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

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**.

```yaml
version: '2'
services:
# kafka-fog
  kafka1:
    image: apache/kafka:3.7.0
    hostname: kafka1
    container_name: kafka1
    ports:
      - 29092:9092
    environment:
      KAFKA_NODE_ID: 1
      KAFKA_PROCESS_ROLES: 'broker,controller'
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
      'CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT'
      KAFKA_CONTROLLER_QUORUM_VOTERS: '1@kafka1:9093,2@kafka2:9093,3@kafka3:9093'
      KAFKA_LISTENERS:
      'PLAINTEXT://:29092,CONTROLLER://:9093,PLAINTEXT_HOST://:9092'
      KAFKA_INTER_BROKER_LISTENER_NAME: 'PLAINTEXT'
      KAFKA_ADVERTISED_LISTENERS:
      PLAINTEXT://kafka1:29092,PLAINTEXT_HOST://localhost:29092
      KAFKA_CONTROLLER_LISTENER_NAMES: 'CONTROLLER'
      CLUSTER_ID: '2UKPPoqNTPezeassrAogQw'
      KAFKA_OFFSETS_TOPIC_NUM_PARTITIONS: 1
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
      KAFKA_LOG_DIRS: '/tmp/kraft-combined-logs'
    networks:
      - lab4_ds_kafka_network

  kafka2:
    image: apache/kafka:3.7.0
    hostname: kafka2
    container_name: kafka2
    ports:
      - 39092:9092
    environment:
      KAFKA_NODE_ID: 2
      KAFKA_PROCESS_ROLES: 'broker,controller'
```

```yaml
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
      'CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT'
      KAFKA_CONTROLLER_QUORUM_VOTERS: '1@kafka1:9093,2@kafka2:9093,3@kafka3:9093'
      KAFKA_LISTENERS:
      'PLAINTEXT://:39092,CONTROLLER://:9093,PLAINTEXT_HOST://:9092'
      KAFKA_INTER_BROKER_LISTENER_NAME: 'PLAINTEXT'
      KAFKA_ADVERTISED_LISTENERS:
      PLAINTEXT://kafka2:39092,PLAINTEXT_HOST://localhost:39092
      KAFKA_CONTROLLER_LISTENER_NAMES: 'CONTROLLER'
      CLUSTER_ID: '2UKPPoqNTPezeassrAogQw'
      KAFKA_OFFSETS_TOPIC_NUM_PARTITIONS: 1
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
      KAFKA_LOG_DIRS: '/tmp/kraft-combined-logs'
    networks:
      - lab4_ds_kafka_network

  kafka3:
    image: apache/kafka:3.7.0
    hostname: kafka3
    container_name: kafka3
    ports:
      - 49092:9092
    environment:
      KAFKA_NODE_ID: 3
      KAFKA_PROCESS_ROLES: 'broker,controller'
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
      'CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT'
      KAFKA_CONTROLLER_QUORUM_VOTERS: '1@kafka1:9093,2@kafka2:9093,3@kafka3:9093'
      KAFKA_LISTENERS:
      'PLAINTEXT://:49092,CONTROLLER://:9093,PLAINTEXT_HOST://:9092'
      KAFKA_INTER_BROKER_LISTENER_NAME: 'PLAINTEXT'
      KAFKA_ADVERTISED_LISTENERS:
      PLAINTEXT://kafka3:49092,PLAINTEXT_HOST://localhost:49092
      KAFKA_CONTROLLER_LISTENER_NAMES: 'CONTROLLER'
      CLUSTER_ID: '2UKPPoqNTPezeassrAogQw'
      KAFKA_OFFSETS_TOPIC_NUM_PARTITIONS: 1
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
      KAFKA_LOG_DIRS: '/tmp/kraft-combined-logs'
    networks:
      - lab4_ds_kafka_network

