Distributed Search Engine Implementation

Erokhin Evgenii
DS-01
e.erokhin@innopolis.university

April 16, 2025

1 System Pipeline Architecture

1.1 Data Preparation Stage

The initial data processing converts raw documents into searchable format:

- **Input:** Parquet files containing documents
- **Processing:**
  - Spark-based sampling
  - Text extraction and cleaning
- **Output:** Clean text files in HDFS

2 Detailed MapReduce Pipeline

2.1 Stage 1: Document Processing

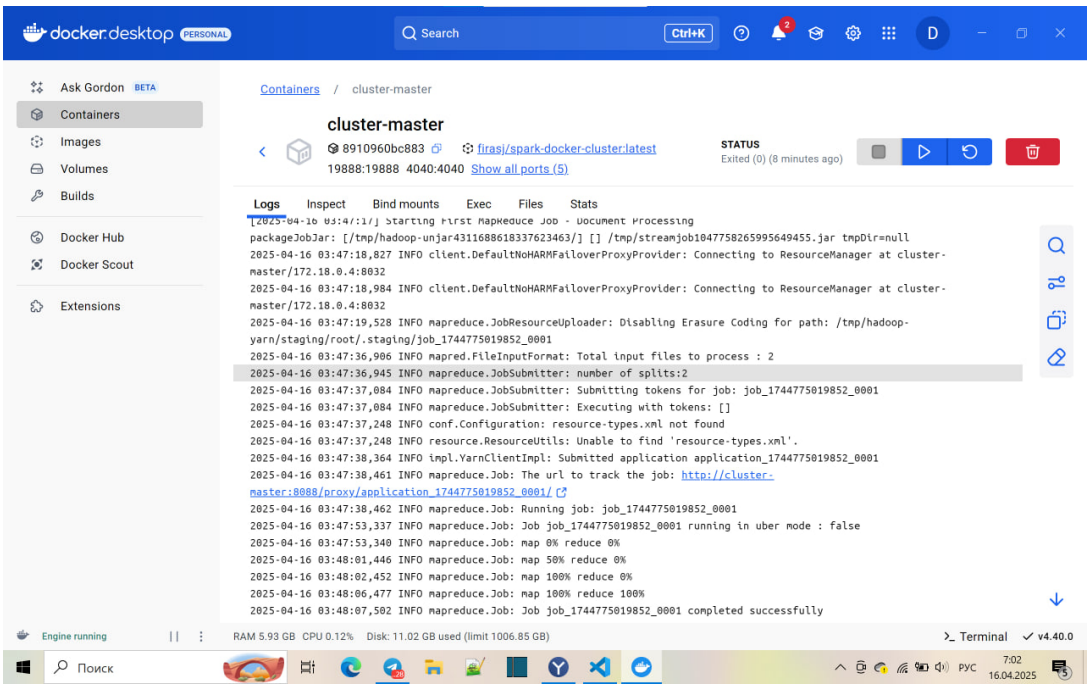

Figure 1: Stage 1: Document Processing Pipeline

The first MapReduce stage performs core text processing:

- **Mapper1 (mapper1.py)**:

```
def process_document(text):
    text = re.sub(r'[^a-zA-Z0-9\s]', ' ', text.lower())
    tokens = word_tokenize(text)
    return [stemmer.stem(t) for t in tokens if t.isalnum()
            and t not in stop_words]
```

  Key functions:

  - Tokenization using NLTK's word_tokenize()
  - Stopword removal (English stopwords)
  - Porter stemming for term normalization

- **Reducer1 (reducer1.py)** maintains:

  - `document_lengths`: {doc_id: length}
  - `term_occurrences`: {term: {doc_id: count}}
  - `texts_of_documents`: {doc_id: raw_text}

  Outputs:

  - Term frequencies per document
  - Document lengths
  - Raw document content

2.2 Stage 2: Index Building

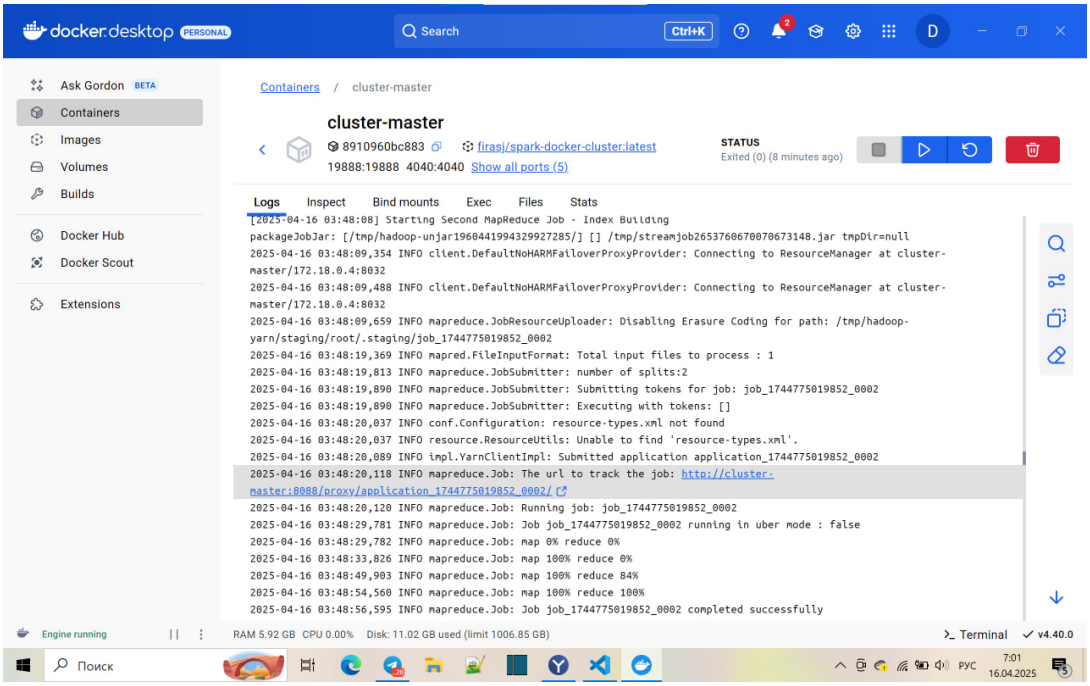

Figure 2: Stage 2: Index Building Pipeline

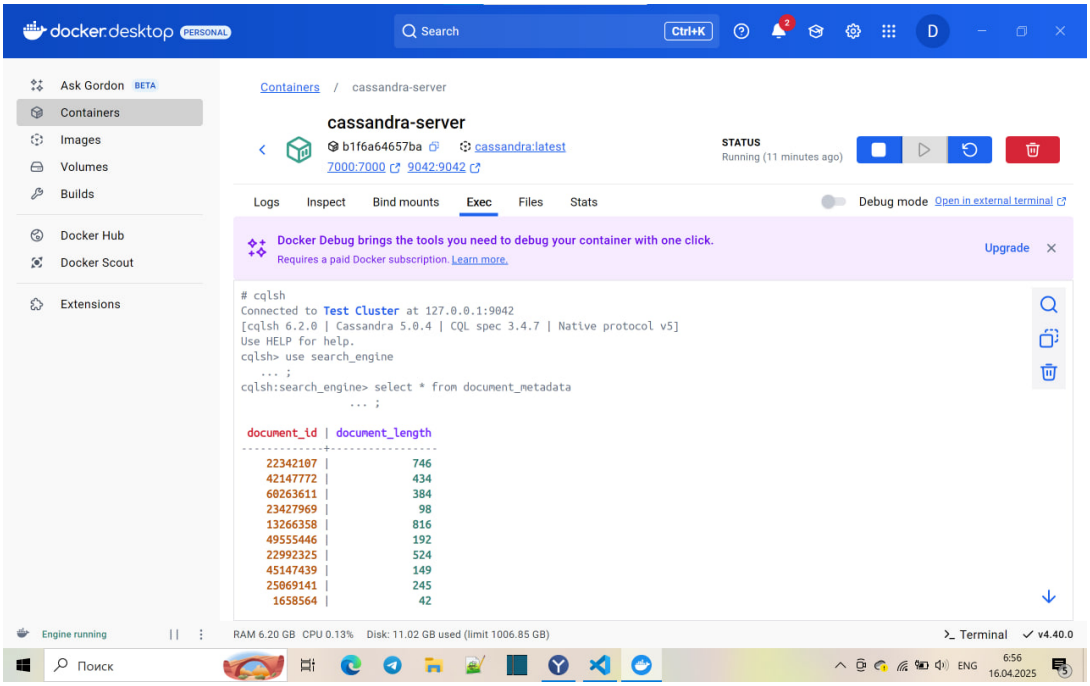

Figure 3: Stage 2: Index Building Pipeline

The second stage handles distributed index construction:

- **Mapper2 (mapper2.py)**: Simple pass-through mapper
- **Reducer2 (reducer2.py)** implements:

```
batch = BatchStatement(consistency_level=ConsistencyLevel.ONE)
for line in sys.stdin:
    if entry_type == "TERM_OCCURRENCE":
        batch.add(insert_term_occurrence,
                  (term, doc_id, count))
    elif entry_type == "DOCUMENT_LENGTH":
        batch.add(insert_document_metadata,
                  (doc_id, length))
    # Executes when batch reaches MAX_BATCH_SIZE
    if batch_size >= MAX_BATCH_SIZE:
        session.execute(batch)
```

Cassandra Tables:

Table	Purpose
document_metadata	Stores document lengths
search_terms	Vocabulary terms
document_content	Raw document texts
term_occurrences	Term frequencies
term_document_frequency	Document frequencies

3 Query Processing System

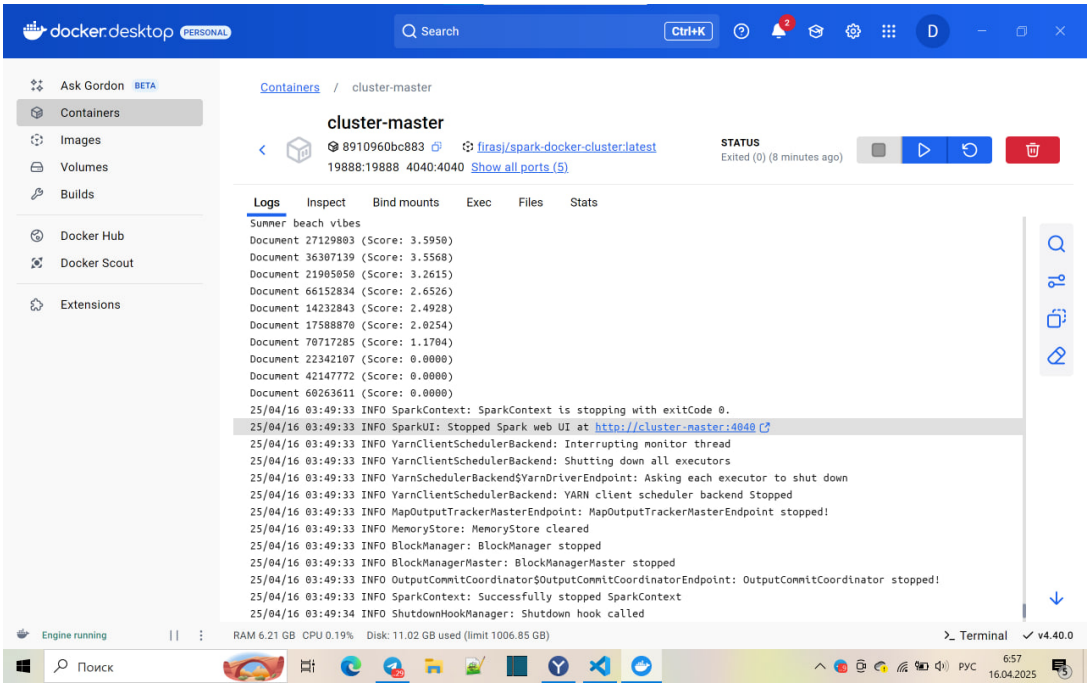

Figure 4: Query Processing Result

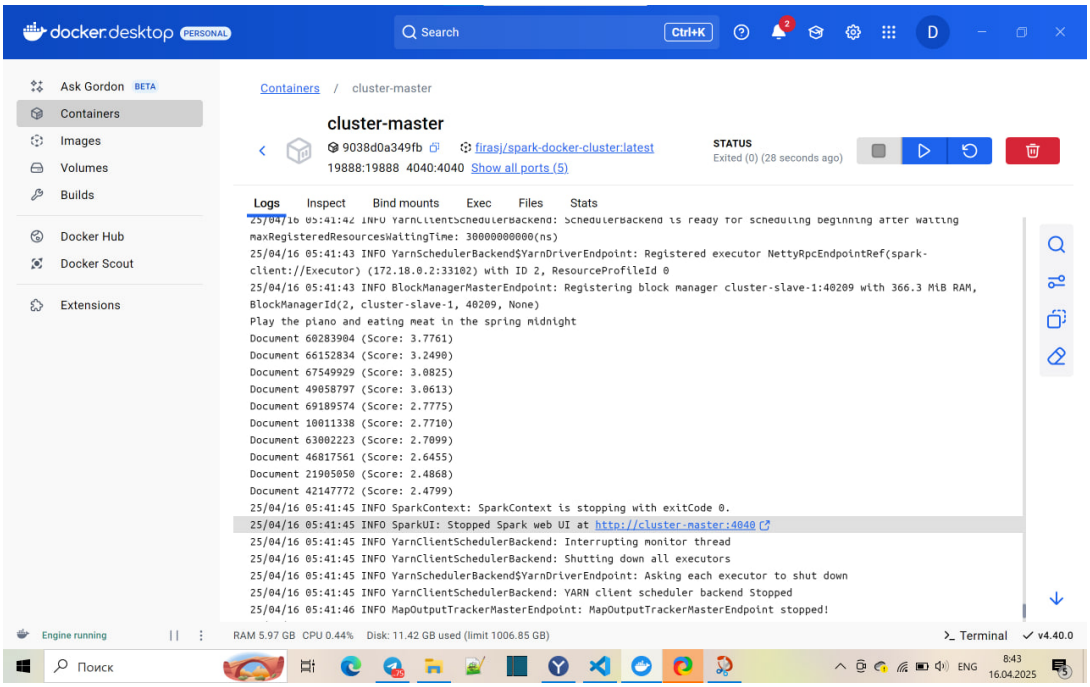

Figure 5: Query Processing Result

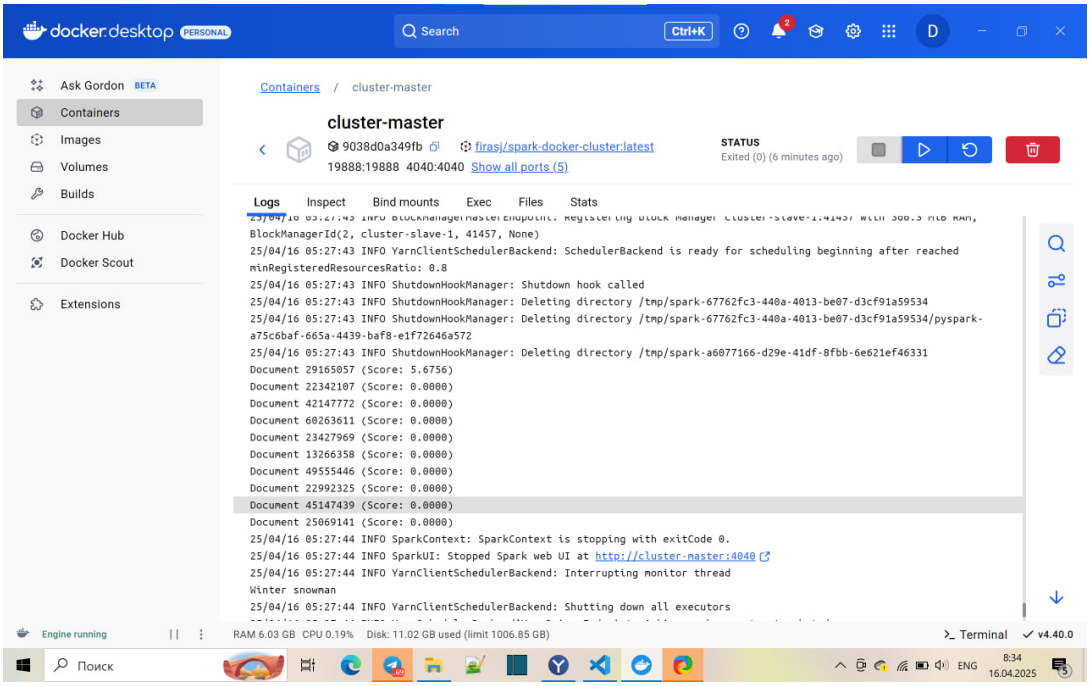

Figure 6: Query Processing Result

3.1 Search Implementation

The query processing system consists of:

- **Search Script (search.sh)**:

```
#!/bin/bash
source .venv/bin/activate
export PYSPARK_DRIVER_PYTHON=$(which python)
export PYSPARK_PYTHON=./.venv/bin/python
spark-submit --master yarn --archives /app/.venv.tar.gz#.venv query.py "$1"
```

- **Core Search Class (query.py)**:

```
class SearchEngine:
    def __init__(self):
        self.spark = SparkSession.builder \
            .appName("DocumentSearch") \
            .config("spark.cassandra.connection.host", CASSANDRA_HOST) \
            .getOrCreate()
        self.cluster, self.session = self._connect_to_cassandra()
```

3.2 BM25 Ranking Algorithm

The system implements the BM25 ranking function:

$$\text{score}(D,Q) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{f(q_i,D) \cdot (k_1+1)}{f(q_i,D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{avgdl}\right)} \tag{1}$$

Implementation in Python:

```
def _calculate_bm25_score(self, query_terms, document_id,
                          document_length, avg_document_length,
                          term_occurrences, term_document_freqs,
                          total_documents):
    score = 0.0
    for term in query_terms:
```

```
    if term in term_occurrences and document_id in term_occurrences[term]:
        tf = term_occurrences[term][document_id]
        df = term_document_freqs.get(term, 0)
        idf = math.log((total_documents - df + 0.5) / (df + 0.5) + 1.0)
        numerator = tf * (BM25_K1 + 1)
        denominator = tf + BM25_K1 * (1 - BM25_B + BM25_B *
                    (document_length / avg_document_length))
        score += idf * (numerator / denominator)
return score
```

4 Implementation Highlights

- **Fault Tolerance**:
  - Cassandra connection retries (5 attempts)
  - Batch processing with error handling
  - YARN log capture for debugging
- **Optimizations**:
  - Memory management (1800MB heap)
  - Batch inserts (MAX_BATCH_SIZE = 1)
  - Columnar data organization