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

Three tables were created:

```
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 calculates the number of tokens. After that it puts the record **<id | title | text | length>** into the docs table. Finally, the mapper 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.

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.
- The average length of documents is calculated.
- 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

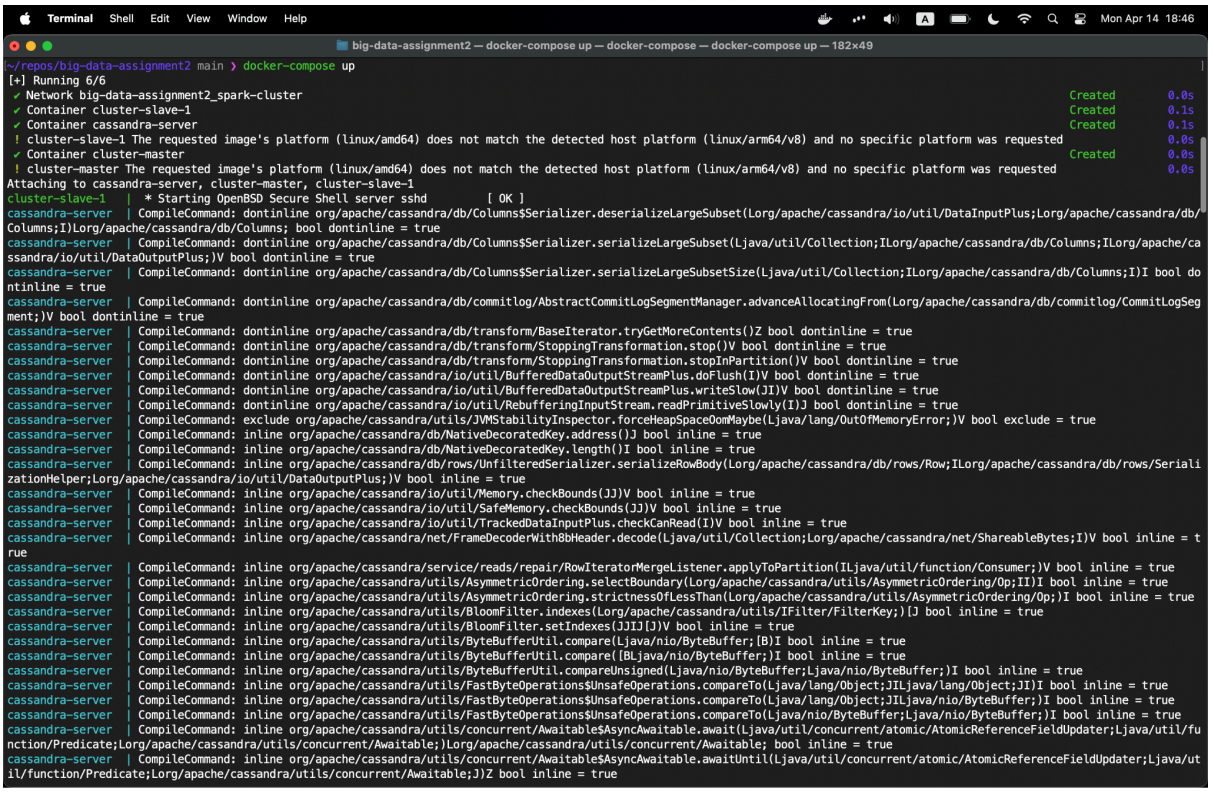
    - The average document length
- Then, the per-term bm25 score is calculated for each token. This per-term bm25 score means the contribution of the token into the document's score.
- Next, the rdd reduceByKey operation is used to sum the scores per document.
- Finally, the scores are sorted and the top 10 documents are printed.

