BigData Assignment 2

Anastasia Pichugina DS-02

Methodology.

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 in HDFS,
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 the 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.
- 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.
    - 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 amount of documents collected change the n parameter in the prepare_data1.py.

- MapReduce pipeline and data loading to the 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, for each documents it processes terms 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 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), and global statistics so that query.py doesn't compute them every time: document count and average document length. The reducer's output is collected in the /tmp/index_output folder (part-00000 and _SUCCESS files).

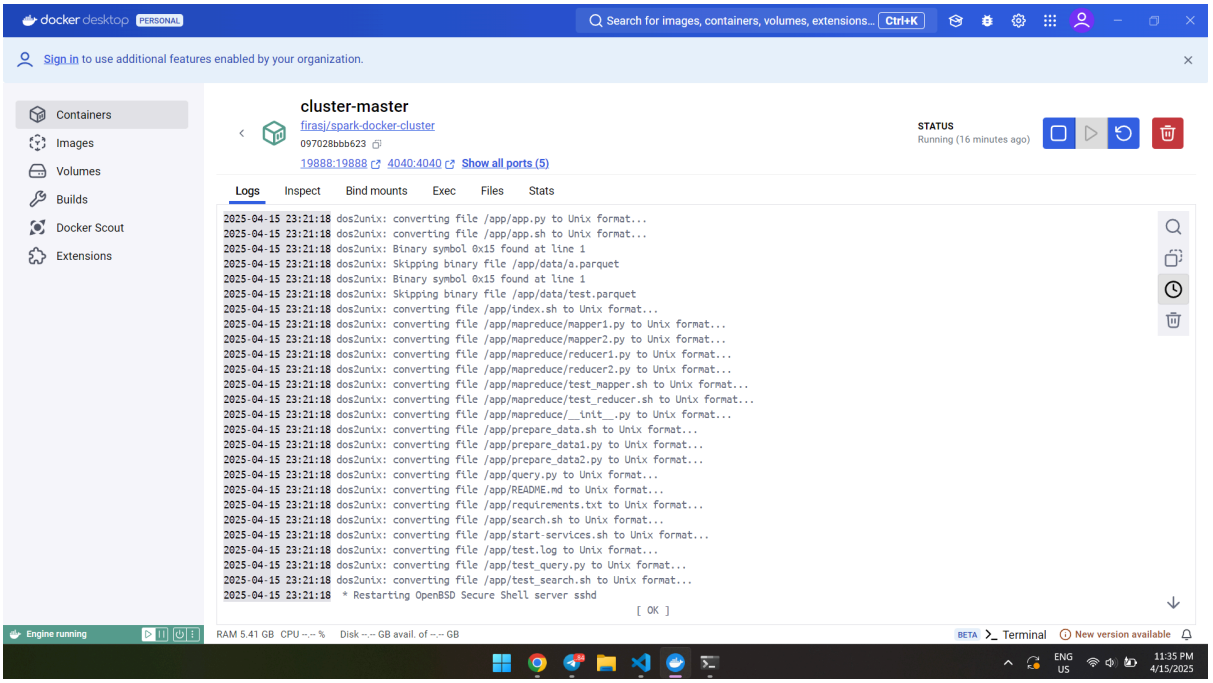
    - app.py connects to the Cassandra-server, creates the keyspace “search_engine” and tables, takes the /tmp/index_output/part-00000 content and inserts it into tables.
- Then search.sh is running. This script must be run with an argument - input query. SparkSession is created and connected to the cassandra server. The input query is tokenized using regex. Cassandra tables are loaded and converted to the RDD objects:
    - terms_rdd: rdd with tuples of “term” and “document frequency”,
    - term_frequencies_rdd: a list of tuples, where each tuple is ((doc_id, term), count). Since term frequencies are defined as “amount of term appearance id doc”, each tuple (doc_id, term) is unique,
    - documents_rdd: rdd with tuples of “doc_id” and “doc_length”
    - stats isn’t converted to RDD but all needed values are extracted.
- IDF map is created for each term in the query. BM25 scores are computed and top 10 documents are printed.

