BigData Assignment 2

Dmitry Dydalin, AAI-02

Methodology

The data collection and preparation phase

It remained unchanged, with only minor changes in code style.

Setting up a Cassandra database

Four tables were created:

```
stats (
            key text PRIMARY KEY,
            value double
        );
```

This table stores only two items

- **N**: the total number of indexed documents
- **dl_avg**: the average document length.

```
df (
            "token" text PRIMARY KEY,
            n_docs int
        );
```

This table stores the document frequency for each token.

- token: the word or term
- n_docs: the number of documents in which the token appears

```
tf (
            document_id text,
            "token" text,
            frequency int,
            PRIMARY KEY (document_id, "token")
        );
```

This table holds term frequency data for each token in each document.

- document_id: the unique identifier of the document
- token: the word or term
- frequency: how many times the token appears in the document

```
docs (
          id text PRIMARY KEY,
          topic text,
          text text,
          len int
      );
```

This table contains the raw documents being indexed.

- id: the unique identifier of the document
- topic: the topic or category of the document
- text: the full text of the document
- len: the length of the document in tokens

MapReduce jobs

1) The first pipeline is used for calculating the token frequency.

   The mapper takes input records in the format **<doc_id | title | text>**, tokenizes the text field, and emits intermediate key-value pairs in the form **<doc_id | token | 1>**.

   The reducer aggregates these values by summing the counts of each token per document and writes the resulting term frequencies to the tf table.

   To sort the emitted values from the mapper using 2 keys (first by **doc_id**, then by **token**) the mapreduce was started using additional parameters as follows:

```
-D mapreduce.partition.keypartitioner.options="-k1,2" \
-D mapreduce.partition.keycomparator.options="-k1,1 -k2,2" \
```

2) The second pipeline calculates document frequency.

   The mapper takes input records in the format **<doc_id | title | text>**, then tokenizes the text field. Afterwards, only the unique tokens are emitted. The intermediate key-value pairs are in the form **<token | 1>**.

   The reducer aggregates these values by summing the counts of each token. Thus, we obtain the number of documents containing each token. Then, the reducer writes the resulting document frequencies to the df table.

3) This pipeline calculates the overall statistics per documents.

   The mapper takes input records in the format **<doc_id | title | text>**, then tokenizes the text field and calculates the length of the text. The mapper puts the record <doc_id | topic | text | len> to a docs table. For further calculation of total **N** and **dl_avg** parameters, the mapper emits a pair **<1 | len>**.

   Thus, the reducer obtains the len for each text. It calculates the number of documents and the average text length. Finally, the reducer puts **N** and **dl_avg** values to the stats table.

Notes:

- Each text is processed the same way: it firstly converted to lowercase, then the regular expression '**\b\w+\b**' is applied to split text only on tokens that consists of **[a-zA-Z0-9_]** symbols.

- The only way I managed to use the external library (cassandra driver) was by using zipimoprt inside the jar with the provided zipped library.

Retrieval

Key steps:

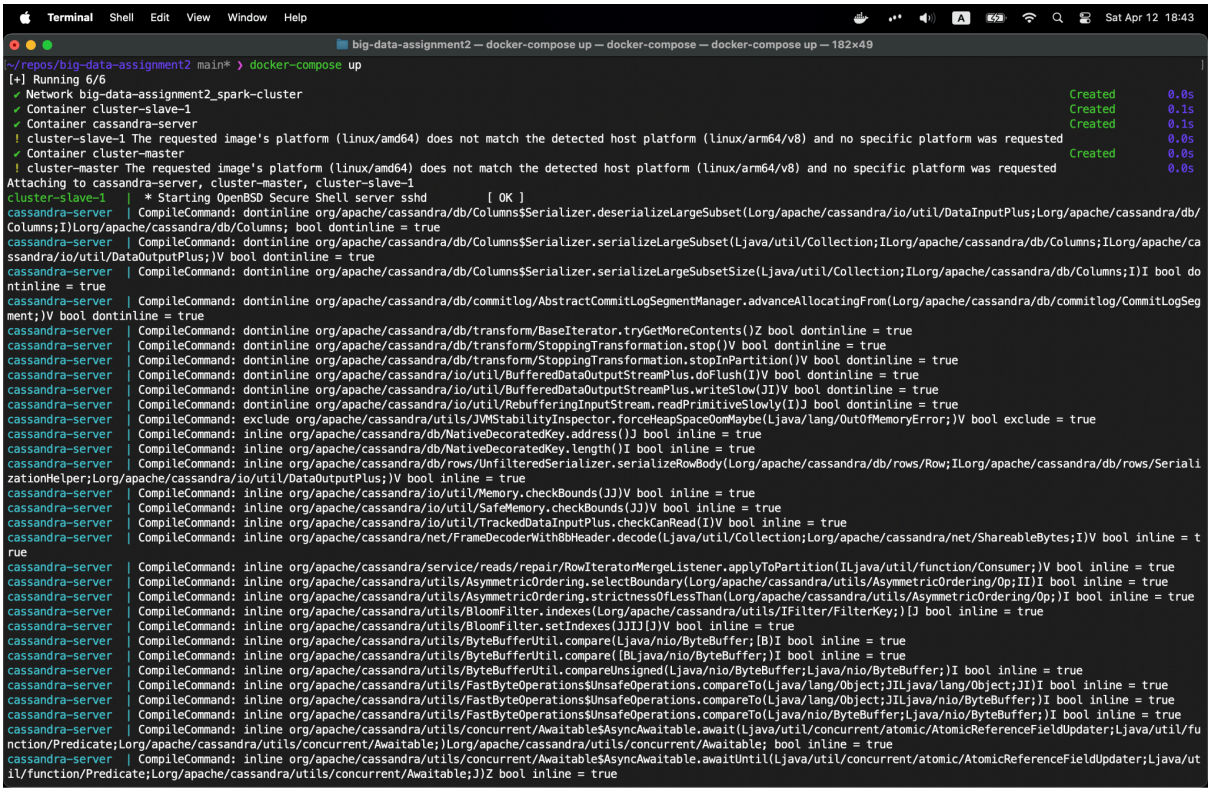
- The search query is tokenized using the same function as in the indexing stage.
- The Apache Spark's Cassandra connector is used to read each table.
- Before scoring, the script broadcasts the following variables to all workers for efficient lookup:
    - A dictionary mapping doc_id to a tuple containing document length, topic, and text
    - Document frequency
    - Total number of documents
    - The average document length
- Then, the partial bm25 score is calculated for each token. The partial bm25 score means the contribution of the token to the document's score.
- Next, the rdd reduceByKey operation is used to sum the scores per document.
- Finally, the scores are sorted and the top 10 documents are printed.