Demonstration

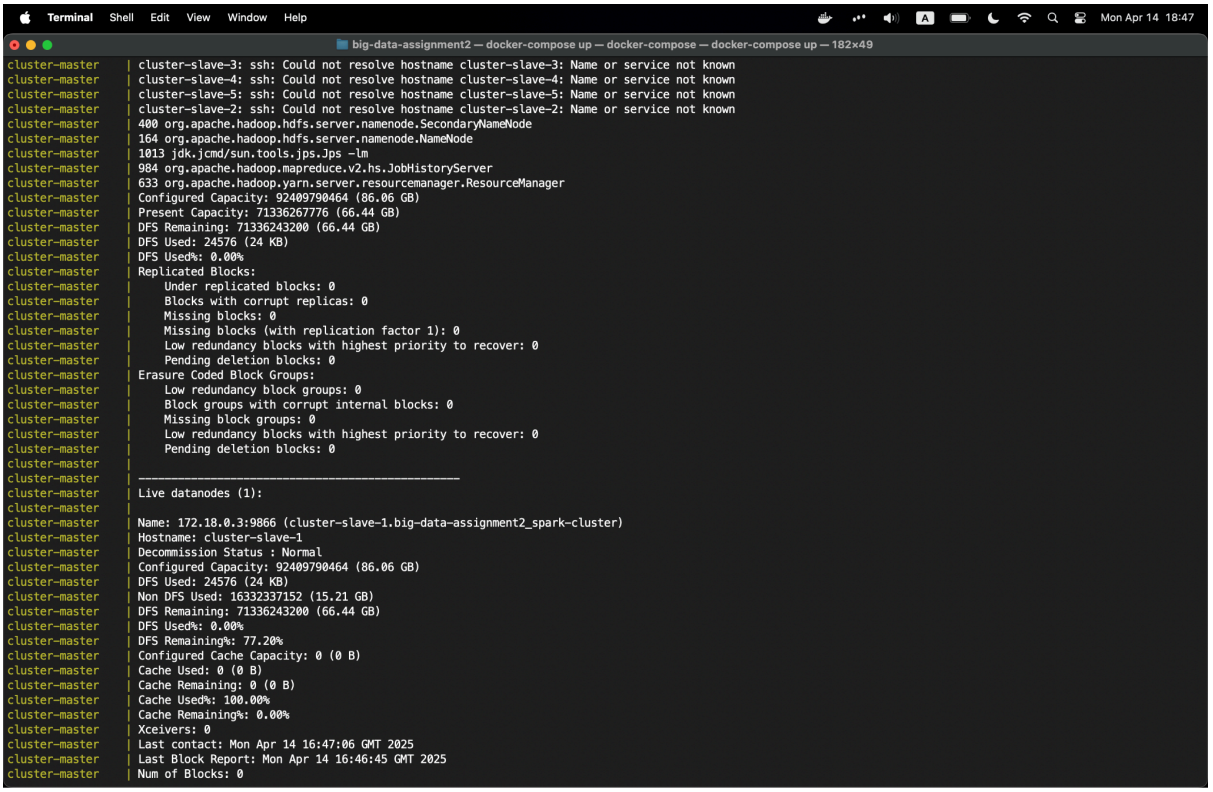
To run the code:

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

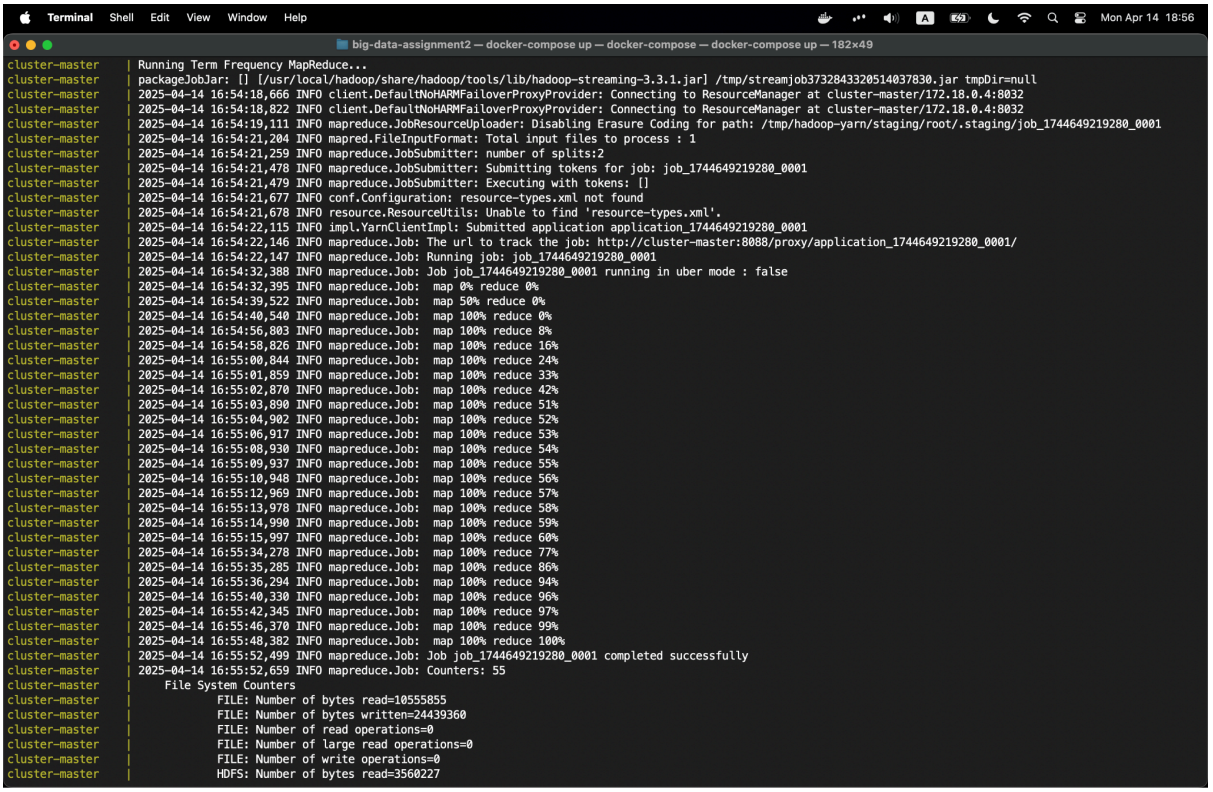
Starting the docker container:

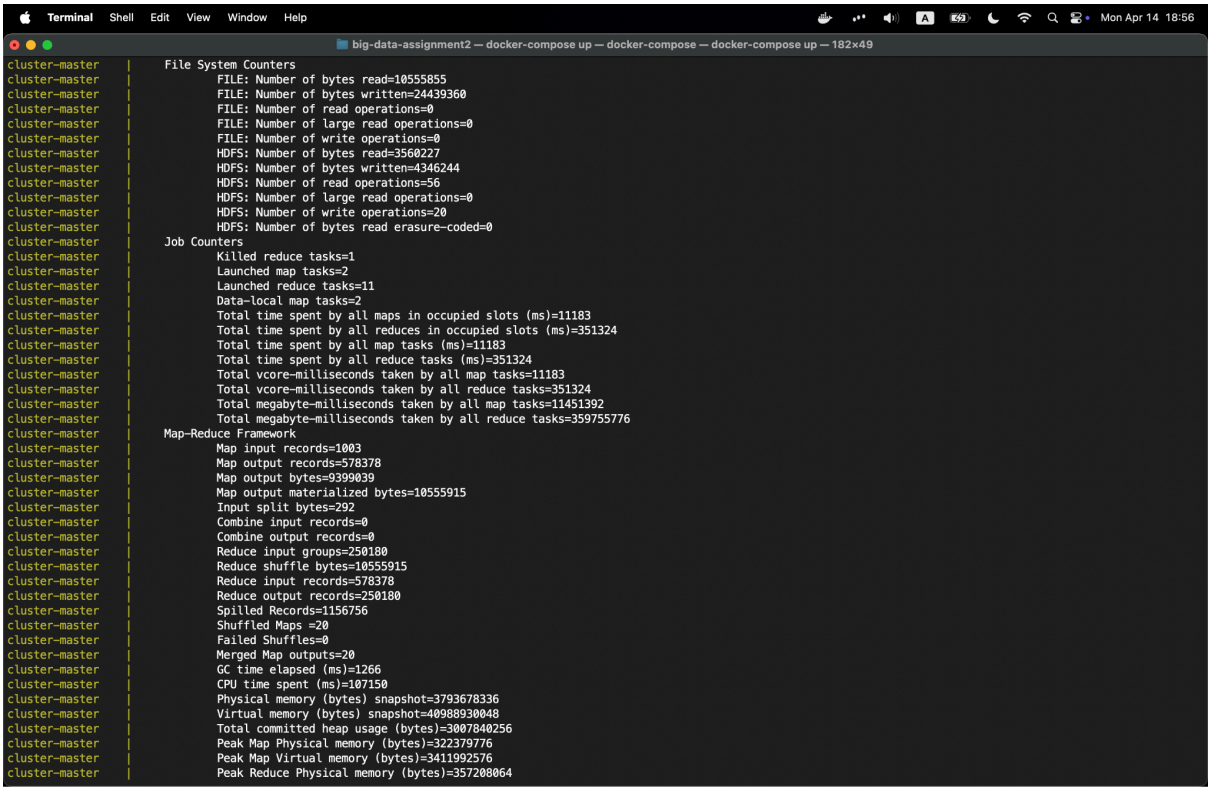


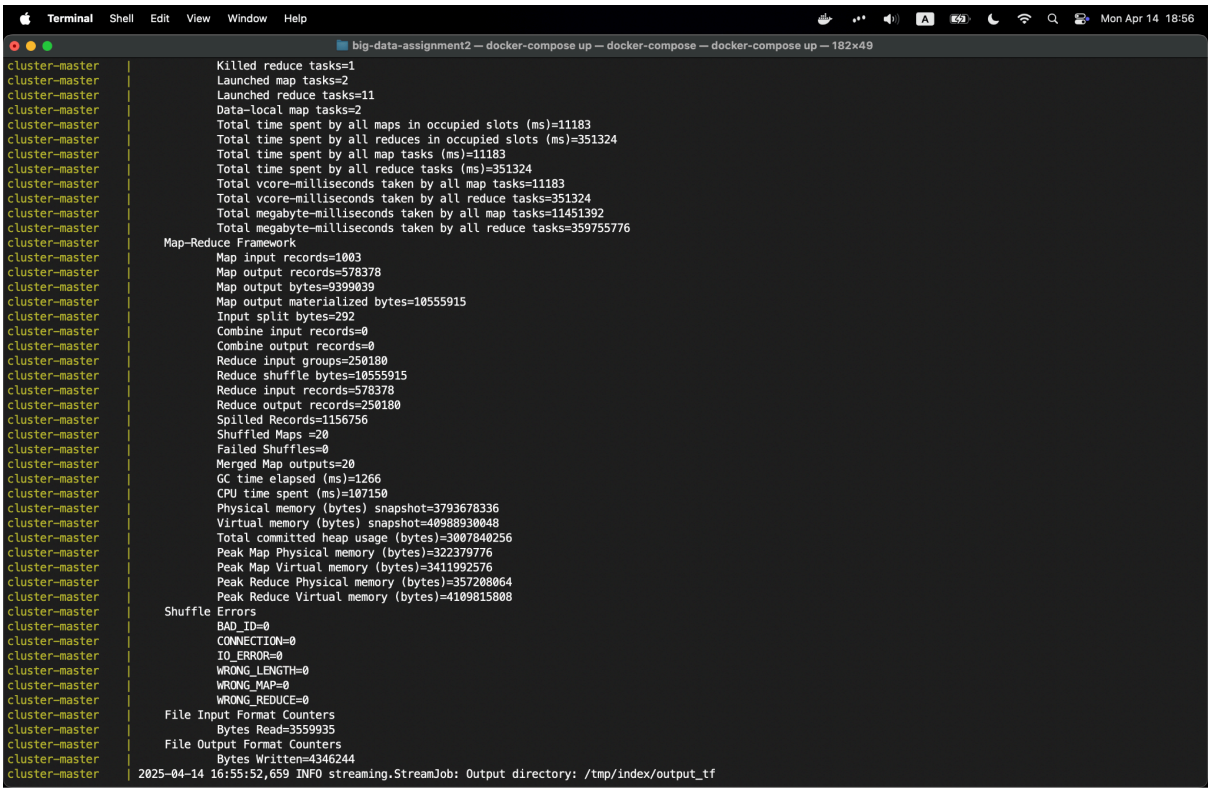
Here we can see that the slave node is up:



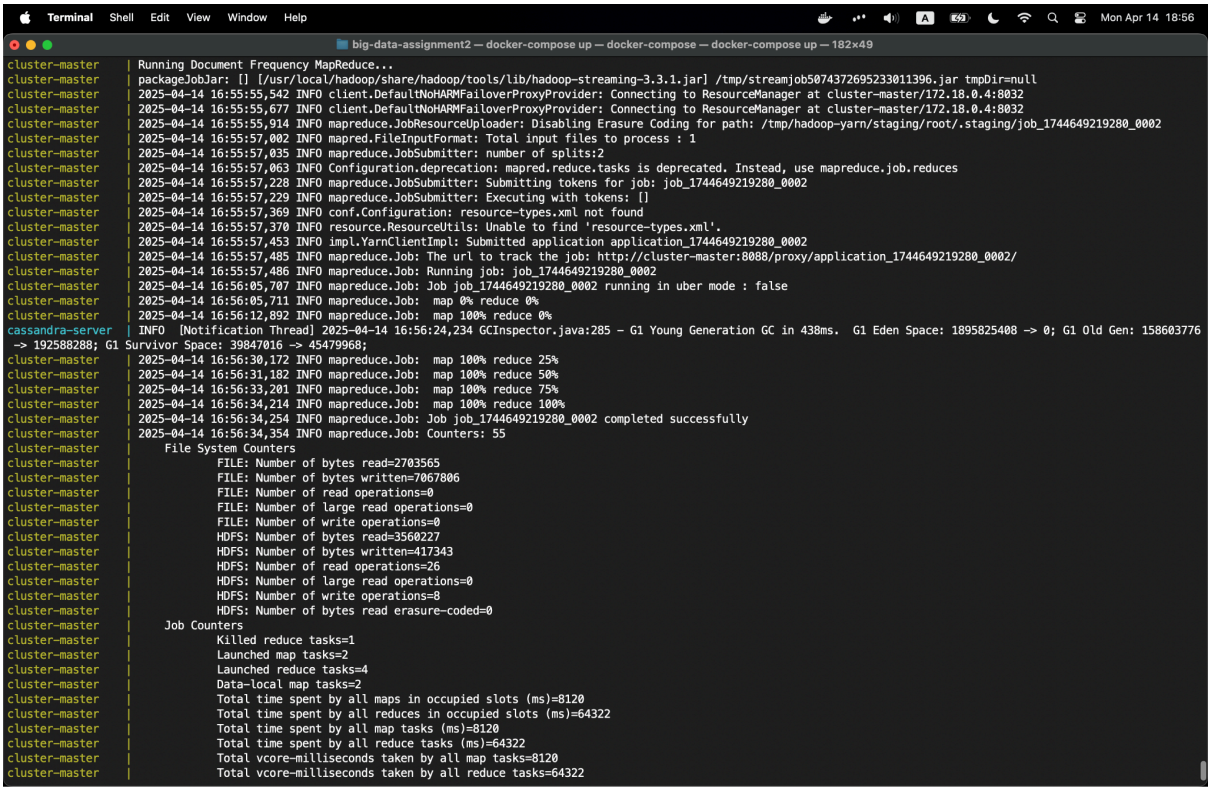
The first mapreduce operation (3 pics):

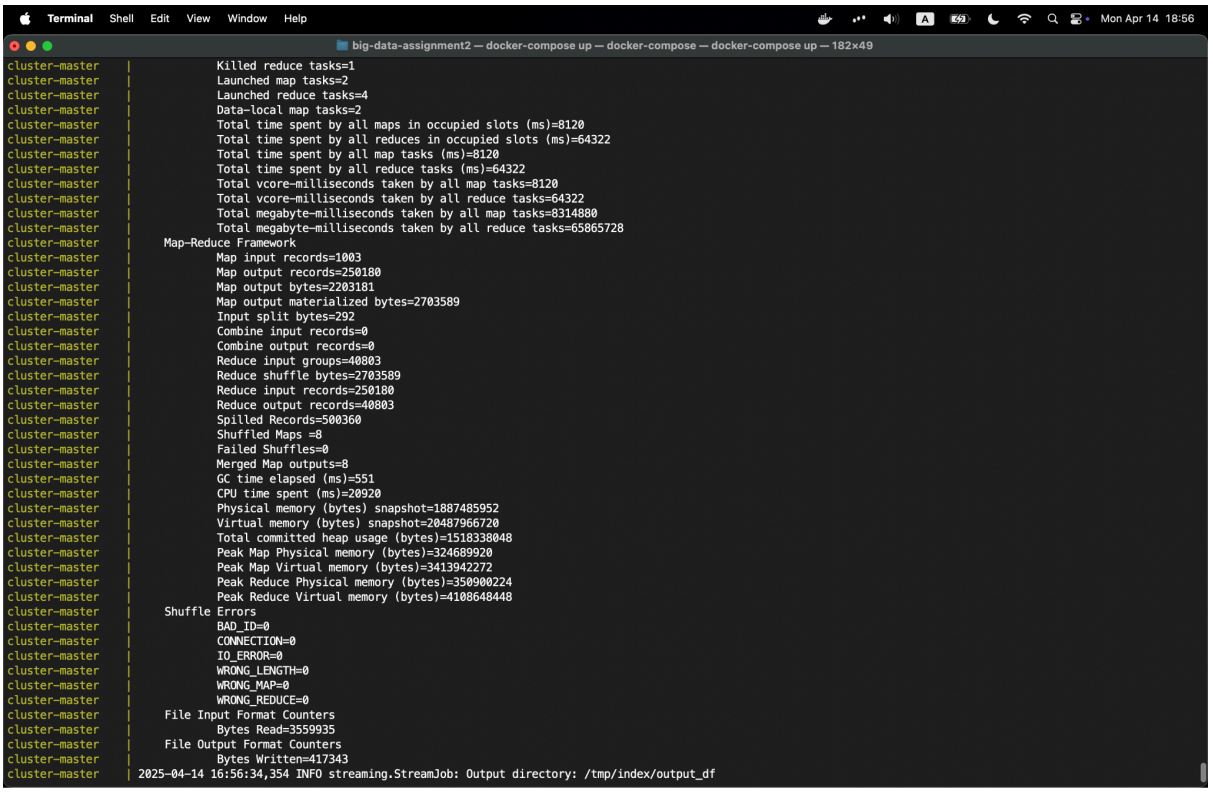



Terminal Shell Edit View Window Help
Mon Apr 14 18:56
big-data-assignment2 — docker-compose up — docker-compose — docker-compose up — 182×49
cluster-master | File System Counters
cluster-master | FILE: Number of bytes read=10555855
cluster-master | FILE: Number of bytes written=24439360
cluster-master | FILE: Number of read operations=0
cluster-master | FILE: Number of large read operations=0
cluster-master | FILE: Number of write operations=0
