

VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



SEMESTER 241 - CO3071

LAB 4 REPORT
DISTRIBUTED SYSTEMS

Instructor: PROF. Nguyễn Mạnh Thìn

Tạ Gia Khang	- 2152642
Nguyễn Quang Thiện	- 2152994

HO CHI MINH CITY, DECEMBER 2024



Contents

1	Homework	3
1.1	Configuration	3
1.2	Previous Topics Data	7
1.3	Configuring PySpark Session	8
1.4	Streaming Data Reader for Kafka Fog Node	10
1.5	Processing Data	11
1.6	Joining Data Streams	12
1.7	Storing Output to HDFS	13
1.8	Demonstration	15

1 Homework

1.1 Configuration

Using Docker and generating a docker-compose.yml file to create 3 brokers for **Kafka Fog**, **Spark Master**, **2 Spark Workers**, a **Hadoop Namenode**, and **3 Datanodes**.

```
1  version: '2'
2  services:
3    # kafka-fog
4    kafka1:
5      image: apache/kafka:3.7.0
6      hostname: kafka1
7      container_name: kafka1
8      ports:
9        - 29092:9092
10     environment:
11       KAFKA_NODE_ID: 1
12       KAFKA_PROCESS_ROLES: 'broker,controller'
13       KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
14         'CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT'
15       KAFKA_CONTROLLER_QUORUM_VOTERS: '1@kafka1:9093,2@kafka2:9093,3@kafka3:9093'
16       KAFKA_LISTENERS:
17         'PLAINTEXT://:29092,CONTROLLER://:9093,PLAINTEXT_HOST://:9092'
18       KAFKA_INTER_BROKER_LISTENER_NAME: 'PLAINTEXT'
19       KAFKA_ADVERTISED_LISTENERS:
20         PLAINTEXT://kafka1:29092,PLAINTEXT_HOST://localhost:29092
21       KAFKA_CONTROLLER_LISTENER_NAMES: 'CONTROLLER'
22       CLUSTER_ID: '2UKPPoqNTPezeassrAogQw'
23       KAFKA_OFFSETS_TOPIC_NUM_PARTITIONS: 1
24       KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
25       KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
26       KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
27       KAFKA_LOG_DIRS: '/tmp/kraft-combined-logs'
28     networks:
29       - lab4_ds_kafka_network
30
31   kafka2:
32     image: apache/kafka:3.7.0
33     hostname: kafka2
34     container_name: kafka2
35     ports:
36       - 39092:9092
37     environment:
38       KAFKA_NODE_ID: 2
39       KAFKA_PROCESS_ROLES: 'broker,controller'
```

```
38 KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
39 'CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT'
40 KAFKA_CONTROLLER_QUORUM_VOTERS: '1@kafka1:9093,2@kafka2:9093,3@kafka3:9093'
41 KAFKA_LISTENERS:
42 'PLAINTEXT://:39092,CONTROLLER://:9093,PLAINTEXT_HOST://:9092'
43 KAFKA_INTER_BROKER_LISTENER_NAME: 'PLAINTEXT'
44 KAFKA_ADVERTISED_LISTENERS:
45 PLAINTEXT://kafka2:39092,PLAINTEXT_HOST://localhost:39092
46 KAFKA_CONTROLLER_LISTENER_NAMES: 'CONTROLLER'
47 CLUSTER_ID: '2UKPPoqNTPezeassrAogQw'
48 KAFKA_OFFSETS_TOPIC_NUM_PARTITIONS: 1
49 KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
50 KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
51 KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
52 KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
53 KAFKA_LOG_DIRS: '/tmp/kraft-combined-logs'
54 networks:
55 - lab4_ds_kafka_network
56
57 kafka3:
58 image: apache/kafka:3.7.0
59 hostname: kafka3
60 container_name: kafka3
61 ports:
62 - 49092:9092
63 environment:
64 KAFKA_NODE_ID: 3
65 KAFKA_PROCESS_ROLES: 'broker,controller'
66 KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
67 'CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT'
68 KAFKA_CONTROLLER_QUORUM_VOTERS: '1@kafka1:9093,2@kafka2:9093,3@kafka3:9093'
69 KAFKA_LISTENERS:
70 'PLAINTEXT://:49092,CONTROLLER://:9093,PLAINTEXT_HOST://:9092'
71 KAFKA_INTER_BROKER_LISTENER_NAME: 'PLAINTEXT'
72 KAFKA_ADVERTISED_LISTENERS:
73 PLAINTEXT://kafka3:49092,PLAINTEXT_HOST://localhost:49092
74 KAFKA_CONTROLLER_LISTENER_NAMES: 'CONTROLLER'
