Simple Search Engine using Hadoop MapReduce

Assignment 2 Report

This report documents the implementation of a simple search engine using Hadoop MapReduce, Apache Cassandra, and Spark RDD. The search engine indexes a collection of text documents from Wikipedia, builds an inverted index, and retrieves documents relevant to user queries based on the BM25 ranking algorithm. The implementation follows a distributed computing approach where document indexing is handled as batch processing using Hadoop MapReduce, while query processing is performed using Spark RDD for efficient real-time retrieval.

Methodology

System Architecture

The search engine implementation follows a distributed architecture with the following components:

1. **Document Processing**: Using PySpark to extract and prepare documents from parquet files
2. **Indexing Engine**: Using Hadoop MapReduce to build an inverted index of terms
3. **Storage Layer**: Using Apache Cassandra for storing the index data
4. **Query Processing**: Using Spark RDD to process queries and retrieve relevant documents

Indexing Process

The indexing engine uses a two-stage MapReduce pipeline to create an inverted index of terms from the document corpus:

First MapReduce Job:

- **Mapper (mapper1.py)**:
    - Reads documents from input files
    - Tokenizes text content (lowercase, remove stopwords, apply stemming)
    - Calculates term frequencies for each document
    - Emits key-value pairs: `(term, (doc_id, term_freq, doc_length))`
- **Reducer (reducer1.py)**:
    - Aggregates information for each term across all documents
    - Calculates document frequency for each term
    - Emits term data with document frequencies and posting lists

- Also emits corpus statistics as a special key-value pair

Second MapReduce Job:

- **Mapper (mapper2.py)**:
    - Processes output from the first reducer
    - Separates vocabulary entries from posting entries
    - Formats data for insertion into Cassandra tables
- **Reducer (reducer2.py)**:
    - Connects to Cassandra cluster
    - Creates required tables if they don't exist
    - Inserts vocabulary, postings, and corpus statistics into respective tables

This two-stage pipeline efficiently distributes the indexing workload across the Hadoop cluster and prepares the data for retrieval operations.

Cassandra Data Model

The search engine uses the following data model in Cassandra:

1. **vocabulary** table:
    - `term` (text, PRIMARY KEY): Stemmed word from the corpus
    - `doc_frequency` (int): Number of documents containing the term
2. **postings** table:
    - `term` (text): Stemmed word
    - `doc_id` (text): Document identifier
    - `term_frequency` (int): Frequency of the term in the document
    - PRIMARY KEY (term, doc_id): Enables efficient lookup of documents by term
3. **corpus_stats** table:
    - `id` (text, PRIMARY KEY): Identifier for corpus statistics
    - `doc_count` (int): Total number of documents in the corpus
    - `avg_doc_length` (float): Average document length in the corpus
4. **document_stats** table:
    - `doc_id` (text, PRIMARY KEY): Document identifier
    - `doc_length` (int): Length of the document (number of terms)

This schema design allows for efficient retrieval of term statistics and document information needed for BM25 scoring.

Query Processing

Query processing is implemented using Spark RDD to provide efficient retrieval of relevant documents:

1. **Query Tokenization**: The user query is tokenized, stemmed, and processed in the same way as documents during indexing.

2. **Term Information Retrieval**: For each query term, the system retrieves:

    - Document frequency from the `vocabulary` table
    - Posting list from the `postings` table
    - Corpus statistics from the `corpus_stats` table
    - Document lengths from the `document_stats` table

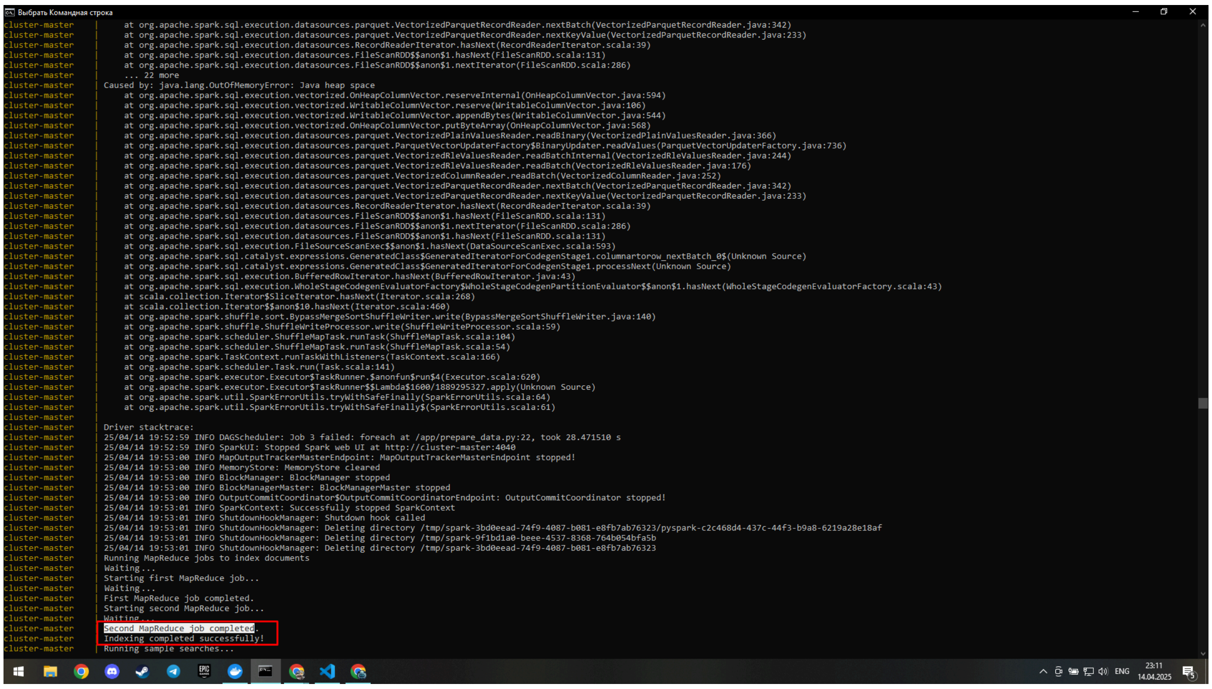
3. **BM25 Scoring**: For each document containing any query term, the system calculates a BM25 score based on term frequency, document frequency, document length, and corpus statistics.

