University of El Oued

Faculty of Exact Sciences
Department of Computer Science

Lab Report 9

Spark Batch & Streaming Processing with Docker

Student: Mohammed Seddik Lifa
Specialization: Master II: AI & Data Science

Date: January 9, 2026

Abstract

This report documents the implementation of a single-node Hadoop/Spark cluster using Docker. We explore three core data processing paradigms: Interactive Processing using Spark Shell (Scala), Batch Processing using a compiled Java application, and Real-Time Streaming using Spark Structured Streaming.

Contents

1 Objective

The primary objective of this lab is to deploy a containerized Big Data environment and perform distributed data processing tasks. The lab involves:

- Deploying a Hadoop/Spark cluster using Docker Compose.
- Performing interactive data analysis with Scala (Spark Shell).
- Developing and submitting a Java Batch Processing job (MapReduce).
- Implementing a Real-Time Streaming pipeline using Spark Structured Streaming.

2 Cluster Installation & Setup

2.1 Environment Initialization

We initialized the cluster using `docker compose`. Post-deployment, we configured SSH keys and user permissions to ensure seamless communication between the NameNode and DataNodes.

```
docker compose up -d --build
docker exec -it hadoop-master bash

# Initialize Hadoop Services
/start-hadoop.sh
jps
```

Listing 1: Cluster Initialization Commands

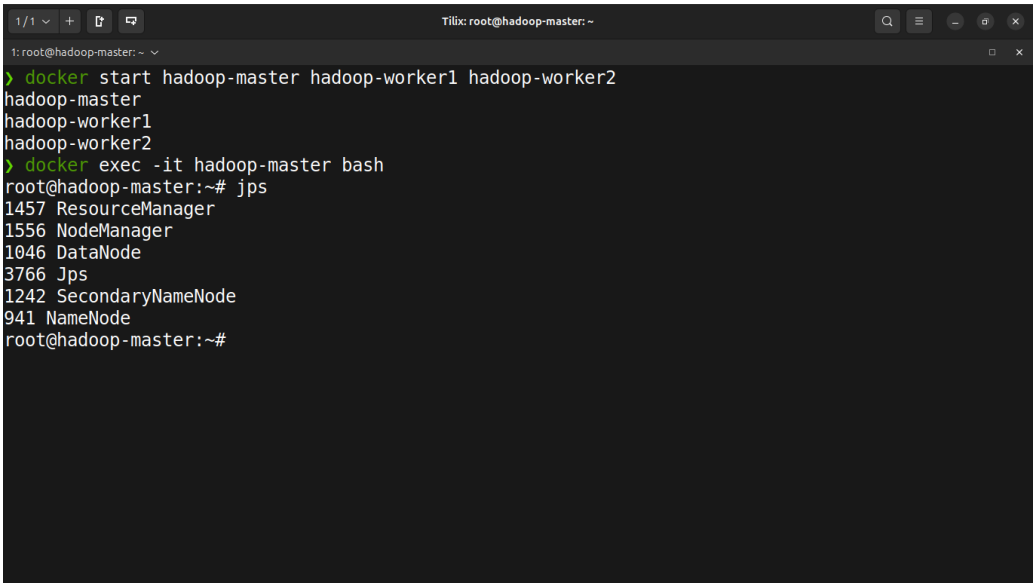
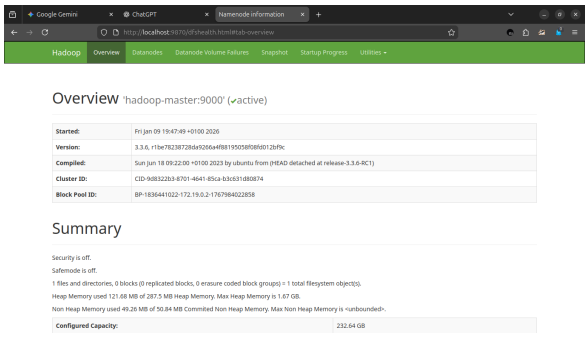


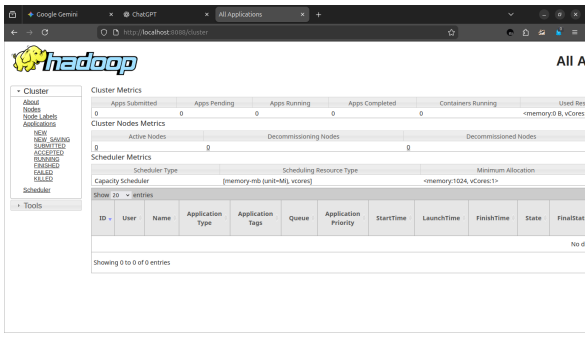
Figure 1: Cluster Startup: The `jps` command confirms that NameNode, DataNode, and ResourceManager are active.