75 CLUSTER_ID: '2UKPPoqNTPezeassrAogQw'
76 KAFKA_OFFSETS_TOPIC_NUM_PARTITIONS: 1
77 KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
78 KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
79 KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
80 KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
81 KAFKA_LOG_DIRS: '/tmp/kraft-combined-logs'
82 networks:
83 - lab4_ds_kafka_network
```

```
79 kafka-ui:
80     container_name: kafka-ui
81     image: provectuslabs/kafka-ui:latest
82     ports:
83         - "8080:8080"
84     environment:
85         DYNAMIC_CONFIG_ENABLED: 'true'
86     depends_on:
87         - kafka1
88         - kafka2
89         - kafka3
90     networks:
91         - lab4_ds_kafka_network
92
93 spark-master:
94     image: bitnami/spark:latest
95     container_name: spark-master
96     environment:
97         - SPARK_MODE=master
98         - SPARK_RPC_AUTHENTICATION_ENABLED=no
99         - SPARK_RPC_ENCRYPTION_ENABLED=no
100        - SPARK_LOCAL_STORAGE_ENCRYPTION_ENABLED=no
101        - SPARK_SSL_ENABLED=no
102     ports:
103         - "7077:7077" # Spark Master port
104     networks:
105         - lab4_ds_kafka_network
106     volumes:
107         - ./pyspark:/opt/spark/python
108
109 spark-worker1:
110     image: bitnami/spark:latest
111     container_name: spark-worker1
112     environment:
113         - SPARK_MODE=worker
114         - SPARK_MASTER_URL=spark://spark-master:7077
115     depends_on:
116         - spark-master
117     networks:
118         - lab4_ds_kafka_network
119
120 spark-worker2:
121     image: bitnami/spark:latest
122     container_name: spark-worker2
123     environment:
124         - SPARK_MODE=worker
125         - SPARK_MASTER_URL=spark://spark-master:7077
```

```
126     depends_on:
127         - spark-master
128     networks:
129         - lab4_ds_kafka_network
130
131     namenode:
132         image: bde2020/hadoop-namenode:latest
133         container_name: namenode
134         environment:
135             - CLUSTER_NAME=hadoop_cluster
136             - CORE_CONF_fs_defaultFS=hdfs://namenode:9000
137         ports:
138             - "9000:9000" # HDFS namenode RPC port
139             - "9870:9870" # HDFS namenode Web UI
140         networks:
141             - lab4_ds_kafka_network
142         volumes:
143             - ./namenode:/hadoop/dfs/name
144
145     datanode:
146         image: bde2020/hadoop-datanode:latest
147         container_name: datanode
148         environment:
149             - CORE_CONF_fs_defaultFS=hdfs://namenode:9000
150         depends_on:
151             - namenode
152         networks:
153             - lab4_ds_kafka_network
154         volumes:
155             - ./datanode:/hadoop/dfs/data
156
157     datanode2:
158         image: bde2020/hadoop-datanode:latest
159         container_name: datanode2
160         environment:
161             - CORE_CONF_fs_defaultFS=hdfs://namenode:9000
162         depends_on:
163             - namenode
164         networks:
165             - lab4_ds_kafka_network
166         volumes:
167             - ./datanode2:/hadoop/dfs/data
168
169     datanode3:
170         image: bde2020/hadoop-datanode:latest
171         container_name: datanode3
172         environment:
```

```

173     - CORE_CONF_fs_defaultFS=hdfs://namenode:9000
174     depends_on:
175     - namenode
176     networks:
177     - lab4_ds_kafka_network
178     volumes:
179     - ./datanode3:/hadoop/dfs/data
180
181     networks:
182     lab4_ds_kafka_network:
183     external: true
184
185     volumes:
186     # Defines volumes for persistent storage of Namenode and Datanode data
187     namenode:
188     datanode:
189     datanode2:
190     datanode3:

```

After that, create the network **lab4_ds_kafka_network** by the cmd below in Docker:

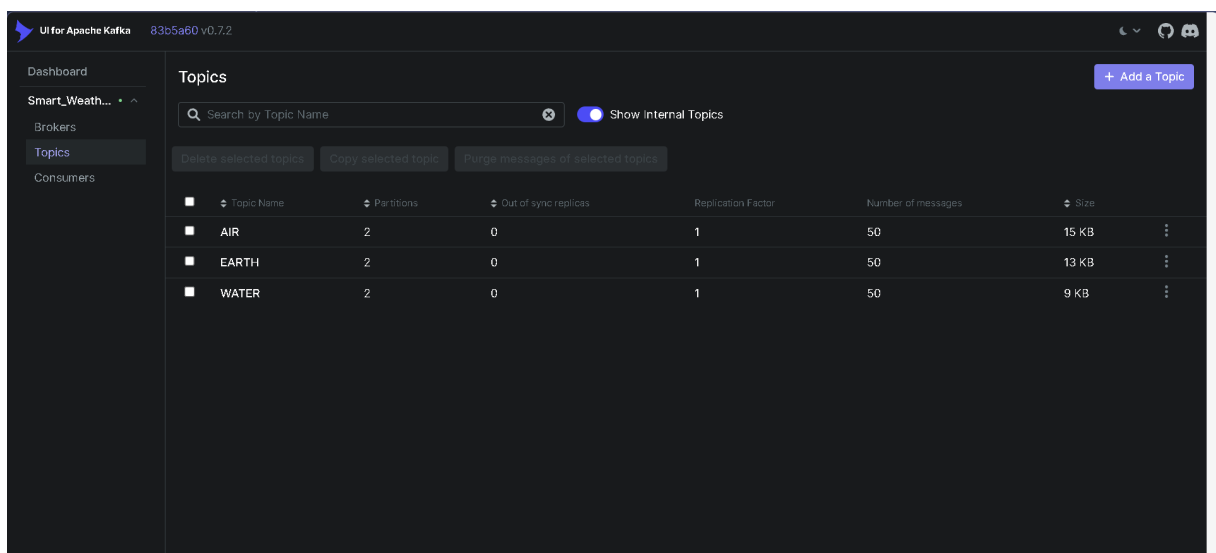
```

1 docker network create lab4_ds_kafka_network

```

1.2 Previous Topics Data

Already have 3 brokers serving **Kafka Fog** with data of 3 topics: **AIR**, **EARTH**, **WATER**



Topic Name	Partitions	Out of sync replicas	Replication Factor	Number of messages	Size
AIR	2	0	1	50	15 KB
EARTH	2	0	1	50	13 KB
WATER	2	0	1	50	9 KB

Figure 1: Data of Topics: AIR, EARTH, WATER

The mission of this lab is to utilize **Spark** to process the streaming data from these topics in real-time and store the results in **HDFS**.

1.3 Configuring PySpark Session

Initialize **SparkSession** using **PySpark** to process streaming data efficiently, serving as the entry point for Spark operations. The session is configured to interact with the Hadoop Distributed File System (HDFS) via the namenode at "**hdfs://namenode:9000**" and enables distributed processing of large environmental datasets.

```
1 from pyspark.sql import SparkSession
2 from pyspark.sql.types import StructType, StructField, StringType, DoubleType,
  IntegerType
3 from pyspark.sql.functions import col, from_json, to_json, struct, when, lit,
  mean, to_timestamp
4
5 from pyspark.sql import functions as F
6 from pyspark.sql.window import Window
7 import random
8
9 # Initialize SparkSession
10 spark = SparkSession.builder \
11     .appName("Environment Data Processing") \
12     .config("spark.hadoop.fs.defaultFS", "hdfs://namenode:9000") \
13     .getOrCreate()
```