Demonstration

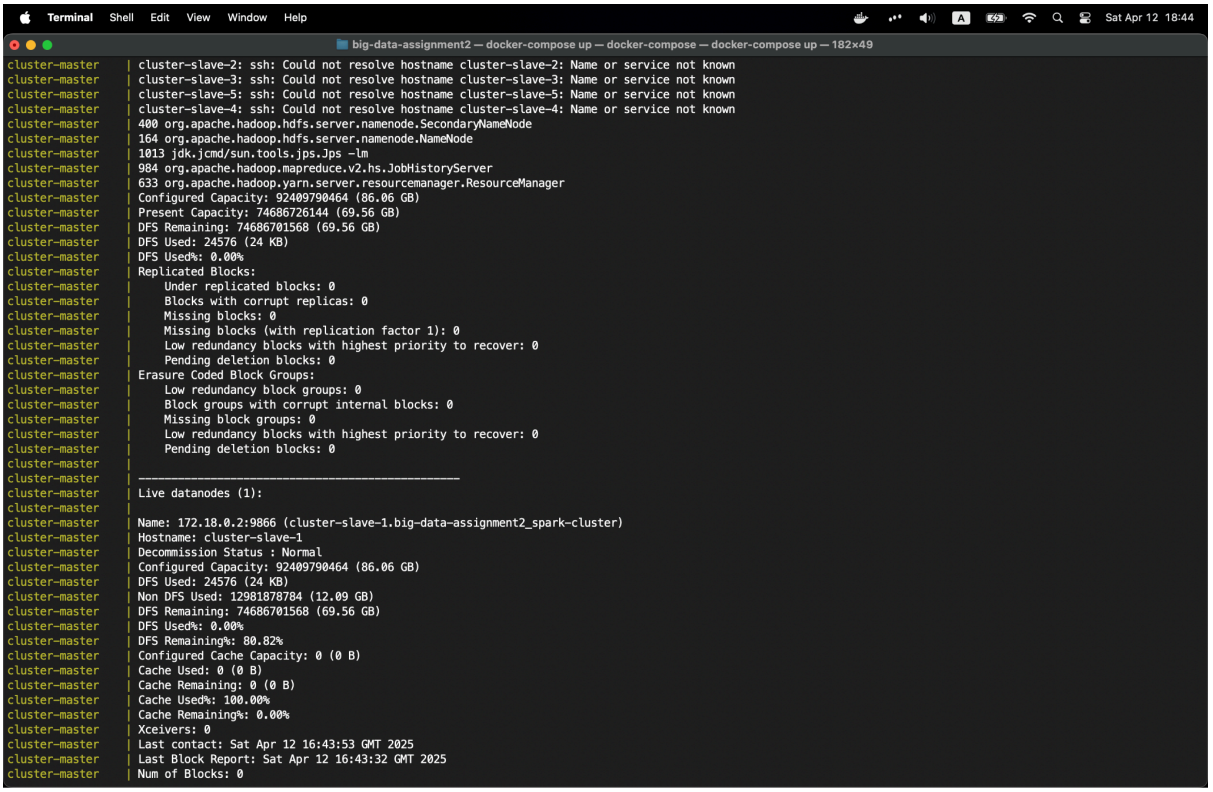
To run the code:

1) git pull https://github.com/Nit31/big-data-assignment2.git
2) Download the file a.parquet and put to an ./app directory
3) Run docker-compose up in the root repository directory