cluster-master | HDFS: Number of bytes read=3560227
cluster-master | HDFS: Number of bytes written=4346244
cluster-master | HDFS: Number of read operations=56
cluster-master | HDFS: Number of large read operations=0
cluster-master | HDFS: Number of write operations=20
cluster-master | HDFS: Number of bytes read erasure-coded=0
cluster-master | Job Counters
cluster-master | Killed reduce tasks=1
cluster-master | Launched map tasks=2
cluster-master | Launched reduce tasks=11
cluster-master | Data-local map tasks=2
cluster-master | Total time spent by all maps in occupied slots (ms)=11183
cluster-master | Total time spent by all reduces in occupied slots (ms)=351324
cluster-master | Total time spent by all map tasks (ms)=11183
cluster-master | Total time spent by all reduce tasks (ms)=351324
cluster-master | Total vcore-milliseconds taken by all map tasks=11183
cluster-master | Total vcore-milliseconds taken by all reduce tasks=351324
cluster-master | Total megabyte-milliseconds taken by all map tasks=11451392
cluster-master | Total megabyte-milliseconds taken by all reduce tasks=359755776
cluster-master | Map-Reduce Framework
cluster-master | Map input records=1003
cluster-master | Map output records=578378
cluster-master | Map output bytes=9399039
cluster-master | Map output materialized bytes=10555915
cluster-master | Input split bytes=292
cluster-master | Combine input records=0
cluster-master | Combine output records=0
cluster-master | Reduce input groups=250180
cluster-master | Reduce shuffle bytes=10555915
cluster-master | Reduce input records=578378
cluster-master | Reduce output records=250180
cluster-master | Spilled Records=1156756
cluster-master | Shuffled Maps =20
cluster-master | Failed Shuffles=0
cluster-master | Merged Map outputs=20
