SECTION 2: LAB ACTIVITIES

THIS LAB HAS BEEN WORKED ON BY BOTH MAHNOOR AND ZUHA. BOTH OF US HAVE USED MAHNOOR'S VM FOR THIS LAB

Activity 1: Spark SQL Fundamentals & Large Dataset Processing

Objectives: Load large datasets, execute SQL queries, understand lazy evaluation and caching

Time: 30 minutes

```
madeel@bdacourse:~/spark-kafka-lab$ source venv/bin/activate
(venv) madeel@bdacourse:~/spark-kafka-lab$ mkdir -p output
(venv) madeel@bdacourse:~/spark-kafka-lab$
```

Part 1: Create the Script

cd ~/spark-kafka-lab source venv/bin/activate

cat > activity1_spark_sql.py << 'PYEOF' from pyspark.sql import SparkSession
from pyspark.sql.functions import col, desc, avg, sum as spark_sum, count as spark_count,
row_number, round as spark_round
from pyspark.sql.window import Window import time

print("=" * 70)
print("Activity 1: Spark SQL Fundamentals") print("=" * 70)

# Create Spark Session
spark = SparkSession.builder \
    .appName("Activity1-SparkSQL") \
    .master("local[*]") \
    .config("spark.driver.memory", "4g") \
    .config("spark.sql.shuffle.partitions", "8") \

```
    .getOrCreate()

spark.sparkContext.setLogLevel("WARN")
print(fSpark Version: {spark.version}")
print(f"Running Mode: {spark.sparkContext.master} ")
PYEOF
```

This part of the script:

- **Imports Spark SQL libraries so we can use SQL functions, window functions, and SparkSession.**
- **Creates a SparkSession with following settings:**
  - **App name: `"Activity1-SparkSQL"`**
  - **Master: `"local[*]"` (uses all CPU cores on machine)**
  - **Driver memory: `"4g"` (Spark gets 4GB RAM)**
  - **Shuffle partitions: `8`**
- **Prints the Spark version + run mode,to ensure Spark is running correctly.**

Part 2: Add Data Loading

```
cat >> activity1_spark_sql.py << 'PYEOF'

# Load dataset print("=" * 70)
print("STEP 1: Loading Dataset") print("=" * 70)
start_time = time.time()
# NOTE: Use the taxi parquet files downloaded in setup (no script logic changed here) df =
spark.read.parquet("data/*.parquet")
load_time = time.time() - start_time
print(f"✓ Schema loaded in {load_time:.2f} seconds")

start_time = time.time() count = df.count()
count_time = time.time() - start_time print(f"✓ Total Records: {count:,}")
print(f"✓ Count time: {count_time:.2f} seconds")

print("
" + "=" * 70)
print("STEP 2: Data Schema and Sample") print("=" * 70)
df.printSchema() df.show(10, truncate=False) PYEOF
```

This part of the script is for loading the dataset into Spark Dataframe (not database) from parquet files. It also prints the time loading the dataset, count of record loaded, and time taken to count the records:

```
================================================================================
STEP 1: Loading Dataset
================================================================================
✓ Schema loaded in 3.71 seconds
✓ Total Records: 9,384,487
✓ Count time: 1.53 seconds
```

It also prints schema and first 10 records from the dataset:

```
================================================================================
STEP 2: Data Schema and Sample
================================================================================
root
 |-- VendorID: long (nullable = true)
 |-- tpep_pickup_datetime: timestamp_ntz (nullable = true)
 |-- tpep_dropoff_datetime: timestamp_ntz (nullable = true)
 |-- passenger_count: double (nullable = true)
 |-- trip_distance: double (nullable = true)
 |-- RatecodeID: double (nullable = true)
 |-- store_and_fwd_flag: string (nullable = true)
 |-- PULocationID: long (nullable = true)
 |-- DOLocationID: long (nullable = true)
 |-- payment_type: long (nullable = true)
 |-- fare_amount: double (nullable = true)
 |-- extra: double (nullable = true)
 |-- mta_tax: double (nullable = true)
 |-- tip_amount: double (nullable = true)
 |-- tolls_amount: double (nullable = true)
 |-- improvement_surcharge: double (nullable = true)
 |-- total_amount: double (nullable = true)
 |-- congestion_surcharge: double (nullable = true)
 |-- airport_fee: double (nullable = true)
```

```
+--------+--------------------+--------------------+---------------+-------------+----------+------------------+------------+------------+------------+-----------+-----+-------+----------+
----------+--------------------+-----------+--------------------+-----------+
|VendorID|tpep_pickup_datetime|tpep_dropoff_datetime|passenger_count|trip_distance|RatecodeID|store_and_fwd_flag|PULocationID|DOLocationID|payment_type|fare_amount|extra|mta_tax|tip_amount|
olls_amount|improvement_surcharge|total_amount|congestion_surcharge|airport_fee|
+--------+--------------------+--------------------+---------------+-------------+----------+------------------+------------+------------+------------+-----------+-----+-------+----------+
----------+--------------------+-----------+--------------------+-----------+
|2       |2023-01-01 00:32:10 |2023-01-01 00:40:36  |1.0            |0.97         |1.0       |N                 |161         |141         |2           |9.3        |1.0  |0.5    |0.0       |
.0        |1.0                 |14.3        |2.5                 |0.0        |
|2       |2023-01-01 00:55:08 |2023-01-01 01:01:27  |1.0            |1.1          |1.0       |N                 |43          |237         |1           |7.9        |1.0  |0.5    |4.0       |
.0        |1.0                 |16.9        |2.5                 |0.0        |
|2       |2023-01-01 00:25:04 |2023-01-01 00:37:49  |1.0            |2.51         |1.0       |N                 |48          |238         |1           |14.9       |1.0  |0.5    |15.0      |
.0        |1.0                 |34.9        |2.5                 |0.0        |
|1       |2023-01-01 00:03:48 |2023-01-01 00:13:25  |0.0            |1.9          |1.0       |N                 |138         |7           |1           |12.1       |7.25 |0.5    |0.0       |
.0        |1.0                 |20.85       |0.0                 |1.25       |
|2       |2023-01-01 00:10:29 |2023-01-01 00:21:19  |1.0            |1.43         |1.0       |N                 |107         |79          |1           |11.4       |1.0  |0.5    |3.28      |
.0        |1.0                 |19.68       |2.5                 |0.0        |
|2       |2023-01-01 00:50:34 |2023-01-01 01:02:52  |1.0            |1.84         |1.0       |N                 |161         |137         |1           |12.8       |1.0  |0.5    |10.0      |
.0        |1.0                 |27.8        |2.5                 |0.0        |
|2       |2023-01-01 00:09:22 |2023-01-01 00:19:49  |1.0            |1.66         |1.0       |N                 |239         |143         |1           |12.1       |1.0  |0.5    |3.42      |
.0        |1.0                 |20.52       |2.5                 |0.0        |
|2       |2023-01-01 00:27:12 |2023-01-01 00:49:56  |1.0            |11.7         |1.0       |N                 |142         |200         |1           |45.7       |1.0  |0.5    |10.74     |
.0        |1.0                 |64.44       |2.5                 |0.0        |
|2       |2023-01-01 00:21:44 |2023-01-01 00:36:40  |1.0            |2.95         |1.0       |N                 |164         |236         |1           |17.7       |1.0  |0.5    |5.68      |
.0        |1.0                 |28.38       |2.5                 |0.0        |
|2       |2023-01-01 00:39:42 |2023-01-01 00:50:36  |1.0            |3.01         |1.0       |N                 |141         |107         |2           |14.9       |1.0  |0.5    |0.0       |
.0        |1.0                 |19.9        |2.5                 |0.0        |
+--------+--------------------+--------------------+---------------+-------------+----------+------------------+------------+------------+------------+-----------+-----+-------+----------+
----------+--------------------+-----------+--------------------+-----------+
```

When we first tried loading the dataset, Spark threw a **schema mismatch error** because of this line:

```
spark.read.parquet("data/*.parquet")
```

This is because the Parquet files had different column names or data types.

We replaced this line of code by **loading data for each month in separate dataframes.**

```
df_jan = spark.read.parquet("data/yellow_tripdata_2023-01.parquet")
df_feb = spark.read.parquet("data/yellow_tripdata_2023-02.parquet")
df_mar = spark.read.parquet("data/yellow_tripdata_2023-03.parquet")
```

Then we **normalized the dataframes and casted columns to type double**, to ensure all have the same columns and data types.

```
#Normalization
common_cols = df_jan.columns
df_feb = df_feb.select(common_cols)
df_mar = df_mar.select(common_cols)
#Typecasting
df_jan = df_jan.withColumn("passenger_count", col("passenger_count").cast("double"))
df_feb = df_feb.withColumn("passenger_count", col("passenger_count").cast("double"))
df_mar = df_mar.withColumn("passenger_count", col("passenger_count").cast("double"))
```

Part 3: Add SQL Queries

cat >> activity1_spark_sql.py << 'PYEOF'

SQL Queries print("
" + "=" * 70)
print("STEP 3: SQL Query - Average Fare by Passenger Count") print("=" * 70)
start_time = time.time() df.createOrReplaceTempView("trips")

result1 = spark.sql(""" SELECT
passenger_count, COUNT(*) as trip_count,
ROUND(AVG(fare_amount), 2) as avg_fare, ROUND(AVG(tip_amount), 2) as avg_tip,
ROUND(AVG(trip_distance), 2) as avg_distance
FROM trips
GROUP BY passenger_count ORDER BY passenger_count

```
""")
result1.show()
query1_time = time.time() - start_time
print(f"✓ Query time: {query1_time:.2f} seconds")

print("
" + "=" * 70)
print("STEP 4: SQL Query - Payment Type Analysis") print("=" * 70)
start_time = time.time() result2 = spark.sql("""
   SELECT
      payment_type,
      COUNT(*) as transaction_count, ROUND(SUM(total_amount), 2) as total_revenue,
      ROUND(AVG(total_amount), 2) as avg_transaction
   FROM trips
   GROUP BY payment_type ORDER BY total_revenue DESC
""")
result2.show()
query2_time = time.time() - start_time
print(f"✓ Query time: {query2_time:.2f} seconds")
PYEOF
```

In this part of the script, we will run 2 queries on the dataset.

Firstly, we created a temporary View, called `trips`, to enable SQL queries on the dataset.

Query1: The first query is used to evaluate the total trips, average fair amount, average trip amount, and average trip distance for each passenger count. The results are ordered by passenger count.

Output:

```
======================================================================
STEP 3: SQL Query - Average Fare by Passenger Count
======================================================================
+---------------+----------+--------+-------+------------+
|passenger_count|trip_count|avg_fare|avg_tip|avg_distance|
+---------------+----------+--------+-------+------------+
|           NULL|    236179|   20.81|   3.59|        21.3|
|            0.0|    156806|   16.48|   2.95|        2.76|
|            1.0|   6950240|   18.01|   3.38|        3.32|
|            2.0|   1345215|   20.41|   3.65|        3.96|
|            3.0|    321857|   19.86|   3.45|        3.71|
|            4.0|    160905|   21.05|   3.36|        3.79|
|            5.0|    129147|   18.07|   3.42|        3.27|
|            6.0|     84083|   18.34|   3.44|         3.3|
|            7.0|        16|   58.55|   8.15|        3.17|
|            8.0|        29|   80.81|   9.61|        4.54|
|            9.0|        10|   75.15|   22.2|        2.91|
+---------------+----------+--------+-------+------------+
```

Query2: The second query is to evaluate the total number of transactions, total revenue, and average transaction amount for each payment type. The results are ordered by total revenue.

Output:

```
======================================================================
STEP 4: SQL Query - Payment Type Analysis
======================================================================
+------------+-----------------+--------------+---------------+
|payment_type|transaction_count| total_revenue|avg_transaction|
+------------+-----------------+--------------+---------------+
|           1|          7410897|2.1164688565E8|          28.56|
|           2|          1575700| 3.654573494E7|          23.19|
|           0|           236179|    6840697.06|          28.96|
|           3|            57707|     602203.84|          10.44|
|           4|           104003|     247001.19|           2.37|
|           5|                1|           0.0|            0.0|
+------------+-----------------+--------------+---------------+

✓ Query time: 0.95 seconds
```

Part 4: Add Caching Demo

cat >> activity1_spark_sql.py << 'PYEOF'

Caching print("
" + "=" * 70)
print("STEP 5: Caching Demonstration") print("=" * 70)

print("Query WITHOUT cache:") start_time = time.time()
df.groupBy("payment_type").count().show() time_no_cache = time.time() - start_time
print(f"✓ Time without cache: {time_no_cache:.2f} seconds")

print("
Caching dataframe...") df.cache()
df.count()

print("
Query WITH cache:") start_time = time.time()
df.groupBy("payment_type").count().show() time_with_cache = time.time() - start_time
print(f"✓ Time with cache: {time_with_cache:.2f} seconds") print(f"✓ Speedup:
{time_no_cache/time_with_cache:.2f}x")

spark.stop() print("
✓ Activity 1 Complete!")
PYEOF

This part of the script compares the computation times for a DataFrame **without caching and**

with caching. It runs a simple query:

```
df.groupBy("payment_type").count().show()
```

with and without caching. The DataFrame is cached using `df.cache()` and an action (`df.count()`) to materialize it.

The script records computation times for both cases and calculates the **speedup**.

Output:

```
======================================================================
STEP 5: Caching Demonstration
======================================================================
Query WITHOUT cache:
+------------+-------+
|payment_type|  count|
+------------+-------+
|           2|1575700|
|           3|  57707|
|           4| 104003|
|           1|7410897|
|           0| 236179|
|           5|      1|
+------------+-------+

✓ Time without cache: 0.42 seconds
Caching dataframe...
Query WITH cache:
+------------+-------+
|payment_type|  count|
+------------+-------+
|           2|1575700|
|           3|  57707|
|           4| 104003|
|           1|7410897|
|           0| 236179|
|           5|      1|
+------------+-------+

✓ Time with cache: 0.56 seconds
✓ Speedup: 0.76x
✓ Activity 1 Complete!
```

Part 5: Run the Script

mkdir -p output
python3 activity1_spark_sql.py

REQUIRED SCREENSHOTS

Screenshot 1: Complete terminal output showing all 5 steps executed successfully

```
================================================================================
STEP 1: Loading Dataset
================================================================================
✓ Schema loaded in 3.71 seconds
✓ Total Records: 9,384,487
✓ Count time: 1.53 seconds
================================================================================
STEP 2: Data Schema and Sample
================================================================================
root
 |-- VendorID: long (nullable = true)
 |-- tpep_pickup_datetime: timestamp_ntz (nullable = true)
 |-- tpep_dropoff_datetime: timestamp_ntz (nullable = true)
 |-- passenger_count: double (nullable = true)
 |-- trip_distance: double (nullable = true)
 |-- RatecodeID: double (nullable = true)
 |-- store_and_fwd_flag: string (nullable = true)
 |-- PULocationID: long (nullable = true)
 |-- DOLocationID: long (nullable = true)
 |-- payment_type: long (nullable = true)
 |-- fare_amount: double (nullable = true)
 |-- extra: double (nullable = true)
 |-- mta_tax: double (nullable = true)
 |-- tip_amount: double (nullable = true)
 |-- tolls_amount: double (nullable = true)
 |-- improvement_surcharge: double (nullable = true)
 |-- total_amount: double (nullable = true)
 |-- congestion_surcharge: double (nullable = true)
 |-- airport_fee: double (nullable = true)
```

```
+--------+--------------------+--------------------+---------------+-------------+----------+------------------+------------+------------+------------+-----------+-----+-------+----------+
----------+---------------------+-----------+-------------------+----------+
|VendorID|tpep_pickup_datetime|tpep_dropoff_datetime|passenger_count|trip_distance|RatecodeID|store_and_fwd_flag|PULocationID|DOLocationID|payment_type|fare_amount|extra|mta_tax|tip_amount|
olls_amount|improvement_surcharge|total_amount|congestion_surcharge|airport_fee|
+--------+--------------------+--------------------+---------------+-------------+----------+------------------+------------+------------+------------+-----------+-----+-------+----------+
----------+---------------------+-----------+-------------------+----------+
|2       |2023-01-01 00:32:10 |2023-01-01 00:40:36 |1.0            |0.97         |1.0       |N                 |161         |141         |2           |9.3        |1.0  |0.5    |0.0       |
.0       |1.0                  |14.3        |2.5                |0.0       |
|2       |2023-01-01 00:55:08 |2023-01-01 01:01:27 |1.0            |1.1          |1.0       |N                 |43          |237         |1           |7.9        |1.0  |0.5    |4.0       |
.0       |1.0                  |16.9        |2.5                |0.0       |
|2       |2023-01-01 00:25:04 |2023-01-01 00:37:49 |1.0            |2.51         |1.0       |N                 |48          |238         |1           |14.9       |1.0  |0.5    |15.0      |
.0       |1.0                  |34.9        |2.5                |0.0       |
|1       |2023-01-01 00:03:48 |2023-01-01 00:13:25 |0.0            |1.9          |1.0       |N                 |138         |7           |1           |12.1       |7.25 |0.5    |0.0       |
.0       |1.0                  |20.85       |0.0                |1.25      |
|2       |2023-01-01 00:10:29 |2023-01-01 00:21:19 |1.0            |1.43         |1.0       |N                 |107         |79          |1           |11.4       |1.0  |0.5    |3.28      |
.0       |1.0                  |19.68       |2.5                |0.0       |
|2       |2023-01-01 00:50:34 |2023-01-01 01:02:52 |1.0            |1.84         |1.0       |N                 |161         |137         |1           |12.8       |1.0  |0.5    |10.0      |
.0       |1.0                  |27.8        |2.5                |0.0       |
|2       |2023-01-01 00:09:22 |2023-01-01 00:19:49 |1.0            |1.66         |1.0       |N                 |239         |143         |1           |12.1       |1.0  |0.5    |3.42      |
.0       |1.0                  |20.52       |2.5                |0.0       |
|2       |2023-01-01 00:27:12 |2023-01-01 00:49:56 |1.0            |11.7         |1.0       |N                 |142         |200         |1           |45.7       |1.0  |0.5    |10.74     |
.0       |1.0                  |64.44       |2.5                |0.0       |
|2       |2023-01-01 00:21:44 |2023-01-01 00:36:40 |1.0            |2.95         |1.0       |N                 |164         |236         |1           |17.7       |1.0  |0.5    |5.68      |
.0       |1.0                  |28.38       |2.5                |0.0       |
|2       |2023-01-01 00:39:42 |2023-01-01 00:50:36 |1.0            |3.01         |1.0       |N                 |141         |107         |2           |14.9       |1.0  |0.5    |0.0       |
.0       |1.0                  |19.9        |2.5                |0.0       |
+--------+--------------------+--------------------+---------------+-------------+----------+------------------+------------+------------+------------+-----------+-----+-------+----------+
----------+---------------------+-----------+-------------------+----------+
```

```
======================================================================
STEP 3: SQL Query - Average Fare by Passenger Count
======================================================================
+---------------+----------+--------+-------+------------+
|passenger_count|trip_count|avg_fare|avg_tip|avg_distance|
+---------------+----------+--------+-------+------------+
|           NULL|    236179|   20.81|   3.59|        21.3|
|            0.0|    156806|   16.48|   2.95|        2.76|
|            1.0|   6950240|   18.01|   3.38|        3.32|
|            2.0|   1345215|   20.41|   3.65|        3.96|
|            3.0|    321857|   19.86|   3.45|        3.71|
|            4.0|    160905|   21.05|   3.36|        3.79|
|            5.0|    129147|   18.07|   3.42|        3.27|
|            6.0|     84083|   18.34|   3.44|         3.3|
|            7.0|        16|   58.55|   8.15|        3.17|
|            8.0|        29|   80.81|   9.61|        4.54|
|            9.0|        10|   75.15|   22.2|        2.91|
+---------------+----------+--------+-------+------------+

✓ Query time: 2.54 seconds
```

```
======================================================================
STEP 4: SQL Query - Payment Type Analysis
======================================================================
+------------+-----------------+-------------+---------------+
|payment_type|transaction_count| total_revenue|avg_transaction|
+------------+-----------------+-------------+---------------+
|           1|          7410897|2.1164688565E8|          28.56|
|           2|          1575700| 3.654573494E7|          23.19|
|           0|           236179|    6840697.06|          28.96|
|           3|            57707|     602203.84|          10.44|
|           4|           104003|     247001.19|           2.37|
|           5|                1|           0.0|            0.0|
+------------+-----------------+-------------+---------------+

✓ Query time: 0.95 seconds
```

```
================================================================================
STEP 5: Caching Demonstration
================================================================================
Query WITHOUT cache:
+------------+-------+
|payment_type|  count|
+------------+-------+
|           2|1575700|
|           3|  57707|
|           4| 104003|
|           1|7410897|
|           0| 236179|
|           5|      1|
+------------+-------+

✓ Time without cache: 0.42 seconds
Caching dataframe...
Query WITH cache:
+------------+-------+
|payment_type|  count|
+------------+-------+
|           2|1575700|
|           3|  57707|
|           4| 104003|
|           1|7410897|
|           0| 236179|
|           5|      1|
+------------+-------+

✓ Time with cache: 0.56 seconds
✓ Speedup: 0.76x
✓ Activity 1 Complete!
```

CRITICAL QUESTIONS - ACTIVITY 1

Question 1: Explain the difference between lazy evaluation and action in Spark. Which operations in your code were transformations and which were actions?