4. **Result Ranking**: Documents are sorted by their BM25 scores, and the top 10 results are returned to the user.

Demonstration

Document Indexing

The indexing process was tested on a collection of 1000 Wikipedia articles. The process included data preparation, two MapReduce jobs, and storage in Cassandra.



```
Выбрать Командная строка
cluster-master    |     at org.apache.spark.sql.execution.datasources.parquet.VectorizedParquetRecordReader.nextBatch(VectorizedParquetRecordReader.java:342)
cluster-master    |     at org.apache.spark.sql.execution.datasources.parquet.VectorizedParquetRecordReader.nextKeyValue(VectorizedParquetRecordReader.java:233)
cluster-master    |     at org.apache.spark.sql.execution.datasources.RecordReaderIterator.hasNext(RecordReaderIterator.scala:39)
cluster-master    |     at org.apache.spark.sql.execution.datasources.FileScanRDD$$anon$1.hasNext(FileScanRDD.scala:131)
cluster-master    |     at org.apache.spark.sql.execution.datasources.FileScanRDD$$anon$1.nextIterator(FileScanRDD.scala:286)
cluster-master    |     ... 22 more
cluster-master    | Caused by: java.lang.OutOfMemoryError: Java heap space
cluster-master    |     at org.apache.spark.sql.execution.vectorized.OnHeapColumnVector.reserveInternal(OnHeapColumnVector.java:594)
cluster-master    |     at org.apache.spark.sql.execution.vectorized.WritableColumnVector.reserve(WritableColumnVector.java:106)
cluster-master    |     at org.apache.spark.sql.execution.vectorized.WritableColumnVector.appendBytes(WritableColumnVector.java:544)
cluster-master    |     at org.apache.spark.sql.execution.vectorized.OnHeapColumnVector.putByteArray(OnHeapColumnVector.java:568)
cluster-master    |     at org.apache.spark.sql.execution.datasources.parquet.VectorizedPlainValuesReader.readBinary(VectorizedPlainValuesReader.java:366)
cluster-master    |     at org.apache.spark.sql.execution.datasources.parquet.ParquetVectorUpdaterFactory$BinaryUpdater.readValues(ParquetVectorUpdaterFactory.java:736)
cluster-master    |     at org.apache.spark.sql.execution.datasources.parquet.VectorizedRleValuesReader.readBatchInternal(VectorizedRleValuesReader.java:244)
cluster-master    |     at org.apache.spark.sql.execution.datasources.parquet.VectorizedRleValuesReader.readBatch(VectorizedRleValuesReader.java:176)
cluster-master    |     at org.apache.spark.sql.execution.datasources.parquet.VectorizedColumnReader.readBatch(VectorizedColumnReader.java:252)
cluster-master    |     at org.apache.spark.sql.execution.datasources.parquet.VectorizedParquetRecordReader.nextBatch(VectorizedParquetRecordReader.java:342)
cluster-master    |     at org.apache.spark.sql.execution.datasources.parquet.VectorizedParquetRecordReader.nextKeyValue(VectorizedParquetRecordReader.java:233)
cluster-master    |     at org.apache.spark.sql.execution.datasources.RecordReaderIterator.hasNext(RecordReaderIterator.scala:39)
cluster-master    |     at org.apache.spark.sql.execution.datasources.FileScanRDD$$anon$1.hasNext(FileScanRDD.scala:131)
cluster-master    |     at org.apache.spark.sql.execution.datasources.FileScanRDD$$anon$1.nextIterator(FileScanRDD.scala:286)
cluster-master    |     at org.apache.spark.sql.execution.datasources.FileScanRDD$$anon$1.hasNext(FileScanRDD.scala:131)
cluster-master    |     at org.apache.spark.sql.execution.FileSourceScanExec$$anon$1.hasNext(DataSourceScanExec.scala:593)
cluster-master    |     at org.apache.spark.sql.catalyst.expressions.GeneratedClass$GeneratedIteratorForCodegenStage1.columnartorow_nextBatch_0$(Unknown Source)
cluster-master    |     at org.apache.spark.sql.catalyst.expressions.GeneratedClass$GeneratedIteratorForCodegenStage1.processNext(Unknown Source)
cluster-master    |     at org.apache.spark.sql.execution.BufferedRowIterator.hasNext(BufferedRowIterator.java:43)
cluster-master    |     at org.apache.spark.sql.execution.WholeStageCodegenEvaluatorFactory$WholeStageCodegenPartitionEvaluator$$anon$1.hasNext(WholeStageCodegenEvaluatorFactory.scala:43)