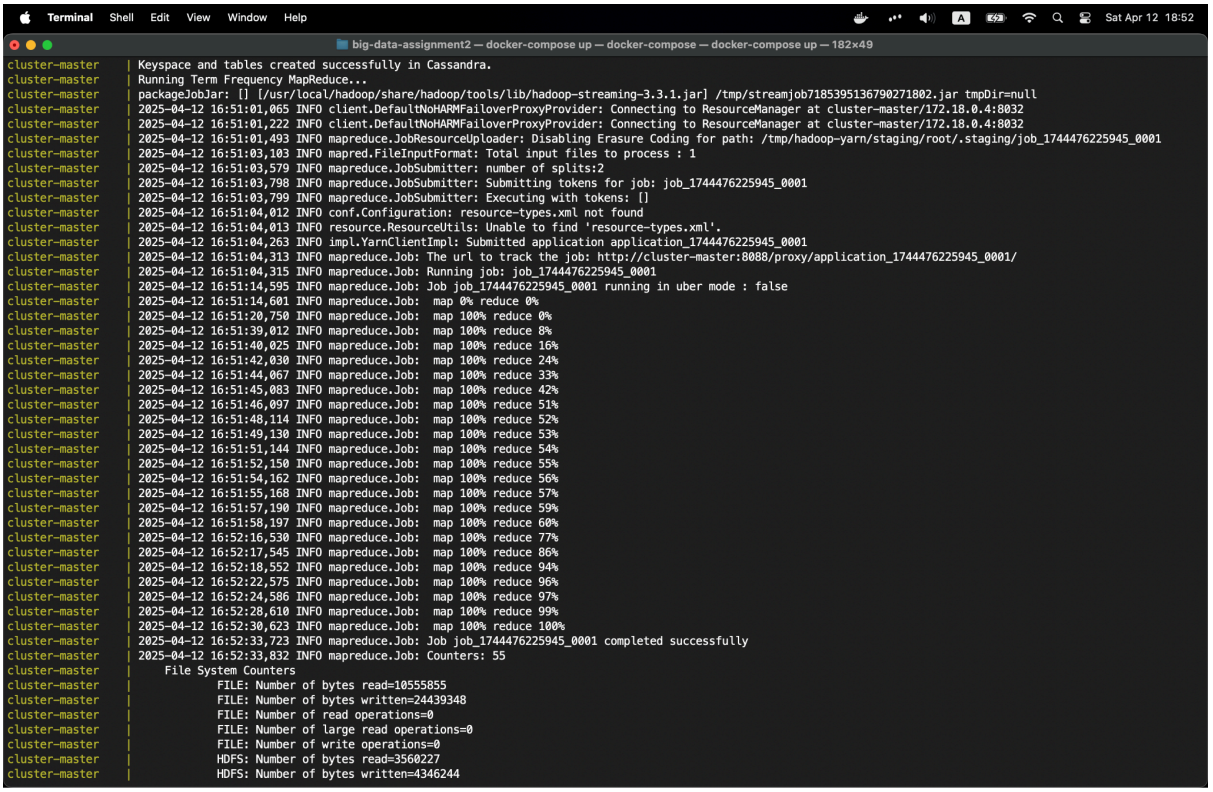
Starting the docker container:

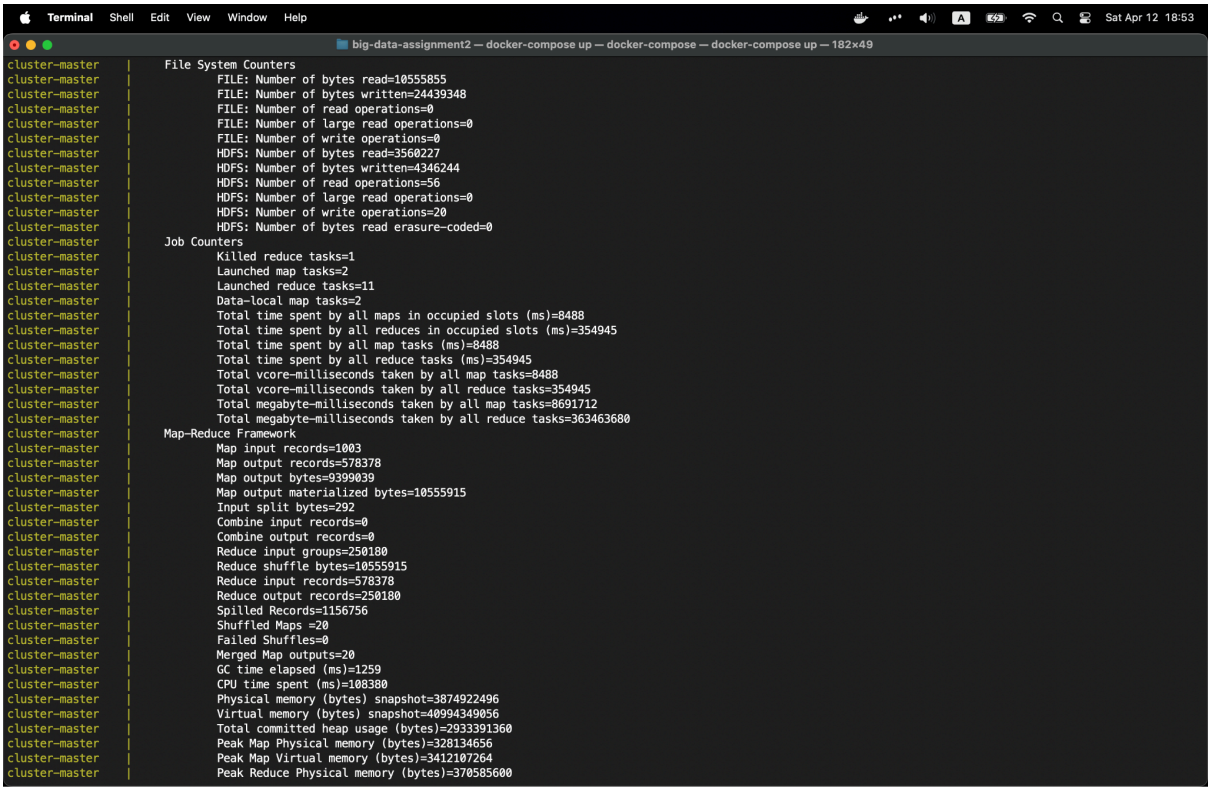


Here we can see that the slave node is up:

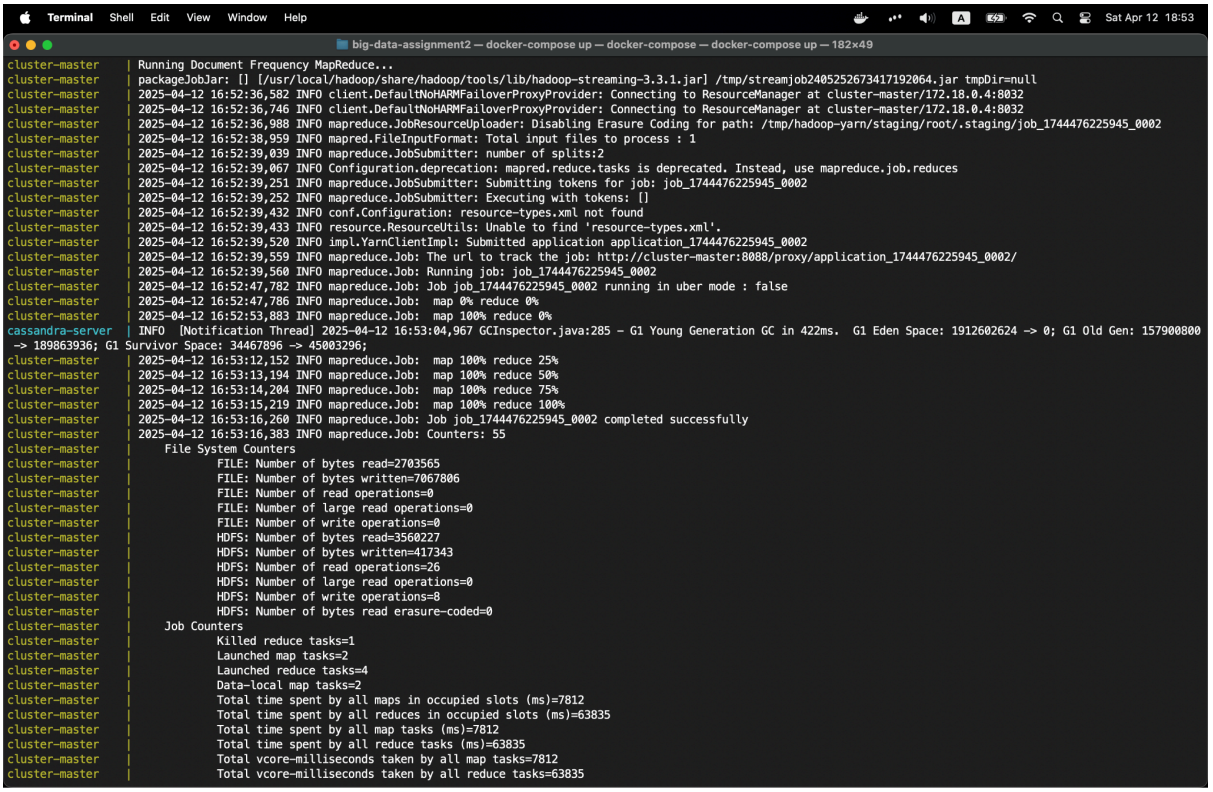


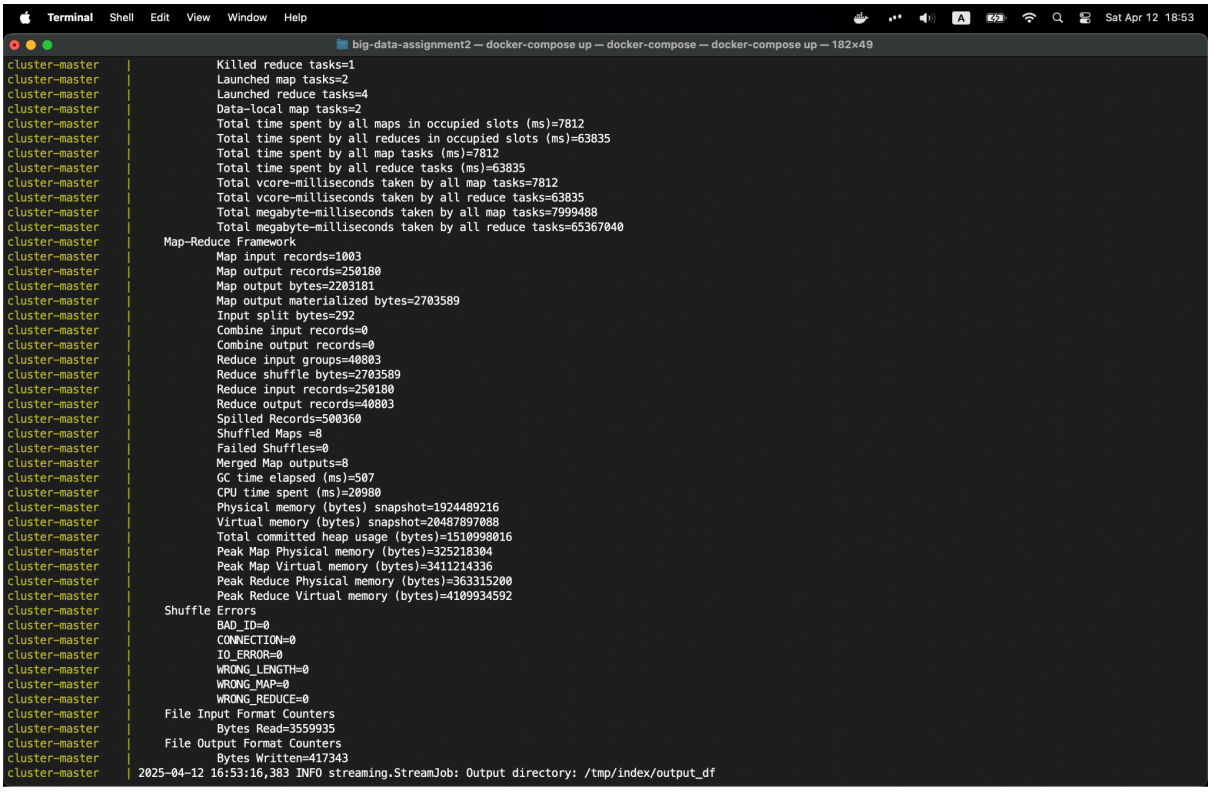
The first mapreduce operation (2 pics):



