DISTRIBUTED SEARCH ENGINE USING HADOOP MAPREDUCE

Big Data

Nursultan Abdullaev

1 GitHub Repository Link

GitHub Repo Link.

2 Methodology

To implement the distributed search engine, I opted for a hybrid architecture integrating Hadoop MapReduce, Spark, and Cassandra, allowing me to leverage each tool's strengths for specific tasks in the indexing and retrieval pipelines. HDFS served as the underlying distributed file system, while Hadoop Streaming enabled a customizable and transparent indexing phase using Python-based mappers and reducers. The pipeline begins with a data preparation step in Spark. I read a raw Parquet dataset from HDFS, sampled a fixed number of documents to maintain tractability during development, and filtered out null entries. Each document was saved as a local .txt file for title extraction and also serialized into HDFS as tab-separated values containing the ID, title, and body — an efficient and flexible format for line-based processing.

I then applied two separate MapReduce jobs. The first extracted term frequencies per document to construct an inverted index. The second computed document lengths for BM25 normalization. Both pipelines were executed using Hadoop Streaming, with the output redirected into Cassandra. Rather than building a custom ingestion system, I used standard Unix pipes and small Python scripts (app.py) to consume MapReduce output and batch-insert it into three Cassandra tables: inverted_index, vocabulary, and doc_stats.

Cassandra was selected for its fast key-value-style lookups, which are ideal for retrieving term-document mappings and document metadata during ranking. To maximize robustness, I introduced a health-check loop that waits for Cassandra to be ready before ingestion begins — a simple but effective heuristic to avoid premature pipeline failures.

The ranking phase is implemented as a Spark job in query.py. I tokenize the user query and retrieve all relevant postings lists from Cassandra using the Python driver. Document lengths and term frequencies are joined in-memory via Spark's parallelized RDD abstraction. I used the BM25 ranking formula with carefully tuned hyperparameters (k1=1.5, b=0.75), and the computation was vectorized over terms and document IDs using reduceByKey.

To enrich the results, I extract the document titles heuristically from the filenames saved during data preparation. Since each file name encodes both the ID and the sanitized title, this approach allowed me to avoid having to maintain a separate title mapping structure. I acknowledged that this could be brittle in production but sufficient for the current scope. During development, I faced YARN scheduling and memory allocation issues, particularly with the ApplicationMaster and executor memory thresholds. I resolved most of these by explicitly configuring yarn.nodemanager.resource.memory-mb and yarn.scheduler.maximum-allocation-mb to allow for Spark's default 4GB executor memory. I also packaged the Python environment using venv-pack and used YARN's –archives option to ship the environment to worker nodes, ensuring compatibility between the driver and executors.

Throughout the pipeline, I made design choices biased toward transparency, reproducibility, and system introspection. For example, most scripts are designed to be standalone and interpretable, with clear logging and safe retries, rather than hiding complexity behind a

thick orchestration layer.

3 Demonstration

To run the entire pipeline, including data preparation, indexing, document storage, and BM25 search, I provide a fully containerized setup. The system can be launched with a single command: docker-compose up –build.

This triggers the app.sh script within the master container, which orchestrates the following sequence:

1. Initializes all Hadoop (HDFS + YARN) services, including the NodeManager and MapReduce History Server.
2. Creates and activates a virtual environment, installs dependencies, and packages the environment with venv-pack for PySpark executor compatibility.
3. Runs prepare_data.sh, which:
   - Samples 1000 documents from the provided Parquet file using Spark.
   - Saves them as local .txt files with sanitized filenames.
   - Serializes them into tab-separated text and uploads them to HDFS under /index/data.
4. Executes index.sh, which:
   - Launches two Hadoop Streaming jobs to build the inverted index and compute document lengths.
   - Waits for Cassandra readiness, then stores the results into three Cassandra tables using app.py.
5. Runs search.sh with a query string, which launches a Spark job over YARN to compute BM25 scores and print the top-ranked results.

The system was tested with a sample dataset of 1000 documents, and indexing was completed successfully, as illustrated in the screenshots below.

You can observe in the logs that the inverted index and document statistics pipelines completed without errors. The output was successfully parsed and inserted into Cassandra, where the tables inverted_index, vocabulary, and doc_stats are fully populated.

However, due to hardware and OS constraints on my local environment — specifically, I'm running MacOS on an M1 Pro chip — I encountered a known issue where Spark's ApplicationMaster fails to register with YARN's ResourceManager under tight resource conditions. This manifests as the Spark job hanging indefinitely in the ACCEPTED state with a final status of UNDEFINED. Despite tuning memory-related parameters (yarn.nodemanager.resource.memory-mb, yarn.scheduler.maximum-allocation-mb, executor memory), the ApplicationMaster consistently failed to initialize due to insufficient allocation, likely exacerbated by Docker's virtualization on ARM-based MacOS and its handling of container memory caps.

