Lab 5: Mini-MapReduce on Amazon EMR

Student: Aruzhan Saparkhankyzy
Group: IT-2306

1 Introduction

Big data processing requires scalable and fault-tolerant distributed systems. MapReduce is a programming model designed to process large datasets in parallel across multiple nodes. Amazon Elastic MapReduce (EMR) is a managed cloud service that simplifies running Hadoop-based workloads. The goal of this laboratory work is to deploy a Hadoop cluster on Amazon EMR, run a MapReduce job using Hadoop Streaming with Python, store data in HDFS, and analyze performance through experimentation.

2 Objectives

The main objective of this laboratory work is to study the MapReduce programming model and understand how it enables distributed processing of large-scale datasets. Another important goal is to deploy and configure a Hadoop cluster using Amazon Elastic MapReduce (EMR) in a cloud environment. The laboratory work also aims to implement and execute a WordCount MapReduce job using Python through Hadoop Streaming. In addition, the use of the Hadoop Distributed File System (HDFS) as a distributed storage system for both input and output data is examined. Finally, the performance of the MapReduce job is analyzed by conducting an experiment that evaluates how execution time changes with different input sizes.

3 Technology Stack

This laboratory work was implemented using Amazon Elastic MapReduce (EMR) as a managed cloud platform for running Hadoop. The MapReduce programming model was applied using Hadoop MapReduce, while the Hadoop Distributed File System (HDFS) was used for storing input and output data. Python was utilized for implementing the mapper and reducer through Hadoop Streaming. Cluster management and job execution were performed using Secure Shell (SSH) and the AWS Management Console.

4 Dataset Description

The dataset used in this laboratory is a small Wikipedia text dump from the Simple English Wikipedia. It contains textual articles suitable for word frequency analysis. The dataset was downloaded directly on the EMR master node and then uploaded to HDFS for distributed processing.

4.1 Dataset Preparation

```
wget https://github.com/LGDoor/Dump-of-Simple-English-Wiki/raw/refs/heads/
   master/corpus.tgz
tar -xvzf corpus.tgz
ls -lh corpus.txt
```

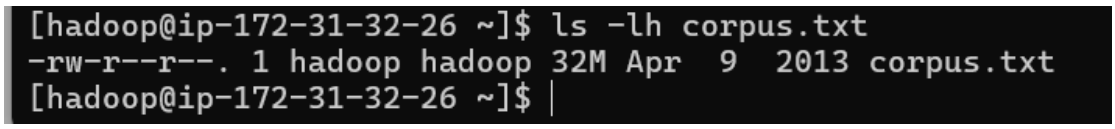

Figure 1: Dataset download and extraction on the EMR master node.

5 Cluster Setup

An Amazon EMR cluster was created using the AWS Academy Learner Lab environment.

5.1 Cluster Configuration

The EMR cluster was configured using EMR version 7.12.0 with Hadoop version 3.4.1. The cluster consisted of one primary (master) node and one core (worker) node, which enabled distributed execution of MapReduce tasks. All instances were launched using the m4.large instance type supported in the AWS Academy environment. The cluster was created with the IAM roles EMR_DefaultRole and EMR_EC2_DefaultRole to ensure correct permissions and access.

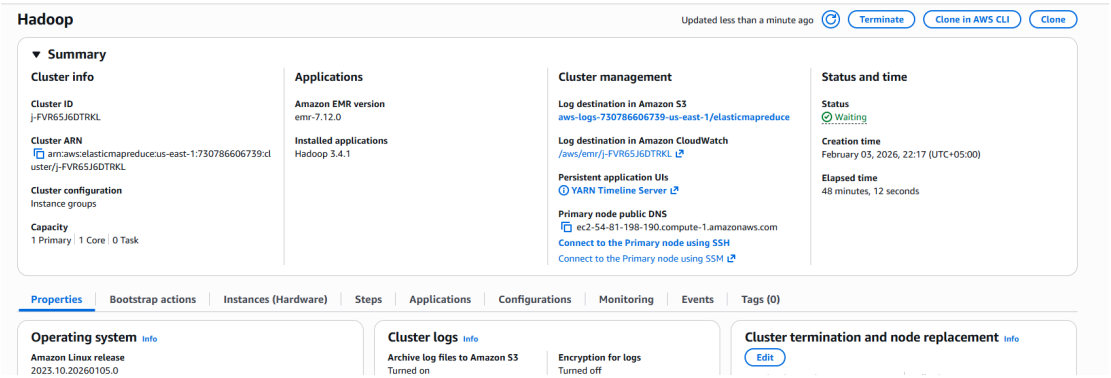

Figure 2: EMR cluster summary and configuration (version, nodes, roles).