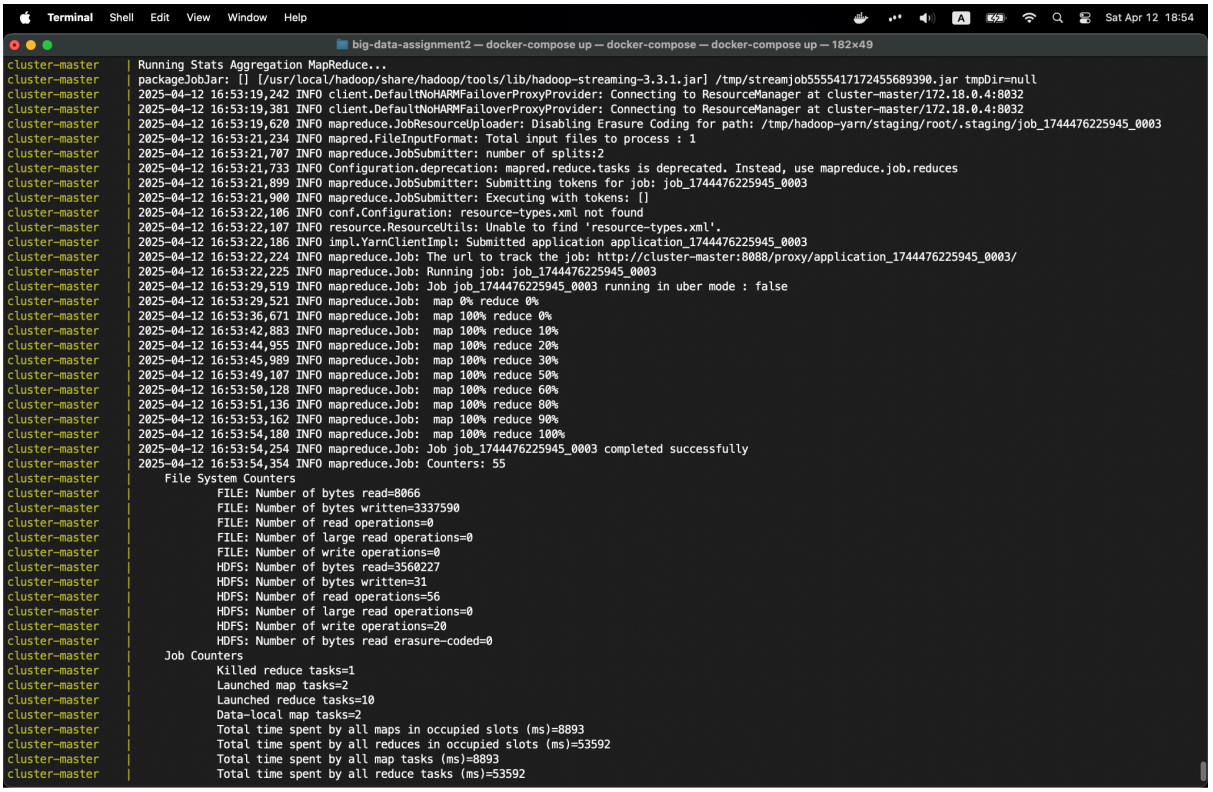


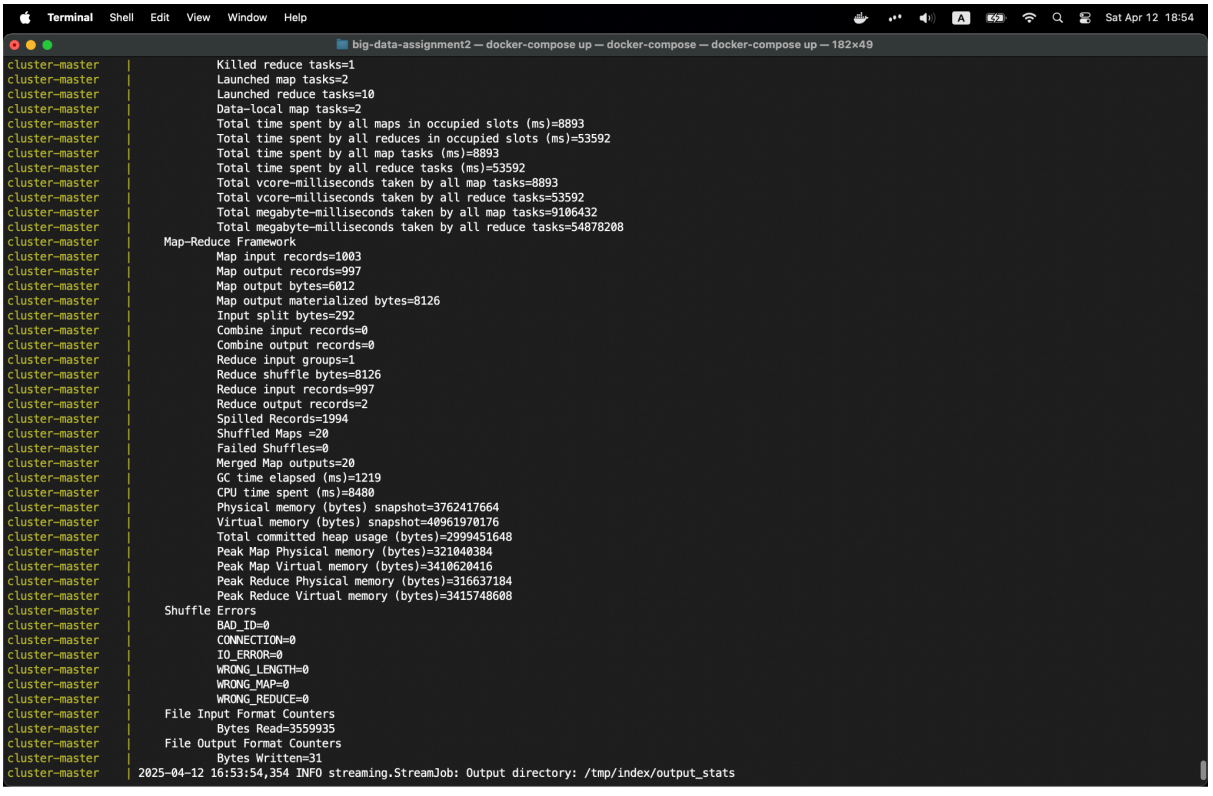
The second mapreduce