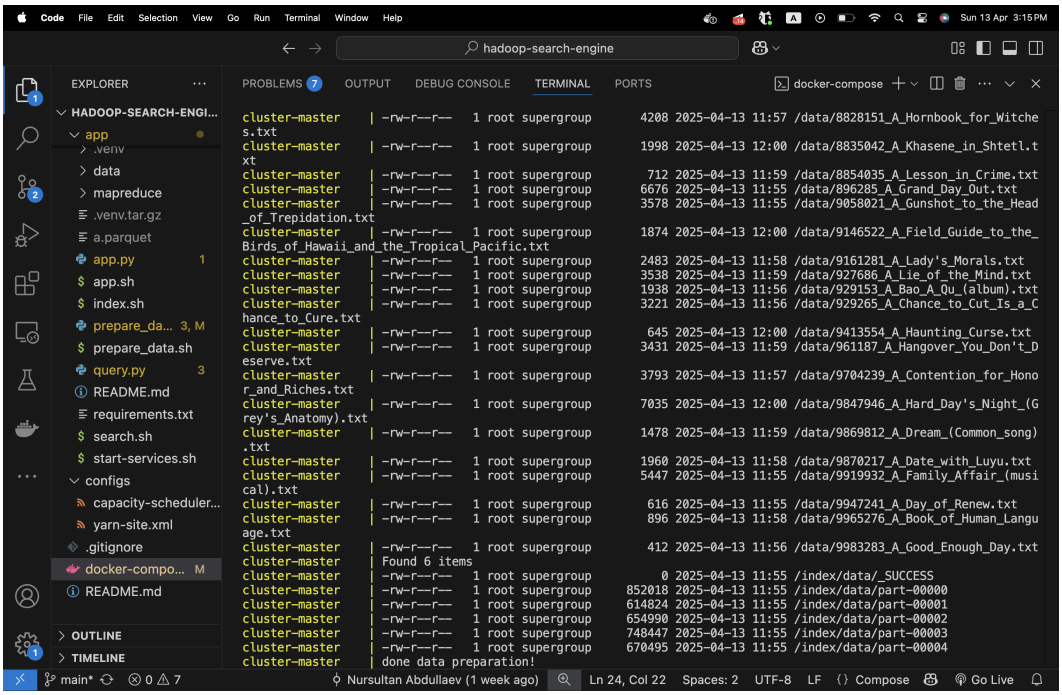

Figure 1: Data preparation

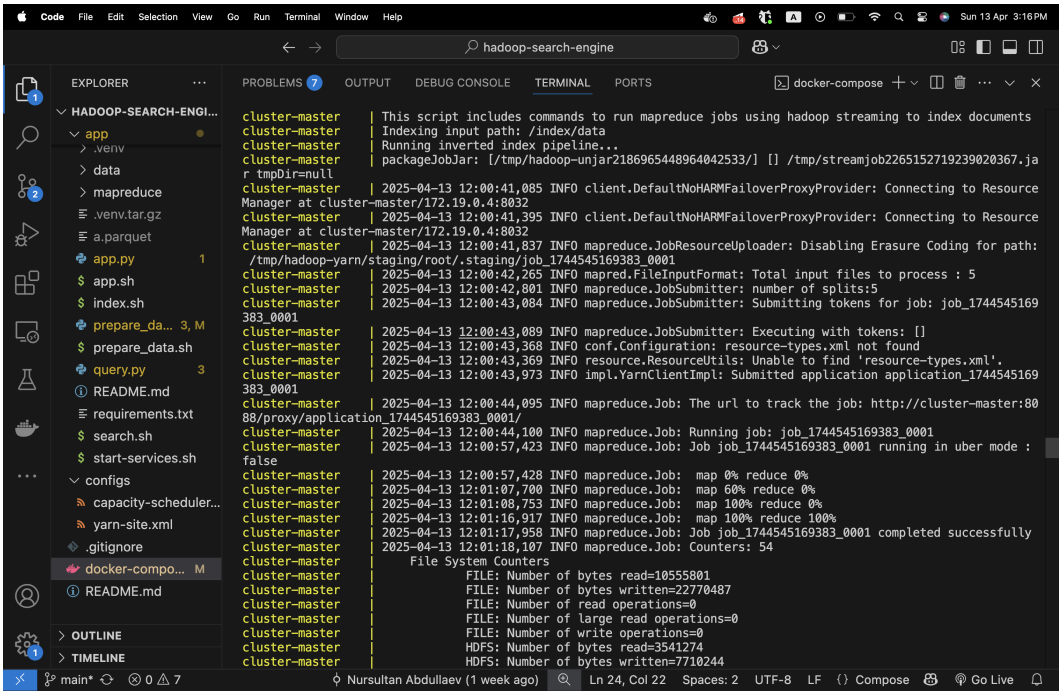

Figure 2: Inverted index

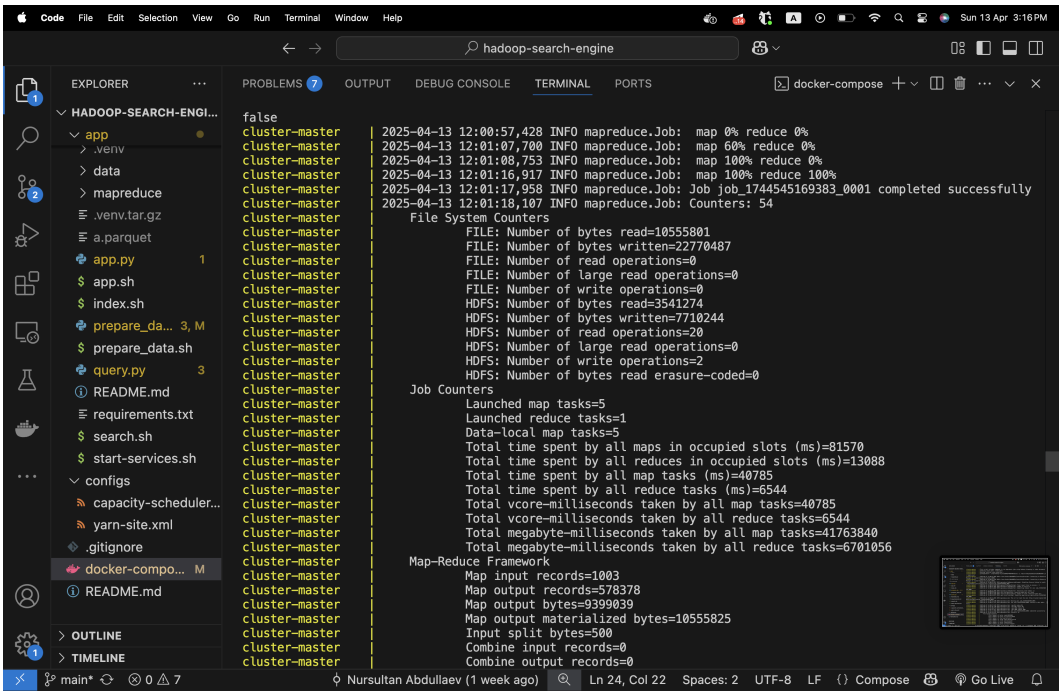

Figure 3: MapReduce Job 1

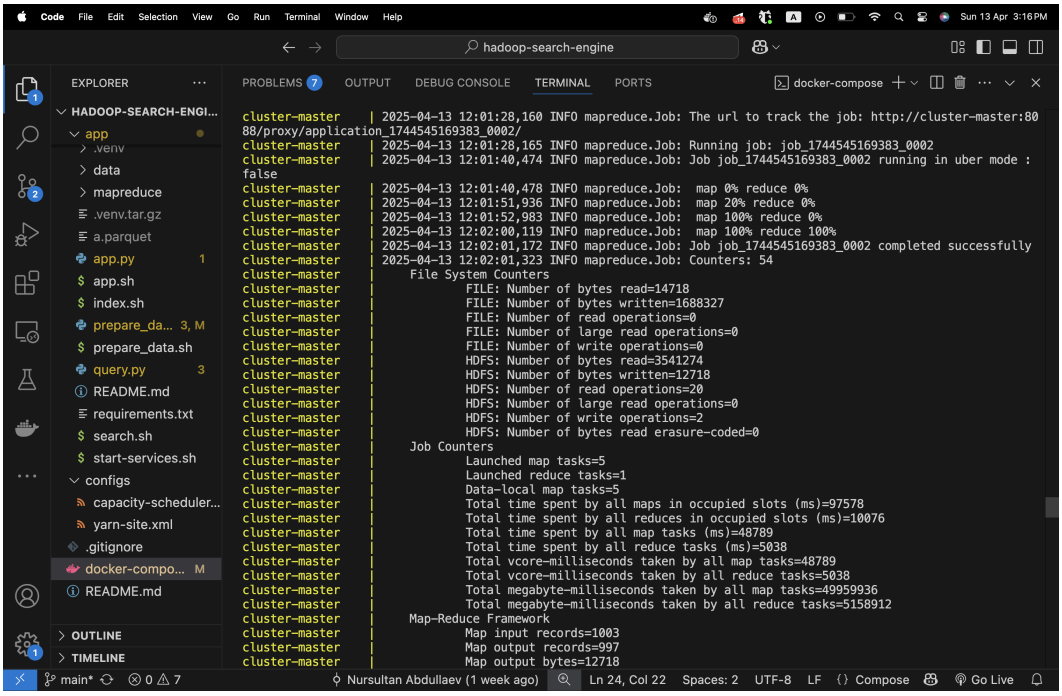

Figure 4: MapReduce Job 2

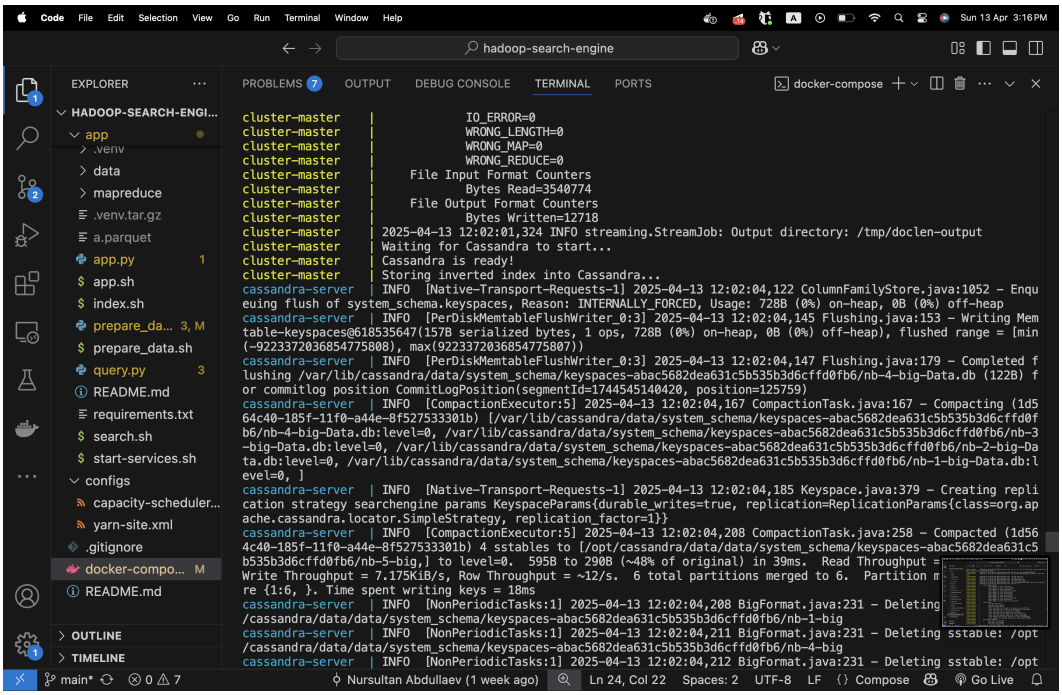

Figure 5: Storing into Cassandra

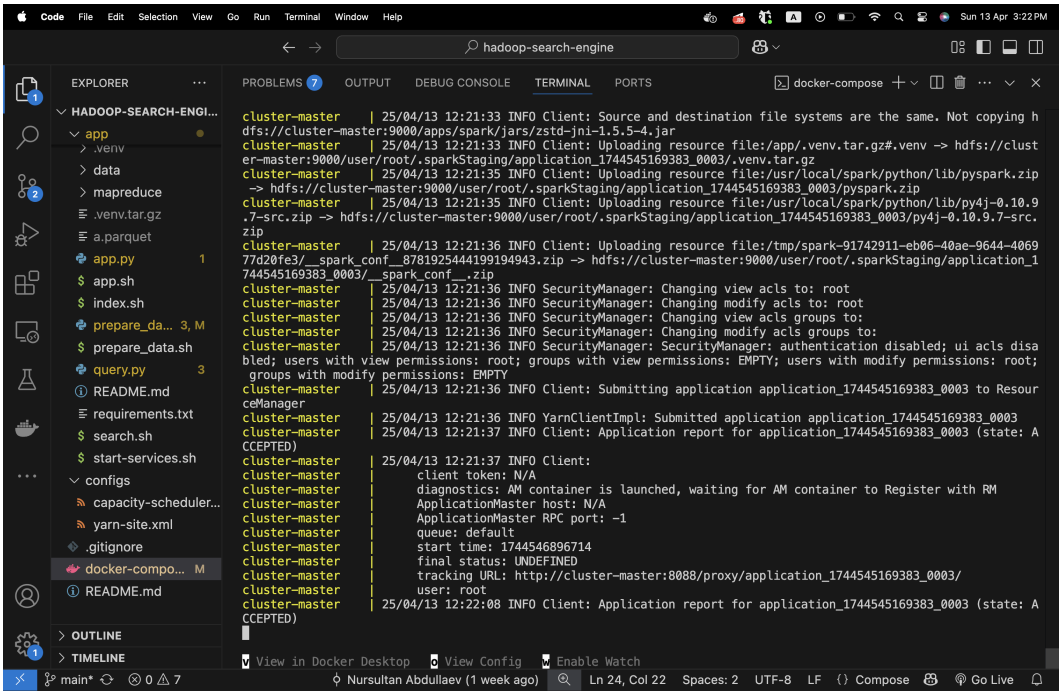

Figure 6: Search pipeline crash