In **Lazy Evaluation**, Spark doesn't immediately execute queries when transformations are defined, instead it builds a **logical plan, DAG** of transformations to optimize execution later, by **combining transformations** and **reducing data shuffles**.

An **Action** triggers Spark to **actually execute the transformations** and produce output.

Transformations: `withColumn(), select(), unionByName(), groupBy(), cache().`

Actions: `count(), show(), df.count()`

Question 2: What was your caching speedup? Explain why caching improves performance and when it would NOT be beneficial to cache a DataFrame.

Caching Speedup: `0.76`

Caching improves performance as it **avoids recomputation of transformations** by storing the DataFrame in memory. This speeds up repeated queries and iterative operations on the same dataset.

In our case caching wasn't beneficial. Some prominent reason for this could be:

- Caching introduces **overhead** to materialize and store ~9.3M rows in memory.
- Only **one query** was run after caching, so the benefit was minimal.

Generally, caching isn't beneficial for **small datasets** or **one-time queries**. Moreover, If **memory is constrained**, caching large data can slow down computation.

Activity 2: Performance Benchmarking (Single-Node vs Cluster)

Objectives: Compare performance with different core counts, analyze parallelism benefits
Time: 40 minutes

```
madeel@bdacourse:~/spark-kafka-lab$ source venv/bin/activate
(venv) madeel@bdacourse:~/spark-kafka-lab$ mkdir -p output
(venv) madeel@bdacourse:~/spark-kafka-lab$
```

Part 1: Create Single-Node Benchmark

cat > activity2_single_node.py << 'PYEOF' from pyspark.sql import SparkSession
from pyspark.sql.functions import col, desc, avg, sum as spark_sum, count as spark_count
import time

def run_benchmark(cores, app_name): print(f"
{'='*70}")
    print(f"BENCHMARK: {cores} Core(s)") print(f"{'='*70}
")

    spark = SparkSession.builder \
        .appName(app_name) \
        .master(f"local[{cores}]") \
        .config("spark.driver.memory", "4g") \
        .config("spark.sql.shuffle.partitions", str(cores * 2)) \
        .getOrCreate()

    spark.sparkContext.setLogLevel("ERROR") results = {}

    # Test 1: Load Data print("Test 1: Loading Data...") start = time.time()
    df = spark.read.parquet('data/*.parquet') record_count = df.count() results['load_time'] =
    time.time() - start
    print(f" ✓ Loaded {record_count:,} records in {results['load_time']:.2f}s")

    df.cache() df.count()

```
    # Test 2: Aggregation print("Test 2: Aggregation...") start = time.time()
    df.groupBy("payment_type", "passenger_count") \
        .agg(spark_count("*").alias("cnt"), avg("fare_amount"), spark_sum("total_amount")) \
        .collect()
    results['aggregation_time'] = time.time() - start
    print(f" ✓ Completed in {results['aggregation_time']:.2f}s")

    # Test 3: Filter
    print("Test 3: Filter & Sort...") start = time.time()
    df.filter(col("fare_amount") > 50).orderBy(desc("fare_amount")).limit(1000).collect()
    results['filter_time'] = time.time() - start
    print(f" ✓ Completed in {results['filter_time']:.2f}s")

    # Test 4: Join print("Test 4: Join...") start = time.time()
    df_sample = df.sample(0.01) df_sample.alias("a").join(df_sample.alias("b"),
    col("a.passenger_count") ==
col("b.passenger_count")).count() results['join_time'] = time.time() - start
    print(f" ✓ Completed in {results['join_time']:.2f}s")

    results['total_time'] = sum(results.values()) spark.stop()
    return results PYEOF
```

Part 2: Add Benchmark Loop and Results

```
cat >> activity2_single_node.py << 'PYEOF'

print("=" * 70)
print("SINGLE-NODE PERFORMANCE BENCHMARK")
print("=" * 70)

configs = [1, 2, 4] all_results = {}

for cores in configs:
    all_results[cores] = run_benchmark(cores, f"Benchmark-{cores}Cores")
```

```
    time.sleep(3)

# Print comparison print("
" + "=" * 85)
print("PERFORMANCE COMPARISON")
print("=" * 85)
print(f"{'Test':<20} {'1 Core':>12} {'2 Cores':>12} {'4 Cores':>12} {'Speedup':>12}")
print("-" * 85)

for test in ['load_time', 'aggregation_time', 'filter_time', 'join_time', 'total_time']: times =
    [all_results[cores][test] for cores in configs]
    speedup = times[0] / times[2]
    print(f"{test:<20} {times[0]:>10.2f}s {times[1]:>10.2f}s {times[2]:>10.2f}s {speedup:>11.2f}x")

print("=" * 85)

# Save results
with open('output/single_node_results.txt', 'w') as f: f.write("Single Node Benchmark Results
")
    f.write("="*50 + "

")
    for cores in configs: f.write(f"{cores} Core(s):
")
        for test, val in all_results[cores].items(): f.write(f" {test}: {val:.2f}s
")
        f.write("
")

print("
✓       Results saved to output/single_node_results.txt") PYEOF
```

Merged both part 1 and part 2 into 1

```
cat > activity2_single_node.py << 'PYEOF'
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, desc, avg, sum as spark_sum,
count as spark_count
import time

def run_benchmark(cores, app_name):
    print("\n" + "=" * 70)
    print(f"BENCHMARK: {cores} Core(s)")
    print("=" * 70)

    # Create a SparkSession in *local* mode with 'cores' cores
```

```
    spark = (
        SparkSession.builder
        .appName(app_name)
        .master(f"local[{cores}]")  # local[1], local[2], local[4]
        .config("spark.driver.memory", "4g")
        .config("spark.sql.shuffle.partitions", str(cores * 2))
        .getOrCreate()
    )

    spark.sparkContext.setLogLevel("ERROR")
    results = {}

    # ----------------------------
    # Test 1: Load Data
    # ----------------------------
    print("Test 1: Loading Data...")
    start = time.time()

    # [CHANGED] Load each month separately and normalize schema
    df_jan =
spark.read.parquet("data/yellow_tripdata_2023-01.parquet")
    df_feb =
spark.read.parquet("data/yellow_tripdata_2023-02.parquet")
    df_mar =
spark.read.parquet("data/yellow_tripdata_2023-03.parquet")

    # Ensure same columns for all three (use Jan as reference)
    common_cols = df_jan.columns
    df_feb = df_feb.select(common_cols)
    df_mar = df_mar.select(common_cols)

    # Normalize passenger_count type
    df_jan = df_jan.withColumn("passenger_count",
col("passenger_count").cast("double"))
    df_feb = df_feb.withColumn("passenger_count",
col("passenger_count").cast("double"))
    df_mar = df_mar.withColumn("passenger_count",
col("passenger_count").cast("double"))

    # Union into single DataFrame
    df = df_jan.unionByName(df_feb).unionByName(df_mar)
```

```
    record_count = df.count()  # action triggers the load
    results["load_time"] = time.time() - start
    print(f" ✓ Loaded {record_count:,} records in
{results['load_time']:.2f}s")

    # Cache the DataFrame in memory so later tests run faster
    df.cache()
    df.count()  # materialize cache

    # ---------------------------
    # Test 2: Aggregation
    # ---------------------------
    print("Test 2: Aggregation...")
    start = time.time()

    (
        df.groupBy("payment_type", "passenger_count")
        .agg(
            spark_count("*").alias("cnt"),
            avg("fare_amount").alias("avg_fare"),
            spark_sum("total_amount").alias("sum_total")
        )
        .collect()  # action
    )

    results["aggregation_time"] = time.time() - start
    print(f" ✓ Completed in {results['aggregation_time']:.2f}s")

    # ---------------------------
    # Test 3: Filter & Sort
    # ---------------------------
    print("Test 3: Filter & Sort...")
    start = time.time()

    (
        df.filter(col("fare_amount") > 50)
        .orderBy(desc("fare_amount"))
        .limit(1000)
        .collect()  # action
    )

    results["filter_time"] = time.time() - start
```

```
    print(f" ✓ Completed in {results['filter_time']:.2f}s")

    # ----------------------------
    # Test 4: Join
    # ----------------------------
    print("Test 4: Join...")
    start = time.time()

    df_sample = df.sample(0.01)
    (
        df_sample.alias("a")
        .join(df_sample.alias("b"),
              col("a.passenger_count") == col("b.passenger_count"))
        .count()  # action
    )

    results["join_time"] = time.time() - start
    print(f" ✓ Completed in {results['join_time']:.2f}s")

    # Total time for this config
    results["total_time"] = sum(results.values())

    spark.stop()
    return results


# ===== MAIN: Benchmark Loop + Results =====

print("=" * 70)
print("SINGLE-NODE PERFORMANCE BENCHMARK")
print("=" * 70)

configs = [1, 2, 4]
all_results = {}

# Run benchmark for each core config
for cores in configs:
    all_results[cores] = run_benchmark(cores,
f"Benchmark-{cores}Cores")
    time.sleep(3)  # small pause between runs

# Print comparison table
```

```
print("\n" + "=" * 85)
print("PERFORMANCE COMPARISON")
print("=" * 85)
print(f"{'Test':<20} {'1 Core':>12} {'2 Cores':>12} {'4 Cores':>12}
{'Speedup':>12}")
print("-" * 85)

for test in ["load_time", "aggregation_time", "filter_time",
"join_time", "total_time"]:
    times = [all_results[cores][test] for cores in configs]
    speedup = times[0] / times[2]  # 1-core time / 4-core time
    print(
        f"{test:<20} "
        f"{times[0]:>10.2f}s {times[1]:>10.2f}s {times[2]:>10.2f}s
{speedup:>11.2f}x"
    )

print("=" * 85)

# Save results to file
with open("output/single_node_results.txt", "w") as f:
    f.write("Single Node Benchmark Results\n")
    f.write("=" * 50 + "\n\n")
    for cores in configs:
        f.write(f"{cores} Core(s):\n")
        for test, val in all_results[cores].items():
            f.write(f"  {test}: {val:.2f}s\n")
        f.write("\n")

print("\nResults saved to output/single_node_results.txt")
PYEOF
```

Part 3: Run Single-Node Benchmark

python3 activity2_single_node.py

```
$ python3 activity2_single_node.py
======================================================================
SINGLE-NODE PERFORMANCE BENCHMARK
======================================================================

======================================================================
BENCHMARK: 1 Core(s)
======================================================================
```

```
25/11/27 20:19:42 WARN Utils: Your hostname, bdacourse resolves to a
loopback address: 127.0.1.1; using 172.17.5.42 instead (on interface
ens18)
25/11/27 20:19:42 WARN Utils: Set SPARK_LOCAL_IP if you need to bind
to another address
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
25/11/27 20:19:43 WARN NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where
applicable
Test 1: Loading Data...
 ✓ Loaded 9,384,487 records in 6.41s
Test 2: Aggregation...
 ✓ Completed in 2.00s
Test 3: Filter & Sort...
 ✓ Completed in 3.10s
Test 4: Join...
 ✓ Completed in 63.37s

======================================================================
BENCHMARK: 2 Core(s)
======================================================================
Test 1: Loading Data...
 ✓ Loaded 9,384,487 records in 0.69s
Test 2: Aggregation...
 ✓ Completed in 1.26s
Test 3: Filter & Sort...
 ✓ Completed in 2.23s
Test 4: Join...
 ✓ Completed in 42.21s

======================================================================
BENCHMARK: 4 Core(s)
======================================================================
Test 1: Loading Data...
 ✓ Loaded 9,384,487 records in 0.59s
Test 2: Aggregation...
 ✓ Completed in 1.42s
Test 3: Filter & Sort...
 ✓ Completed in 1.71s
Test 4: Join...
 ✓ Completed in 23.57s

======================================================================
===============
PERFORMANCE COMPARISON
======================================================================
```

```
===============
Test                          1 Core      2 Cores      4 Cores
Speedup
-------------------------------------------------------------------------
---------------
load_time                     6.41s       0.69s       0.59s       10.81x
aggregation_time              2.00s       1.26s       1.42s        1.40x
filter_time                   3.10s       2.23s       1.71s        1.81x
join_time                    63.37s      42.21s      23.57s        2.69x
total_time                   74.88s      46.39s      27.30s        2.74x
=========================================================================
===============

Results saved to output/single_node_results.txt
```

Part 4: Create Cluster Benchmark

cat > activity2_cluster.py << 'PYEOF' from pyspark.sql import SparkSession
from pyspark.sql.functions import col, desc, avg, sum as spark_sum, count as spark_count
import time

def run_cluster_benchmark(cores, app_name): print(f"
{'='*70}")
    print(f"CLUSTER BENCHMARK: {cores} Cores") print(f"{'='*70}
")

    spark = SparkSession.builder \
        .appName(app_name) \
        .master("spark://localhost:7077") \
        .config("spark.executor.memory", "2g") \
        .config("spark.driver.memory", "2g") \
        .config("spark.cores.max", str(cores)) \
        .config("spark.sql.shuffle.partitions", str(cores * 4)) \
        .getOrCreate()

    spark.sparkContext.setLogLevel("ERROR") results = {}

    print("Test 1: Loading...") start = time.time()
    df = spark.read.parquet('data/*.parquet') df.count()
    results['load_time'] = time.time() - start print(f" ✓ {results['load_time']:.2f}s")

    df = df.repartition(cores * 4).cache() df.count()

    print("Test 2: Aggregation...") start = time.time()

```
    df.groupBy("payment_type").agg(spark_count("*"), avg("fare_amount")).collect()
    results['aggregation_time'] = time.time() - start print(f" ✓ {results['aggregation_time']:.2f}s")

    print("Test 3: Filter...") start = time.time()
    df.filter(col("fare_amount") > 50).limit(10000).collect() results['filter_time'] = time.time() - start
    print(f" ✓ {results['filter_time']:.2f}s")

    print("Test 4: Join...") start = time.time()
    df_left = df.sample(0.02).repartition(cores, "passenger_count") df_right =
    df.sample(0.02).repartition(cores, "passenger_count") df_left.join(df_right,
    "passenger_count").count() results['join_time'] = time.time() - start
    print(f" ✓ {results['join_time']:.2f}s")

    results['total_time'] = sum(results.values()) spark.stop()
    time.sleep(5) return results

print("=" * 70)
print("CLUSTER PERFORMANCE BENCHMARK")
print("=" * 70)

configurations = [2, 4, 6] all_results = {}

for cores in configurations:
    all_results[cores] = run_cluster_benchmark(cores, f"Cluster-{cores}Cores")

# Comparison print("
" + "=" * 85)
print("CLUSTER COMPARISON")
print("=" * 85)
print(f"{'Test':<20} {'2 Cores':>12} {'4 Cores':>12} {'6 Cores':>12} {'Speedup':>12}")
print("-" * 85)

for test in ['load_time', 'aggregation_time', 'filter_time', 'join_time', 'total_time']: times =
    [all_results[cores][test] for cores in configurations]
    speedup = times[0] / times[2]
    print(f"{test:<20} {times[0]:>10.2f}s {times[1]:>10.2f}s {times[2]:>10.2f}s {speedup:>11.2f}x")
    print("=" * 85)

with open('output/cluster_results.txt', 'w') as f: f.write("Cluster Benchmark Results
")
    f.write("="*50 + "

")
    for cores in configurations: f.write(f"{cores} Cores:
")
        for test, val in all_results[cores].items(): f.write(f" {test}: {val:.2f}s
")
        f.write("
```