cluster-master    |     at scala.collection.Iterator$SliceIterator.hasNext(Iterator.scala:268)
cluster-master    |     at scala.collection.Iterator$$anon$10.hasNext(Iterator.scala:460)
cluster-master    |     at org.apache.spark.shuffle.sort.BypassMergeSortShuffleWriter.write(BypassMergeSortShuffleWriter.java:140)
cluster-master    |     at org.apache.spark.shuffle.ShuffleWriteProcessor.write(ShuffleWriteProcessor.scala:59)
cluster-master    |     at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:104)
cluster-master    |     at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:54)
cluster-master    |     at org.apache.spark.TaskContext.runTaskWithListeners(TaskContext.scala:166)
cluster-master    |     at org.apache.spark.scheduler.Task.run(Task.scala:141)
cluster-master    |     at org.apache.spark.executor.Executor$TaskRunner.$anonfun$run$4(Executor.scala:620)
cluster-master    |     at org.apache.spark.executor.Executor$TaskRunner$$Lambda$1600/1889295327.apply(Unknown Source)
cluster-master    |     at org.apache.spark.util.SparkErrorUtils.tryWithSafeFinally(SparkErrorUtils.scala:64)
cluster-master    |     at org.apache.spark.util.SparkErrorUtils.tryWithSafeFinally$(SparkErrorUtils.scala:61)
cluster-master    |
cluster-master    | Driver stacktrace:
cluster-master    | 25/04/14 19:52:59 INFO DAGScheduler: Job 3 failed: foreach at /app/prepare_data.py:22, took 28.471510 s
cluster-master    | 25/04/14 19:52:59 INFO SparkUI: Stopped Spark web UI at http://cluster-master:4040
cluster-master    | 25/04/14 19:53:00 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
cluster-master    | 25/04/14 19:53:00 INFO MemoryStore: MemoryStore cleared
cluster-master    | 25/04/14 19:53:00 INFO BlockManager: BlockManager stopped
cluster-master    | 25/04/14 19:53:00 INFO BlockManagerMaster: BlockManagerMaster stopped
cluster-master    | 25/04/14 19:53:00 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
cluster-master    | 25/04/14 19:53:01 INFO SparkContext: Successfully stopped SparkContext
cluster-master    | 25/04/14 19:53:01 INFO ShutdownHookManager: Shutdown hook called
cluster-master    | 25/04/14 19:53:01 INFO ShutdownHookManager: Deleting directory /tmp/spark-3bd0eead-74f9-4087-b081-e8fb7ab76323/pyspark-c2c468d4-437c-44f3-b9a8-6219a28e18af
cluster-master    | 25/04/14 19:53:01 INFO ShutdownHookManager: Deleting directory /tmp/spark-9f1bd1a0-beee-4537-8368-764b054bfa5b
cluster-master    | 25/04/14 19:53:01 INFO ShutdownHookManager: Deleting directory /tmp/spark-3bd0eead-74f9-4087-b081-e8fb7ab76323
cluster-master    | Running MapReduce jobs to index documents
cluster-master    | Waiting...
cluster-master    | Starting first MapReduce job...
cluster-master    | Waiting...
cluster-master    | First MapReduce job completed.
cluster-master    | Starting second MapReduce job...
cluster-master    | Waiting...
cluster-master    | Second MapReduce job completed.
cluster-master    | Indexing completed successfully!
cluster-master    | Running sample searches...
```

Conclusion

This implementation demonstrates a functional search engine using Hadoop MapReduce for indexing, Cassandra for storage, and Spark RDD for query processing. The BM25 ranking algorithm provides relevant search results by considering term frequency, document frequency, and document length.

The distributed architecture allows the system to handle large document collections efficiently, with batch processing for indexing and low-latency retrieval for query processing. While this implementation focuses on basic text search functionality, it provides a foundation that could be extended with additional features such as phrase matching, relevance feedback, or vector space models.

The project successfully demonstrates the application of Big Data technologies (Hadoop, Cassandra, Spark) to the problem of information retrieval, highlighting both the benefits and challenges of distributed computing approaches for search applications.