Cassandra schema:

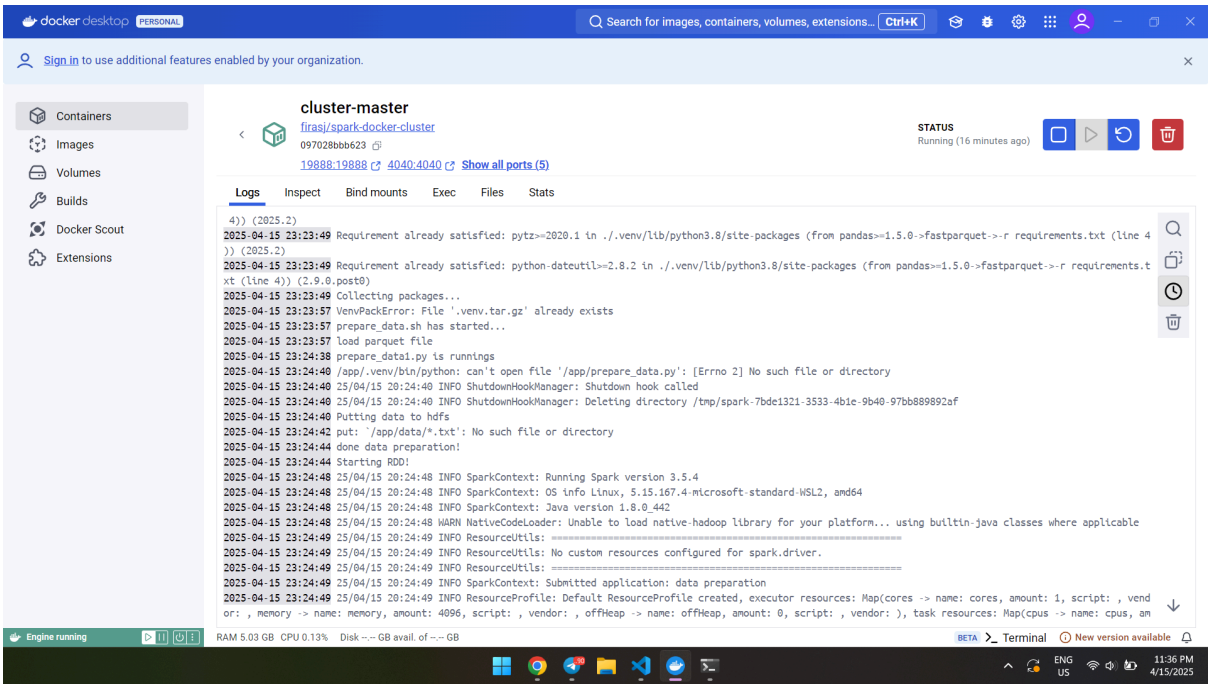
Keyspace: “”
Tables:
terms: term text Primary Key, df int. Since the BM25 formula requires document frequency for each term, this table is created.
term_frequencies: term text, doc_id int, tf int, Primary Key is (term, doc_id). Here the primary key is purple, because Term Frequency is defined by term and doc_id
documents: doc_id int Primary Key, length int. For each document we need to know its length, thus this table keeps these values.
global_stats: key text Primary Key, value int. 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.

Demonstration.

The whole project can be run via docker-compose:
docker-compose up –build
This command will run app.sh that 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. app.sh sets the virtual environment and installs all packages from requirements.txt, runs prepare_data.sh (here you can see logs like "load parquet file", "prepare_data1.py is running", etc.).