")

print("
✓ Results saved to output/cluster_results.txt") PYEOF

```
cat > activity2_cluster.py << 'PYEOF'
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, desc, avg, sum as spark_sum,
count as spark_count
import time

def run_cluster_benchmark(cores, app_name):
    print("\n" + "=" * 70)
    print(f"CLUSTER BENCHMARK: {cores} Cores")
    print("=" * 70)

    # Connect to Spark standalone cluster
    spark = (
        SparkSession.builder
        .appName(app_name)
        .master("spark://localhost:7077")
        .config("spark.executor.memory", "2g")
        .config("spark.driver.memory", "2g")
        .config("spark.cores.max", str(cores))
        .config("spark.sql.shuffle.partitions", str(cores * 4))
        .getOrCreate()
    )

    spark.sparkContext.setLogLevel("ERROR")
    results = {}

    # ----------------------------
    # Test 1: Load + cache
    # ----------------------------
    print("Test 1: Loading...")
    start = time.time()

    # [CHANGED] Load each month separately and normalize schema
    df_jan =
spark.read.parquet("data/yellow_tripdata_2023-01.parquet")
    df_feb =
spark.read.parquet("data/yellow_tripdata_2023-02.parquet")
    df_mar =
```

```
spark.read.parquet("data/yellow_tripdata_2023-03.parquet")

    common_cols = df_jan.columns
    df_feb = df_feb.select(common_cols)
    df_mar = df_mar.select(common_cols)

    df_jan = df_jan.withColumn("passenger_count",
col("passenger_count").cast("double"))
    df_feb = df_feb.withColumn("passenger_count",
col("passenger_count").cast("double"))
    df_mar = df_mar.withColumn("passenger_count",
col("passenger_count").cast("double"))

    df = df_jan.unionByName(df_feb).unionByName(df_mar)

    df.count()
    results["load_time"] = time.time() - start
    print(f" ✓ {results['load_time']:.2f}s")

    # Repartition across executors and cache
    df = df.repartition(cores * 4).cache()
    df.count()  # materialize cache

    # ----------------------------
    # Test 2: Aggregation
    # ----------------------------
    print("Test 2: Aggregation...")
    start = time.time()

    (
        df.groupBy("payment_type")
        .agg(
            spark_count("*").alias("cnt"),
            avg("fare_amount").alias("avg_fare")
        )
        .collect()
    )

    results["aggregation_time"] = time.time() - start
    print(f" ✓ {results['aggregation_time']:.2f}s")

    # ----------------------------
```

```
    # Test 3: Filter
    # ----------------------------
    print("Test 3: Filter...")
    start = time.time()

    df.filter(col("fare_amount") > 50).limit(10000).collect()

    results["filter_time"] = time.time() - start
    print(f" ✓ {results['filter_time']:.2f}s")

    # ----------------------------
    # Test 4: Join
    # ----------------------------
    print("Test 4: Join...")
    start = time.time()

    df_left = df.sample(0.02).repartition(cores, "passenger_count")
    df_right = df.sample(0.02).repartition(cores, "passenger_count")

    df_left.join(df_right, "passenger_count").count()

    results["join_time"] = time.time() - start
    print(f" ✓ {results['join_time']:.2f}s")

    results["total_time"] = sum(results.values())

    spark.stop()
    time.sleep(5)  # small pause between cluster runs
    return results


# ===== MAIN: Cluster Benchmark Loop + Results =====

print("=" * 70)
print("CLUSTER PERFORMANCE BENCHMARK")
print("=" * 70)

configurations = [2, 4, 6]
all_results = {}

for cores in configurations:
    all_results[cores] = run_cluster_benchmark(cores,
```

```
f"Cluster-{cores}Cores")

# Print comparison table
print("\n" + "=" * 85)
print("CLUSTER COMPARISON")
print("=" * 85)
print(f"{'Test':<20} {'2 Cores':>12} {'4 Cores':>12} {'6 Cores':>12}
{'Speedup':>12}")
print("-" * 85)

for test in ["load_time", "aggregation_time", "filter_time",
"join_time", "total_time"]:
    times = [all_results[cores][test] for cores in configurations]
    speedup = times[0] / times[2]  # 2-core / 6-core
    print(
        f"{test:<20} "
        f"{times[0]:>10.2f}s {times[1]:>10.2f}s {times[2]:>10.2f}s
{speedup:>11.2f}x"
    )

print("=" * 85)

# Save results
with open("output/cluster_results.txt", "w") as f:
    f.write("Cluster Benchmark Results\n")
    f.write("=" * 50 + "\n\n")
    for cores in configurations:
        f.write(f"{cores} Cores:\n")
        for test, val in all_results[cores].items():
            f.write(f"  {test}: {val:.2f}s\n")
        f.write("\n")

print("\nResults saved to output/cluster_results.txt")
PYEOF
```

Part 5: Run Cluster Benchmark

jps | grep -E "Master|Worker" # Verify cluster running python3 activity2_cluster.py

```
(venv) madeel@bdacourse:~/spark-kafka-lab$ jps | grep -E "Master|Worker"
657422 Worker
656307 Master
(venv) madeel@bdacourse:~/spark-kafka-lab$
```

```
$ python3 activity2_cluster.py
======================================================================
CLUSTER PERFORMANCE BENCHMARK
======================================================================

======================================================================
CLUSTER BENCHMARK: 2 Cores
======================================================================
25/11/27 20:37:39 WARN Utils: Your hostname, bdacourse resolves to a
loopback address: 127.0.1.1; using 172.17.5.42 instead (on interface
ens18)
25/11/27 20:37:39 WARN Utils: Set SPARK_LOCAL_IP if you need to bind
to another address
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
25/11/27 20:37:40 WARN NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where
applicable
Test 1: Loading...
 ✓ 8.44s
Test 2: Aggregation...
 ✓ 2.18s
Test 3: Filter...
 ✓ 0.80s
Test 4: Join...
 ✓ 223.84s

======================================================================
CLUSTER BENCHMARK: 4 Cores
======================================================================
Test 1: Loading...
 ✓ 4.52s
Test 2: Aggregation...
 ✓ 2.29s
Test 3: Filter...
 ✓ 0.44s
Test 4: Join...
 ✓ 220.93s

======================================================================
CLUSTER BENCHMARK: 6 Cores
```

```
================================================================================
Test 1: Loading...
 ✓ 4.36s
Test 2: Aggregation...
 ✓ 1.49s
Test 3: Filter...
 ✓ 0.43s
Test 4: Join...
 ✓ 216.94s

================================================================================
===============
CLUSTER COMPARISON
================================================================================
===============
Test                        2 Cores      4 Cores      6 Cores
Speedup
--------------------------------------------------------------------------------
---------------
load_time                     8.44s        4.52s        4.36s         1.94x
aggregation_time              2.18s        2.29s        1.49s         1.46x
filter_time                   0.80s        0.44s        0.43s         1.85x
join_time                   223.84s      220.93s      216.94s         1.03x
total_time                  235.27s      228.18s      223.22s         1.05x
================================================================================
===============

Results saved to output/cluster_results.txt
```

Part 6: View Results File

cat output/single_node_results.txt

cat output/cluster_results.txt

REQUIRED SCREENSHOTS

Screenshot 2.1: Terminal showing single-node benchmark comparison table (1, 2, 4 cores)

```
======================================================================
Test 1: Loading Data...
 ✓ Loaded 9,384,487 records in 0.59s
Test 2: Aggregation...
 ✓ Completed in 1.42s
Test 3: Filter & Sort...
 ✓ Completed in 1.71s
Test 4: Join...
 ✓ Completed in 23.57s

================================================================================
PERFORMANCE COMPARISON
================================================================================
Test                        1 Core      2 Cores      4 Cores      Speedup
--------------------------------------------------------------------------------
load_time                   6.41s       0.69s       0.59s       10.81x
aggregation_time            2.00s       1.26s       1.42s        1.40x
filter_time                 3.10s       2.23s       1.71s        1.81x
join_time                  63.37s      42.21s      23.57s        2.69x
total_time                 74.88s      46.39s      27.30s        2.74x
================================================================================

Results saved to output/single_node_results.txt
(venv) madeel@bdacourse:~/spark-kafka-lab$
```

Screenshot 2.2: Terminal showing cluster benchmark comparison table (2, 4, 6 cores)

```
======================================================================
Test 1: Loading...
 ✓ 4.36s
Test 2: Aggregation...
 ✓ 1.49s
Test 3: Filter...
 ✓ 0.43s
Test 4: Join...
 ✓ 216.94s

================================================================================
CLUSTER COMPARISON
================================================================================
Test                        2 Cores      4 Cores      6 Cores      Speedup
--------------------------------------------------------------------------------
load_time                   8.44s       4.52s       4.36s        1.94x
aggregation_time            2.18s       2.29s       1.49s        1.46x
filter_time                 0.80s       0.44s       0.43s        1.85x
join_time                 223.84s     220.93s     216.94s        1.03x
total_time                235.27s     228.18s     223.22s        1.05x
================================================================================

Results saved to output/cluster_results.txt
```

Screenshot 2.3: Content of output files showing detailed timing for all core configurations

```
(venv) madeel@bdacourse:~/spark-kafka-lab$ cat output/single_node_results.txt
Single Node Benchmark Results
==================================================

1 Core(s):
  load_time: 6.41s
  aggregation_time: 2.00s
  filter_time: 3.10s
  join_time: 63.37s
  total_time: 74.88s

2 Core(s):
  load_time: 0.69s
  aggregation_time: 1.26s
  filter_time: 2.23s
  join_time: 42.21s
  total_time: 46.39s

4 Core(s):
  load_time: 0.59s
  aggregation_time: 1.42s
  filter_time: 1.71s
  join_time: 23.57s
  total_time: 27.30s

(venv) madeel@bdacourse:~/spark-kafka-lab$
```

```
(venv) madeel@bdacourse:~/spark-kafka-lab$ cat output/cluster_results.txt
Cluster Benchmark Results
==================================================

2 Cores:
  load_time: 8.44s
  aggregation_time: 2.18s
  filter_time: 0.80s
  join_time: 223.84s
  total_time: 235.27s

4 Cores:
  load_time: 4.52s
  aggregation_time: 2.29s
  filter_time: 0.44s
  join_time: 220.93s
  total_time: 228.18s

6 Cores:
  load_time: 4.36s
  aggregation_time: 1.49s
  filter_time: 0.43s
  join_time: 216.94s
  total_time: 223.22s

(venv) madeel@bdacourse:~/spark-kafka-lab$
```

CRITICAL QUESTIONS

Question 1: Which operation (load, aggregation, filter, join) showed the best speedup and why? Which operation showed the least improvement with more cores?

Test	1 Core	4 Cores	Speedup
load_time	6.41 s	0.59 s	10.81x
aggregation	2.00 s	1.42 s	1.40x
filter	3.10 s	1.71 s	1.81x
join	63.37 s	23.57 s	2.69x

In our single-node results, the highest measured speedup from 1 to 4 cores is on the load step (about 10.8×). However, this is mainly because the first run includes Spark/JVM startup and cold disk reads, while later runs benefit from warm caches.

Among the true processing operations, the join shows the best scaling (around 2.7× faster), since joins are very CPU- and shuffle-intensive and can exploit more parallel tasks.

The aggregation step shows the least improvement (about 1.4×), because a lot of its cost is in combining partial aggregates and job overhead, which doesn't parallelize as well, so extra cores give only moderate gains.

Question 2: Compare your single-node 4-core results with cluster 2-core results. Is cluster mode always faster? Explain the trade-offs between single-node and cluster execution.

In our experiment, the single-node 4-core run (≈27.3 s total) was much faster than the 2-core cluster run (≈235.3 s). So cluster mode was not faster here.

The reason is that the cluster setup adds a lot of overhead: communication with the Spark master, starting executors, serialization, and distributed shuffles. In this lab, the "cluster" is effectively on the same machine, so I don't gain extra physical hardware, but I still pay the distributed-system overhead, especially during the join step.

A single machine with 4 cores can process this dataset entirely in memory with low overhead, so it outperforms a small cluster. Cluster execution only becomes clearly beneficial when the data volume is large enough or when we have multiple real nodes and need to scale beyond the capacity of one machine.

Test Operation	Single-Node (4 Cores)	Cluster (2 Cores)	Faster Setup

Load	0.59 s	8.44 s	**Single-node**
Aggregation	1.42 s	2.18 s	**Single-node**
Filter	1.71 s	0.80 s	**Cluster**
Join	23.57 s	223.84 s	**Single-node**
Total Time	**27.30 s**	**235.27 s**	**Single-node**

From the table, single-node execution with 4 cores is clearly faster overall (27.30s) compared to the cluster with 2 cores (235.27s).

The only exception is the filter operation, where the cluster performs slightly better. However, the join operation dominates total time and is much slower on the cluster, making the overall performance worse.

This shows that cluster execution is not always faster; for moderate datasets that fit in memory, a single multi-core machine can outperform a distributed cluster due to lower overhead.

Activity 3: Kafka Producer-Consumer & Real-Time Anomaly Detection

Objectives: Implement Kafka producer and consumer, detect anomalies in streaming sensor data

Part 1: Verify Kafka Topics

kafka-topics.sh --list --bootstrap-server localhost:9092 # Should show: sensor-data, transactions, alerts

```
(venv) madeel@bdacourse:~/spark-kafka-lab$ kafka-topics.sh --list --bootstrap-server localhost:9092
alerts
sensor-data
transactions
(venv) madeel@bdacourse:~/spark-kafka-lab$
```

Part 2: Create Kafka Producer

cat > activity3_producer.py << 'PYEOF' from kafka import KafkaProducer import json, time, random
from datetime import datetime from faker import Faker

```
fake = Faker()
producer = KafkaProducer( bootstrap_servers=['localhost:9092'], value_serializer=lambda v:
   json.dumps(v).encode('utf-8')
)

def gen_sensor(): return {
       'sensor_id': f"SENSOR_{random.randint(1, 20):03d}", 'timestamp':
       datetime.now().isoformat(), 'temperature': round(random.uniform(15, 35), 2),
       'humidity': round(random.uniform(30, 80), 2),
       'pressure': round(random.uniform(980, 1050), 2),
       'status': random.choice(['normal', 'normal', 'normal', 'warning', 'critical'])
   }

print("=" * 70)
print("Kafka Producer - Sending Sensor Data") print("=" * 70)

for i in range(500):
msg = gen_sensor() producer.send('sensor-data', msg)

   if (i + 1) % 100 == 0:
       print(f"Sent {i+1} messages...") time.sleep(0.05)

producer.flush() producer.close() print("
✓       Complete! Sent 500 sensor readings") PYEOF
```

```
cat > activity3_producer.py << 'PYEOF'
from kafka import KafkaProducer
import json, time, random
from datetime import datetime
from faker import Faker

fake = Faker()

# Kafka producer: sends Python dict as JSON to localhost:9092
producer = KafkaProducer(
    bootstrap_servers=['localhost:9092'],
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

def gen_sensor():
    """
    Generate one fake sensor reading:
    - sensor_id: SENSOR_001 ... SENSOR_020
    - timestamp: current time in ISO format
    - temperature: 15–35 °C
    - humidity: 30–80 %
    - pressure: 980–1050 hPa
    - status: mostly 'normal', sometimes 'warning' or 'critical'
    """
    return {
        'sensor_id': f"SENSOR_{random.randint(1, 20):03d}",
        'timestamp': datetime.now().isoformat(),
        'temperature': round(random.uniform(15, 35), 2),
        'humidity': round(random.uniform(30, 80), 2),
        'pressure': round(random.uniform(980, 1050), 2),
              'status':  random.choice(['normal',  'normal',  'normal',
'warning', 'critical'])
    }

print("=" * 70)
print("Kafka Producer - Sending Sensor Data")
print("=" * 70)

# Send 500 sensor messages
for i in range(500):
    msg = gen_sensor()
    producer.send('sensor-data', msg)
```

```
    if (i + 1) % 100 == 0:
        print(f"Sent {i+1} messages...")

    time.sleep(0.05)

producer.flush()
producer.close()
print("\nComplete! Sent 500 sensor readings")
PYEOF
```

Part 3: Create Kafka Consumer with Anomaly Detection

cat > activity3_consumer.py << 'PYEOF' from kafka import KafkaConsumer import json, signal, sys

consumer = KafkaConsumer('sensor-data',
bootstrap_servers=['localhost:9092'], auto_offset_reset='earliest',
group_id='anomaly-detector',
value_deserializer=lambda x: json.loads(x.decode('utf-8'))
)

def signal_handler(sig, frame): consumer.close() sys.exit(0)
signal.signal(signal.SIGINT, signal_handler) print("=" * 70)

print("Kafka Consumer - Real-Time Anomaly Detection")
print("=" * 70)
print("Listening for sensor data... (Press Ctrl+C to stop) ")

count = 0
anomaly_count = 0
for msg in consumer: data = msg.value count += 1

Anomaly Detection Logic is_anomaly = False reasons = []

if data['temperature'] > 32: is_anomaly = True
reasons.append(f"High temp: {data['temperature']}°C")

if data['status'] == 'critical': is_anomaly = True reasons.append(f"Critical status")

if is_anomaly: anomaly_count += 1 print(f"
⚠ ANOMALY DETECTED (#{anomaly_count})") print(f" Sensor: {data['sensor_id']}")
print(f" Time: {data['timestamp']}")
print(f" Reasons: {', '.join(reasons)}") print(f" Full Data: {data}")

```
   if count % 50 == 0: print(f"
--- Processed: {count} | Anomalies: {anomaly_count} ---") PYEOF

cat > activity3_consumer.py << 'PYEOF'
from kafka import KafkaConsumer
import json, signal, sys

# Create Kafka consumer listening to 'sensor-data'
consumer = KafkaConsumer(
    'sensor-data',
    bootstrap_servers=['localhost:9092'],
    auto_offset_reset='earliest',      # start from earliest messages
if no offset
    group_id='anomaly-detector',       # consumer group name
    value_deserializer=lambda x: json.loads(x.decode('utf-8'))
)

def signal_handler(sig, frame):
    """Graceful shutdown when you press Ctrl+C."""
    consumer.close()
    sys.exit(0)

signal.signal(signal.SIGINT, signal_handler)

print("=" * 70)
print("Kafka Consumer - Real-Time Anomaly Detection")
print("=" * 70)
print("Listening for sensor data... (Press Ctrl+C to stop)\n")

count = 0
anomaly_count = 0

for msg in consumer:
    data = msg.value      # this is the dict we sent from the producer
    count += 1

    # -------------------------
    # Anomaly Detection Logic
    # -------------------------
    is_anomaly = False
    reasons = []
```

```
    # Condition 1: High temperature
    if data['temperature'] > 32:
        is_anomaly = True
        reasons.append(f"High temp: {data['temperature']}°C")

    # Condition 2: Critical status
    if data['status'] == 'critical':
        is_anomaly = True
        reasons.append("Critical status")

    # Condition 3: (EXTRA) very high humidity
    if data['humidity'] > 75:  # //extra — new anomaly rule
        is_anomaly = True      # //extra
        reasons.append(f"High humidity: {data['humidity']}%")  #
//extra

    # If any condition triggered, print anomaly
    if is_anomaly:
        anomaly_count += 1
        print(f"\nANOMALY DETECTED (#{anomaly_count})")
        print(f"  Sensor:  {data['sensor_id']}")
        print(f"  Time:    {data['timestamp']}")
        print(f"  Reasons: {', '.join(reasons)}")
        print(f"  Full Data: {data}")

    # Progress log every 50 messages
    if count % 50 == 0:
        print(f"\n--- Processed: {count} | Anomalies: {anomaly_count}
---")
PYEOF
```