cluster-master | GC time elapsed (ms)=1266
cluster-master | CPU time spent (ms)=107150
cluster-master | Physical memory (bytes) snapshot=3793678336
cluster-master | Virtual memory (bytes) snapshot=40988930048
cluster-master | Total committed heap usage (bytes)=3007840256
cluster-master | Peak Map Physical memory (bytes)=322379776
cluster-master | Peak Map Virtual memory (bytes)=3411992576
cluster-master | Peak Reduce Physical memory (bytes)=357208064

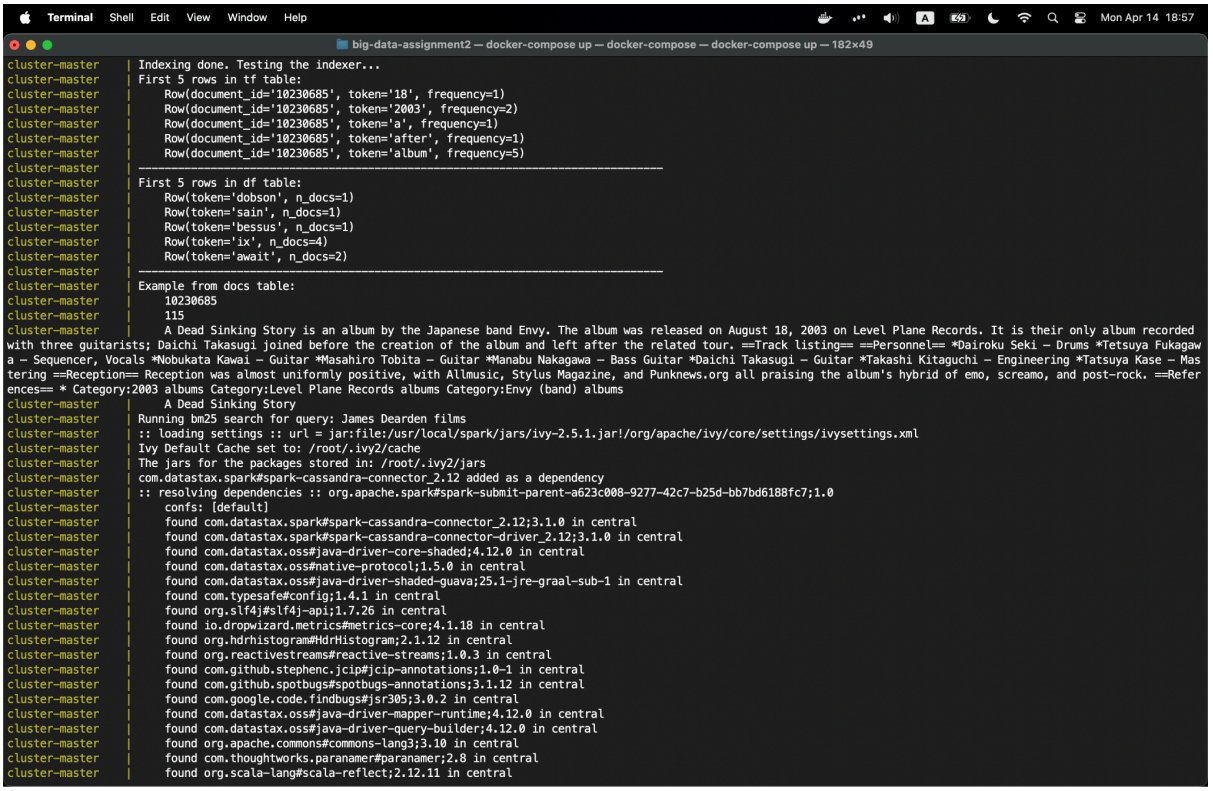

Terminal Shell Edit View Window Help
Mon Apr 14 18:56
big-data-assignment2 — docker-compose up — docker-compose — docker-compose up — 182×49
cluster-master | Killed reduce tasks=1
cluster-master | Launched map tasks=2
cluster-master | Launched reduce tasks=11
cluster-master | Data-local map tasks=2
cluster-master | Total time spent by all maps in occupied slots (ms)=11183
cluster-master | Total time spent by all reduces in occupied slots (ms)=351324