```

```yaml
  kafka-ui:
    container_name: kafka-ui
    image: provectuslabs/kafka-ui:latest
    ports:
      - "8080:8080"
    environment:
      DYNAMIC_CONFIG_ENABLED: 'true'
    depends_on:
      - kafka1
      - kafka2
      - kafka3
    networks:
      - lab4_ds_kafka_network

  spark-master:
    image: bitnami/spark:latest
    container_name: spark-master
    environment:
      - SPARK_MODE=master
      - SPARK_RPC_AUTHENTICATION_ENABLED=no
      - SPARK_RPC_ENCRYPTION_ENABLED=no
      - SPARK_LOCAL_STORAGE_ENCRYPTION_ENABLED=no
      - SPARK_SSL_ENABLED=no
    ports:
      - "7077:7077" # Spark Master port
    networks:
      -  lab4_ds_kafka_network
    volumes:
      - ./pyspark:/opt/spark/python

  spark-worker1:
    image: bitnami/spark:latest
    container_name: spark-worker1
    environment:
      - SPARK_MODE=worker
      - SPARK_MASTER_URL=spark://spark-master:7077
    depends_on:
      - spark-master
    networks:
      -  lab4_ds_kafka_network

  spark-worker2:
    image: bitnami/spark:latest
    container_name: spark-worker2
    environment:
      - SPARK_MODE=worker
      - SPARK_MASTER_URL=spark://spark-master:7077
```

```
    depends_on:
      - spark-master
    networks:
      -  lab4_ds_kafka_network

  namenode:
    image: bde2020/hadoop-namenode:latest
    container_name: namenode
    environment:
      - CLUSTER_NAME=hadoop_cluster
      - CORE_CONF_fs_defaultFS=hdfs://namenode:9000
    ports:
      - "9000:9000" # HDFS namenode RPC port
      - "9870:9870" # HDFS namenode Web UI
    networks:
      -  lab4_ds_kafka_network
    volumes:
      - ./namenode:/hadoop/dfs/name

  datanode:
    image: bde2020/hadoop-datanode:latest
    container_name: datanode
    environment:
      - CORE_CONF_fs_defaultFS=hdfs://namenode:9000
    depends_on:
      - namenode
    networks:
      -  lab4_ds_kafka_network
    volumes:
      - ./datanode:/hadoop/dfs/data

  datanode2:
    image: bde2020/hadoop-datanode:latest
    container_name: datanode2
    environment:
      - CORE_CONF_fs_defaultFS=hdfs://namenode:9000
    depends_on:
      - namenode
    networks:
      -  lab4_ds_kafka_network
    volumes:
      - ./datanode2:/hadoop/dfs/data

  datanode3:
    image: bde2020/hadoop-datanode:latest
    container_name: datanode3
    environment:
```

```
      - CORE_CONF_fs_defaultFS=hdfs://namenode:9000
    depends_on:
      - namenode
    networks:
      -  lab4_ds_kafka_network
    volumes:
      - ./datanode3:/hadoop/dfs/data

networks:
  lab4_ds_kafka_network:
    external: true

volumes:
  # Defines volumes for persistent storage of Namenode and Datanode data
  namenode:
  datanode:
  datanode2:
  datanode3:
```