2.2 Web Interfaces Verification

We verified the cluster health via the HDFS and YARN web interfaces to ensure resources were correctly allocated.



(a) HDFS Web UI (Port 9870)



(b) YARN Resource Manager (Port 8088)

Figure 2: Cluster Monitoring Dashboards

3 Phase 1: Interactive Processing (Spark Shell)

In this phase, we uploaded a text file to HDFS and performed a word count using Scala in the interactive Spark Shell.

3.1 Data Preparation

We created a sample file inside the container and uploaded it to the distributed file system.

```
hdfs dfs -rm -r -f file1.count
echo -e "Hello Spark Wordcount\nHello Hadoop Also" > file1.txt
hdfs dfs -mkdir -p /user/root
hdfs dfs -put file1.txt
```

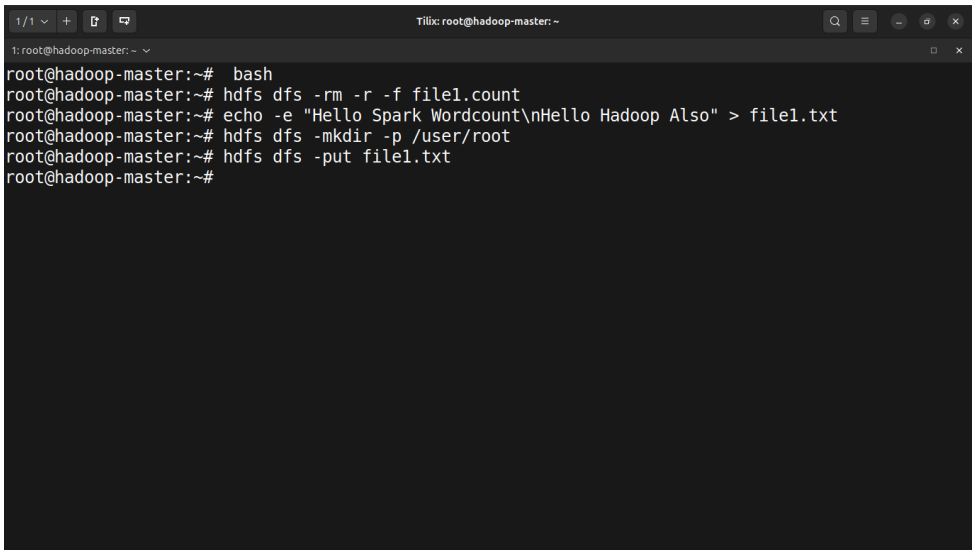


Figure 3: Uploading input data to HDFS.

3.2 Execution Results

We executed the MapReduce logic using Scala's functional API:

```
val lines = sc.textFile("file1.txt")
val wc = lines.flatMap(_.split("\\s+")).map(w => (w, 1)).reduceByKey(_ + _)
wc.saveAsTextFile("hdfs://hadoop-master:9000/user/root/file1.count")
```