Schemas for the data streams are predefined to ensure proper parsing and validation during data processing. Each schema defines the expected structure and types for the incoming data.

```
1 # Define schemas for AIR, EARTH, and WATER data
2 air_schema = StructType([
3     StructField("Time", StringType(), True),
4     StructField("Station", StringType(), True),
5     StructField("Temperature", DoubleType(), True),
6     StructField("Moisture", DoubleType(), True),
7     StructField("Light", DoubleType(), True),
8     StructField("Total_Rainfall", DoubleType(), True),
9     StructField("Rainfall", DoubleType(), True),
10    StructField("Wind_Direction", IntegerType(), True),
11    StructField("PM2.5", DoubleType(), True),
12    StructField("PM10", DoubleType(), True),
13    StructField("CO", DoubleType(), True),
14    StructField("NOx", DoubleType(), True),
15    StructField("SO2", DoubleType(), True)
16 ])
17
18 earth_schema = StructType([
19     StructField("Time", StringType(), True),
20     StructField("Station", StringType(), True),
21     StructField("Moisture", DoubleType(), True),
```



```
22     StructField("Temperature", DoubleType(), True),
23     StructField("Salinity", DoubleType(), True),
24     StructField("pH", DoubleType(), True),
25     StructField("Water_Root", DoubleType(), True),
26     StructField("Water_Leaf", DoubleType(), True),
27     StructField("Water_Level", DoubleType(), True),
28     StructField("Voltage", DoubleType(), True)
29 ])
```

```
30
31 water_schema = StructType([
32     StructField("Time", StringType(), True),
33     StructField("Station", StringType(), True),
34     StructField("pH", DoubleType(), True),
35     StructField("DO", DoubleType(), True),
36     StructField("Temperature", DoubleType(), True),
37     StructField("Salinity", DoubleType(), True)
38 ])
```

1.4 Streaming Data Reader for Kafka Fog Node

The streaming data is read from three Kafka topics: AIR, EARTH, and WATER. Using Spark Structured Streaming connect to the Kafka brokers ("kafka1:29092", "kafka2:39092", "kafka3:49092") and parse the incoming JSON messages into structured data based on the predefined schemas.

```
1  # Read streams from Kafka
2  air_stream = spark.readStream \
3      .format("kafka") \
4      .option("kafka.bootstrap.servers", "kafka1:29092") \
5      .option("subscribe", "AIR") \
6      .option("failOnDataLoss", "false") \
7      .option("startingOffsets", "earliest") \
8      .load()
9
10 earth_stream = spark.readStream \
11     .format("kafka") \
12     .option("kafka.bootstrap.servers", "kafka2:39092") \
13     .option("subscribe", "EARTH") \
14     .option("failOnDataLoss", "false") \
15     .option("startingOffsets", "earliest") \
16     .load()
17
18 water_stream = spark.readStream \
19     .format("kafka") \
20     .option("kafka.bootstrap.servers", "kafka3:49092") \
21     .option("subscribe", "WATER") \
22     .option("failOnDataLoss", "false") \
23     .option("startingOffsets", "earliest") \
24     .load()
25
26 # Parse JSON msgs into structured data with Time column cast to Timestamp
27 air_data = air_stream.select(from_json(col("value").cast("string"),
28     air_schema).alias("data")).select(col("data.*"),)
29
30 earth_data = earth_stream.select(from_json(col("value").cast("string"),
31     earth_schema).alias("data")).select(col("data.*"),)
32
33 water_data = water_stream.select(from_json(col("value").cast("string"),
34     water_schema).alias("data")).select(col("data.*"),)
```