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

```
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**

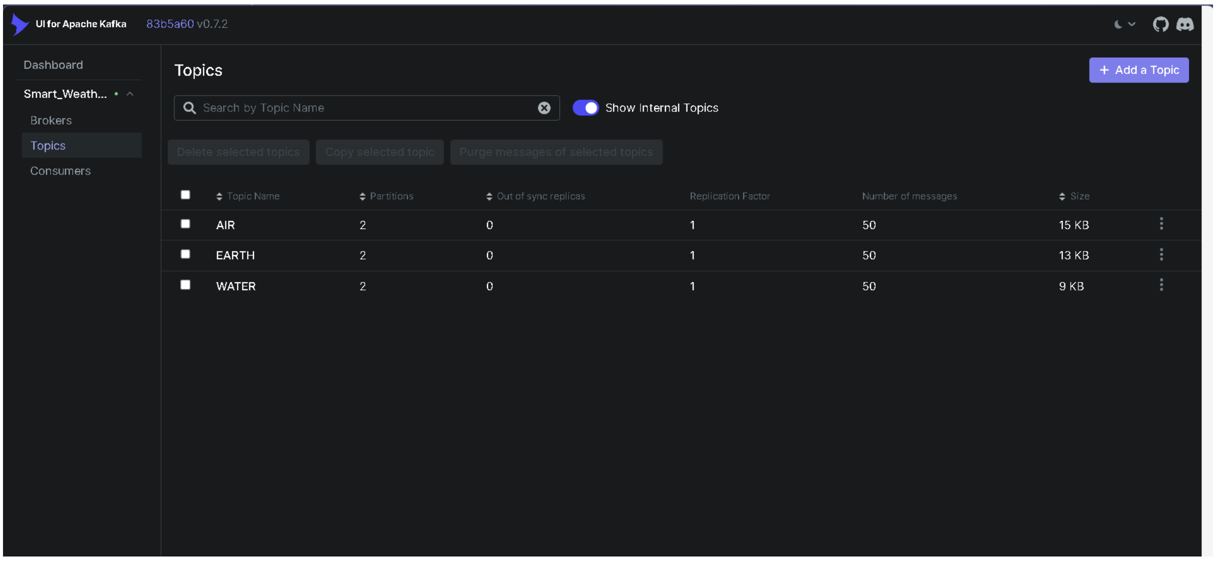

Figure 1: Data of Topics: AIR, EARTH, WATER

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.

```python
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, DoubleType,
IntegerType
from pyspark.sql.functions import col, from_json, to_json, struct, when, lit,
mean, to_timestamp

from pyspark.sql import functions as F
from pyspark.sql.window import Window
import random

# Initialize SparkSession
spark = SparkSession.builder \
    .appName("Environment Data Processing") \
    .config("spark.hadoop.fs.defaultFS", "hdfs://namenode:9000") \
    .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.

```python
# Define schemas for AIR, EARTH, and WATER data
air_schema = StructType([
    StructField("Time", StringType(), True),
    StructField("Station", StringType(), True),
    StructField("Temperature", DoubleType(), True),
    StructField("Moisture", DoubleType(), True),
    StructField("Light", DoubleType(), True),
    StructField("Total_Rainfall", DoubleType(), True),
    StructField("Rainfall", DoubleType(), True),
    StructField("Wind_Direction", IntegerType(), True),
    StructField("PM2.5", DoubleType(), True),
    StructField("PM10", DoubleType(), True),
    StructField("CO", DoubleType(), True),
    StructField("NOx", DoubleType(), True),
    StructField("SO2", DoubleType(), True)
])

earth_schema = StructType([
    StructField("Time", StringType(), True),
    StructField("Station", StringType(), True),
    StructField("Moisture", DoubleType(), True),
```

```
    StructField("Temperature", DoubleType(), True),
    StructField("Salinity", DoubleType(), True),
    StructField("pH", DoubleType(), True),
    StructField("Water_Root", DoubleType(), True),
    StructField("Water_Leaf", DoubleType(), True),
    StructField("Water_Level", DoubleType(), True),
    StructField("Voltage", DoubleType(), True)
])

water_schema = StructType([
    StructField("Time", StringType(), True),
    StructField("Station", StringType(), True),
    StructField("pH", DoubleType(), True),
    StructField("DO", DoubleType(), True),
    StructField("Temperature", DoubleType(), True),
    StructField("Salinity", DoubleType(), True)
])
```

