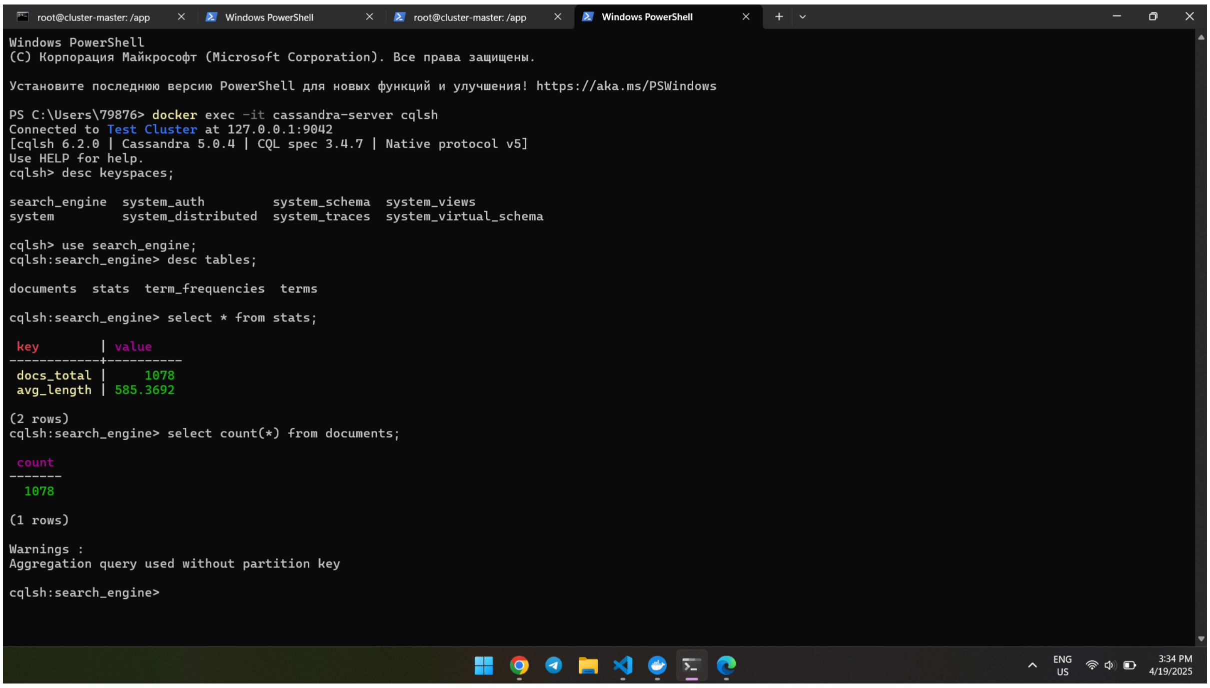
And here is the output of the reducer (in /tmp/undex_output/part-00000)



After the MapReduce pipeline app.py is running. It creates the keyspace, the tables, and inserts data (in the end wee see “Data is loaded into Cassandra”):