Part 4: Run Producer and Consumer

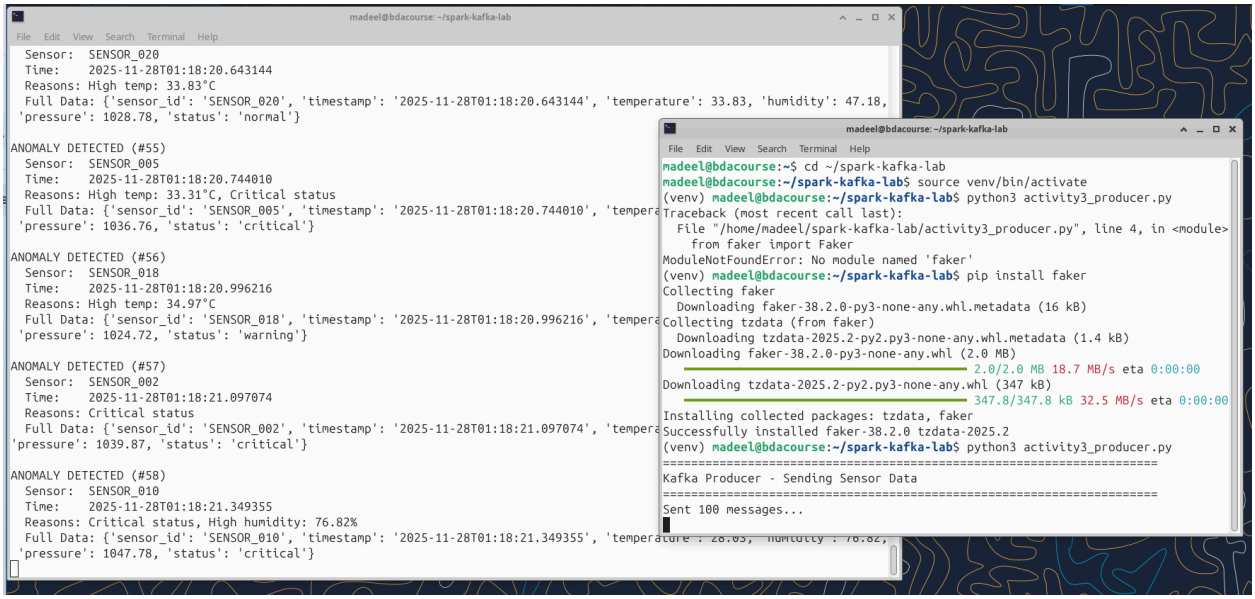
Terminal 1 - Consumer (Start First):

python3 activity3_consumer.py

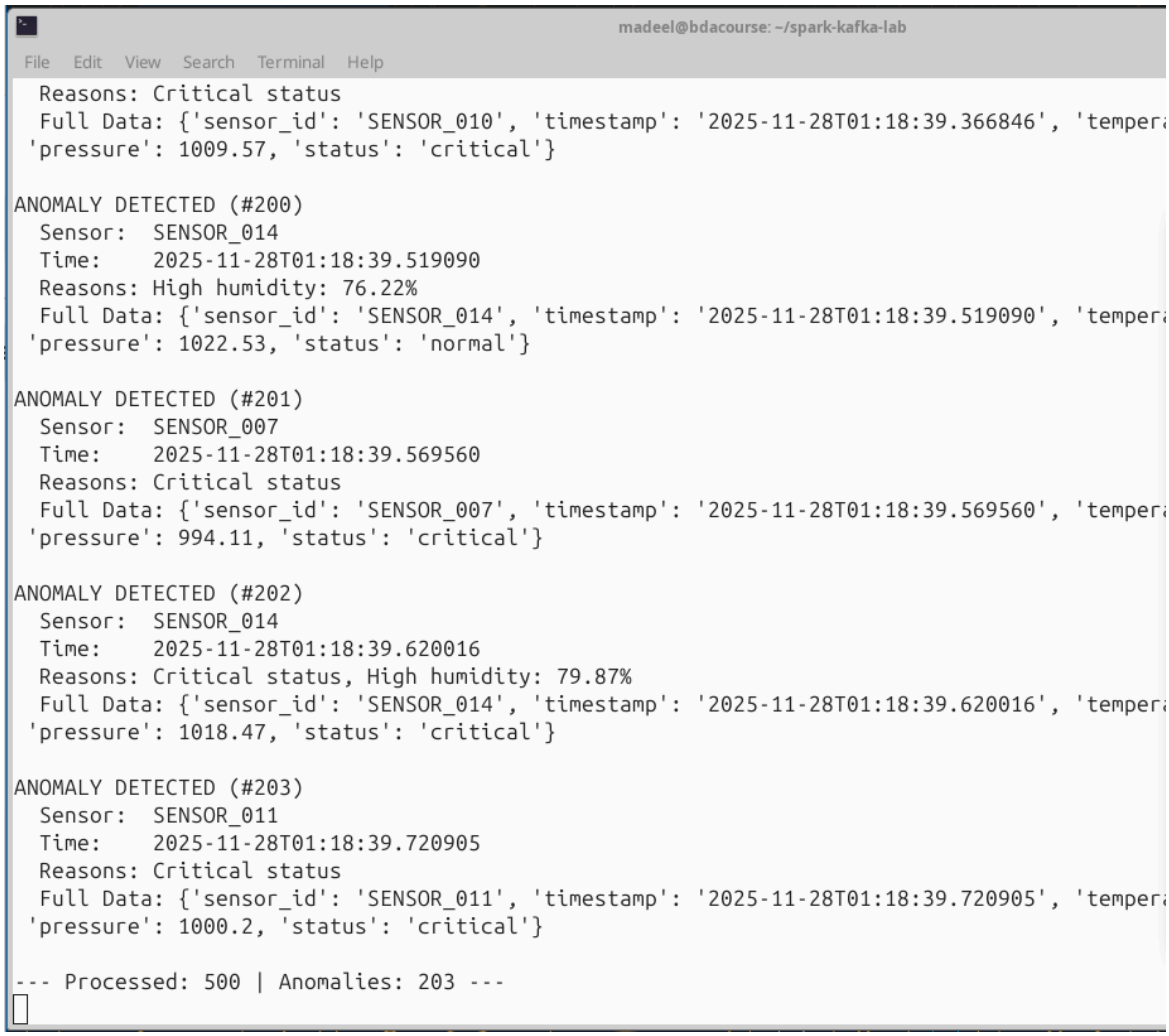
Terminal 2 - Producer:

python3 activity3_producer.py

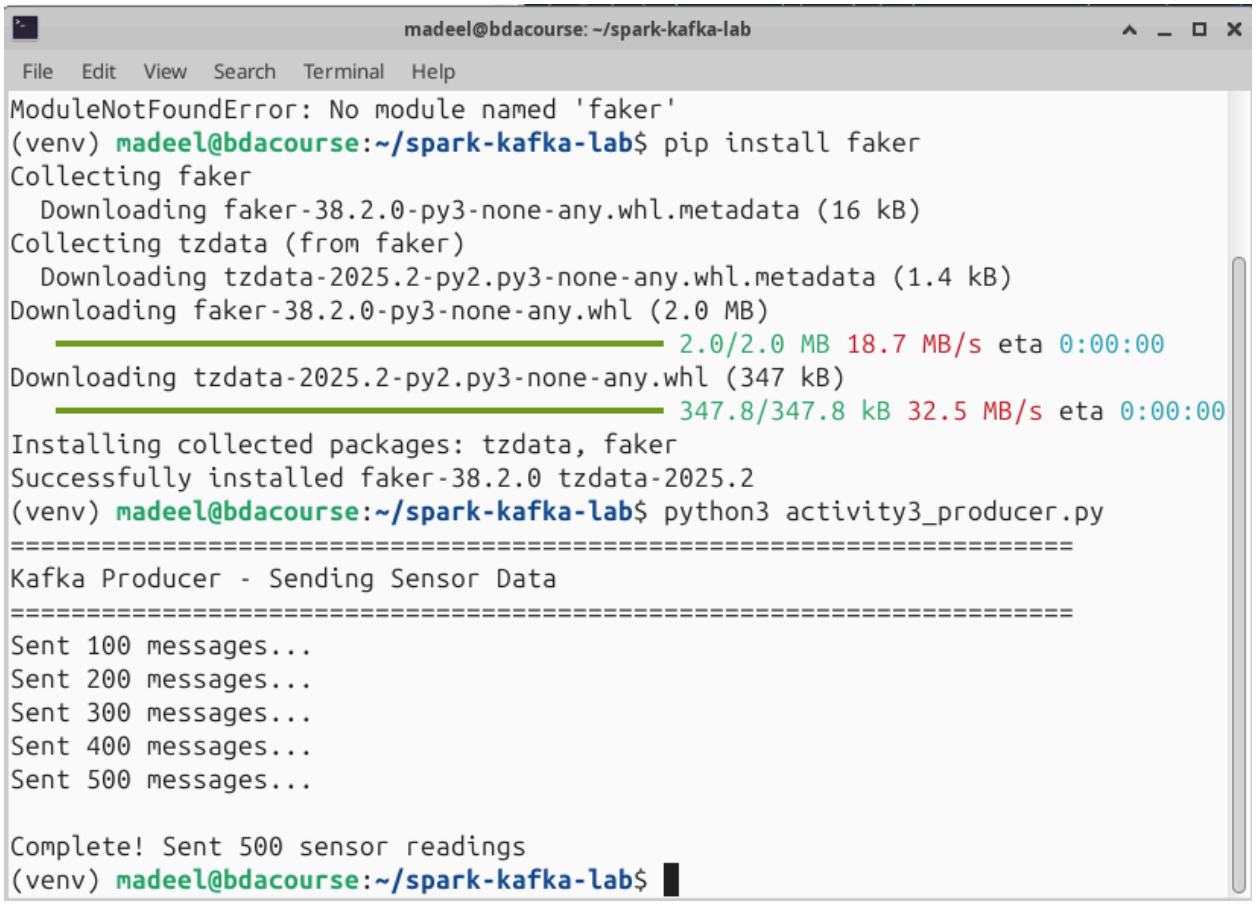
REQUIRED SCREENSHOTS



Screenshot 3.1: Terminal 1 (Consumer) showing anomaly detection in action with multiple anomalies detected



Screenshot 3.2: Terminal 2 (Producer) showing messages being sent successfully



CRITICAL QUESTIONS

Question 1: Explain how Kafka's producer-consumer model works. What happens if the consumer is slower than the producer?

Producer–consumer decoupling:
In Kafka, producers send messages to *topics* on the broker (e.g. `sensor-data`). Consumers subscribe to those topics and read messages independently. Producers don't know who is consuming; they just append data to the log.

Offsets and consumer groups:
Each consumer in a consumer group keeps track of an *offset* in each partition (i.e., "where I last read"). Kafka stores messages on disk for a configured retention time, and consumers can read at their own speed, replay, or resume from the saved offset.

If the consumer is slower than the producer:

- Messages will accumulate in Kafka; this is seen as **consumer lag** (producer keeps writing new offsets faster than consumer reads them).

- As long as Kafka has enough storage and retention time hasn't expired, the consumer can eventually catch up.

- If the consumer is *too* slow for a long time, data may get deleted by retention before being read, or partitions may fill disk, so some messages would effectively be lost for that consumer.

Question 2: What anomalies did your system detect? Modify the anomaly detection logic to add one more condition (e.g., humidity threshold) and explain why that would be useful.

High temperature:

- Example:
  `High temp: 34.18°C` for `SENSOR_014` and many others.
- Logic in code: `if data['temperature'] > 32: ...`
- This catches overheating or abnormally hot environments around the sensor.

Critical status from the sensor:

- Example:
  `Reasons: Critical status` for sensors like `SENSOR_010`, `SENSOR_009`, `SENSOR_020`, etc.
- Logic: `if data['status'] == 'critical': ...`
- This flags readings where the device itself reports a critical condition (e.g., internal error or serious fault).

(added by us) High humidity condition:

- Example anomalies you saw:
  - `High humidity: 76.95%` for `SENSOR_017`
  - `High humidity: 79.28%` for `SENSOR_018`
  - `High temp: 34.1°C, High humidity: 77.89%` for `SENSOR_018`
  - `High humidity: 75.07%` for `SENSOR_008`

Code we added:

```
if data['humidity'] > 75:
    is_anomaly = True
    reasons.append(f"High humidity: {data['humidity']}%")
```

- **Why this is useful:**
  - Very high humidity can indicate risky environmental conditions (condensation, corrosion, mold, or risk to electronic equipment).
  - Combined with high temperature, it's especially dangerous (heat + humidity = more stress on hardware and people).
  - Adding this rule means the system doesn't rely only on temperature/status; it can also catch "silent" environmental issues where temperature is normal but humidity is becoming unsafe.

Activity 4: Spark Structured Streaming with Kafka

Objectives: Use Spark to consume Kafka streams, perform windowed aggregations, write to output topic
Time: 30 minutes

Part 1: Create Spark Streaming Script

```
cat > activity4_streaming.py << 'PYEOF' from pyspark.sql import SparkSession from
pyspark.sql.functions import *
from pyspark.sql.types import *

schema = StructType([ StructField("sensor_id", StringType()), StructField("timestamp",
   StringType()), StructField("temperature", DoubleType()), StructField("humidity",
   DoubleType()), StructField("pressure", DoubleType()), StructField("status", StringType())
])

spark = SparkSession.builder \
   .appName("Activity4-Streaming") \
   .master("spark://localhost:7077") \
   .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0") \
   .getOrCreate()

spark.sparkContext.setLogLevel("WARN") print("=" * 70)
print("Spark Structured Streaming - Kafka Integration") print("=" * 70)
print("Reading from topic: sensor-data ")

# Read stream
raw = spark.readStream \
   .format("kafka") \
   .option("kafka.bootstrap.servers", "localhost:9092") \
   .option("subscribe", "sensor-data") \
   .option("startingOffsets", "latest") \
   .load()

# Parse JSON
parsed = raw.selectExpr("CAST(value AS STRING)") \
   .select(from_json(col("value"), schema).alias("data")) \
   .select("data.*") \
   .withColumn("event_time", to_timestamp(col("timestamp")))

# Filter for alerts
alerts = parsed.filter((col("temperature") > 30) | (col("status") == "critical")) \
   .select(col("sensor_id"), col("temperature"), col("status")) \
```

```
    .withColumn("alert_msg", concat(lit("ALERT: "), col("sensor_id"),
                                    lit(" - temp="), col("temperature"),
                                    lit(" status="), col("status")))

# Windowed aggregation (5-second windows) windowed = parsed \
    .withWatermark("event_time", "5 seconds") \
    .filter(col("status") == "critical") \
    .groupBy(window(col("event_time"), "5 seconds"), col("sensor_id")) \
    .count() \
    .withColumnRenamed("count", "critical_count")

# Write alerts to Kafka kafka_output = alerts.select(
    col("sensor_id").cast("string").alias("key"),
    to_json(struct("sensor_id", "temperature", "status", "alert_msg")).alias("value")
)

query1 = kafka_output.writeStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("topic", "alerts") \
    .option("checkpointLocation", "checkpoint/alerts") \
    .start()

# Write windowed counts to console query2 = windowed.writeStream \
    .format("console") \
    .outputMode("update") \
    .option("truncate", "false") \
    .start()
    print("Streaming queries active... Press Ctrl+C to stop") spark.streams.awaitAnyTermination()
PYEOF

cat > activity4_streaming.py << 'PYEOF'
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *

# Schema of the JSON messages coming from Kafka
schema = StructType([
    StructField("sensor_id", StringType()),
    StructField("timestamp", StringType()),
    StructField("temperature", DoubleType()),
    StructField("humidity", DoubleType()),
    StructField("pressure", DoubleType()),
    StructField("status", StringType())
])

# SparkSession connected to the Spark standalone cluster
spark = (
    SparkSession.builder
```

```
    .appName("Activity4-Streaming")
    .master("spark://localhost:7077")
    .config(
        "spark.jars.packages",
        "org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0"
    )
    .getOrCreate()
)

spark.sparkContext.setLogLevel("WARN")

print("=" * 70)
print("Spark Structured Streaming - Kafka Integration")
print("=" * 70)
print("Reading from topic: sensor-data")

# --------------------------
# 1) Read stream from Kafka
# --------------------------
raw = (
    spark.readStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "localhost:9092")
    .option("subscribe", "sensor-data")
    .option("startingOffsets", "latest")
    .load()
)

# 'raw' schema: key (binary), value (binary), topic, partition,
offset, timestamp, etc.

# -----------------------------------
# 2) Parse JSON from Kafka 'value'
# -----------------------------------
parsed = (
    raw.selectExpr("CAST(value AS STRING)")
       .select(from_json(col("value"), schema).alias("data"))
       .select("data.*")
       .withColumn("event_time", to_timestamp(col("timestamp")))
)

# -----------------------------------
# 3) Build "alerts" stream
#    - Filter high temp OR critical status
#    - Create an alert message string
# -----------------------------------
alerts = (
    parsed
```

```
    .filter((col("temperature") > 30) | (col("status") == "critical"))
    .select(col("sensor_id"), col("temperature"), col("status"))
    .withColumn(
        "alert_msg",
        concat(
            lit("ALERT: "), col("sensor_id"),
            lit(" - temp="), col("temperature"),
            lit(" status="), col("status")
        )
    )
)

# -------------------------------------
# 4) Windowed aggregation (5-second windows)
#    - Count how many 'critical' statuses per sensor_id
#    - Use watermark to bound late events
# -------------------------------------
windowed = (
    parsed
    .withWatermark("event_time", "5 seconds")
    .filter(col("status") == "critical")
    .groupBy(window(col("event_time"), "5 seconds"), col("sensor_id"))
    .count()
    .withColumnRenamed("count", "critical_count")
)

# -------------------------------------
# 5) Prepare alerts to be written back to Kafka
#    - key: sensor_id (string)
#    - value: JSON with sensor_id, temperature, status, alert_msg
# -------------------------------------
kafka_output = alerts.select(
    col("sensor_id").cast("string").alias("key"),
    to_json(struct("sensor_id", "temperature", "status",
"alert_msg")).alias("value")
)

# Write alerts to Kafka topic "alerts"
query1 = (
    kafka_output.writeStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "localhost:9092")
    .option("topic", "alerts")
    .option("checkpointLocation", "checkpoint/alerts")
    .start()
)

# Write windowed counts to console
```

```
query2 = (
    windowed.writeStream
    .format("console")
    .outputMode("update")
    .option("truncate", "false")
    .start()
)

print("Streaming queries active... Press Ctrl+C to stop")
spark.streams.awaitAnyTermination()
PYEOF
```

Part 2: Run Streaming Pipeline (3 Terminals)