5.2 Cluster Verification

After connecting to the master node via SSH, the cluster status was verified using YARN and HDFS administrative commands. The output confirmed that the cluster services were running correctly and that at least one node manager and datanode were available.

```
yarn node -list
hdfs dfsadmin -report | head -n 30
```

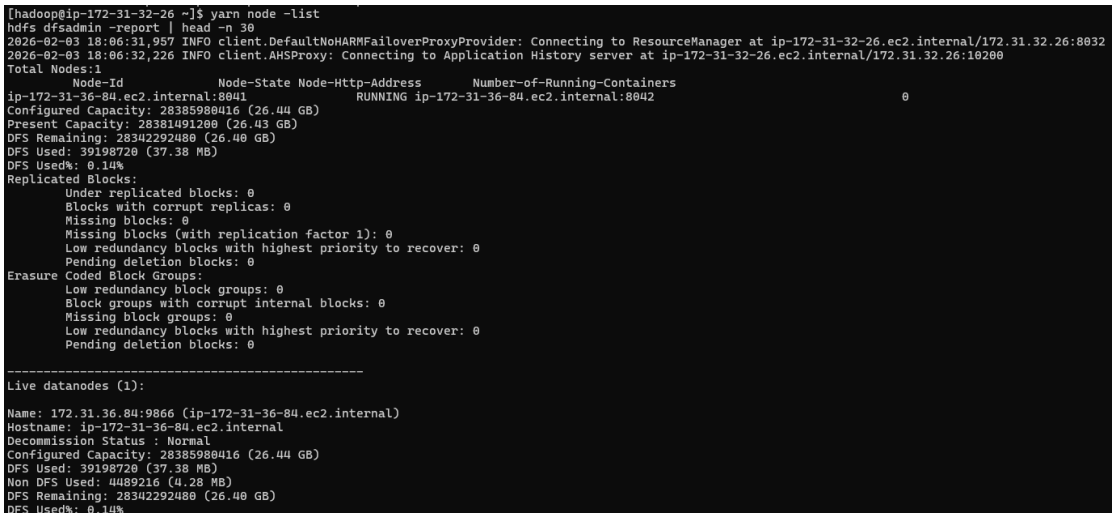

Figure 3: Cluster verification: YARN node list and HDFS report.

6 HDFS Data Storage

To ensure distributed processing, the input dataset was uploaded into HDFS. An input directory was created and the extracted `corpus.txt` file was copied to HDFS.

```
hdfs dfs -mkdir -p /user/hadoop/input
hdfs dfs -put -f corpus.txt /user/hadoop/input/
hdfs dfs -ls /user/hadoop/input
```

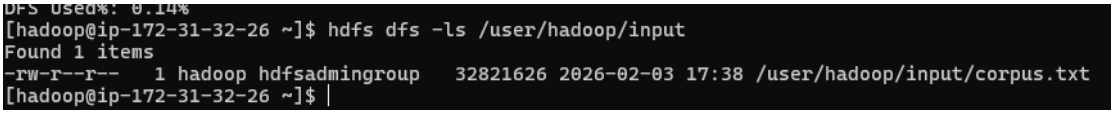

Figure 4: The dataset stored in HDFS under `/user/hadoop/input`.

7 MapReduce Programming Model

The MapReduce programming model consists of three main stages. During the Map phase, input data is processed and transformed into intermediate key–value pairs. The Shuffle phase is automatically handled by the Hadoop framework and is responsible for grouping all intermediate values associated with the same key. Finally, during the Reduce phase, these grouped values are aggregated to produce the final output. This model enables parallel and scalable processing of large datasets across multiple nodes.

8 WordCount Implementation

The WordCount task counts the frequency of each word in the dataset. It was implemented using Hadoop Streaming with Python.