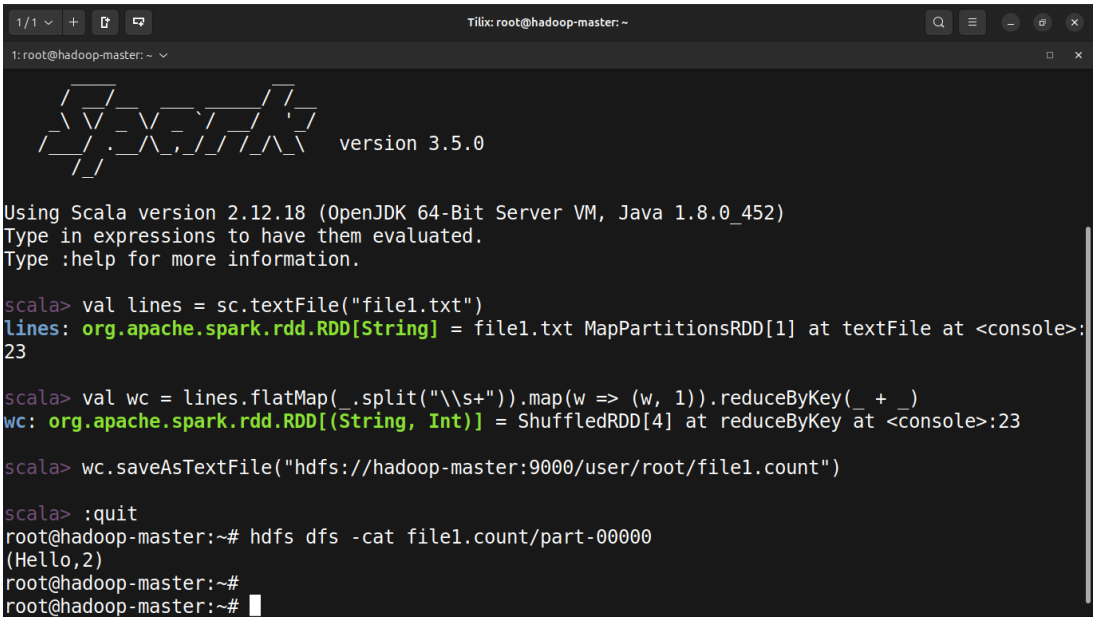


Figure 4: Phase 1 Result: Successful word count execution in Spark Shell.

4 Phase 2: Batch Processing (Java)

We developed a standalone Java application to process transaction data (Date, City, Item, Price) stored in HDFS.

4.1 Implementation

The `WordCountTask.java` reads tab-separated values and counts occurrences.

```
// Map Phase
JavaPairRDD<String, Integer> counts = textFile
    .flatMap(s -> Arrays.asList(s.split("\t")).iterator())
    .mapToPair(word -> new Tuple2<>(word, 1))
    .reduceByKey((a, b) -> a + b); // Reduce Phase
```

Listing 2: WordCountTask.java Snippet

4.2 Compilation and Submission

The code was compiled into a JAR and submitted to the Spark cluster using 'spark-submit'.

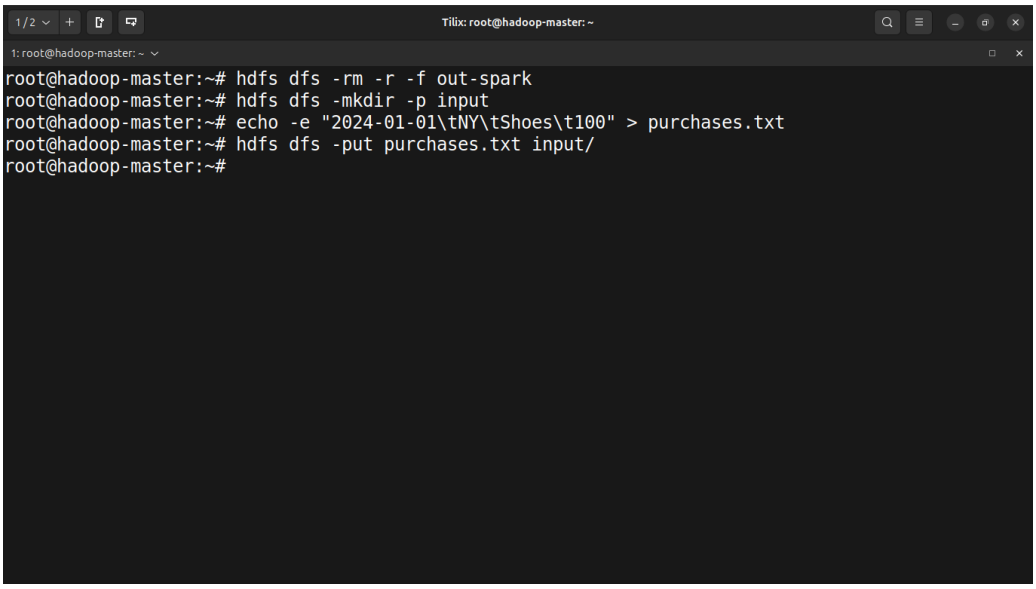


Figure 5: Preparing the purchases.txt input file.