Terminal 1 - Start Spark Streaming:

rm -rf checkpoint/
python3 activity4_streaming.py

```
        ----------------------------------------------------------------
:: retrieving :: org.apache.spark#spark-submit-parent-d1c219be-046a-4be0-8964-8690ba784ee8
        confs: [default]
        11 artifacts copied, 0 already retrieved (56767kB/43ms)
25/11/28 02:04:31 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classe
s where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
================================================================================
Spark Structured Streaming - Kafka Integration
================================================================================
Reading from topic: sensor-data
25/11/28 02:04:35 WARN ResolveWriteToStream: spark.sql.adaptive.enabled is not supported in streaming DataFrames/Datasets an
d will be disabled.
25/11/28 02:04:35 WARN ResolveWriteToStream: Temporary checkpoint location created which is deleted normally when the query
didn't fail: /tmp/temporary-5e6abd5a-54de-4363-b3d8-ac8e8792df0c. If it's required to delete it under any circumstances, ple
ase set spark.sql.streaming.forceDeleteTempCheckpointLocation to true. Important to know deleting temp checkpoint folder is
best effort.
25/11/28 02:04:35 WARN ResolveWriteToStream: spark.sql.adaptive.enabled is not supported in streaming DataFrames/Datasets an
d will be disabled.
Streaming queries active... Press Ctrl+C to stop
25/11/28 02:04:36 WARN AdminClientConfig: These configurations '[key.deserializer, value.deserializer, enable.auto.commit, m
ax.poll.records, auto.offset.reset]' were supplied but are not used yet.
25/11/28 02:04:36 WARN AdminClientConfig: These configurations '[key.deserializer, value.deserializer, enable.auto.commit, m
ax.poll.records, auto.offset.reset]' were supplied but are not used yet.
-------------------------------------------
Batch: 0
-------------------------------------------
+------+---------+--------------+
|window|sensor_id|critical_count|
+------+---------+--------------+
+------+---------+--------------+
```

Terminal 2 - Start Producer (continuous):

cat > continuous_producer.py << 'PYEOF' from kafka import KafkaProducer
import json, time, random from datetime import datetime

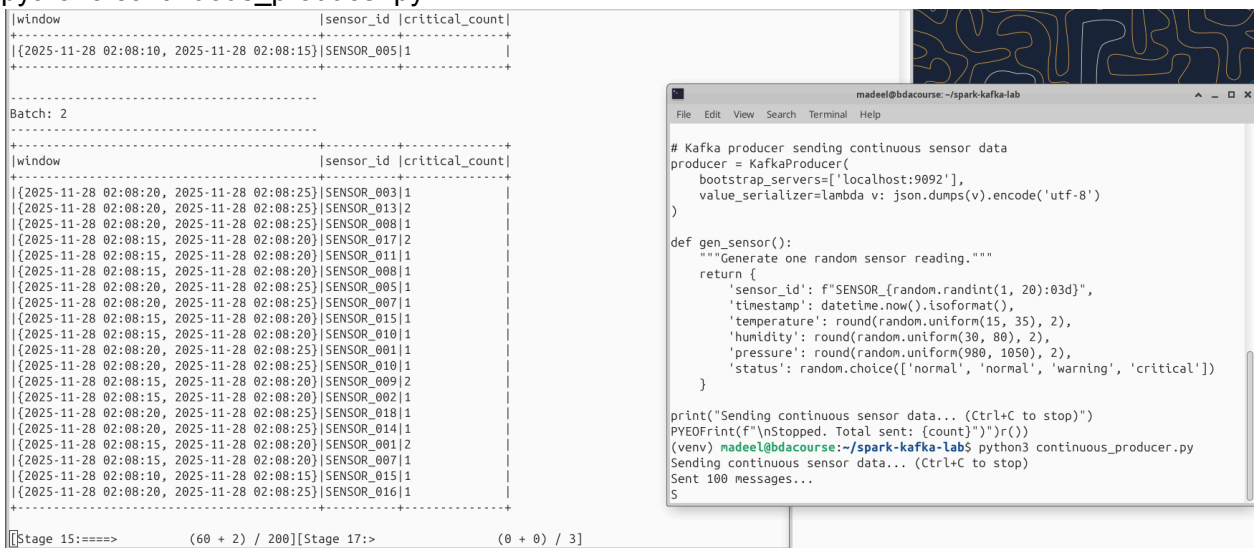
producer = KafkaProducer(bootstrap_servers=['localhost:9092'], value_serializer=lambda v:
  json.dumps(v).encode('utf-8')
)

def gen_sensor(): return {

```
        'sensor_id': f"SENSOR_{random.randint(1, 20):03d}", 'timestamp':
        datetime.now().isoformat(), 'temperature': round(random.uniform(15, 35), 2),
        'humidity': round(random.uniform(30, 80), 2),
        'pressure': round(random.uniform(980, 1050), 2),
        'status': random.choice(['normal', 'normal', 'warning', 'critical'])
    }

print("Sending continuous sensor data... (Ctrl+C to stop)") count = 0
try:
    while True:
        producer.send('sensor-data', gen_sensor()) count += 1
        if count % 100 == 0:
            print(f"Sent {count} messages...") time.sleep(0.1)
            except KeyboardInterrupt: producer.close() print(f"
Stopped. Total sent: {count}") PYEOF

cat > continuous_producer.py << 'PYEOF'
from kafka import KafkaProducer
import json, time, random
from datetime import datetime

# Kafka producer sending continuous sensor data
producer = KafkaProducer(
    bootstrap_servers=['localhost:9092'],
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

def gen_sensor():
    """Generate one random sensor reading."""
    return {
        'sensor_id': f"SENSOR_{random.randint(1, 20):03d}",
        'timestamp': datetime.now().isoformat(),
        'temperature': round(random.uniform(15, 35), 2),
        'humidity': round(random.uniform(30, 80), 2),
        'pressure': round(random.uniform(980, 1050), 2),
        'status': random.choice(['normal', 'normal', 'warning',
'critical'])
    }

print("Sending continuous sensor data... (Ctrl+C to stop)")
count = 0

try:
    while True:
```

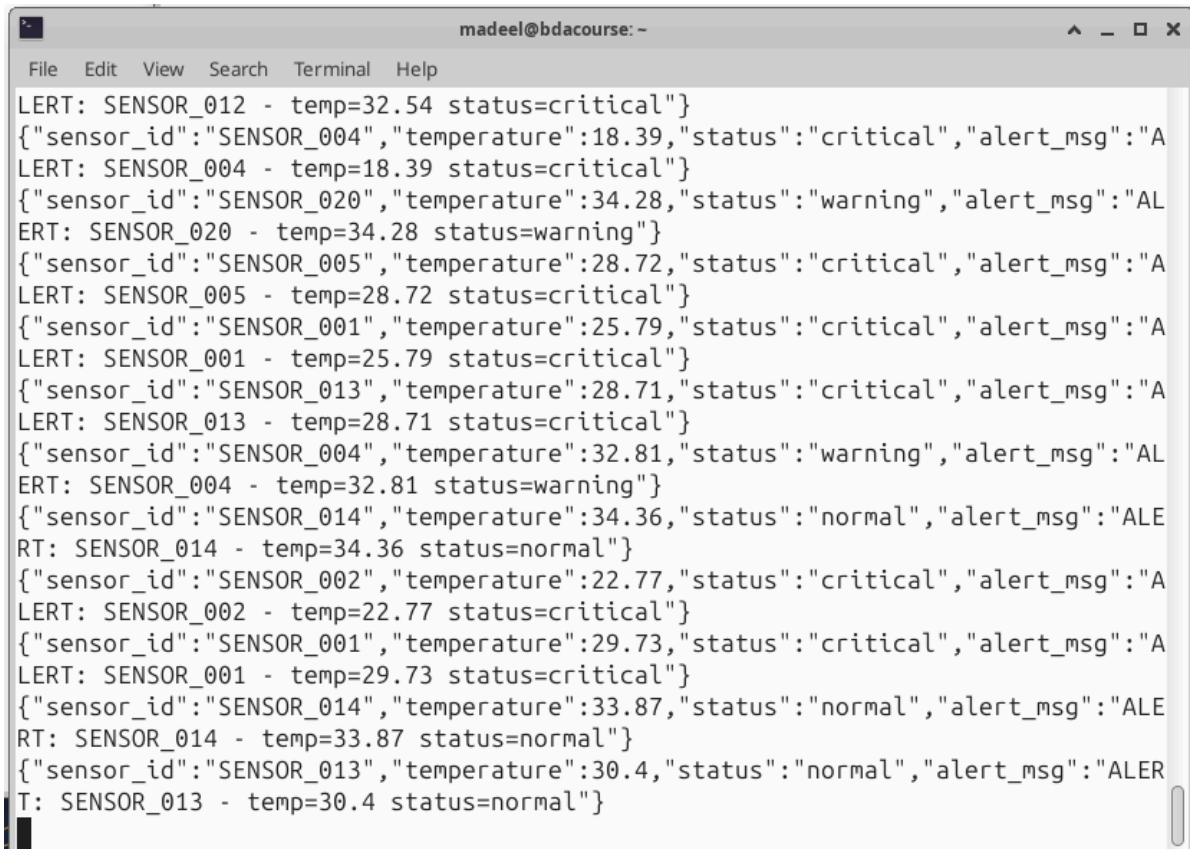
```
        producer.send('sensor-data', gen_sensor())
        count += 1
        if count % 100 == 0:
            print(f"Sent {count} messages...")
        time.sleep(0.1)
except KeyboardInterrupt:
    producer.close()
    print(f"\nStopped. Total sent: {count}")
PYEOF
```

python3 continuous_producer.py



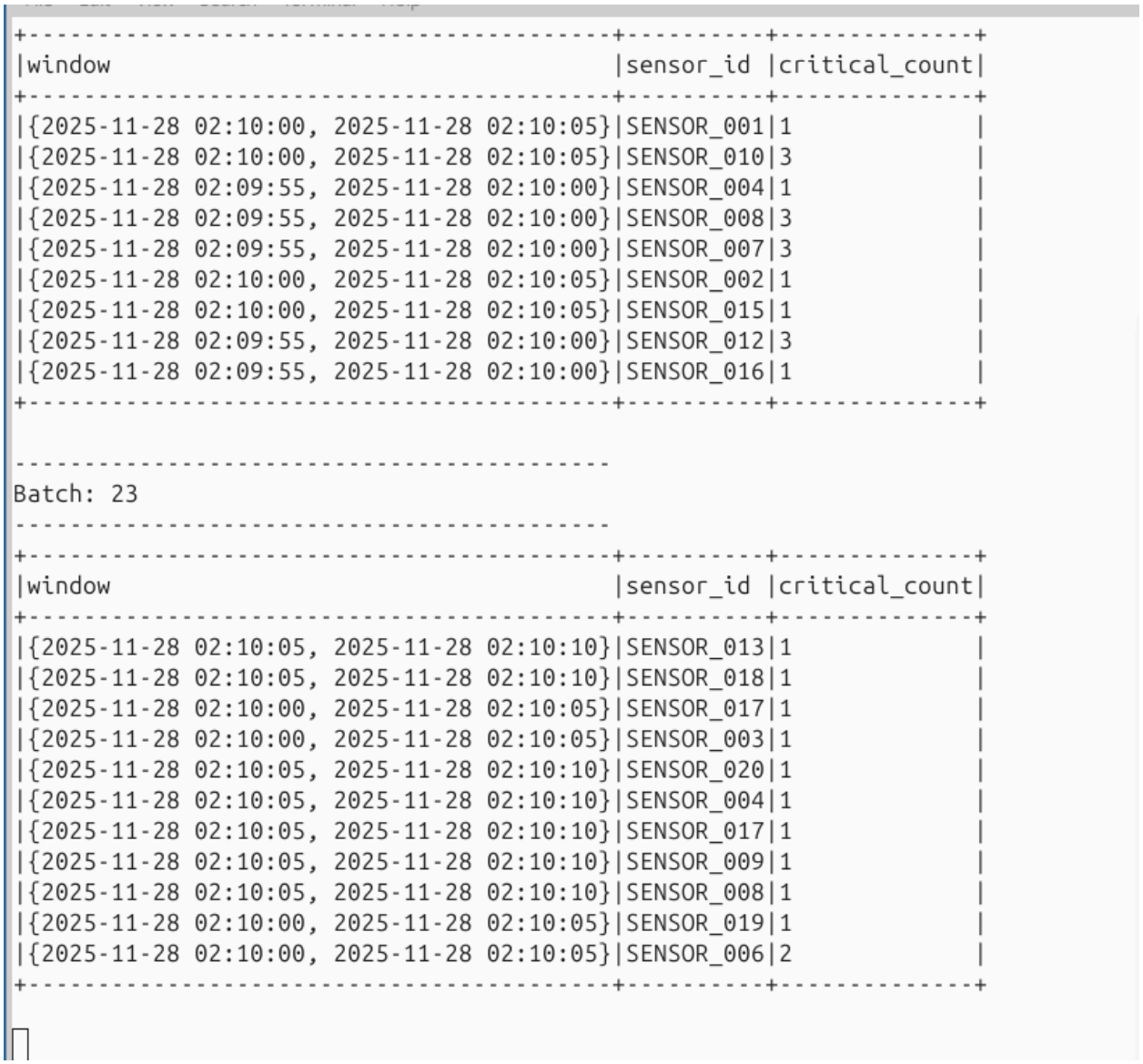
Terminal 3 - Monitor Alerts Topic:

```
kafka-console-consumer.sh \
  --bootstrap-server localhost:9092 \
  --topic alerts \
  --from-beginning
```



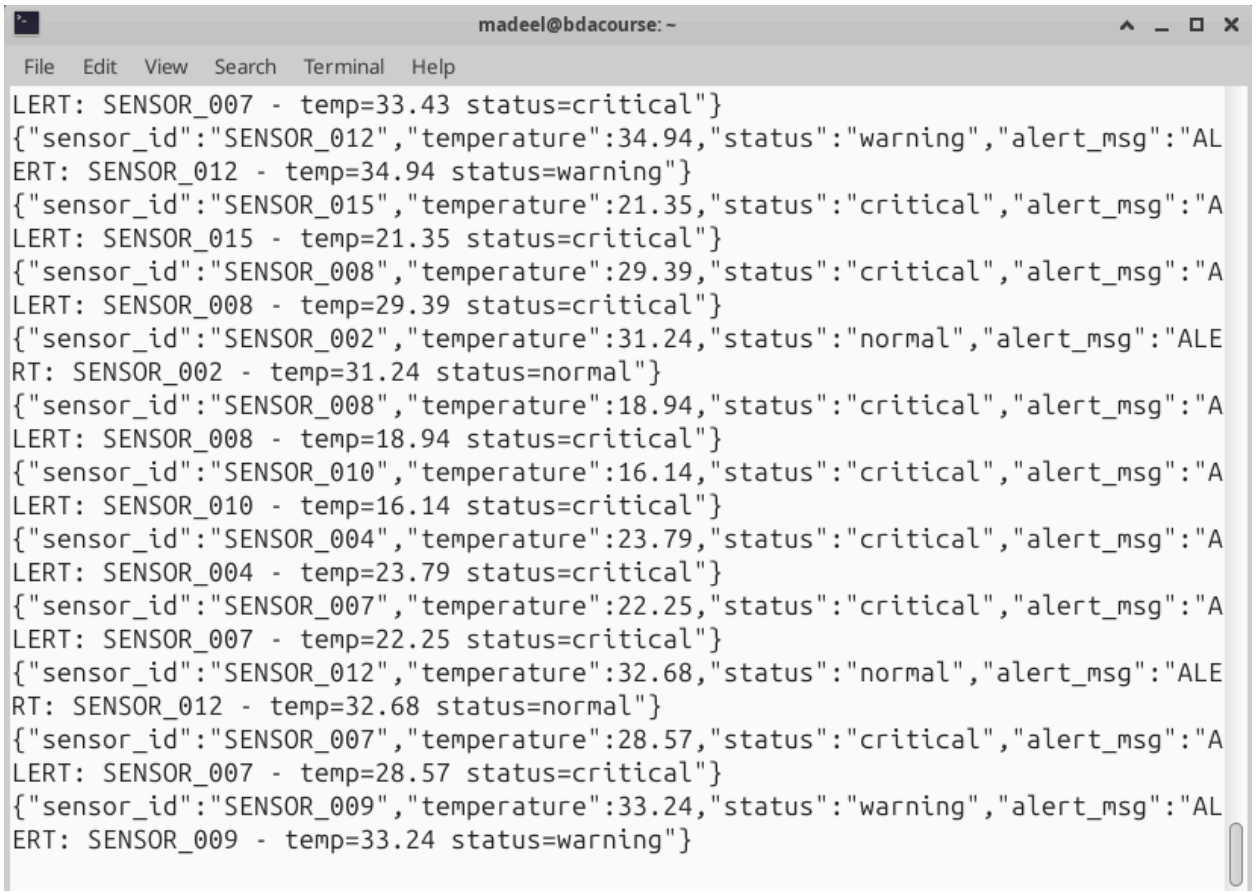
REQUIRED SCREENSHOTS

Screenshot 4.1: Terminal 1 showing Spark Streaming console output with windowed aggregation results (5-second windows showing critical counts)



```
+------------------------------------------+----------+--------------+
|window                                    |sensor_id |critical_count|
+------------------------------------------+----------+--------------+
|{2025-11-28 02:10:00, 2025-11-28 02:10:05}|SENSOR_001|1             |
|{2025-11-28 02:10:00, 2025-11-28 02:10:05}|SENSOR_010|3             |
|{2025-11-28 02:09:55, 2025-11-28 02:10:00}|SENSOR_004|1             |
|{2025-11-28 02:09:55, 2025-11-28 02:10:00}|SENSOR_008|3             |
|{2025-11-28 02:09:55, 2025-11-28 02:10:00}|SENSOR_007|3             |
|{2025-11-28 02:10:00, 2025-11-28 02:10:05}|SENSOR_002|1             |
|{2025-11-28 02:10:00, 2025-11-28 02:10:05}|SENSOR_015|1             |
|{2025-11-28 02:09:55, 2025-11-28 02:10:00}|SENSOR_012|3             |
|{2025-11-28 02:09:55, 2025-11-28 02:10:00}|SENSOR_016|1             |
+------------------------------------------+----------+--------------+


-------------------------------------------
Batch: 23
-------------------------------------------
+------------------------------------------+----------+--------------+
|window                                    |sensor_id |critical_count|
+------------------------------------------+----------+--------------+
|{2025-11-28 02:10:05, 2025-11-28 02:10:10}|SENSOR_013|1             |
|{2025-11-28 02:10:05, 2025-11-28 02:10:10}|SENSOR_018|1             |
|{2025-11-28 02:10:00, 2025-11-28 02:10:05}|SENSOR_017|1             |
|{2025-11-28 02:10:00, 2025-11-28 02:10:05}|SENSOR_003|1             |
|{2025-11-28 02:10:05, 2025-11-28 02:10:10}|SENSOR_020|1             |
|{2025-11-28 02:10:05, 2025-11-28 02:10:10}|SENSOR_004|1             |
|{2025-11-28 02:10:05, 2025-11-28 02:10:10}|SENSOR_017|1             |
|{2025-11-28 02:10:05, 2025-11-28 02:10:10}|SENSOR_009|1             |
|{2025-11-28 02:10:05, 2025-11-28 02:10:10}|SENSOR_008|1             |
|{2025-11-28 02:10:00, 2025-11-28 02:10:05}|SENSOR_019|1             |
|{2025-11-28 02:10:00, 2025-11-28 02:10:05}|SENSOR_006|2             |
+------------------------------------------+----------+--------------+
```

Screenshot 4.2: Terminal 3 showing alerts being written to Kafka alerts topic in real-time



CRITICAL QUESTIONS

Question 1: Explain what watermarking does in the windowed aggregation. Why is it necessary for streaming applications?