8.1 Mapper

The mapper reads lines of text, extracts alphabetic words, normalizes them to lowercase, and emits `(word, 1)` pairs.

```
#!/usr/bin/env python3
import sys, re

for line in sys.stdin:
    line = line.strip().lower()
    for w in re.findall(r"[a-z]+", line):
        print(f"{w}\t1")
```

8.2 Reducer

The reducer receives grouped values for each word and sums them to produce total counts per word.

```
#!/usr/bin/env python3
import sys

cur = None
```

```
cnt = 0

for line in sys.stdin:
    line = line.strip()
    if not line:
        continue
    w, c = line.split("\t", 1)
    c = int(c)

    if w == cur:
        cnt += c
    else:
        if cur is not None:
            print(f"{cur}\t{cnt}")
        cur = w
        cnt = c

if cur is not None:
    print(f"{cur}\t{cnt}")
```

9 Running MapReduce on EMR

The Hadoop Streaming job was executed on the EMR cluster. Before running the job, the output directory was removed to avoid conflicts.

```
hdfs dfs -rm -r -f /user/hadoop/output/wordcount

hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
  -D mapreduce.job.name="wordcount-streaming" \
  -input /user/hadoop/input/corpus.txt \
  -output /user/hadoop/output/wordcount \
  -mapper "python3␣mapper.py" \
  -reducer "python3␣reducer.py" \
  -file mapper.py \
  -file reducer.py
```

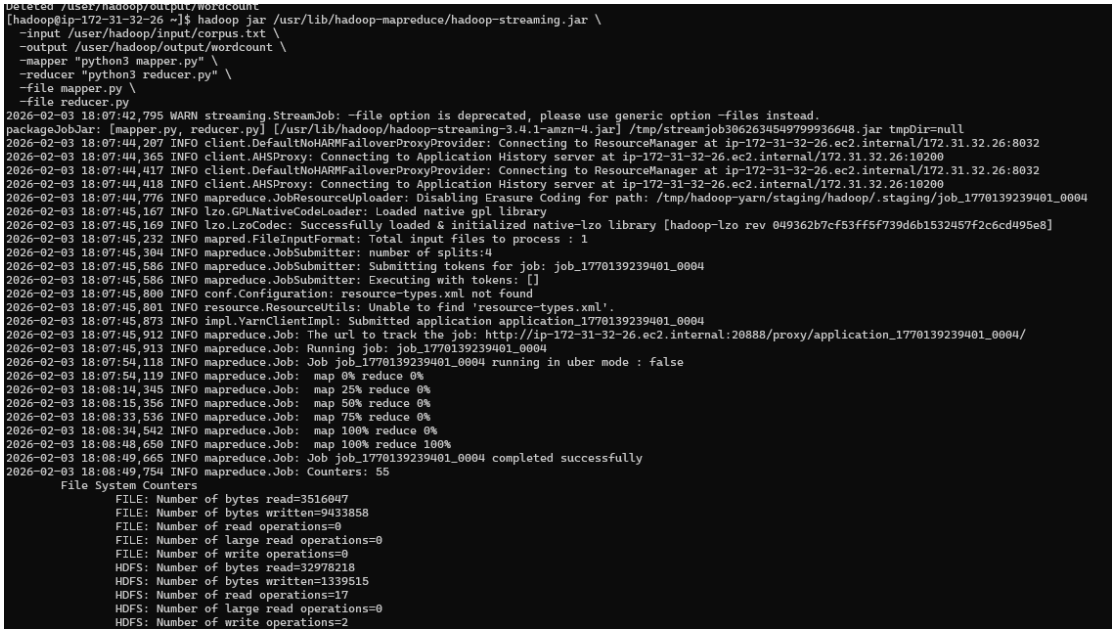

Figure 5: Hadoop Streaming job execution and successful completion.

Figure 6: Hadoop Streaming job execution and successful completion.