operation (2 pics):



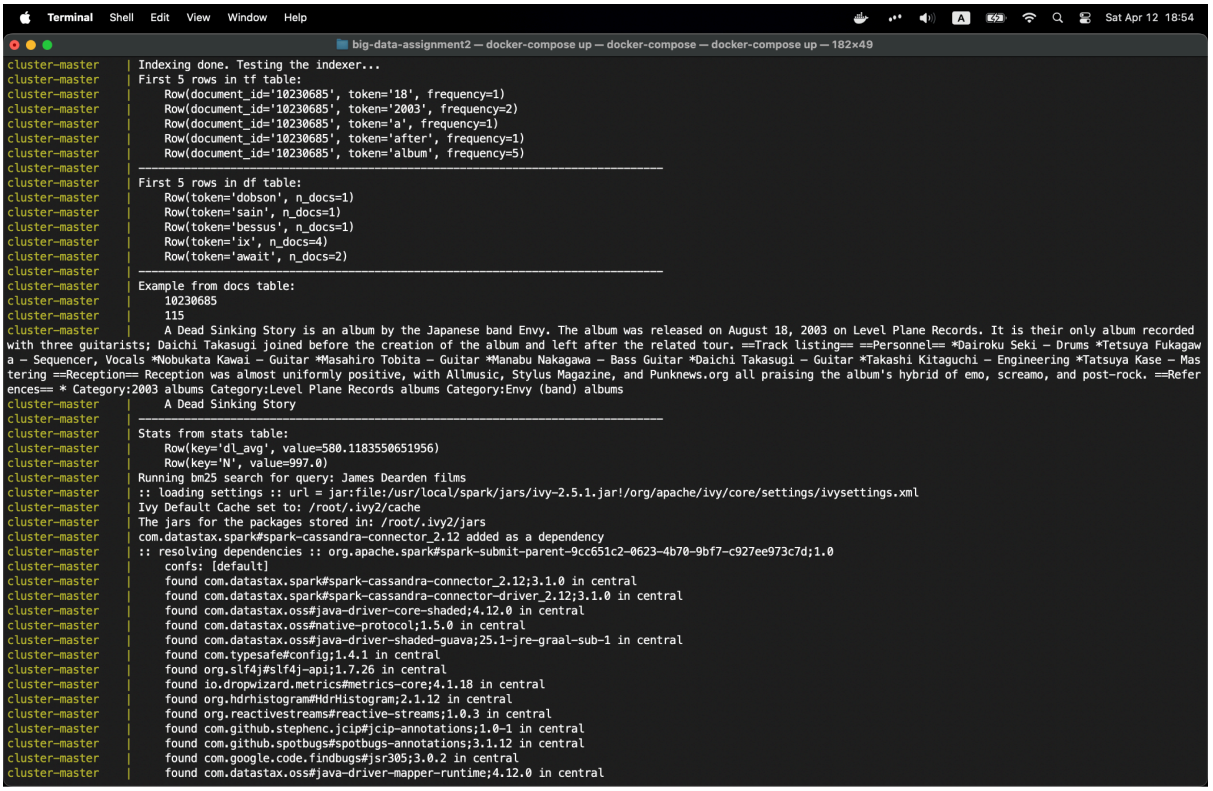


The third mapreduce operation (2 pics):