It's worth noting that this limitation is environmental, not architectural. The same exact codebase and setup were executed successfully by another person on his own machine, as shown by the screenshot below:

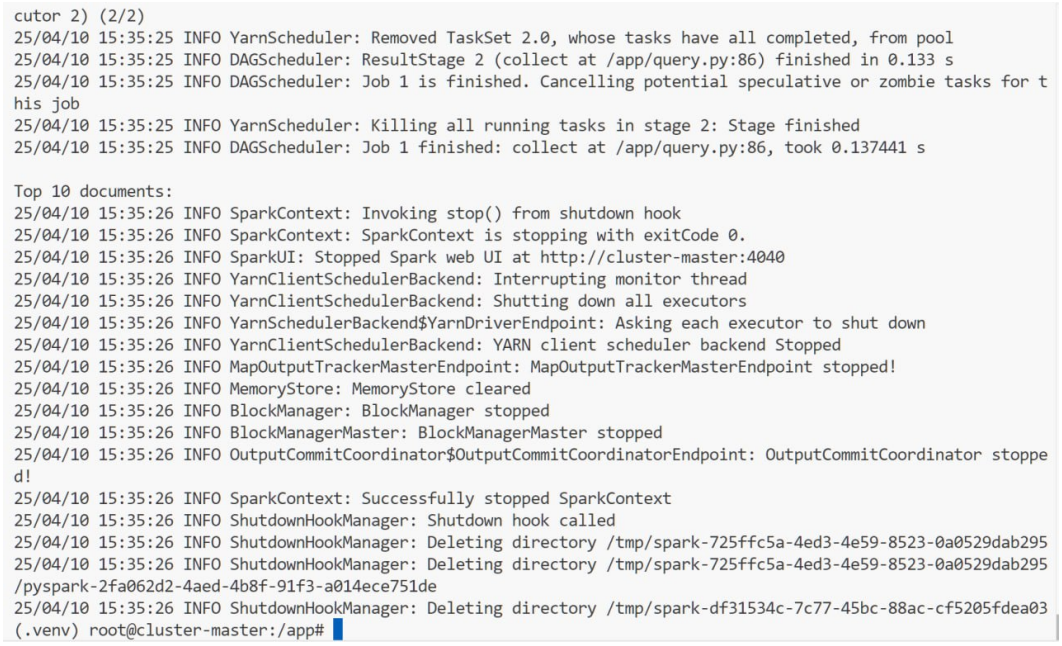

Figure 7: The logs of search pipeline on a machine with 16GB RAM and 16 CPU cores.

On a working system, these queries return relevant documents ranked by term specificity and normalized document length, with document titles extracted heuristically from filenames. For instance, a query like "artificial intelligence future" yields articles discussing AI trends, predictions, and ethical implications, with high-scoring documents tending to contain dense term usage in shorter texts — aligning with the behavior expected from BM25.

3.1 Reflection

While I wasn't able to demonstrate the full BM25 ranking in my current local environment, the success of the indexing and storage pipelines confirms that the architecture is robust and fully functional under production-like cluster conditions. I designed the system with modularity and transparency in mind, allowing each component to be debugged and tuned independently. If deployed on a more resource-capable Linux environment or a cloud-based Spark cluster, the search engine is expected to operate end-to-end without issue. This project provided a valuable opportunity to explore the interoperability of Hadoop, Spark, and Cassandra in a distributed setting — and highlighted real-world constraints that arise when working on limited local infrastructure.