10 Output Validation

After execution, the output was verified in HDFS. The output directory contained the _SUCCESS marker and the reducer output file part-00000. To validate correctness, the first lines were displayed and the top-10 most frequent words were extracted.

```
hdfs dfs -ls /user/hadoop/output/wordcount
hdfs dfs -head /user/hadoop/output/wordcount/part-00000

hdfs dfs -cat /user/hadoop/output/wordcount/part-00000 | sort -k2 -nr | head -
    n 10
```

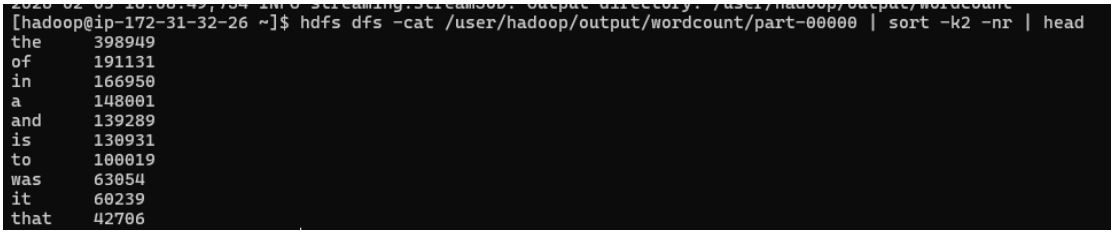

Figure 7: Output validation: HDFS output listing and top-10 most frequent words.

11 Experiment: Input Size Analysis

To analyze performance, an input-size experiment was conducted. The WordCount job was executed on a small input (approximately 1 MB) and on the full dataset (approximately 32 MB), and runtimes were measured using the time command.

11.1 Small Dataset

A smaller dataset was created by taking the first one million bytes from the original file and uploading it to HDFS. The WordCount job executed on the small dataset completed in **50.184 seconds**.

```
head -c 1000000 corpus.txt > corpus_small.txt
hdfs dfs -mkdir -p /user/hadoop/input_small
hdfs dfs -put -f corpus_small.txt /user/hadoop/input_small/

hdfs dfs -rm -r -f /user/hadoop/output/wordcount_small
time hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
  -input /user/hadoop/input_small/corpus_small.txt \
  -output /user/hadoop/output/wordcount_small \
  -mapper "python3␣mapper.py" \
  -reducer "python3␣reducer.py" \
```

```
  -file mapper.py \
  -file reducer.py
```

11.2 Large Dataset

The same job was executed on the full dataset stored in HDFS. The runtime for the large dataset was **69.150 seconds**.

```
hdfs dfs -rm -r -f /user/hadoop/output/wordcount_big
time hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
  -input /user/hadoop/input/corpus.txt \
  -output /user/hadoop/output/wordcount_big \
  -mapper "python3␣mapper.py" \
  -reducer "python3␣reducer.py" \
  -file mapper.py \
  -file reducer.py
```

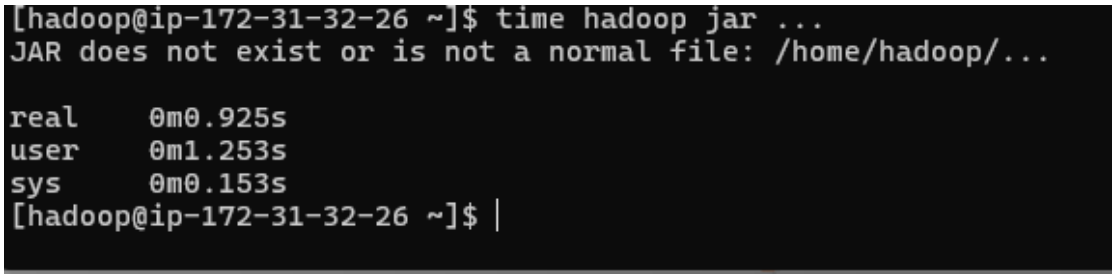

Figure 8: Experiment results: runtime comparison for small vs large input size.

11.3 Experiment Analysis

The experiment results show that increasing input size leads to increased execution time. This happens because a larger dataset produces more intermediate key–value pairs during the Map stage, which increases shuffle and sort overhead before the Reduce stage. Therefore, MapReduce performance scales with input volume, and runtime generally grows as the amount of processed data increases.

12 Conclusion

In this laboratory work, a Hadoop cluster was successfully deployed using Amazon EMR. A WordCount MapReduce job was implemented in Python using Hadoop Streaming and executed on the EMR cluster with input and output stored in HDFS. The output was validated by inspecting reducer results and extracting the most frequent words. Finally, an input-size experiment confirmed that MapReduce execution time increases

when processing a larger dataset due to higher map output and shuffle overhead. Overall, this laboratory demonstrates practical application of distributed computing concepts and performance analysis using MapReduce in a cloud environment.