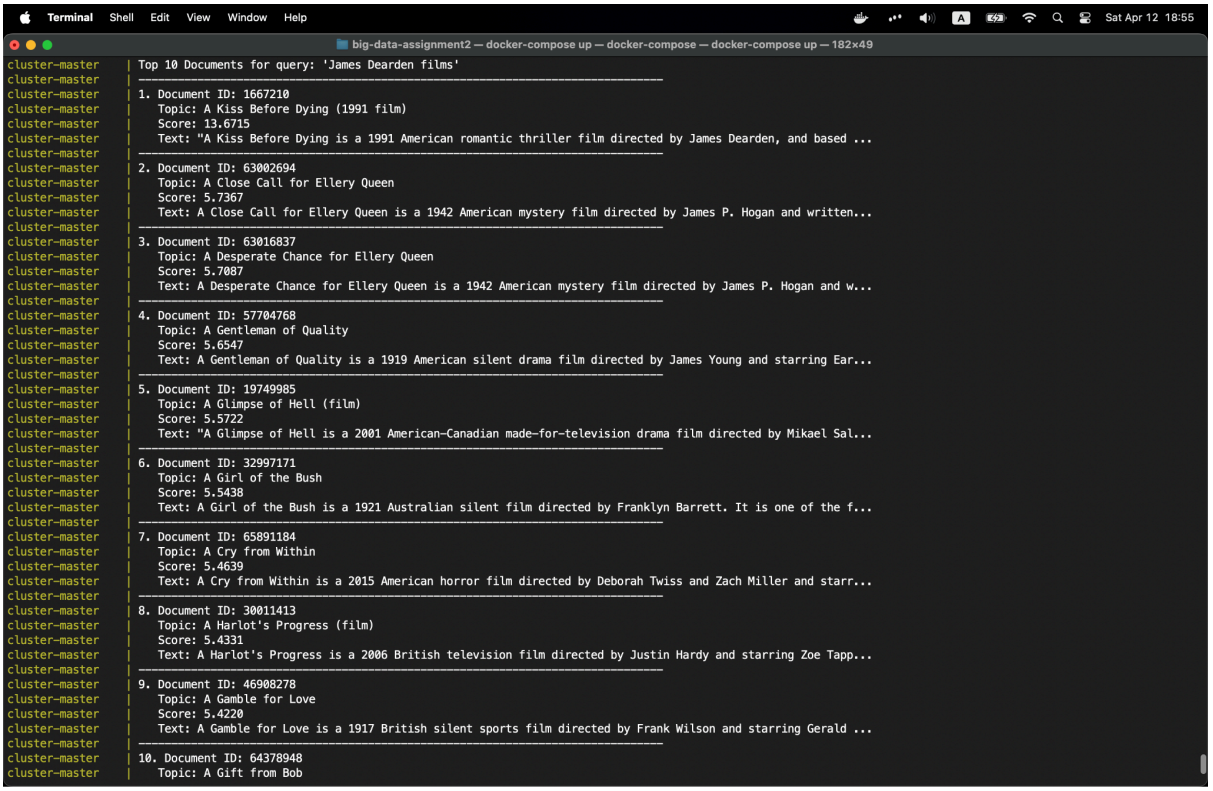


I have created a simple python script (app/test_cassandra.py) to print several examples from the created tables:

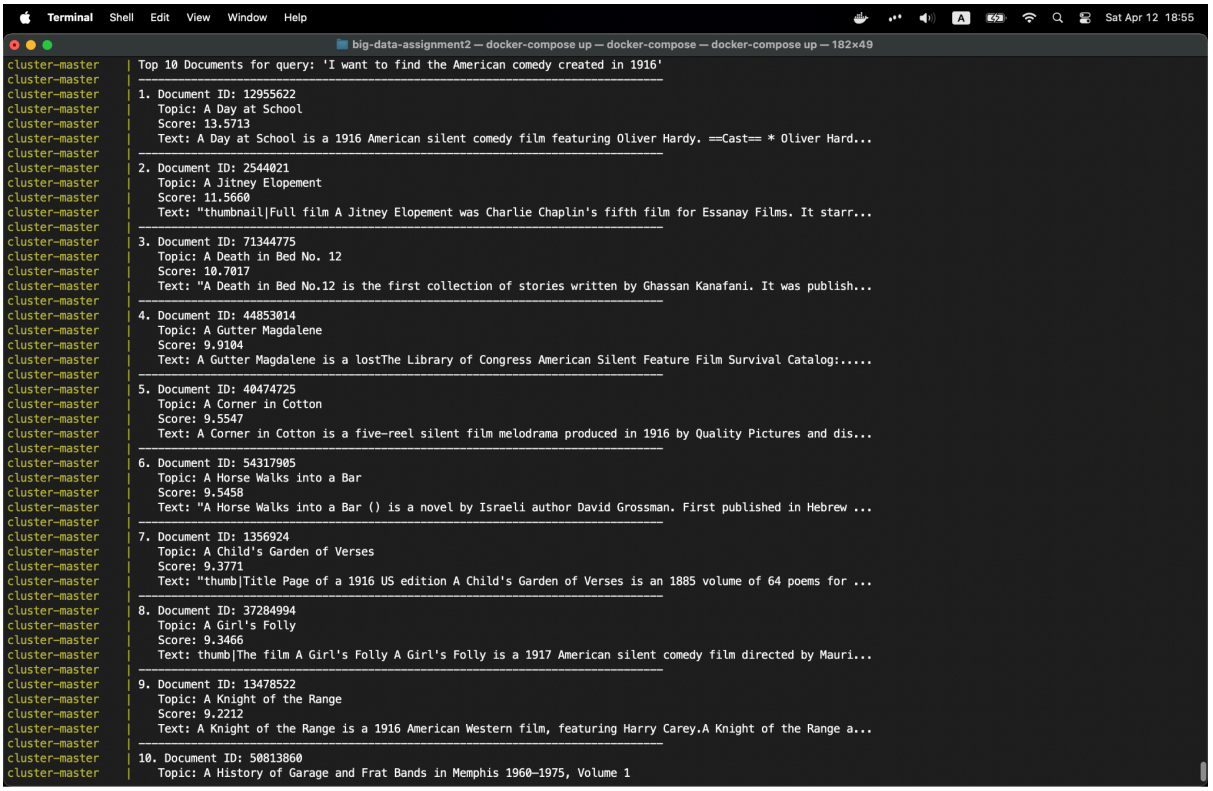


Finally, we got to the retrieval! Below you can see results for 3 queries

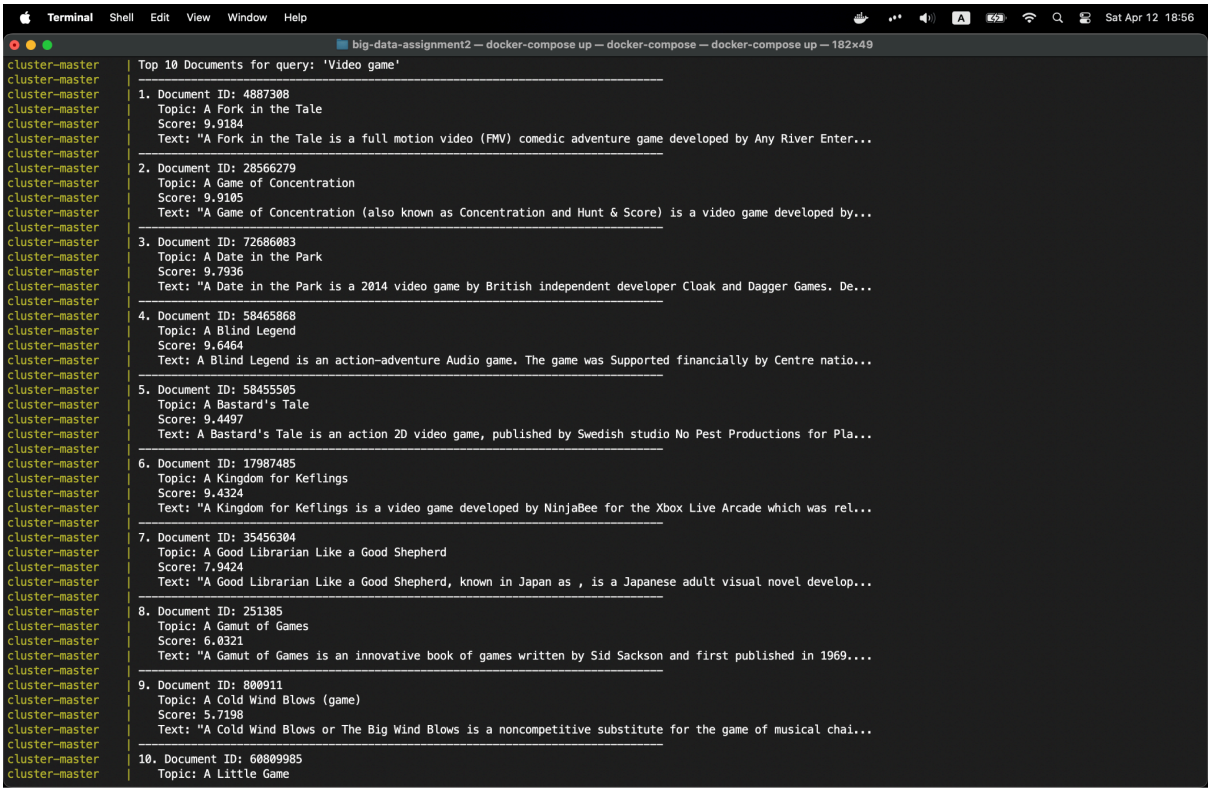
1) **The first result is a perfect hit. The documents 2-4 are films directed by some "James" and the documents 5-10 were chosen only for the "film" hit**



2) **Again, the first result is the most relevant for the query. Other documents were chosen for film/date/genre hits.**



3) **Nothing to add. Video games are found perfectly.**



Conclusion

The BM25 algorithm performs well, delivering the expected results. However, in my experience, MapReduce feels limited in terms of functionality - it is constrained on how data can be processed.

Additionally, the execution speed is currently slower than running the equivalent logic in pure Python without Spark. This is primarily due to the relatively small size of the dataset. For smaller workloads, the overhead of initializing and coordinating distributed tasks outweighs the benefits. I understand that with the number of documents growing, the distributed approach becomes more apparent, and Spark must outperform single-node Python execution significantly.