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.

```python
# Read streams from Kafka
air_stream = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "kafka1:29092") \
    .option("subscribe", "AIR") \
    .option("failOnDataLoss", "false") \
    .option("startingOffsets", "earliest") \
    .load()

earth_stream = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "kafka2:39092") \
    .option("subscribe", "EARTH") \
    .option("failOnDataLoss", "false") \
    .option("startingOffsets", "earliest") \
    .load()

water_stream = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "kafka3:49092") \
    .option("subscribe", "WATER") \
    .option("failOnDataLoss", "false") \
    .option("startingOffsets", "earliest") \
    .load()

# Parse JSON msgs into structured data with Time column cast to Timestamp
air_data = air_stream.select(from_json(col("value").cast("string"),
air_schema).alias("data")).select(col("data.*"),)

earth_data = earth_stream.select(from_json(col("value").cast("string"),
earth_schema).alias("data")).select(col("data.*"),)

water_data = water_stream.select(from_json(col("value").cast("string"),
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.

```
def compute_statistics(df, columns, topic):
    # Define the window specification to calculate running average and standard deviation
    window_spec = Window.orderBy("timestamp").rowsBetween(Window.unboundedPreceding, Window.currentRow)

    # Create a list of expressions to calculate incremental average and stddev for each column
    select_exprs = []
    for column in columns:
        # Calculate running average (mean) and standard deviation
        avg_col = F.avg(F.col(column)).over(window_spec).alias(f"{column}_avg")
        stddev_col = F.stddev(F.col(column)).over(window_spec).alias(f"{column}_stddev")

        # Apply the formula: avg(v_i_0_n1) + rand(-std_i_0_n1, std_i_0_n1)
        # rand(-std, std)
        random_component = F.rand() * (stddev_col) * 2 - stddev_col
        new_value = avg_col + random_component

        select_exprs.append(new_value.alias(f"`{column}_calculated`"))

    # Perform the transformation and return the stream
    return df.select(select_exprs) \
             .writeStream \
             .outputMode("complete") \
             .format("memory") \
             .queryName(f"stats_table_{topic}") \
             .start()

# Start calculating statistics for each stream
air_columns = ["Temperature", "Moisture", "Light", "Total_Rainfall", "Rainfall", "`PM2.5`", "PM10", "CO", "NOx", "SO2"]
earth_columns = ["Moisture", "Temperature", "Salinity", "pH", "Water_Root", "Water_Leaf", "Water_Level", "Voltage"]
water_columns = ["pH", "DO", "Temperature", "Salinity"]

air_stats = compute_statistics(air_data, air_columns, "air")
earth_stats = compute_statistics(earth_data, earth_columns, "earth")
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.

```
# Join streams
environment_data = air_data.alias("air").join(
        earth_data.alias("earth"),["Time"],"inner"
    ).join(
        water_data.alias("water"),["Time"],"inner"
    )

# Debugging print joining result
environment_data.writeStream \
    .format("console") \
    .outputMode("append") \
    .trigger(processingTime="1 minute") \
    .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`.

```
# Convert the joined data to JSON
environment_json = environment_data.select(
    to_json(
        struct(
            col("Time"),
            struct(
                col("`air`.`Station`"),
                col("`air`.`Temperature`"),
                col("`air`.`Moisture`"),
                col("Light"),
                col("Total_Rainfall"),
                col("Rainfall"),
                col("Wind_Direction"),
                col("`PM2.5`"),
                col("PM10"),
                col("CO"),
                col("NOx"),
                col("SO2")
            ).alias("Air"),
            struct(
                col("`earth`.`Station`"),
                col("`earth`.`Moisture`"),
                col("`earth`.`Temperature`"),
                col("`earth`.`Salinity`"),
                col("`earth`.`pH`"),
                col("Water_Root"),
                col("Water_Leaf"),
                col("Water_Level"),
                col("Voltage")
            ).alias("Earth"),
            struct(
                col("`water`.`Station`"),
                col("`water`.`pH`"),
                col("DO"),
                col("`water`.`Temperature`"),
                col("`water`.`Salinity`")
            ).alias("Water")
```

```
        )
    ).alias("value")
)

# Write final stream to HDFS
query = environment_json.writeStream \
    .format("json") \
    .option("path", "/hdfs/environment_data/") \
    .option("checkpointLocation", "/hdfs/checkpoints/environment_data/") \
    .outputMode("append") \
    .trigger(processingTime="1 minute") \
    .start()