What watermarking does:
Watermarking tells Spark how late events are allowed to arrive for a given event-time column. Here, `withWatermark("event_time", "5 seconds")` means “accept data that is up to 5 seconds late compared to the latest event_time seen.”

How it affects windows:
Spark keeps window state (e.g., counts per 5-second window) in memory. Once the watermark passes the end of a window + 5 seconds, Spark considers that window complete and can safely drop its state. Late data beyond that watermark is ignored for that window.

Why it’s necessary:
In streaming, data can be delayed or arrive out of order. Watermarking gives a **balance**:

- Allows some lateness (so slightly late events are still counted),
- But **prevents infinite state growth** by eventually closing old windows and freeing memory

Without watermarking, a long-running streaming job would keep window state forever.

Question 2: What is the difference between batch processing and stream processing? Give an example from this activity where streaming provides an advantage.

Batch vs stream (concept):

- **Batch processing** works on a **fixed, finite dataset** (e.g., a file or table). You load all data, run the job once, and get a result.
- **Stream processing** works on an **unbounded, continuous flow** of events (e.g., Kafka topic). The job runs continuously and updates results as new data arrives.

Example from this activity (why streaming helps):

In Activity 4, Spark Structured Streaming reads live sensor events from Kafka (`sensor-data`), computes 5-second windowed `critical_count` per sensor, and writes alerts to the `alerts` topic in **near real time**.

This means if a sensor suddenly starts sending many `critical` statuses in a short window, we see it immediately in the console and in the alerts stream—no need to wait for a batch to finish.

Advantage phrased clearly:

For monitoring sensors, streaming is better than batch because operators can react **within seconds** of a spike in critical events (like your `Batch: 35–36` windows with multiple criticals), instead of discovering the issue only after the whole dataset is collected and a batch job is run.

Activity 5: Real-Time Analytics Dashboard

Objectives: Build web dashboard with Flask-SocketIO to display live Kafka metrics
Time: 30 minutes

Part 1: Create Dashboard Server

```
cat > activity5_dashboard.py << 'PYEOF' from flask import Flask, render_template from
flask_socketio import SocketIO, emit from kafka import KafkaConsumer
import json, threading, time

app = Flask(_name_) app.config['SECRET_KEY'] = 'secret'
socketio = SocketIO(app, cors_allowed_origins="*")

metrics = { 'sensor_count': 0,
   'txn_count': 0,
   'current_temp': 0.0,
   'total_revenue': 0.0, 'status': 'Running'
}

class KafkaThread(threading.Thread): def _init_(self, topic):
       threading.Thread._init_(self, daemon=True) self.topic = topic

   def run(self):
      consumer = KafkaConsumer( self.topic, bootstrap_servers=['localhost:9092'],
         auto_offset_reset='latest', group_id=f'{self.topic}_dash',
         value_deserializer=lambda x: json.loads(x.decode('utf-8'))
      )

      for msg in consumer: data = msg.value
         if self.topic == 'sensor-data':
         metrics['sensor_count'] += 1 metrics['current_temp'] = data.get('temperature', 0)
         elif self.topic == 'transactions': metrics['txn_count'] += 1
            metrics['total_revenue'] += data.get('amount', 0) socketio.emit('update', metrics,

         namespace='/live')

@app.route('/') def index():
   return render_template('dashboard.html')

@socketio.on('connect', namespace='/live') def handle_connect():
   emit('update', metrics)

if _name____== '_main_': KafkaThread('sensor-data').start() KafkaThread('transactions').start()
   print("=" * 70)
   print("Real-Time Dashboard Server") print("=" * 70)
```

print("Dashboard URL: http://localhost:5000") print("Press Ctrl+C to stop")
print("=" * 70)
socketio.run(app, host='0.0.0.0', port=5000, allow_unsafe_werkzeug=True) PYEOF

```
cat > activity5_dashboard.py << 'PYEOF'
from flask import Flask, render_template
from flask_socketio import SocketIO, emit
from kafka import KafkaConsumer
import json, threading, time

# Flask app and SocketIO setup
app = Flask(__name__)
app.config['SECRET_KEY'] = 'secret'
socketio = SocketIO(app, cors_allowed_origins="*")

# Shared metrics dictionary (global state)
metrics = {
    'sensor_count': 0,
    'txn_count': 0,
    'current_temp': 0.0,
    'total_revenue': 0.0,
    'status': 'Running'
}

class KafkaThread(threading.Thread):
    """
    Background thread that consumes from a Kafka topic
    and updates the global 'metrics' dict. Then emits
    updates to all connected SocketIO clients.
    """
    def __init__(self, topic):
        super().__init__(daemon=True)
        self.topic = topic

    def run(self):
        consumer = KafkaConsumer(
            self.topic,
            bootstrap_servers=['localhost:9092'],
            auto_offset_reset='latest',
            group_id=f'{self.topic}_dash',
            value_deserializer=lambda x: json.loads(x.decode('utf-8'))
        )
```

```
        for msg in consumer:
            data = msg.value

            # Update metrics depending on topic
            if self.topic == 'sensor-data':
                metrics['sensor_count'] += 1
                metrics['current_temp'] = data.get('temperature', 0.0)

            elif self.topic == 'transactions':
                metrics['txn_count'] += 1
                metrics['total_revenue'] += data.get('amount', 0.0)

            # Push update to all connected dashboards
            socketio.emit('update', metrics, namespace='/live')


@app.route('/')
def index():
    """Serve the dashboard HTML."""
    return render_template('dashboard.html')


@socketio.on('connect', namespace='/live')
def handle_connect():
    """
    When a new browser connects to /live namespace,
    immediately send it the current metrics snapshot.
    """
    emit('update', metrics)


if __name__ == '__main__':
    # Start Kafka consumer threads
    KafkaThread('sensor-data').start()
    KafkaThread('transactions').start()

    print("=" * 70)
    print("Real-Time Dashboard Server")
    print("=" * 70)
    print("Dashboard URL: http://localhost:5000")
    print("Press Ctrl+C to stop")
```

```
    print("=" * 70)

    socketio.run(app, host='0.0.0.0', port=5000,
allow_unsafe_werkzeug=True)
PYEOF
```

Part 2: Create Dashboard HTML

```
mkdir -p templates
cat > templates/dashboard.html << 'HTMLEOF'
<!DOCTYPE html>
<html>
<head>
    <title>Real-Time Analytics Dashboard</title>
    <script src="https://cdn.socket.io/4.0.0/socket.io.min.js"></script>
    <style>
        body {
            font-family: Arial, sans-serif;
            background: linear-gradient(135deg, #667eea 0%, #764ba2 100%); padding: 20px;
            margin: 0;
            }
        h1 {
            text-align: center; color: white;
            margin-bottom: 40px;
            text-shadow: 2px 2px 4px rgba(0,0,0,0.3);
        }
        .grid {
            display: grid;
            grid-template-columns: repeat(4, 1fr); gap: 20px;
            max-width: 1400px; margin: 0 auto;
        }
        .card {
            background: white; padding: 30px; border-radius: 15px;
            box-shadow: 0 10px 30px rgba(0,0,0,0.2); text-align: center;
            transition: transform 0.3s;
        }
        .card:hover {
            transform: translateY(-5px);
        }
        .value {
            font-size: 3em; font-weight: bold; color: #667eea; margin: 15px 0;
        }
        .label {
            font-size: 0.9em; color: #666;
            text-transform: uppercase; letter-spacing: 1px;
        }
        .status-indicator { position: fixed; top: 20px; right: 20px;
```

```
        padding: 10px 20px; background: #4caf50; color: white;
        border-radius: 20px; font-weight: bold;
        box-shadow: 0 4px 10px rgba(0,0,0,0.2);
      }
  </style>
</head>
<body>
  <div class="status-indicator">● Live</div>
  <h1>ç/¡#  Real-Time Analytics Dashboard</h1>
  <div class="grid">
    <div class="card">
      <div class="label">Sensor Readings</div>
      <div id="sensor" class="value">0</div>
    </div>
    <div class="card">
      <div class="label">Temperature (°C)</div>
      <div id="temp" class="value">0.0</div>
    </div>
    <div class="card">
      <div class="label">Transactions</div>
      <div id="txn" class="value">0</div>
    </div>
    <div class="card">
      <div class="label">Revenue ($)</div>
      <div id="revenue" class="value">0.00</div>
    </div>
  </div>
  <script>
    var socket = io('/live'); socket.on('update', function(data) {
      document.getElementById('sensor').innerText = data.sensor_count.toLocaleString();
      document.getElementById('temp').innerText = data.current_temp.toFixed(1);
      document.getElementById('txn').innerText = data.txn_count.toLocaleString();
      document.getElementById('revenue').innerText = data.total_revenue.toLocaleString('en',
{minimumFractionDigits: 2});
    });
  </script>
</body>
</html> HTMLEOF

mkdir -p templates
cat > templates/dashboard.html << 'HTMLEOF'
<!DOCTYPE html>
<html>
<head>
    <title>Real-Time Analytics Dashboard</title>
    <script
```

```
src="https://cdn.socket.io/4.0.0/socket.io.min.js"></script>
    <style>
        body {
            font-family: Arial, sans-serif;
            background: linear-gradient(135deg, #667eea 0%, #764ba2
100%);
            padding: 20px;
            margin: 0;
        }
        h1 {
            text-align: center;
            color: white;
            margin-bottom: 40px;
            text-shadow: 2px 2px 4px rgba(0,0,0,0.3);
        }
        .grid {
            display: grid;
            grid-template-columns: repeat(4, 1fr);
            gap: 20px;
            max-width: 1400px;
            margin: 0 auto;
        }
        .card {
            background: white;
            padding: 30px;
            border-radius: 15px;
            box-shadow: 0 10px 30px rgba(0,0,0,0.2);
            text-align: center;
            transition: transform 0.3s;
        }
        .card:hover {
            transform: translateY(-5px);
        }
        .value {
            font-size: 3em;
            font-weight: bold;
            color: #667eea;
            margin: 15px 0;
```

```
        }
        .label {
            font-size: 0.9em;
            color: #666;
            text-transform: uppercase;
            letter-spacing: 1px;
        }
        .status-indicator {
            position: fixed;
            top: 20px;
            right: 20px;
            padding: 10px 20px;
            background: #4caf50;
            color: white;
            border-radius: 20px;
            font-weight: bold;
            box-shadow: 0 4px 10px rgba(0,0,0,0.2);
        }
    </style>
</head>
<body>
    <div class="status-indicator">● Live</div>
    <h1>Real-Time Analytics Dashboard</h1>

    <div class="grid">
        <div class="card">
            <div class="label">Sensor Readings</div>
            <div id="sensor" class="value">0</div>
        </div>
        <div class="card">
            <div class="label">Temperature (°C)</div>
            <div id="temp" class="value">0.0</div>
        </div>
        <div class="card">
            <div class="label">Transactions</div>
            <div id="txn" class="value">0</div>
        </div>
        <div class="card">
```

```
            <div class="label">Revenue ($)</div>
            <div id="revenue" class="value">0.00</div>
        </div>
    </div>

    <script>
        // Connect to Socket.IO namespace /live
        var socket = io('/live');

        socket.on('update', function(data) {
            document.getElementById('sensor').innerText =
                data.sensor_count.toLocaleString();

            document.getElementById('temp').innerText =
                data.current_temp.toFixed(1);

            document.getElementById('txn').innerText =
                data.txn_count.toLocaleString();

            document.getElementById('revenue').innerText =
                data.total_revenue.toLocaleString('en', {
                    minimumFractionDigits: 2
                });
        });
    </script>
</body>
</html>
HTMLEOF
```

Part 3: Create Transaction Producer

cat > transaction_producer.py << 'PYEOF' from kafka import KafkaProducer
import json, time, random from datetime import datetime from faker import Faker

fake = Faker()
producer = KafkaProducer(bootstrap_servers=['localhost:9092'], value_serializer=lambda v:
  json.dumps(v).encode('utf-8')
)

def gen_transaction(): return {

```
    'transaction_id': fake.uuid4(), 'timestamp': datetime.now().isoformat(),
    'user_id': f"USER_{random.randint(1, 500):05d}", 'amount': round(random.uniform(5, 500),
    2), 'merchant': fake.company(),
    'category': random.choice(['Shopping', 'Food', 'Transport']),
  }

print("Sending transactions... (Ctrl+C to stop)") count = 0
try:
  while True:
    producer.send('transactions', gen_transaction()) count += 1
    if count % 50 == 0:
      print(f"Sent {count} transactions...") time.sleep(0.2)
except KeyboardInterrupt: producer.close() print(f"
Stopped. Total: {count}") PYEOF

cat > transaction_producer.py << 'PYEOF'
from kafka import KafkaProducer
import json, time, random
from datetime import datetime
from faker import Faker

fake = Faker()

producer = KafkaProducer(
    bootstrap_servers=['localhost:9092'],
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

def gen_transaction():
    """Generate a fake transaction event."""
    return {
        'transaction_id': fake.uuid4(),
        'timestamp': datetime.now().isoformat(),
        'user_id': f"USER_{random.randint(1, 500):05d}",
        'amount': round(random.uniform(5, 500), 2),
        'merchant': fake.company(),
        'category': random.choice(['Shopping', 'Food', 'Transport']),
    }

print("Sending transactions... (Ctrl+C to stop)")
count = 0

try:
    while True:
```

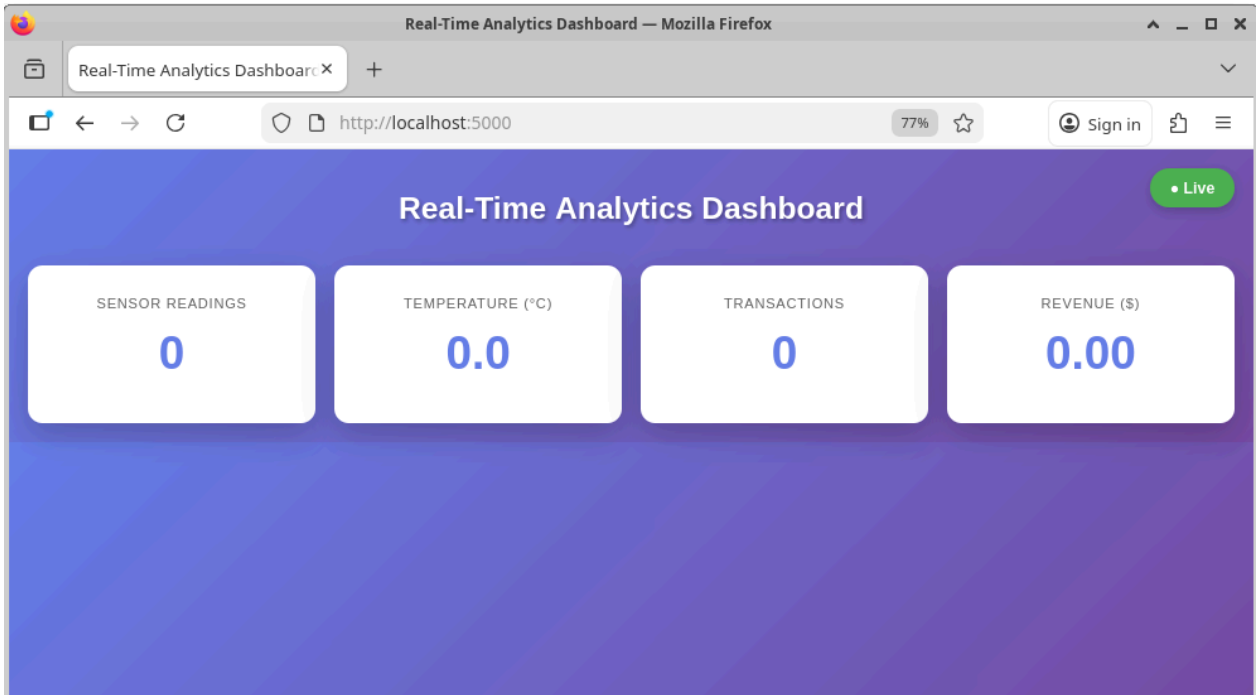
```
        producer.send('transactions', gen_transaction())
        count += 1
        if count % 50 == 0:
            print(f"Sent {count} transactions...")
        time.sleep(0.2)
except KeyboardInterrupt:
    producer.close()
    print(f"\nStopped. Total: {count}")
PYEOF
```

Part 4: Run Dashboard (3 Terminals)

Terminal 1 - Dashboard Server:

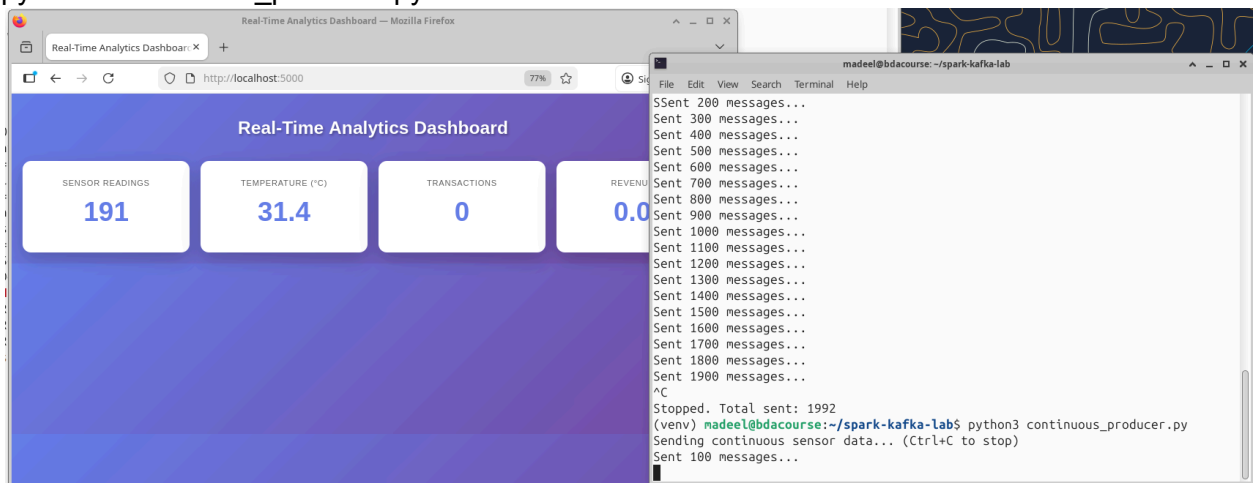
python3 activity5_dashboard.py
Open browser: http://localhost:5000

```
(venv) madeel@bdacourse:~/spark-kafka-lab$ python3 activity5_dashboard.py
======================================================================
Real-Time Dashboard Server
======================================================================
Dashboard URL: http://localhost:5000
Press Ctrl+C to stop
======================================================================
 * Serving Flask app 'activity5_dashboard'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://172.17.5.42:5000
Press CTRL+C to quit
```