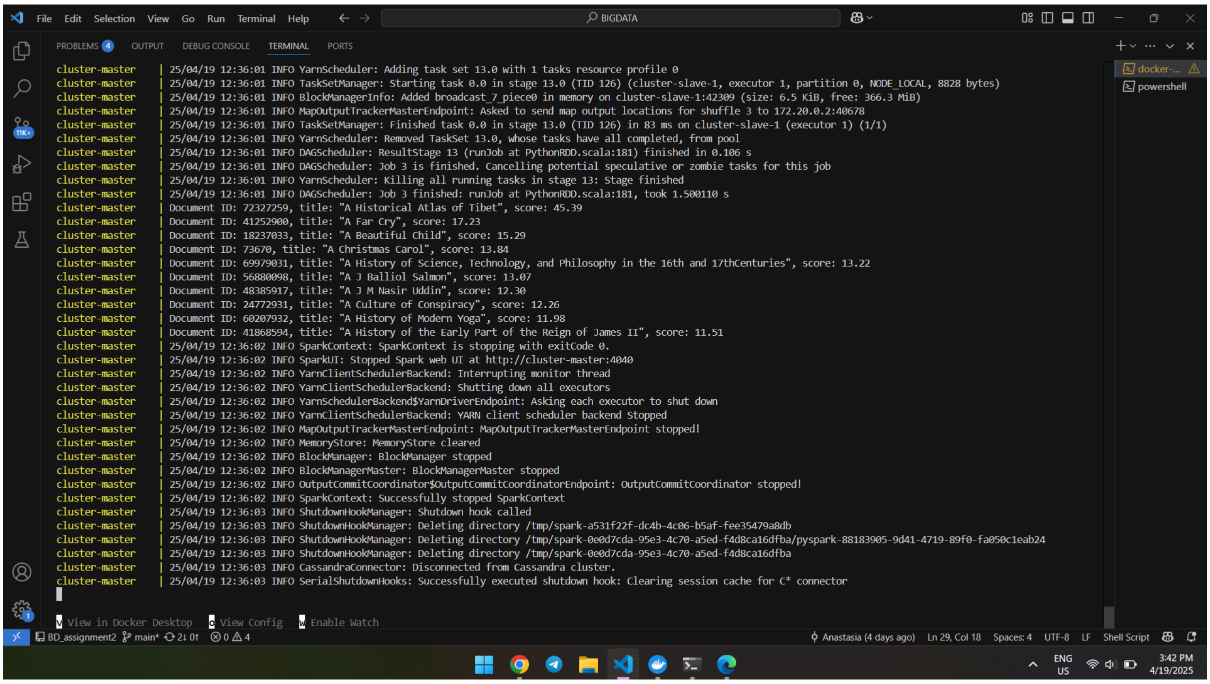


And the Cassandra tables check:

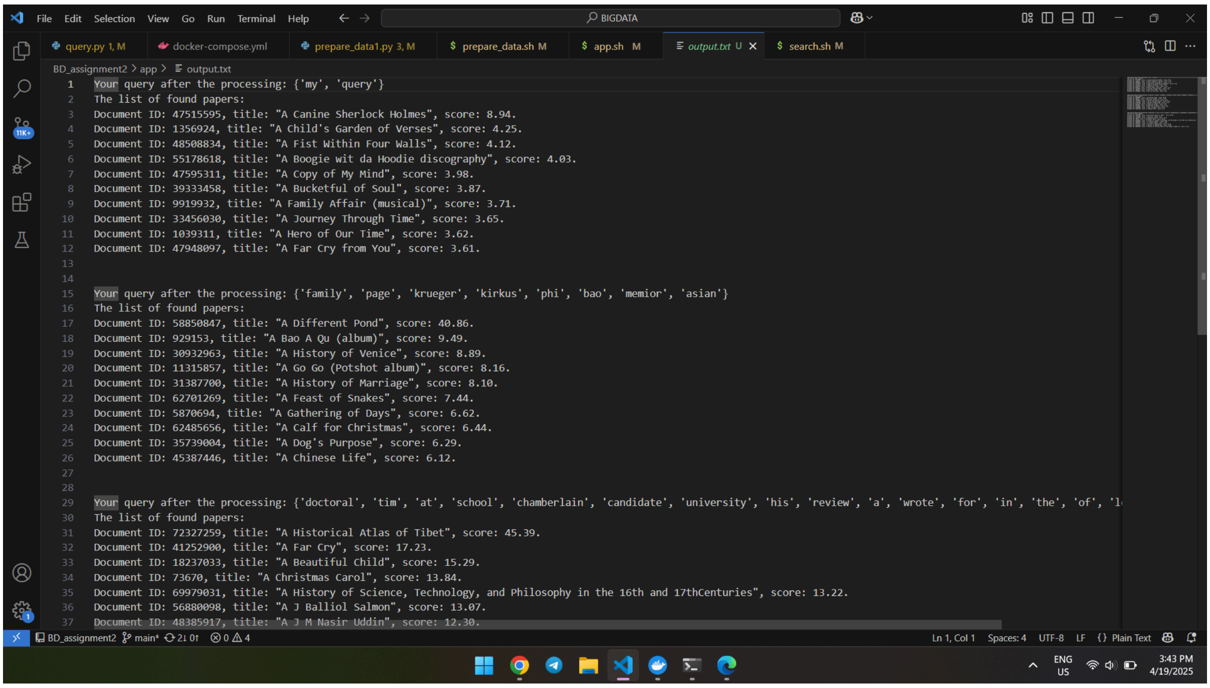


We don’t see 1100 documents in total (as written in data_preparation1.py: n=1100) because some documents have a problem with their titles: characters are not ASCII. As we are required to use at least 1000 documents I decided to ignore documents with not relevant titles and load 1100 (1078 in real life) documents.

Finally, the search.sh is running. According to the Assignment requirements, I ran this script 3 times for different queries: “my query” to see the ability to find anything, “asian family page memior kirkus krueger bao phi”, and “Tim Chamberlain, a doctoral candidate at BirkBeck, University of London wrote in his review for the London School”. The last 2 are from particular Wikipedia pages to see if the search engine found them (yes).