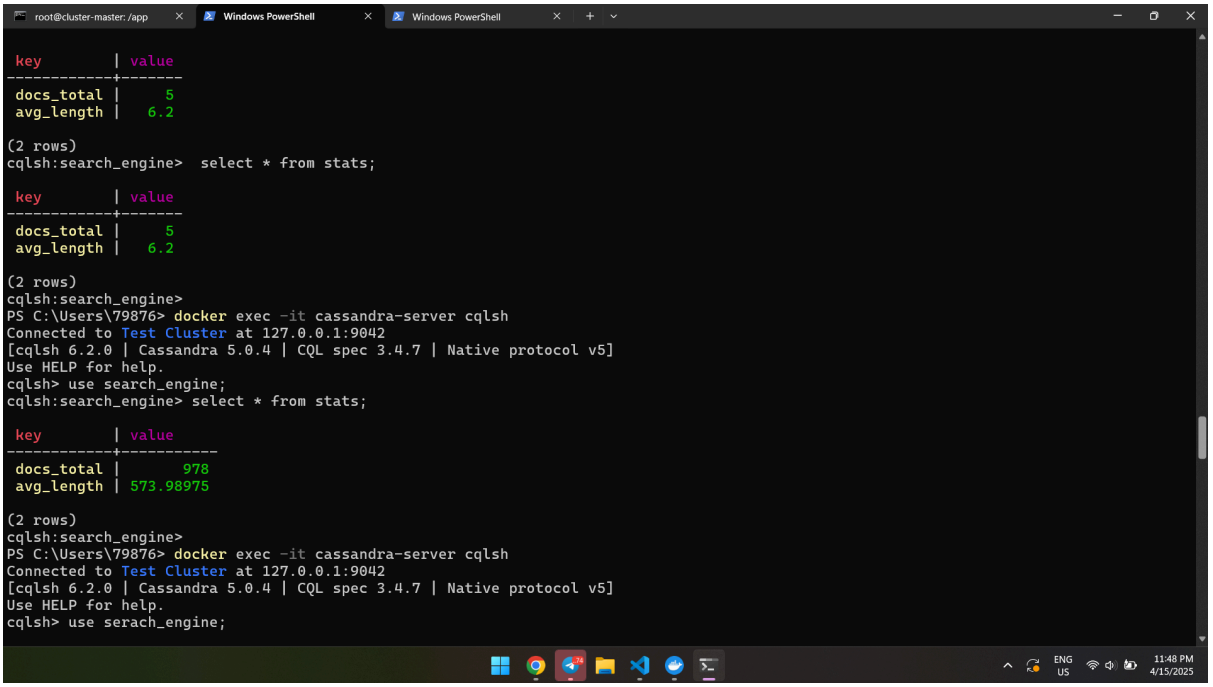


As a result hdfs will collect all needed data ready for MapReduce pipeline. Next, other scripts will be run. As a result, found documents will be saved to the output.txt.

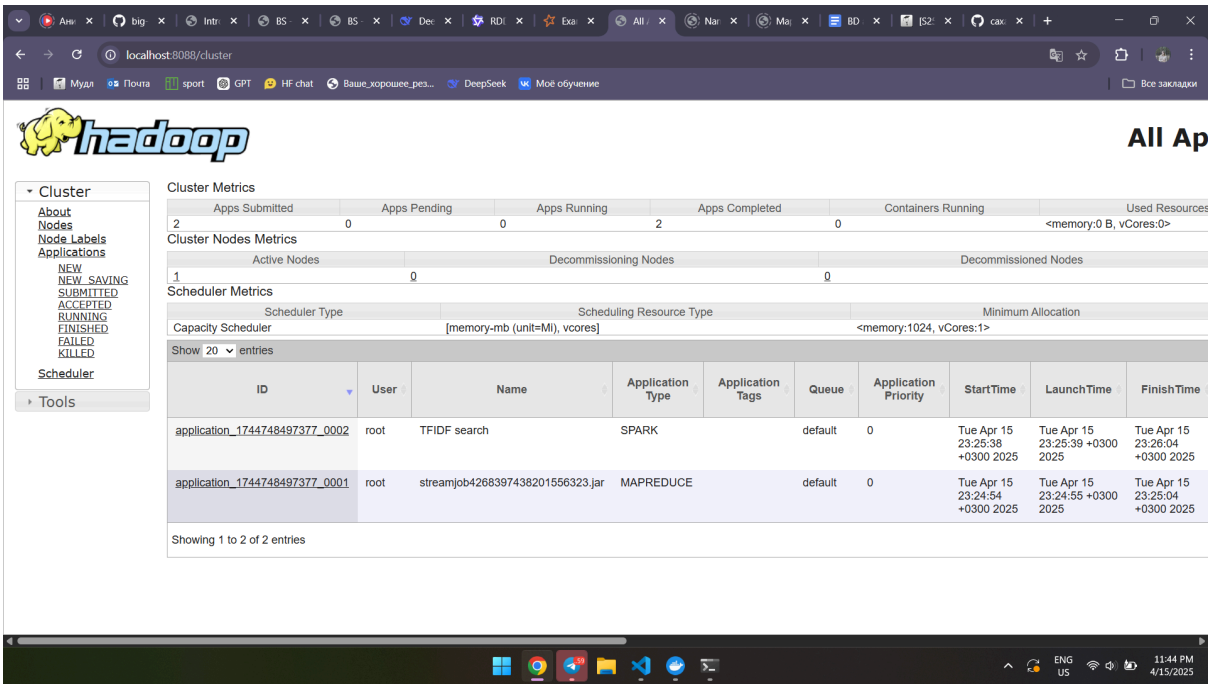
My results:

I can see all files in HDFS, results of Mapreduce, jobs, Cassandra tables. However, my output.txt is empty, and I assume join on RDDs in the query.py works somehow wrong.

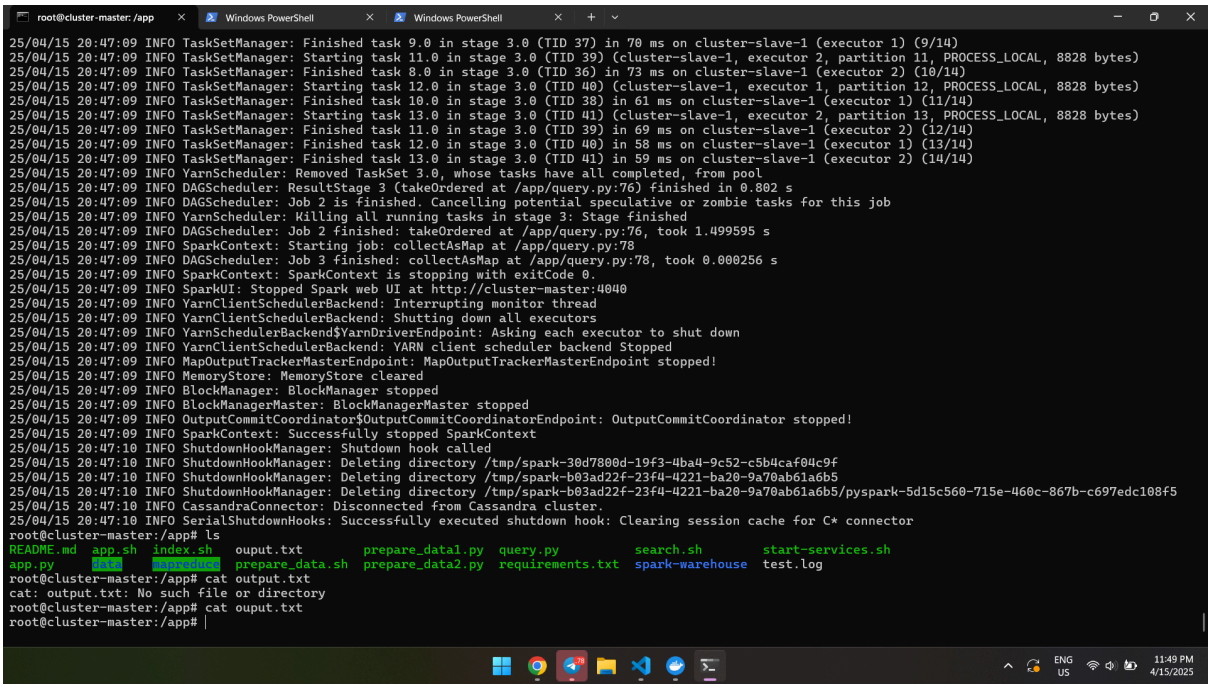
Here I tested the system on test.parquet and global stats are small. But the last output shows the actual stats for a.parquet.



Here you can see stages of the application that are finished:



Here is the screen of the search.sh logs. As I said, output.txt is empty



I also added “tail -f /dev/null” to the end of the app.sh file. So you can run
docker exec -it cluster-master bash
./search.sh “your query”