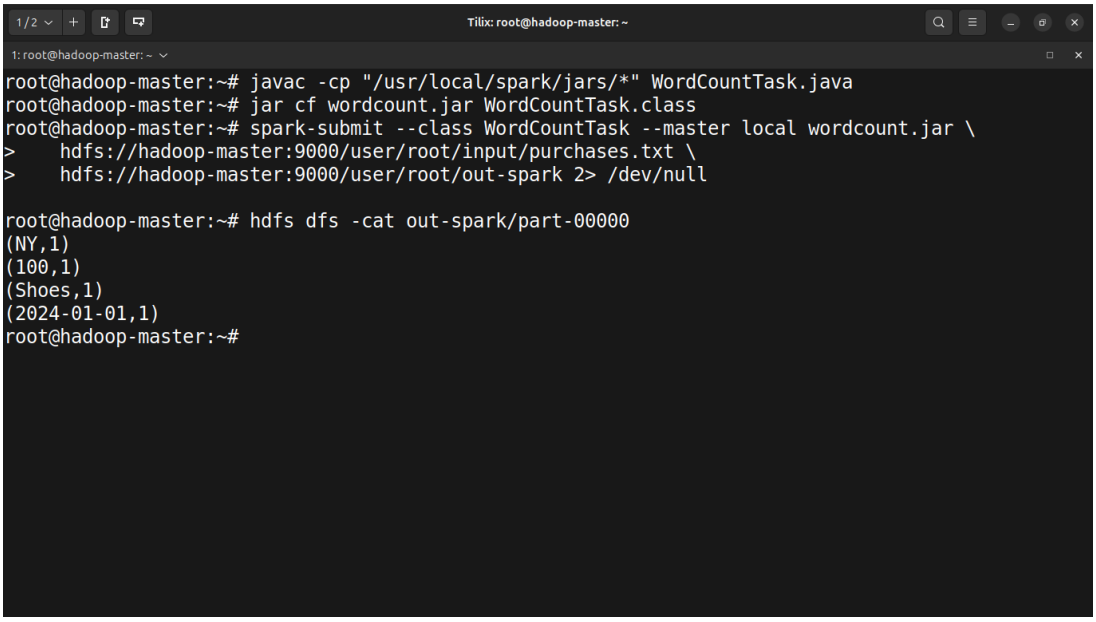


Figure 6: Phase 2 Result: The output in HDFS shows correct counts for the transaction data.

5 Phase 3: Real-Time Streaming

The final phase demonstrated Spark Structured Streaming by listening to a TCP socket on port 9999.

5.1 Streaming Configuration

The application reads from the socket and updates counts every second.

```
1 Dataset<Row> lines = spark.readStream()
```

```
    .format("socket")
    .option("host", "localhost")
    .option("port", 9999)
    .load();

// Output to Console
StreamingQuery query = wordCounts.writeStream()
    .outputMode("complete")
    .format("console")
    .trigger(Trigger.ProcessingTime("1 second"))
    .start();
```

Listing 3: Stream.java Configuration

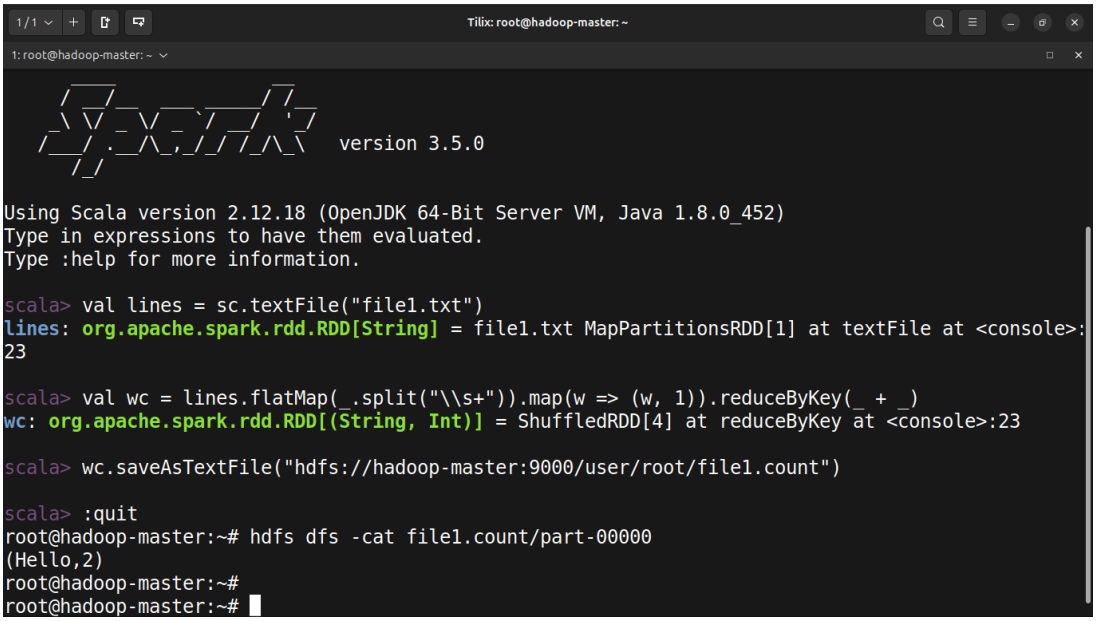


Figure 7: Phase 1 Result: Successful word count execution in Spark Shell.