1.5 Processing Data

A compute statistics function is implemented to calculate incremental statistics (mean and standard deviation) for specific columns in the data streams. The function uses Spark's Window specification to calculate running statistics over time.

```
1 def compute_statistics(df, columns, topic):
2     # Define the window specification to calculate running average and standard
3     deviation
4     window_spec =
5     Window.orderBy("timestamp").rowsBetween(Window.unboundedPreceding,
6     Window.currentRow)
7
8     # Create a list of expressions to calculate incremental average and stddev for
9     each column
10    select_exprs = []
11    for column in columns:
12        # Calculate running average (mean) and standard deviation
13        avg_col = F.avg(F.col(column)).over(window_spec).alias(f"{column}_avg")
14        stddev_col =
15        F.stddev(F.col(column)).over(window_spec).alias(f"{column}_stddev")
16
17        # Apply the formula: avg(v_i_0_n1) + rand(-std_i_0_n1, std_i_0_n1)
18        # rand(-std, std)
19        random_component = F.rand() * (stddev_col) * 2 - stddev_col
20        new_value = avg_col + random_component
21
22        select_exprs.append(new_value.alias(f"`{column}_calculated`"))
23
24    # Perform the transformation and return the stream
25    return df.select(select_exprs) \
26        .writeStream \
27        .outputMode("complete") \
28        .format("memory") \
29        .queryName(f"stats_table_{topic}") \
30        .start()
31
32    # Start calculating statistics for each stream
33    air_columns = ["Temperature", "Moisture", "Light", "Total_Rainfall", "Rainfall",
34    "`PM2.5`", "PM10", "CO", "NOx", "SO2"]
35    earth_columns = ["Moisture", "Temperature", "Salinity", "pH", "Water_Root",
36    "Water_Leaf", "Water_Level", "Voltage"]
37    water_columns = ["pH", "DO", "Temperature", "Salinity"]
38
39    air_stats = compute_statistics(air_data, air_columns, "air")
40    earth_stats = compute_statistics(earth_data, earth_columns, "earth")
41    water_stats = compute_statistics(water_data, water_columns, "water")
```

1.6 Joining Data Streams

Performs an inner join on three streams of environmental data (*air_data*, *earth_data*, *water_data*) based on the **Time** column, creating a unified dataset called *environment_data* that combines corresponding data from all three sources. The joined data is written to the **console** format in real-time using **writeStream** and **append** output mode, displaying newly appended data every minute to ensure accurate and seamless data integration.

```
1  # Join streams
2  environment_data = air_data.alias("air").join(
3      earth_data.alias("earth"), ["Time"], "inner"
4  ).join(
5      water_data.alias("water"), ["Time"], "inner"
6  )
7
8  # Debugging print joining result
9  environment_data.writeStream \
10     .format("console") \
11     .outputMode("append") \
12     .trigger(processingTime="1 minute") \
13     .start()
```

1.7 Storing Output to HDFS

The combined dataset (*environment_data*) is transformed into **json** format, organizing it into three primary sections: *Air*, *Earth*, and *Water*, each containing relevant fields from their respective streams. The processed data is then stored in HDFS under the directory `/hdfs/environment_data/` in JSON format. Checkpoint directory at `/hdfs/checkpoints/environment_data/` is utilized to monitor the progress of the stream to ensure fault tolerance and enable recovery. The system processes data every minute using `writeStream` and `append` output mode and continues streaming until explicitly terminated via `awaitAnyTermination`.

```
1  # Convert the joined data to JSON
2  environment_json = environment_data.select(
3      to_json(
4          struct(
5              col("Time"),
6              struct(
7                  col("`air`.`Station`"),
8                  col("`air`.`Temperature`"),
9                  col("`air`.`Moisture`"),
10                 col("Light"),
11                 col("Total_Rainfall"),
12                 col("Rainfall"),
13                 col("Wind_Direction"),
14                 col("`PM2.5`"),
15                 col("PM10"),
16                 col("CO"),
17                 col("NOx"),
18                 col("SO2")
19             ).alias("Air"),
20             struct(
21                 col("`earth`.`Station`"),
22                 col("`earth`.`Moisture`"),
23                 col("`earth`.`Temperature`"),
24                 col("`earth`.`Salinity`"),
25                 col("`earth`.`pH`"),
26                 col("Water_Root"),
27                 col("Water_Leaf"),
28                 col("Water_Level"),
29                 col("Voltage")
30             ).alias("Earth"),
31             struct(
32                 col("`water`.`Station`"),
33                 col("`water`.`pH`"),
34                 col("DO"),
35                 col("`water`.`Temperature`"),
36                 col("`water`.`Salinity`")
37             ).alias("Water")
```

```
38     )
39     ).alias("value")
40 )
41
42 # Write final stream to HDFS
43 query = environment_json.writeStream \
44     .format("json") \
45     .option("path", "/hdfs/environment_data/") \
46     .option("checkpointLocation", "/hdfs/checkpoints/environment_data/") \
47     .outputMode("append") \
48     .trigger(processingTime="1 minute") \
49     .start()
50
51 # Wait for termination
52 spark.streams.awaitAnyTermination()
```