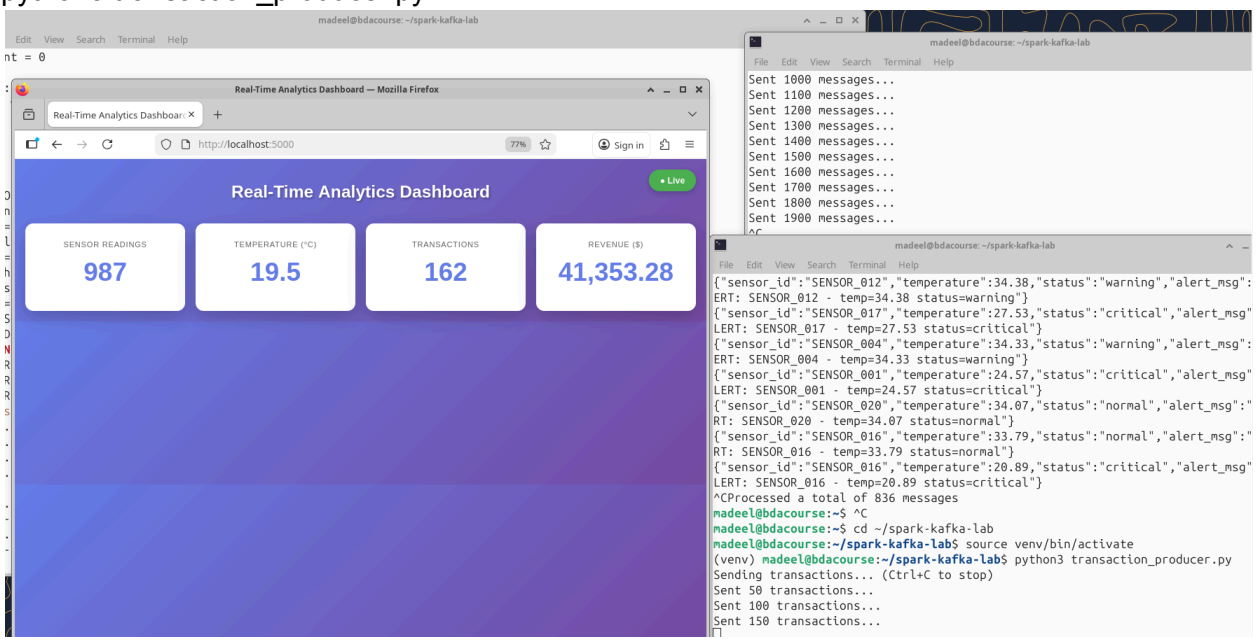
Terminal 2 - Sensor Data Producer:

python3 continuous_producer.py



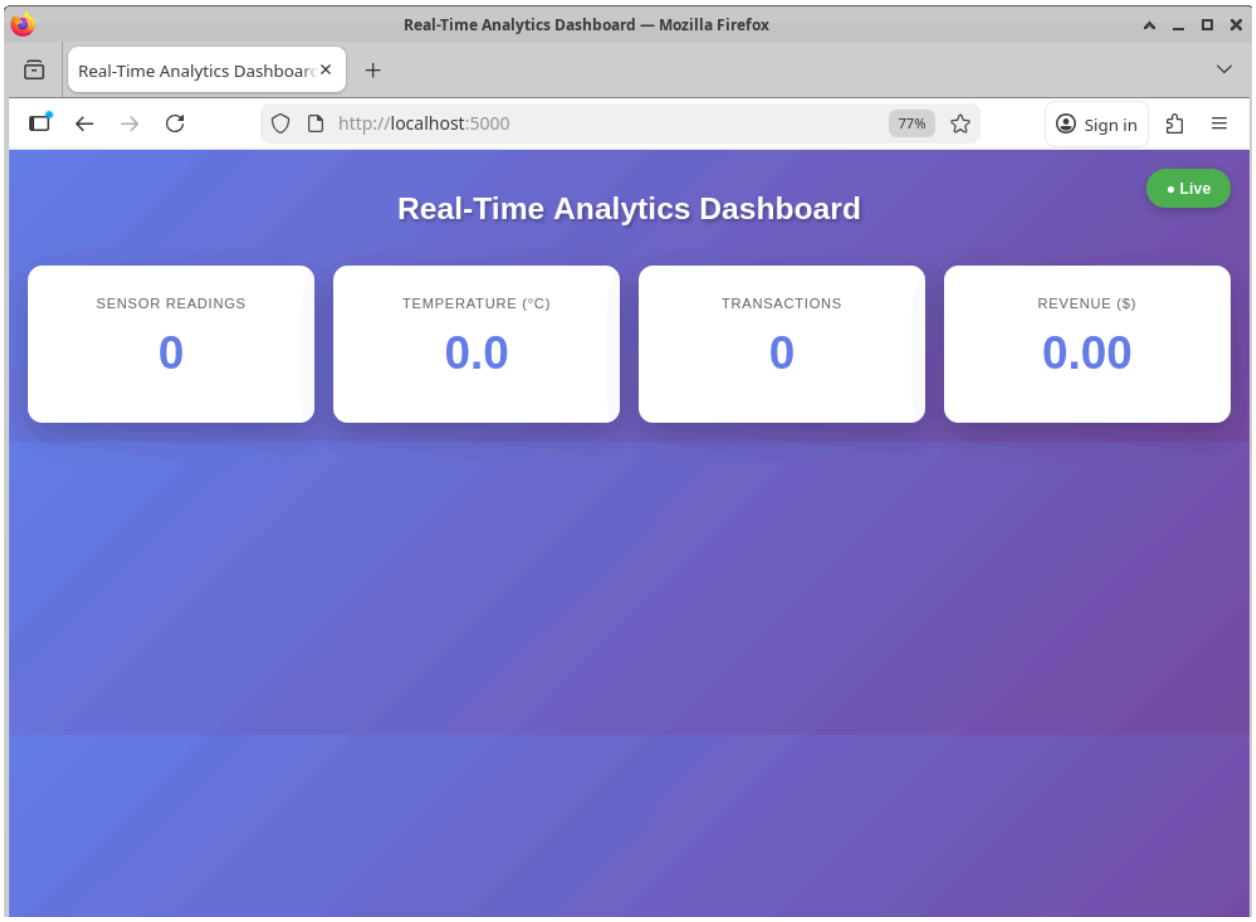
Terminal 3 - Transaction Producer:

python3 transaction_producer.py

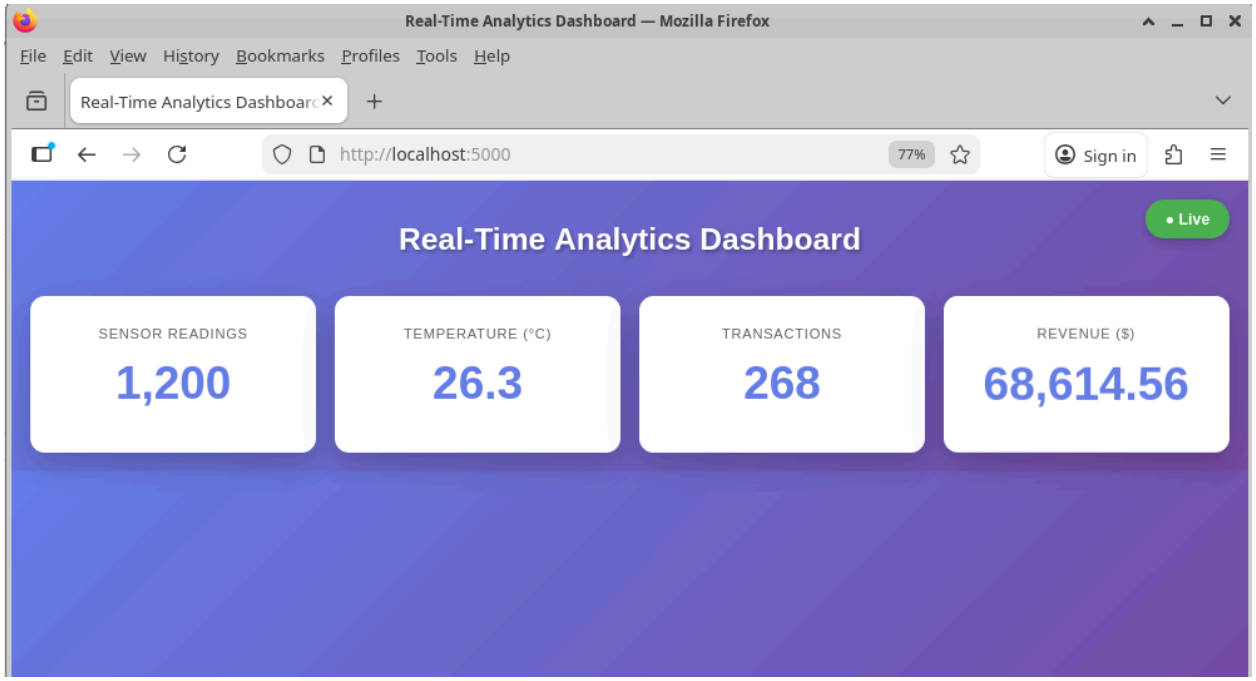


REQUIRED SCREENSHOTS

Screenshot 5.1: Web browser showing the live dashboard with all 4 metrics updating in real-time (initial state)



Screenshot 5.2: Same dashboard after 1–2 minutes showing updated metrics (sensor count, temperature, transactions, revenue)

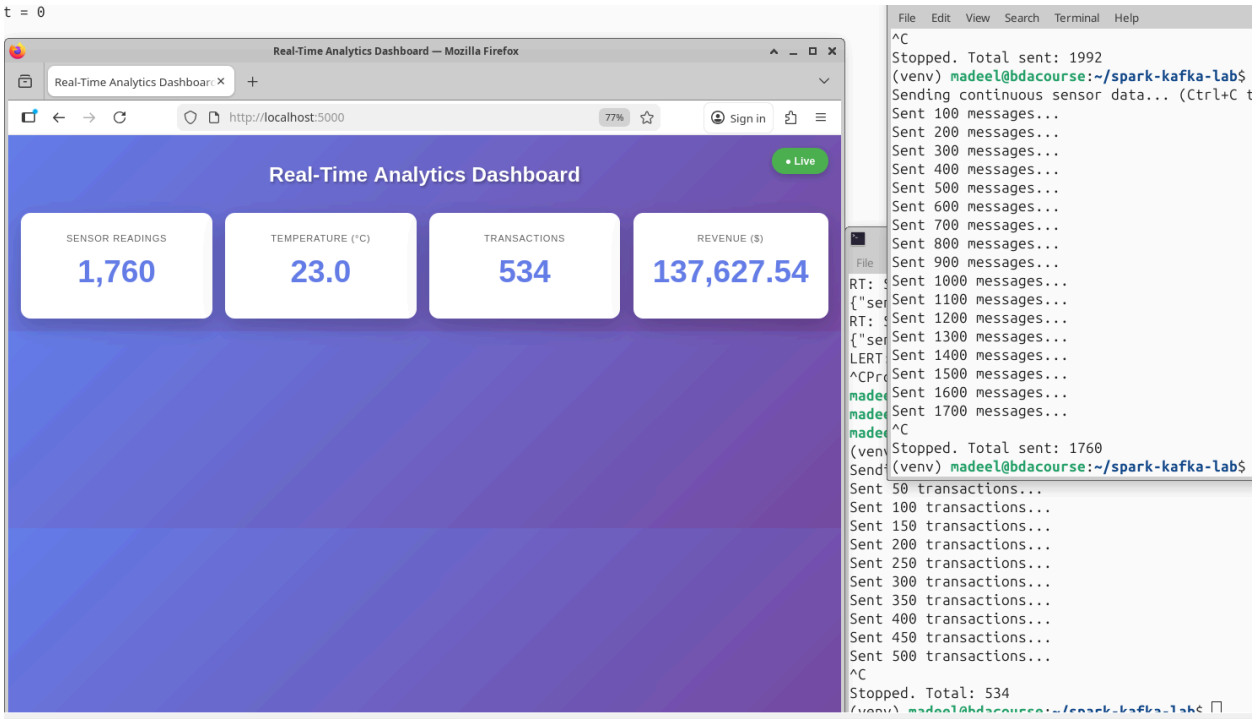


Screenshot 5.3: Terminal showing dashboard server logs with connection

messages

```
(venv) madeel@bdacourse:~/spark-kafka-lab$ python3 activity5_dashboard.py
======================================================================
Real-Time Dashboard Server
======================================================================
Dashboard URL: http://localhost:5000
Press Ctrl+C to stop
======================================================================
 * Serving Flask app 'activity5_dashboard'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://172.17.5.42:5000
Press CTRL+C to quit
127.0.0.1 - - [28/Nov/2025 02:28:54] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [28/Nov/2025 02:28:54] "GET /socket.io/?EIO=4&transport=polling&t=Ph7E1io HTTP/1.1" 200 -
127.0.0.1 - - [28/Nov/2025 02:28:54] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [28/Nov/2025 02:28:54] "POST /socket.io/?EIO=4&transport=polling&t=Ph7E1j3&sid=S8kTjwN4YZdFtPh8AAAA HTTP/1.1"
200 -
127.0.0.1 - - [28/Nov/2025 02:28:54] "GET /socket.io/?EIO=4&transport=polling&t=Ph7E1j7&sid=S8kTjwN4YZdFtPh8AAAA HTTP/1.1" 2
00 -
127.0.0.1 - - [28/Nov/2025 02:28:54] "GET /socket.io/?EIO=4&transport=polling&t=Ph7E1jJ&sid=S8kTjwN4YZdFtPh8AAAA HTTP/1.1" 2
00 -
```

Now we stopped it



CRITICAL QUESTIONS

Question 1: How does Flask-SocketIO enable real-time communication between the server and browser? What happens when multiple users access the dashboard simultaneously?

Flask-SocketIO enables real-time communication by using **WebSockets**, which create a permanent two-way connection between the server and the browser.

Instead of the browser repeatedly asking the server for updates (like normal HTTP), the server **pushes updates instantly** whenever new data arrives from Kafka.

In our dashboard:

The server uses: `socketio.emit('update', metrics, namespace='/live')` to send updated metrics to all connected clients.

The browser uses: `socket.on('update', function(data) { ... });` to receive data and update the dashboard instantly.

What happens when multiple users connect?
When multiple users access the dashboard:

1. Each browser opens its own WebSocket connection to the Flask server.
2. Flask-SocketIO keeps track of all connected clients.
3. Whenever new Kafka data arrives, the server broadcasts the updated metrics to **all connected users at the same time**.
4. So every user sees the **same live dashboard** and the metrics update simultaneously across all screens.

This is called **broadcasting in real-time systems**, and Flask-SocketIO handles it automatically.

Question 2: Why do we use threading for Kafka consumers in the dashboard? What would happen if we didn't use threads?

We use **threads** so that Kafka data consumption runs **in the background** and does not block the main Flask application.

In our code:

```
KafkaThread('sensor-data').start()

KafkaThread('transactions').start()
```

Each thread runs this infinite loop:

```
for msg in consumer:

    ...
```

Kafka consumers never stop; they continuously wait for messages.

Why threading is necessary:

If we did **not use threads**:

- The Flask server would get stuck inside the Kafka consumer loop.
- Flask would not be able to:
    - Serve the web page (`/`)
    - Accept new browser connections
    - Send WebSocket updates

- The dashboard would either never load or freeze completely.

With threading:

- Flask runs normally on the main thread.
- Kafka consumers run in parallel background threads.
- The system stays responsive:
    - The dashboard loads
    - Metrics update live
    - Multiple users can connect smoothly.

Activity 6: Machine Learning with Spark MLlib

Objectives: Train a classification model using Spark MLlib, evaluate model performance

Part 1: Create ML Script

bash
cat > activity6_ml.py << 'PYEOF'
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler, StringIndexer from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator from pyspark.ml import Pipeline

print("=" * 70)
print("Activity 6: Machine Learning with Spark MLlib") print("=" * 70)

spark = SparkSession.builder \
.appName("Activity6-ML") \
.master("local[*]") \
.config("spark.driver.memory", "2g") \
.getOrCreate() spark.sparkContext.setLogLevel("ERROR")

# Load Iris dataset
print("\nSTEP 1: Loading Iris Dataset") print("=" * 70)
data = spark.read.csv("data/iris.csv", header=True, inferSchema=True) print(f"Total samples: {data.count()}")
data.printSchema() data.show(5)

PYEOF

This part of the script:

- Imports relevant libraries, like **SparkSession**, **VectorAssembler**, **StringIndexer**, **DecisionTreeClassifier**, **MulticlassClassificationEvaluator** to train a model using spark MLib .
- Creates a **SparkSession** with following settings:
  - **App name: `"Activity6-ML"`**
  - **Master: `"local[*]"` (uses all CPU cores on machine)**
  - **Driver memory: `2g"` (Spark gets 2GB RAM)**
- Loads the dataset from **iris.csv** file into a Spark Dataframe.
- Counts the **number of samples** in the dataset and prints it.
- Prints **schema of the dataset** and first 5 rows.

Part 2: Add Feature Engineering

bash
cat >> activity6_ml.py << 'PYEOF'

# Feature Engineering
print("\nSTEP 2: Feature Engineering") print("=" * 70)

# Convert species labels to numeric
indexer = StringIndexer(inputCol="species", outputCol="label")

# Combine features into vector
feature_cols = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width'] assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")

print("Feature columns:", feature_cols)
print("Label column: species (converted to numeric)")

PYEOF

This part of the script prepares the raw data columns into a format that the Spark MLlib Decision Tree model can use.

- It creates an `indexer` stage to convert the **text labels** in the `species` column into **numerical indices**.
- Next, it creates an `assembler` stage that takes the four separate feature columns and combines them into a **single dense vector column** named `features`.