5.2 Live Demonstration

We used `nc -lk 9999` to simulate a data stream. As words were typed into the terminal, Spark updated the count matrix in real-time.

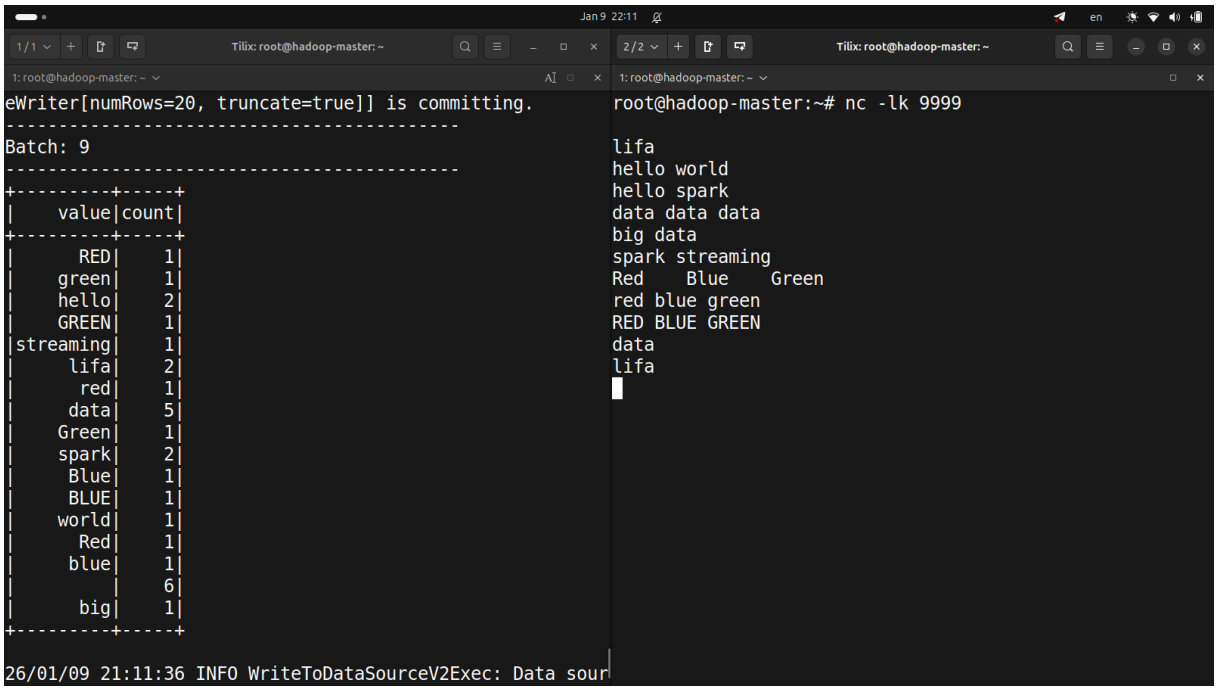


Figure 8: Real-Time Execution: Left terminal shows Spark outputting updated batches; Right terminal shows the user input stream.

6 Troubleshooting

During the lab, we encountered and resolved several issues:

Error	**Solution**
`FileAlreadyExistsException`	Delete existing directory: `hdfs dfs -rm -r <dir>`
Hadoop services fail to start	Remove faulty package: `apt remove pdsh`
Connection Refused (Port 9000)	Ensure NameNode is running via `jps`
Output missing in HDFS	Verify HDFS URI path in code

Table 1: Common Issues and Solutions

7 Conclusion

This lab successfully demonstrated the versatility of Apache Spark within a Dockerized Hadoop environment. We moved from simple interactive commands to compiled batch jobs and finally to real-time stream processing, verifying the results at every stage using HDFS and console outputs. The containerized approach allowed for a reproducible and isolated Big Data environment.