1.8 Demonstration

```

1 Prepare HDFS:
2 docker exec -it namenode bash -c 'hdfs dfs -mkdir /hdfs'
3 docker exec -it namenode bash -c 'hdfs dfs -mkdir /hdfs/
  environment_data'
4 docker exec -it namenode bash -c 'hdfs dfs -mkdir /hdfs/checkpoints/'
5 docker exec -it namenode bash -c 'hdfs dfs -mkdir /hdfs/checkpoints/
  environment_data/'
6 docker exec -it namenode bash -c 'hdfs dfs -chown -R spark:supergroup
  /hdfs'
7 Push code to Spark:
8 docker cp spark_process.py spark-master:/opt/spark/python/
  spark_process.py
9 Run code:
10 docker exec -it spark-master bash
11 spark-submit \
12 --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.3 \
13 --master spark://spark-master:7077 \
14 /opt/spark/python/spark_process.py

```

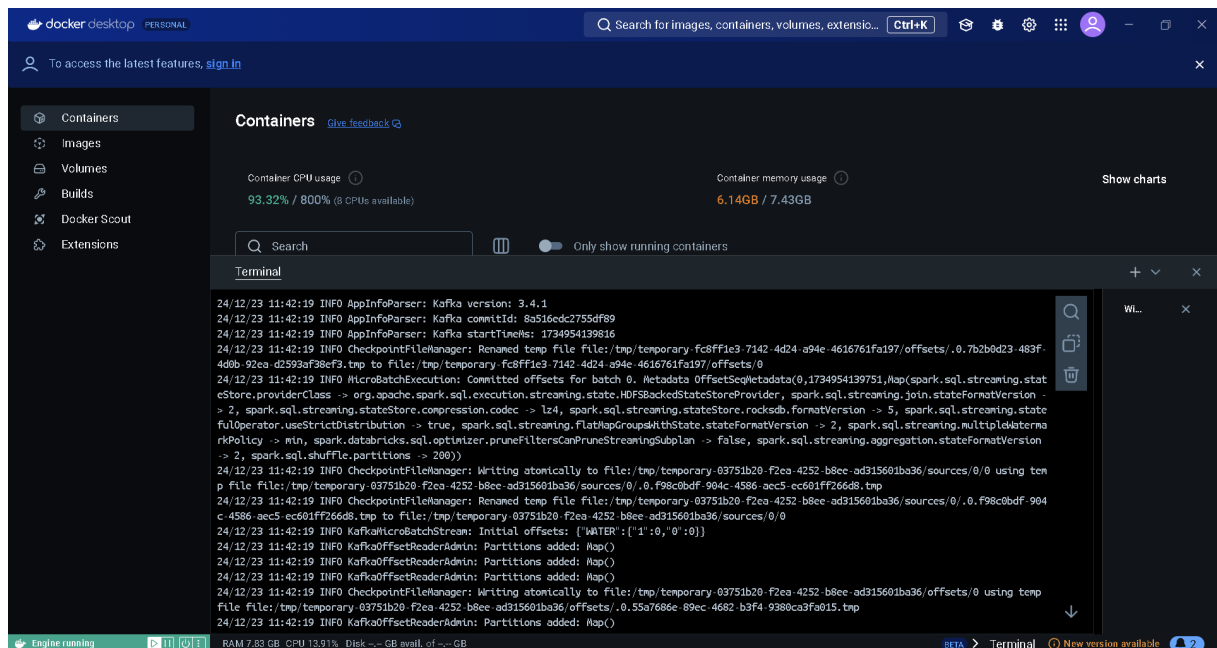


Figure 2: Run integration

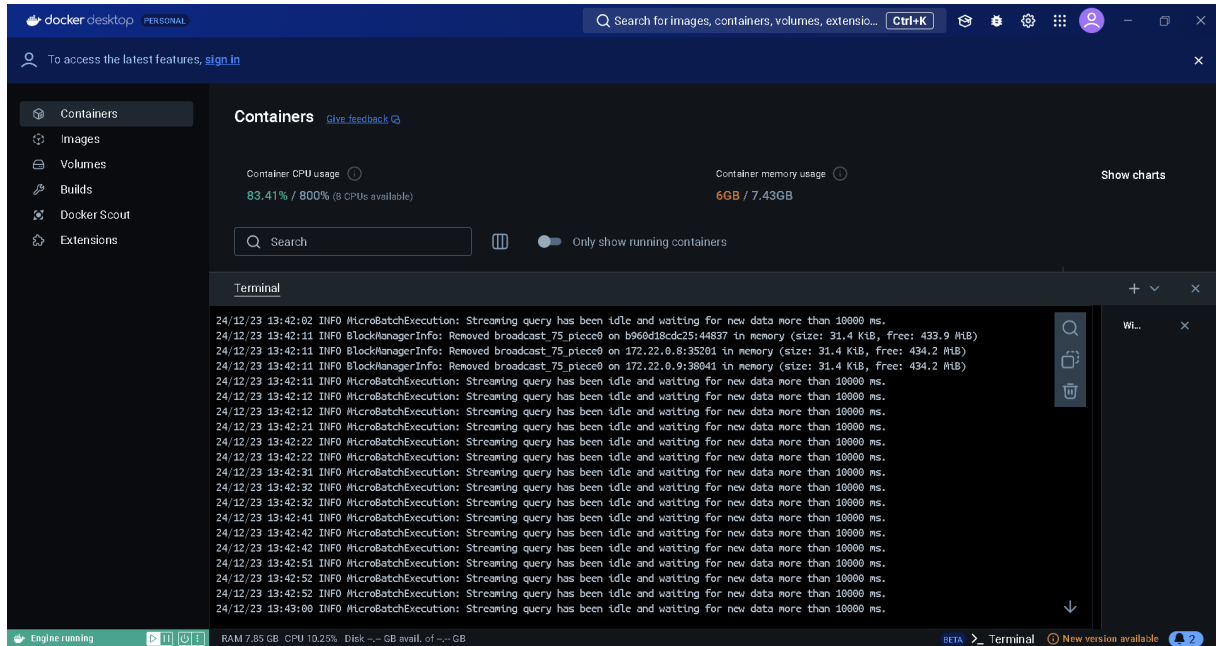


Figure 3: Finish integration

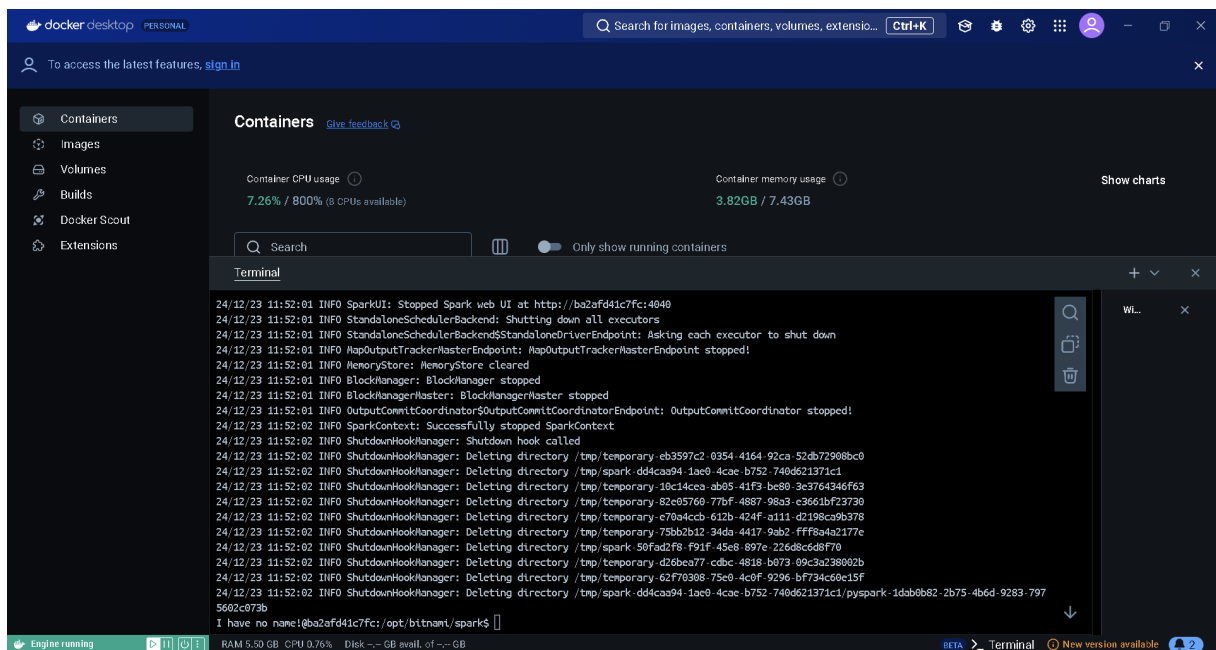


Figure 4: Terminate the execution

1 Check Output:

2 `docker exec -it namenode bash -c 'hdfs dfs -cat /hdfs/environment_data /*.json'`

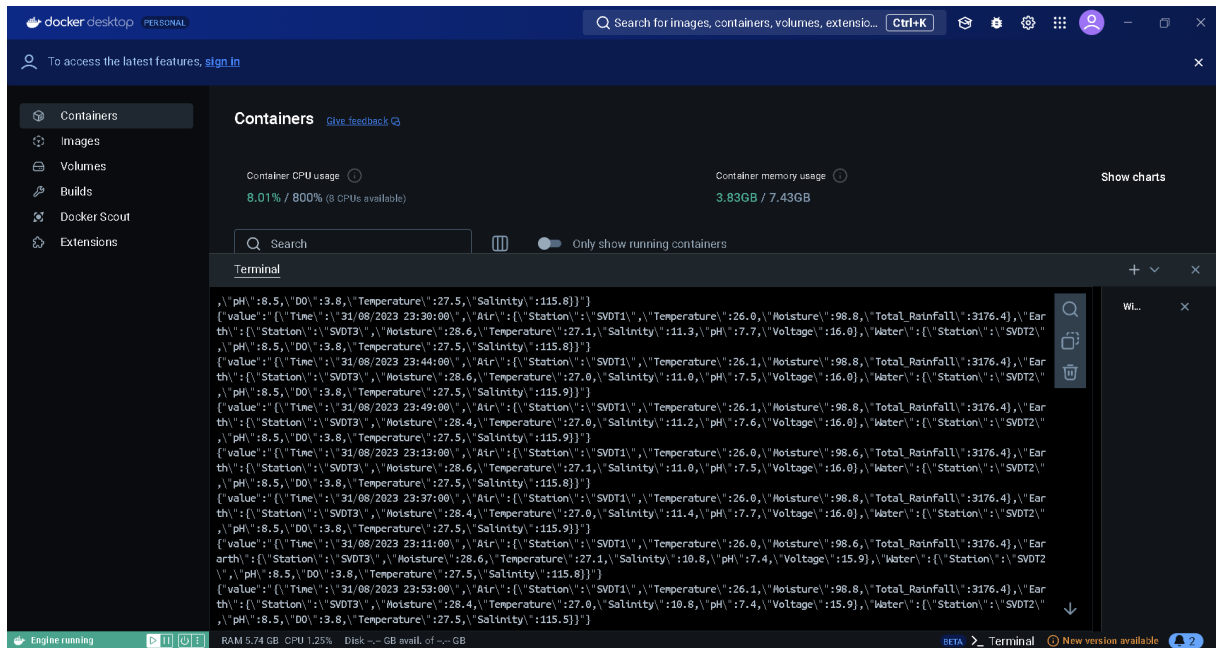


Figure 5: Output