cluster-master | Total time spent by all map tasks (ms)=11183
cluster-master | Total time spent by all reduce tasks (ms)=351324
cluster-master | Total vcore-milliseconds taken by all map tasks=11183
cluster-master | Total vcore-milliseconds taken by all reduce tasks=351324
cluster-master | Total megabyte-milliseconds taken by all map tasks=11451392
cluster-master | Total megabyte-milliseconds taken by all reduce tasks=359755776
cluster-master | Map-Reduce Framework
cluster-master | Map input records=1003
cluster-master | Map output records=578378
cluster-master | Map output bytes=9399039
cluster-master | Map output materialized bytes=10555915
cluster-master | Input split bytes=292
cluster-master | Combine input records=0
cluster-master | Combine output records=0
cluster-master | Reduce input groups=250180
cluster-master | Reduce shuffle bytes=10555915
cluster-master | Reduce input records=578378
cluster-master | Reduce output records=250180
cluster-master | Spilled Records=1156756
cluster-master | Shuffled Maps =20
cluster-master | Failed Shuffles=0
cluster-master | Merged Map outputs=20
cluster-master | GC time elapsed (ms)=1266
cluster-master | CPU time spent (ms)=107150
cluster-master | Physical memory (bytes) snapshot=3793678336
cluster-master | Virtual memory (bytes) snapshot=40988930048
cluster-master | Total committed heap usage (bytes)=3007840256
cluster-master | Peak Map Physical memory (bytes)=322379776
cluster-master | Peak Map Virtual memory (bytes)=3411992576
cluster-master | Peak Reduce Physical memory (bytes)=357208064
cluster-master | Peak Reduce Virtual memory (bytes)=4109815808
cluster-master | Shuffle Errors
cluster-master | BAD_ID=0
cluster-master | CONNECTION=0
cluster-master | IO_ERROR=0
cluster-master | WRONG_LENGTH=0
