

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	2
2 Cluster Installation & Setup	2
2.1 Environment Initialization	2
2.2 Web Interfaces Verification	3
3 Phase 1: Interactive Processing (Spark Shell)	3
3.1 Data Preparation	3
3.2 Execution Results	4
4 Phase 2: Batch Processing (Java)	4
4.1 Implementation	4
4.2 Compilation and Submission	4
5 Phase 3: Real-Time Streaming	5
5.1 Streaming Configuration	5
5.2 Live Demonstration	6
6 Troubleshooting	7
7 Conclusion	7

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.

```

1 docker compose up -d --build
2 docker exec -it hadoop-master bash
3
4 # Initialize Hadoop Services
5 /start-hadoop.sh
6 jps

```

Listing 1: Cluster Initialization Commands

```

1:root@hadoop-master:~ ~
> docker start hadoop-master hadoop-worker1 hadoop-worker2
hadoop-master
hadoop-worker1
hadoop-worker2
> docker exec -it hadoop-master bash
root@hadoop-master:~# jps
1457 ResourceManager
1556 NodeManager
1046 DataNode
3766 Jps
1242 SecondaryNameNode
941 NameNode
root@hadoop-master:~#

```

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.

```

1 hdfs dfs -rm -r -f file1.count
2 echo -e "Hello Spark Wordcount\nHello Hadoop Also" > file1.txt
3 hdfs dfs -mkdir -p /user/root
4 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:

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

```
1/1 + Tilix: root@hadoop-master:~  
1: root@hadoop-master:~  
  
Using Scala version 2.12.18 (OpenJDK 64-Bit Server VM, Java 1.8.0_452)  
Type in expressions to have them evaluated.  
Type :help for more information.  
  
scala> val lines = sc.textFile("file1.txt")  
lines: org.apache.spark.rdd.RDD[String] = file1.txt MapPartitionsRDD[1] at textFile at <console>:23  
  
scala> val wc = lines.flatMap(_.split("\\s+")).map(w => (w, 1)).reduceByKey(_ + _)  
wc: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:23  
  
scala> wc.saveAsTextFile("hdfs://hadoop-master:9000/user/root/file1.count")  
  
scala> :quit  
root@hadoop-master:~# hdfs dfs -cat file1.count/part-00000  
(Hello,2)  
root@hadoop-master:~#  
root@hadoop-master:~#
```

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’.

```
1:root@hadoop-master:~ 
root@hadoop-master:~# hdfs dfs -rm -r -f out-spark
root@hadoop-master:~# hdfs dfs -mkdir -p input
root@hadoop-master:~# echo -e "2024-01-01\tNY\tShoes\t100" > purchases.txt
root@hadoop-master:~# hdfs dfs -put purchases.txt input/
root@hadoop-master:~#
```

Figure 5: Preparing the purchases.txt input file.

```
1/2 + 🔍 Tilix: root@hadoop-master: ~
1: root@hadoop-master:~ # javac -cp "/usr/local/spark/jars/*" WordCountTask.java
root@hadoop-master:~ # jar cf wordcount.jar WordCountTask.class
root@hadoop-master:~ # spark-submit --class WordCountTask --master local wordcount.jar \
>      hdfs://hadoop-master:9000/user/root/input/purchases.txt \
>      hdfs://hadoop-master:9000/user/root/out-spark 2> /dev/null

root@hadoop-master:~ # hdfs dfs -cat out-spark/part-00000
(NY,1)
(100,1)
(Shoes,1)
(2024-01-01,1)
root@hadoop-master:~ #
```

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()
```

```
2     .format("socket")
3     .option("host", "localhost")
4     .option("port", 9999)
5     .load();
6
7 // Output to Console
8 StreamingQuery query = wordCounts.writeStream()
9     .outputMode("complete")
10    .format("console")
11    .trigger(Trigger.ProcessingTime("1 second"))
12    .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.

```

eWriter[numRows=20, truncate=true]] is committing.
Batch: 9
+-----+-----+
| value|count|
+-----+-----+
|    RED|    1|
|   green|    1|
|   hello|    2|
|   GREEN|    1|
|streaming|    1|
|    lifa|    2|
|     red|    1|
|    data|    5|
|   Green|    1|
|   spark|    2|
|    Blue|    1|
|    BLUE|    1|
|  world|    1|
|    Red|    1|
|    blue|    1|
|      |    6|
|    big|    1|
+-----+-----+
26/01/09 21:11:36 INFO WriteToDataSourceV2Exec: Data sour

```

```

root@hadoop-master:~# nc -lk 9999
lifa
hello world
hello spark
data data data
big data
spark streaming
Red Blue Green
red blue green
RED BLUE GREEN
data
lifa

```

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
<code>FileNotFoundException</code>	Delete existing directory: <code>hdfs dfs -rm -r <dir></code>
Hadoop services fail to start	Remove faulty package: <code>apt remove pdsh</code>
Connection Refused (Port 9000)	Ensure NameNode is running via <code>jps</code>
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.