The lists of found files are appended to the output.txt:

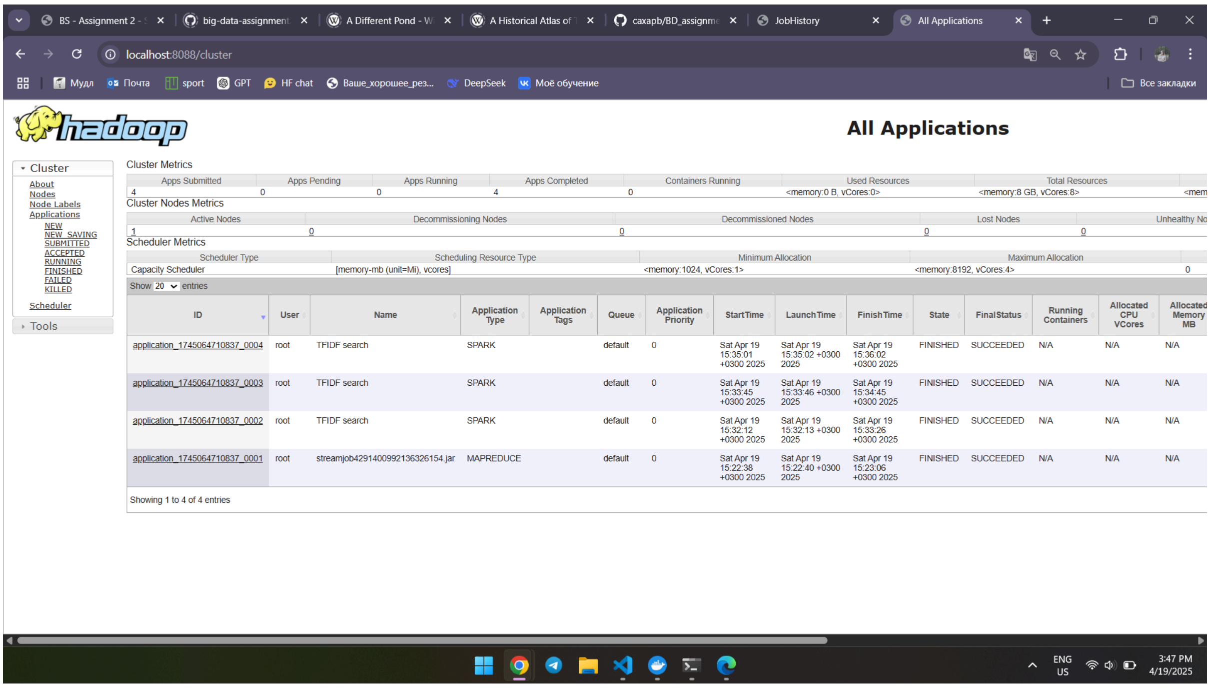


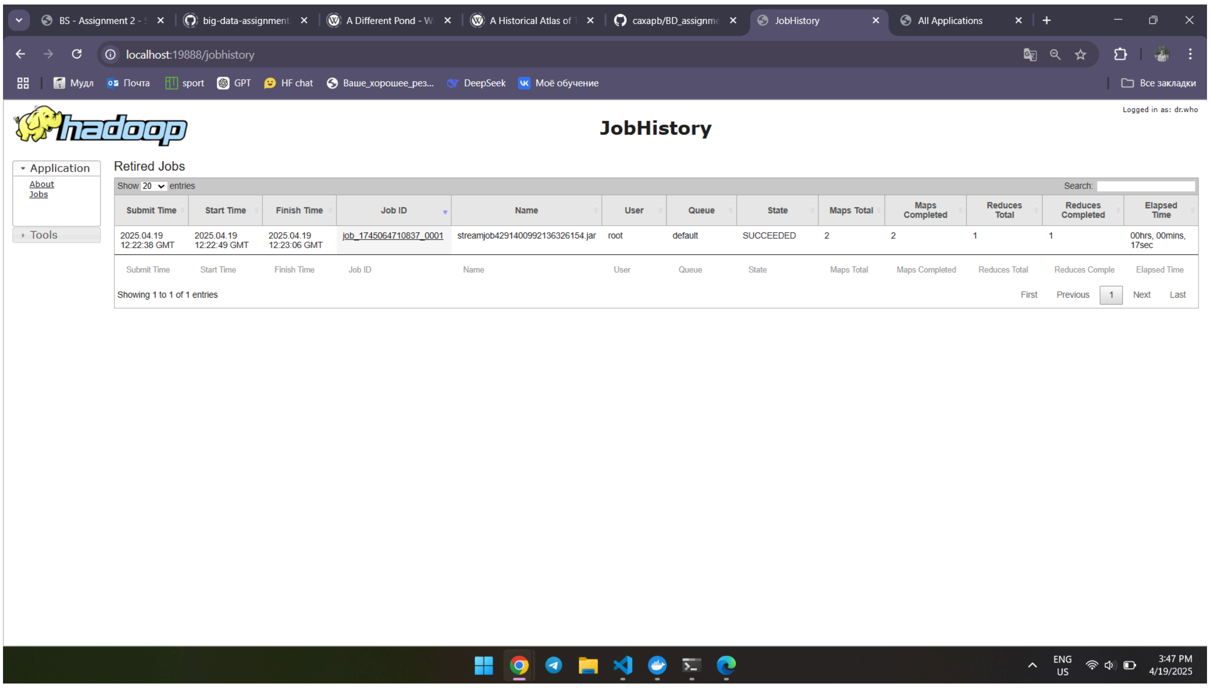
To run your own query you can run the following in a new terminal:

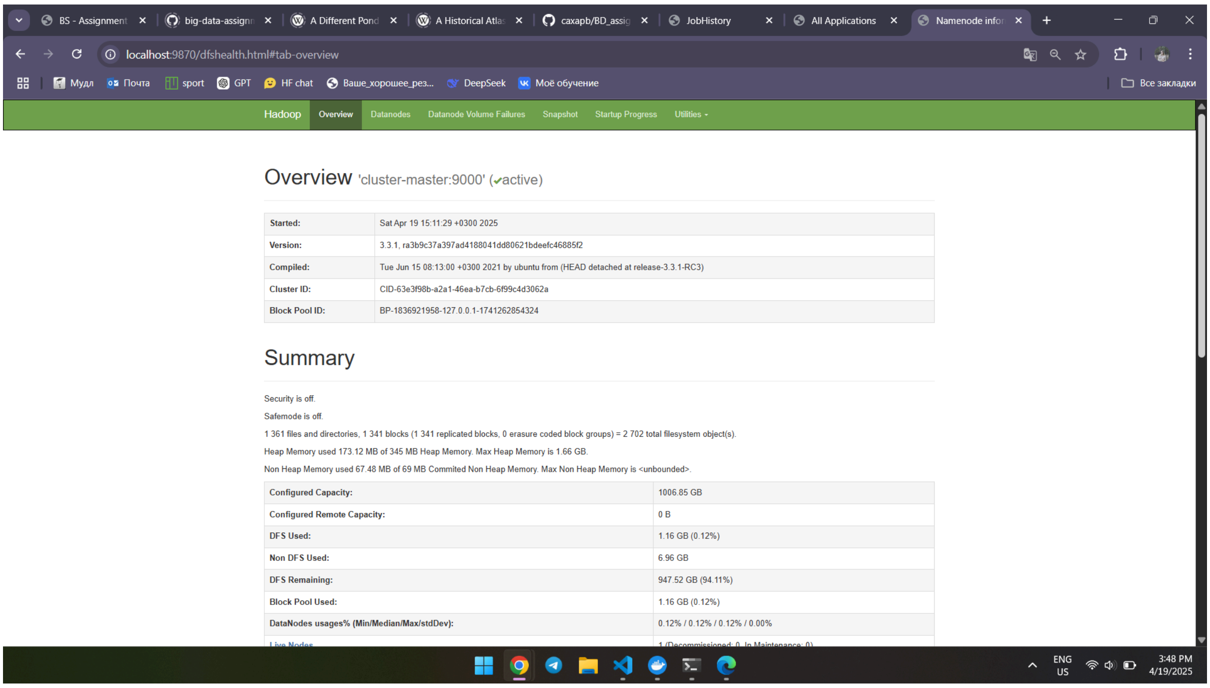
```
docker exec -it cluster-master bash
./search.sh “your own query”
```

To avoid cluster-master stops I added “tail -f /dev/null” at the end of the app.sh. It allows connection to the container from other terminals even after the app.sh is done.

Some web UIs:







In total the application run took about 30 minutes (with searching for 3 queries but without packages installation). The most time consuming: Dos2unix conversion (6 minutes for me), packages installation (about 10 minutes), data loading to HDFS (5 minutes), data loading to Cassandra (8 minutes). Query search takes about 1.5 minutes.