cluster-master | WRONG_MAP=0
cluster-master | WRONG_REDUCE=0
cluster-master | File Input Format Counters
cluster-master | Bytes Read=3559935
cluster-master | File Output Format Counters
cluster-master | Bytes Written=4346244
cluster-master | 2025-04-14 16:55:52,659 INFO streaming.StreamJob: Output directory: /tmp/index/output_tf

The second mapreduce operation (2 pics):



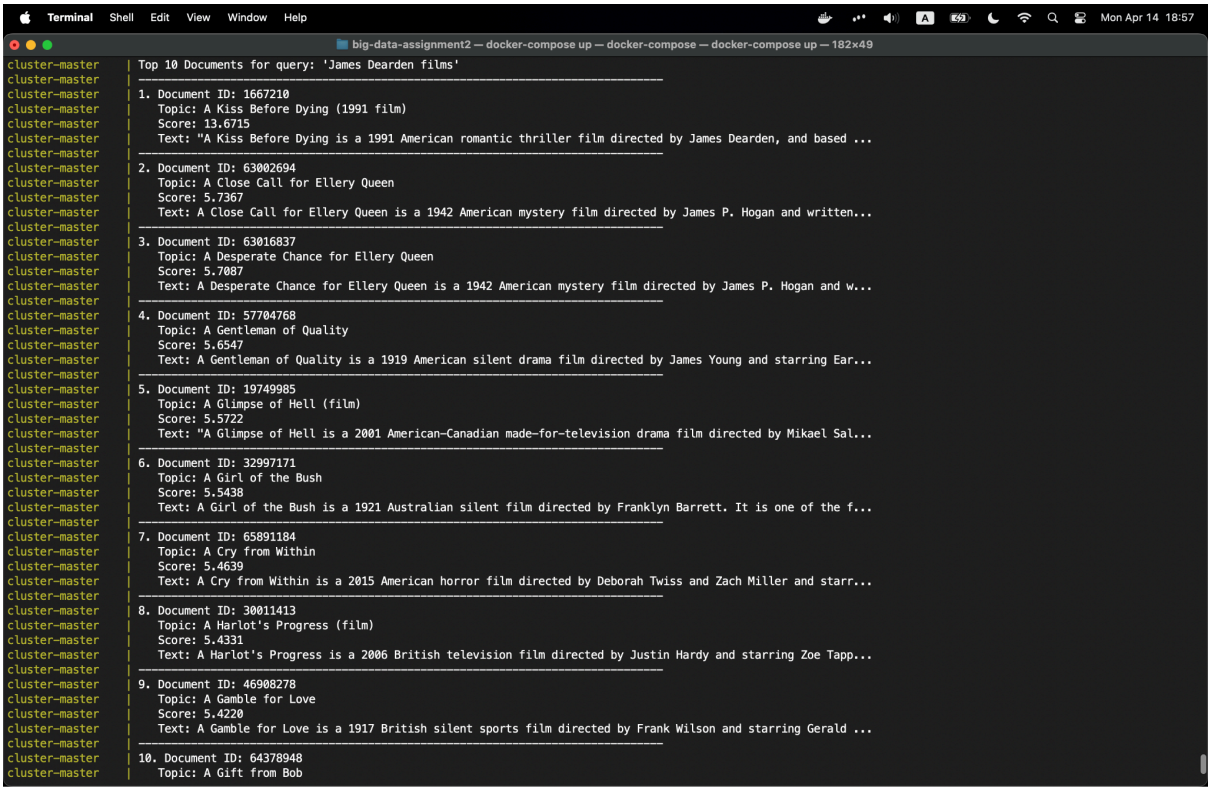


I have created a simple python script (app/test_cassandra.py) to print several examples from the created tables:

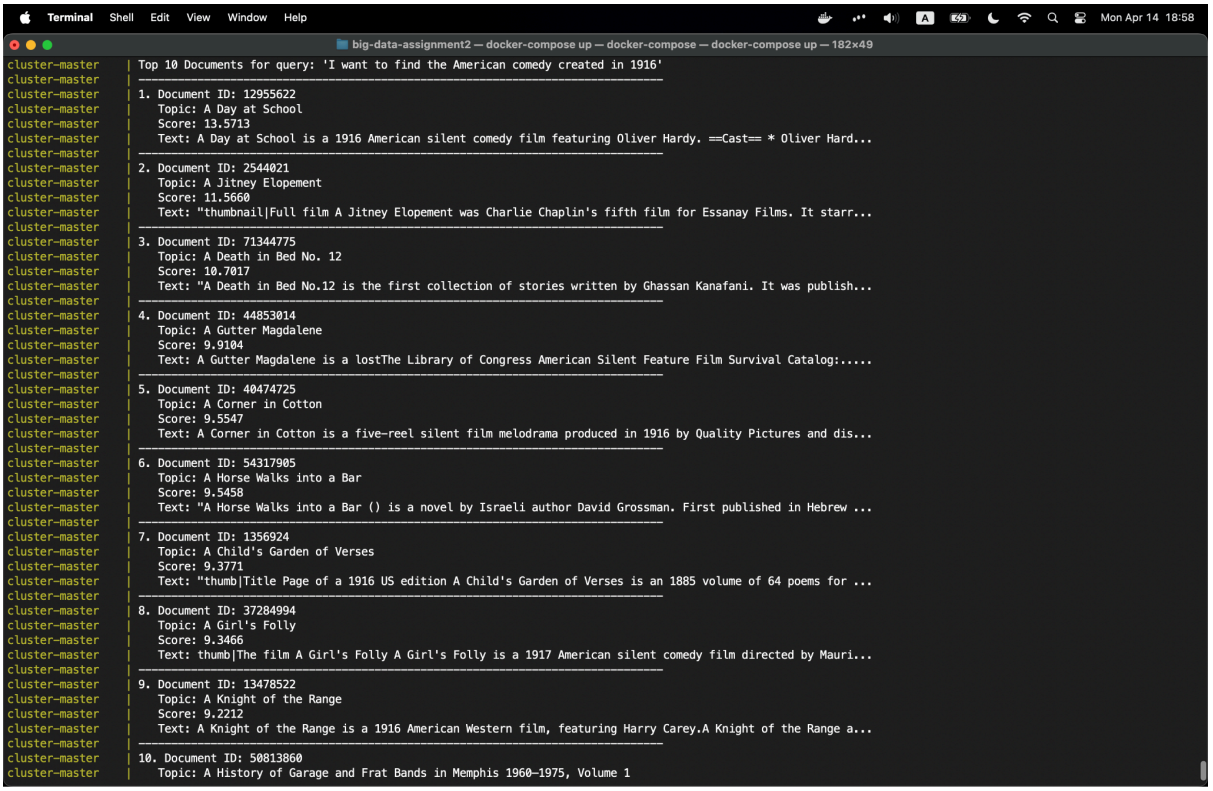


Finally, we got to the retrieval! Below you can see results for 3 queries

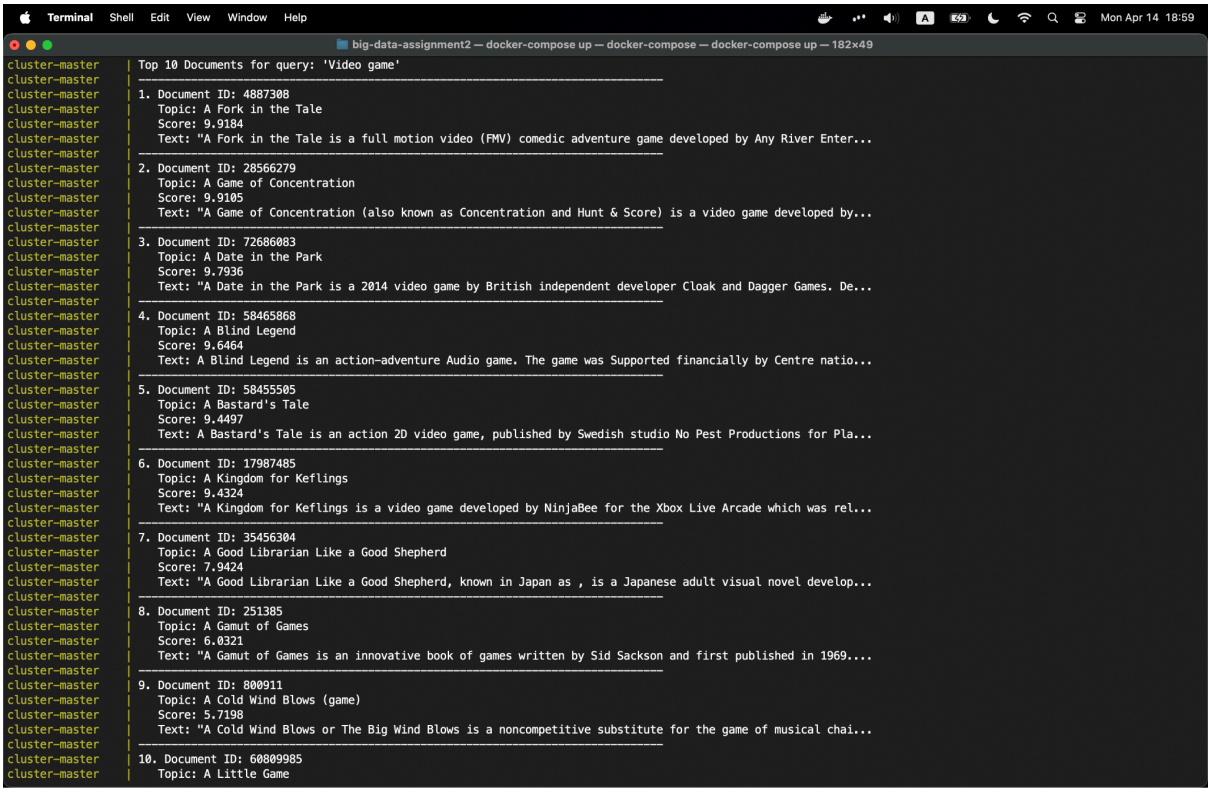
1) **The first result is a perfect hit. The documents 2-4 are films directed by some "James" and the documents 5-10 were chosen only for the "film" hit**



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.