# Wait for termination
spark.streams.awaitAnyTermination()
```

1.8 Demonstration

```
Prepare HDFS:
  docker exec -it namenode bash -c 'hdfs dfs -mkdir /hdfs'
  docker exec -it namenode bash -c 'hdfs dfs -mkdir /hdfs/
   environment_data'
  docker exec -it namenode bash -c 'hdfs dfs -mkdir /hdfs/checkpoints/'
  docker exec -it namenode bash -c 'hdfs dfs -mkdir /hdfs/checkpoints/
   environment_data/'
  docker exec -it namenode bash -c 'hdfs dfs -chown -R spark:supergroup
   /hdfs'
Push code to Spark:
  docker cp spark_process.py spark-master:/opt/spark/python/
   spark_process.py
Run code:
  docker exec -it spark-master bash
  spark-submit \
    --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.3 \
    --master spark://spark-master:7077 \
  /opt/spark/python/spark_process.py
```

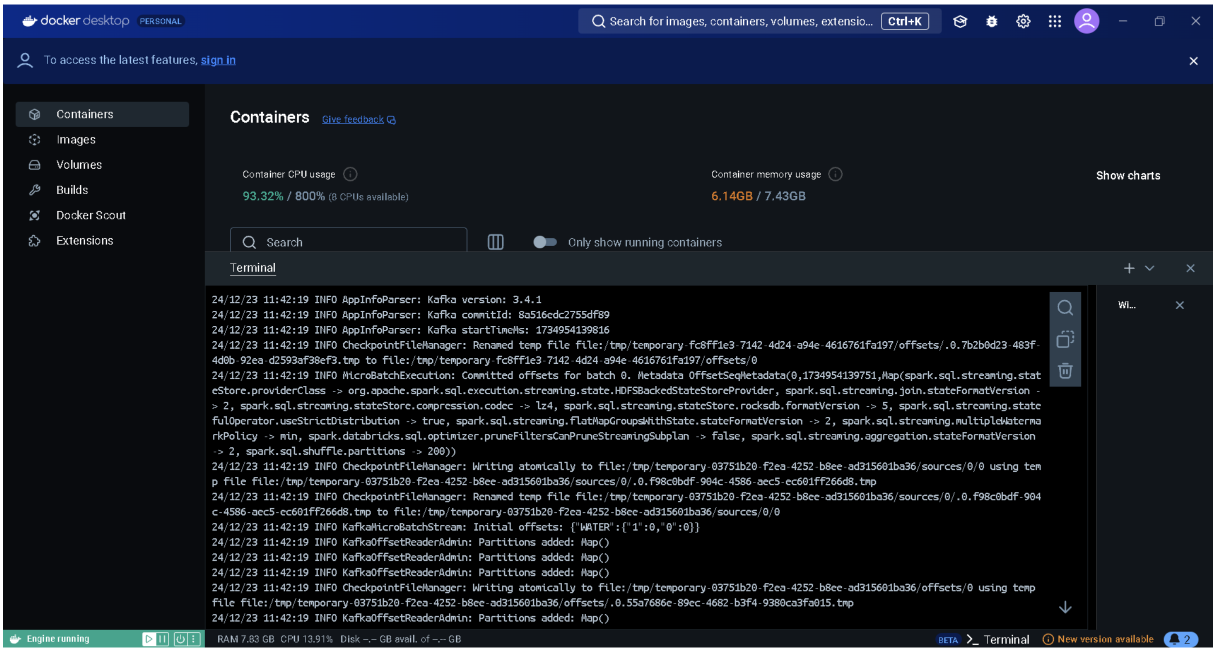

Figure 2: Run integration

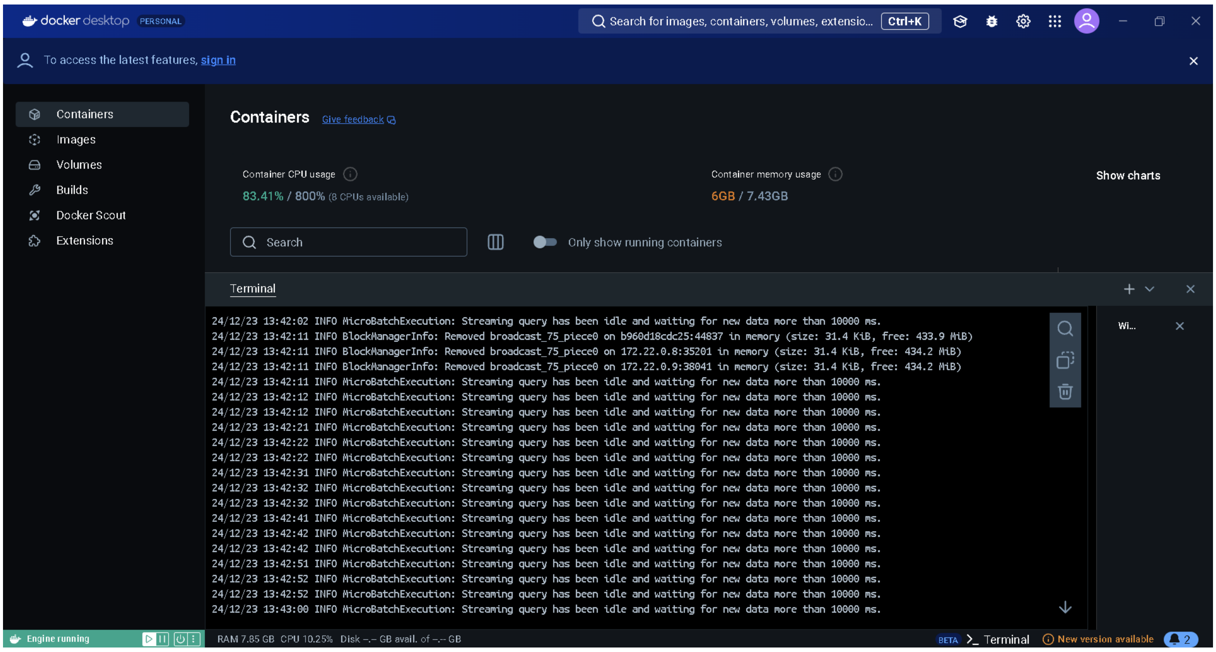

Figure 3: Finish integration

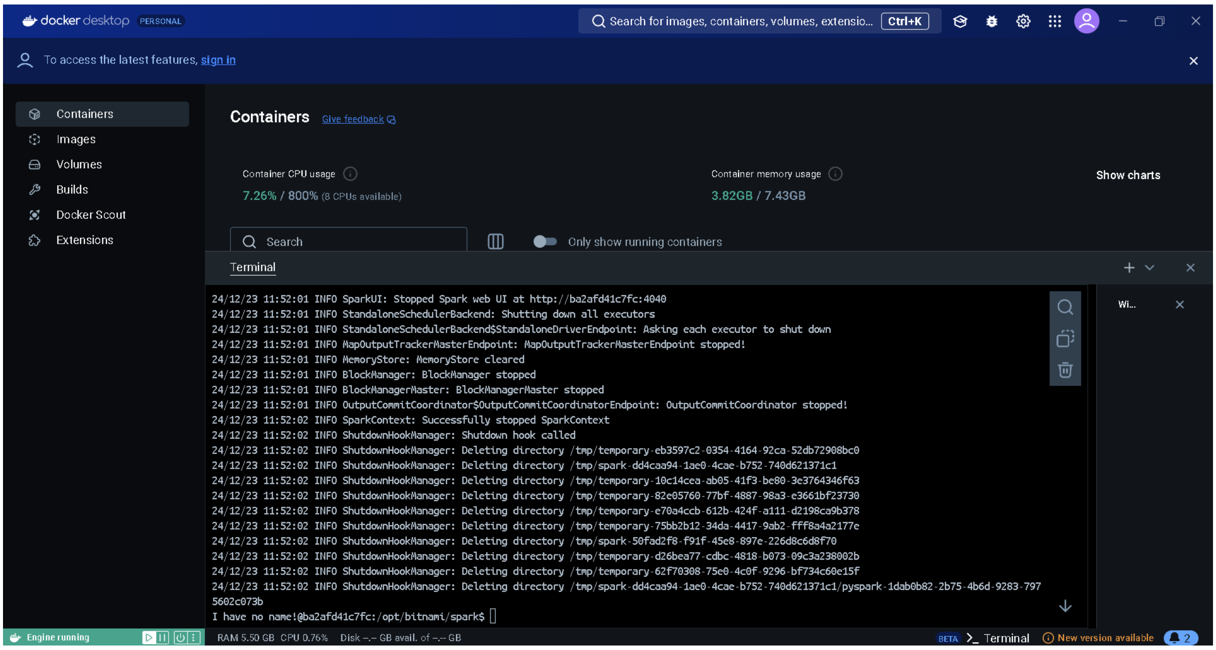

Figure 4: Terminate the execution

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

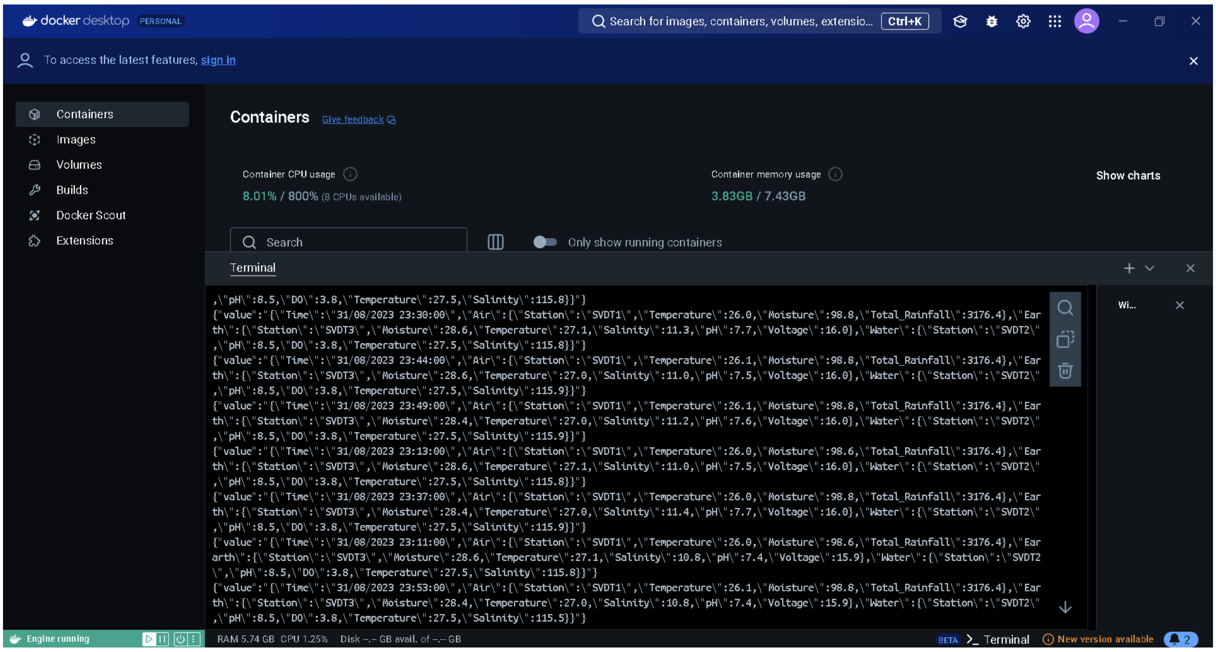

Figure 5: Output