Part 3: Add Training and Evaluation

bash
cat >> activity6_ml.py << 'PYEOF'

# Train/Test Split
print("\nSTEP 3: Train/Test Split") print("=" * 70)
train_data, test_data = data.randomSplit([0.7, 0.3], seed=42) print(f"Training samples: {train_data.count()}")
print(f"Test samples: {test_data.count()}")

# Build Pipeline
print("\nSTEP 4: Training Decision Tree Model") print("=" * 70)
dt = DecisionTreeClassifier(labelCol="label", featuresCol="features", maxDepth=5) pipeline = Pipeline(stages=[indexer, assembler, dt])

print("Training model...") model = pipeline.fit(train_data) print("✓ Training complete!")

Make Predictions

```
print("\nSTEP 5: Making Predictions") print("=" * 70)
predictions = model.transform(test_data) predictions.select("features", "label", "prediction",
"species").show(10)

# Evaluate Model
print("\nSTEP 6: Model Evaluation") print("=" * 70)
evaluator = MulticlassClassificationEvaluator( labelCol="label", predictionCol="prediction",
    metricName="accuracy"
)

accuracy = evaluator.evaluate(predictions)
print(f"Test Accuracy: {accuracy:.4f} ({accuracy*100:.2f}%)")

# Confusion analysis print("\nPrediction Summary:")
predictions.groupBy("label", "prediction").count().show()

spark.stop()
print("\n✓ Activity 6 Complete!") PYEOF
```

This part of the script:

- splits dataset into **70% for training** and **30% for testing**.
- Chains the feature engineering steps with the `DecisionTreeClassifier` into a `Pipeline`, and then **trains the entire pipeline using the training data**.
- Uses the trained model to make **`predictions`** on the **unseen test data**.
- Measures the model's performance on the test data, calculating and printing the classification **accuracy** and a confusion matrix summary.

Part 4: Run ML Pipeline

bash

python3 activity6_ml.py

REQUIRED SCREENSHOTS

Screenshot 6.1: Terminal output showing dataset loading, schema, and sample data (Steps 1-2)

```
STEP 1: Loading Iris Dataset
======================================================================
Total samples: 150
root
 |-- sepal_length: double (nullable = true)
 |-- sepal_width: double (nullable = true)
 |-- petal_length: double (nullable = true)
 |-- petal_width: double (nullable = true)
 |-- species: string (nullable = true)

+------------+-----------+------------+-----------+-----------+
|sepal_length|sepal_width|petal_length|petal_width|    species|
+------------+-----------+------------+-----------+-----------+
|         5.1|        3.5|         1.4|        0.2|Iris-setosa|
|         4.9|        3.0|         1.4|        0.2|Iris-setosa|
|         4.7|        3.2|         1.3|        0.2|Iris-setosa|
|         4.6|        3.1|         1.5|        0.2|Iris-setosa|
|         5.0|        3.6|         1.4|        0.2|Iris-setosa|
+------------+-----------+------------+-----------+-----------+
only showing top 5 rows
```

```
STEP 2: Feature Engineering
======================================================================
Feature columns: ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
Label column: species (converted to numeric)

STEP 3: Train/Test Split
======================================================================
Training samples: 104
Test samples: 46

STEP 4: Training Decision Tree Model
======================================================================
Training model...
✓ Training complete!
```

Screenshot 6.2: Terminal showing predictions with features, actual label, and predicted label (Step 5)

```
STEP 5: Making Predictions
================================================================================
+-----------------+-----+----------+---------------+
|         features|label|prediction|        species|
+-----------------+-----+----------+---------------+
|[4.4,3.0,1.3,0.2]|  2.0|       2.0|    Iris-setosa|
|[4.6,3.2,1.4,0.2]|  2.0|       2.0|    Iris-setosa|
|[4.6,3.6,1.0,0.2]|  2.0|       2.0|    Iris-setosa|
|[4.7,3.2,1.3,0.2]|  2.0|       2.0|    Iris-setosa|
|[4.8,3.1,1.6,0.2]|  2.0|       2.0|    Iris-setosa|
|[4.8,3.4,1.6,0.2]|  2.0|       2.0|    Iris-setosa|
|[4.8,3.4,1.9,0.2]|  2.0|       2.0|    Iris-setosa|
|[4.9,3.1,1.5,0.1]|  2.0|       2.0|    Iris-setosa|
|[4.9,3.1,1.5,0.1]|  2.0|       2.0|    Iris-setosa|
|[5.0,2.3,3.3,1.0]|  1.0|       1.0|Iris-versicolor|
+-----------------+-----+----------+---------------+
only showing top 10 rows
```

Screenshot 6.3: Terminal showing final model accuracy and prediction summary (Step 6)

```
STEP 6: Model Evaluation
================================================================================
Test Accuracy: 0.9783 (97.83%)

Prediction Summary:
+-----+----------+-----+
|label|prediction|count|
+-----+----------+-----+
|  1.0|       1.0|   14|
|  2.0|       2.0|   22|
|  1.0|       0.0|    1|
|  0.0|       0.0|    9|
+-----+----------+-----+
```

CRITICAL QUESTIONS

Question 1: What was your model's accuracy? Analyze the prediction summary - which species (if any) was harder to classify correctly and why might that be?

Model Accuracy: 97.83%

This high accuracy is primarily due to the **Iris dataset's simplicity**. The differences in petal and sepal measurements between the three flower species are **highly separable**, making it **easy** for even a simple model like the **Decision Tree** to find clean boundaries and classify samples correctly.

Since there's **only one error**, it's more appropriate to label that sample as an **outlier** rather than concluding that the entire species **(Label 1.0)** is inherently hard to classify. If Label 1.0 had missed 5 or 6 samples, that could suggest a systemic problem. The misclassified sample **(True Label 1.0, Predicted Label 0.0)** likely represents a specific flower whose measurements fell outside the typical range for its species, making it look much closer to the very distinct **Label 0.0** class than it should have been.

Lab Completion & Submission

Final Checklist

- Activity 1: Spark SQL completed with 1 screenshot
- Activity 2: Performance benchmarks with 3 screenshots
- Activity 3: Kafka anomaly detection with 2 screenshots
- Activity 4: Spark streaming with 2 screenshots
- Activity 5: Dashboard with 3 screenshots
- Activity 6: Machine learning with 3 screenshots

Stop All Services

bash
 # Stop producers (Ctrl+C in terminals)

 # Stop Spark Streaming
 # Ctrl+C in streaming terminal

 # Stop Dashboard
 # Ctrl+C in dashboard terminal

 # Stop Spark Cluster
stop-worker.sh stop-master.sh

 # Stop Kafka
kafka-server-stop.sh zookeeper-server-stop.sh

 # Verify all stopped

jps *# Should only show Jp*

Submission Requirements

1. Lab Report Document

Create a document containing:

- **Cover Page:** Name, ERP, Date, Lab Title
- **Activity 1-6 Sections:**

- All required screenshots (labeled clearly)
- Answers to all critical questions
- Brief observations/notes for each activity

2. Code Files

Submit a ZIP file containing:

```
spark-kafka-lab/
├── activity1_spark_sql.py
├── activity2_single_node.py
├── activity2_cluster.py
├── activity3_producer.py
├── activity3_consumer.py
├── activity4_streaming.py
├── continuous_producer.py
├── activity5_dashboard.py
├── transaction_producer.py
├── activity6_ml.py
├── templates/
│   └── dashboard.html
└── output/
    ├── single_node_results.txt
    └── cluster_results.txt
```

Outcome & Achievements (Concise)

- **Dataset:** Implemented use of NYC Yellow Taxi Parquet (Jan–Mar 2023) as the primary large dataset; fallback sample created if download fails. Parquet improves IO and preserves schema.

- **Activity 1:** Loaded taxi data, ran SQL aggregations (avg fare by passenger_count, payment-type revenue), demonstrated caching speedup.

- **Activity 2:** Benchmarked single-node vs cluster runs (load, aggregation, filter, join) to measure parallelism benefits.

- **Activity 3:** Implemented Kafka producer and consumer; detected anomalies and validated message flow.

- **Activity 4:** Built Spark Structured Streaming pipeline consuming Kafka, produced alerts back to Kafka and performed windowed aggregations with watermarking.

- **Activity 5:** Built Flask-SocketIO dashboard streaming live metrics from Kafka topics; multi-client capable and thread-safe via consumer threads.

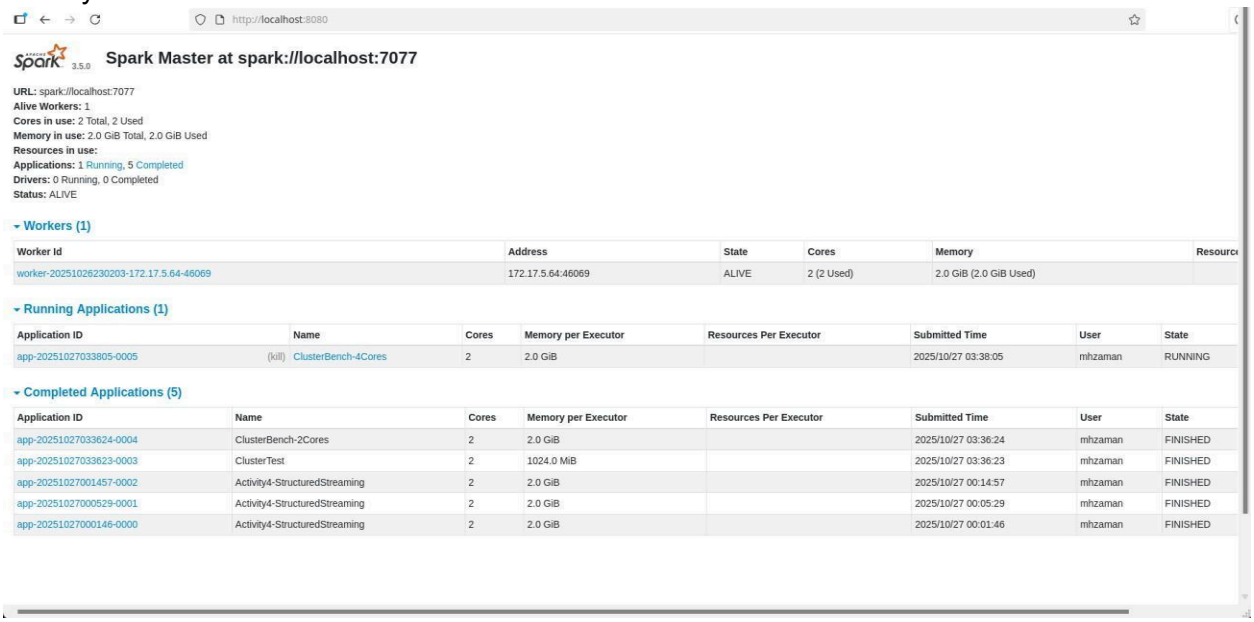
- **Activity 6:** Trained Decision Tree on Iris dataset using Spark MLlib; evaluated and printed accuracy and confusion counts.

- **Overall:** End-to-end pipeline demonstrating batch (Parquet) processing, streaming integration (Kafka + Spark Streaming), real-time visualization, and ML — suitable for coursework and demos.

Activity 1:

```
|payment_type|  count|
+------------+-------+
|        Cash|3332750|
|      Credit|3333563|
|      Mobile|3333687|
+------------+-------+

Time with cache: 1.07 seconds
Speedup: 5.47x

=== Query 6: Complex Aggregation - Trip Categories ===
+------------+-----------------+----------+--------+---------+
|payment_type|distance_category|trip_count|avg_fare|avg_total|
+------------+-----------------+----------+--------+---------+
|        Cash|             Long|    114727|   52.56|    62.41|
|        Cash|           Medium|    510648|    52.5|    62.48|
|        Cash|            Short|   1080771|   52.51|    62.46|
|        Cash|        Very Long|      4295|    52.3|    62.61|
|        Cash|       Very Short|   1622309|   52.44|    62.47|
|      Credit|             Long|    114596|   52.49|    62.62|
|      Credit|           Medium|    509211|   52.49|     62.6|
|      Credit|            Short|   1082262|   52.45|    62.51|
|      Credit|        Very Long|      4344|   51.74|    63.12|
|      Credit|       Very Short|   1623150|   52.49|     62.5|
|      Mobile|             Long|    114628|   52.35|    62.66|
|      Mobile|           Medium|    511355|   52.51|     62.5|
|      Mobile|            Short|   1080778|   52.54|    62.49|
|      Mobile|        Very Long|      4099|   52.69|    63.41|
|      Mobile|       Very Short|   1622827|   52.52|    62.51|
+------------+-----------------+----------+--------+---------+

Query execution time: 1.54 seconds

=== Saving Results ===
Results saved to output/activity1_results
Parquet results saved to output/activity1_parquet

============================================================
Activity 1 Complete!
============================================================
(venv) mhzaman@bdacourse:~/spark-kafka-lab$
```

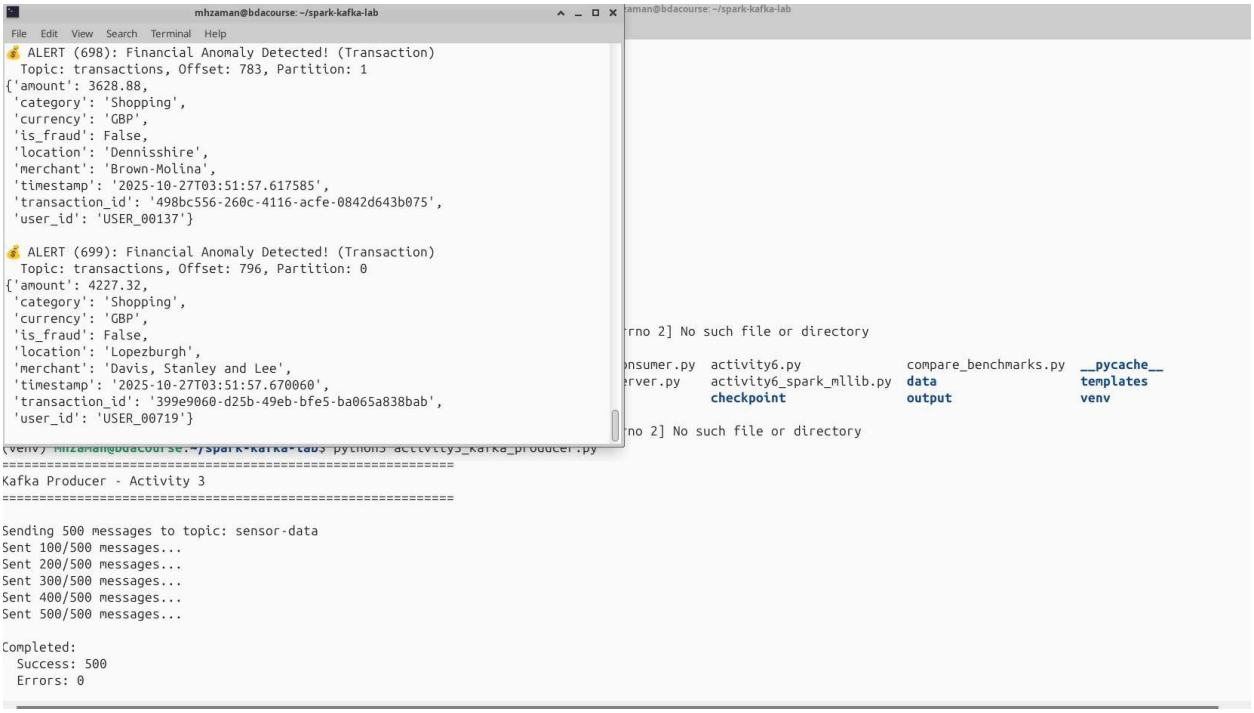
Activity 2:


http://localhost:8080
Spark 3.5.0
Spark Master at spark://localhost:7077
URL: spark://localhost:7077
Alive Workers: 1
Cores in use: 2 Total, 2 Used
Memory in use: 2.0 GiB Total, 2.0 GiB Used
Resources in use:
Applications: 1 Running, 5 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE
Workers (1)
Worker Id | Address | State | Cores | Memory
worker-20251026230203-172.17.5.64-46069 | 172.17.5.64:46069 | ALIVE | 2 (2 Used) | 2.0 GiB (2.0 GiB Used)
Running Applications (1)
Application ID | Name | Cores | Memory per Executor | Resources Per Executor | Submitted Time | User | State
app-20251027033805-0005 (kill) | ClusterBench-4Cores | 2 | 2.0 GiB | | 2025/10/27 03:38:05 | mhzaman | RUNNING
Completed Applications (5)
Application ID | Name | Cores | Memory per Executor | Resources Per Executor | Submitted Time | User | State
app-20251027033624-0004 | ClusterBench-2Cores | 2 | 2.0 GiB | | 2025/10/27 03:36:24 | mhzaman | FINISHED
app-20251027033623-0003 | ClusterTest | 2 | 1024.0 MiB | | 2025/10/27 03:36:23 | mhzaman | FINISHED
app-20251027001457-0002 | Activity4-StructuredStreaming | 2 | 2.0 GiB | | 2025/10/27 00:14:57 | mhzaman | FINISHED
app-20251027000529-0001 | Activity4-StructuredStreaming | 2 | 2.0 GiB | | 2025/10/27 00:05:29 | mhzaman | FINISHED
app-20251027000146-0000 | Activity4-StructuredStreaming | 2 | 2.0 GiB | | 2025/10/27 00:01:46 | mhzaman | FINISHED

Generate an output file with all relevant results, show how number of cores impacted your execution/processing time for different functions/queries on a dataset:

```
(venv) mhzaman@bdacourse:~/spark-kafka-lab$ cat output/cluster_benchmark.txt
Cluster Benchmark Results
================================================


2 Cores:
------------------------------
  Data Loading        :    27.83s
  Aggregation         :     2.39s
  Filter & Sort       :     2.26s
  Join Operation      :    20.11s
  Window Functions    :    11.16s
  TOTAL TIME          :    63.76s

4 Cores:
------------------------------
  Data Loading        :    26.17s
  Aggregation         :     2.03s
  Filter & Sort       :     2.36s
  Join Operation      :    19.76s
  Window Functions    :     8.62s
  TOTAL TIME          :    58.94s

6 Cores:
------------------------------
  Data Loading        :    25.48s
  Aggregation         :     2.04s
  Filter & Sort       :     2.61s
  Join Operation      :    20.35s
  Window Functions    :     8.62s
  TOTAL TIME          :    59.10s


Speedups (2 cores → 6 cores):
------------------------------
  Data Loading        :     1.09x
  Aggregation         :     1.18x
  Filter & Sort       :     0.87x
  Join Operation      :     0.99x
  Window Functions    :     1.29x
(venv) mhzaman@bdacourse:~/spark-kafka-lab$ S
```

Activity 3:
Detecting Anamolies in real time data stream

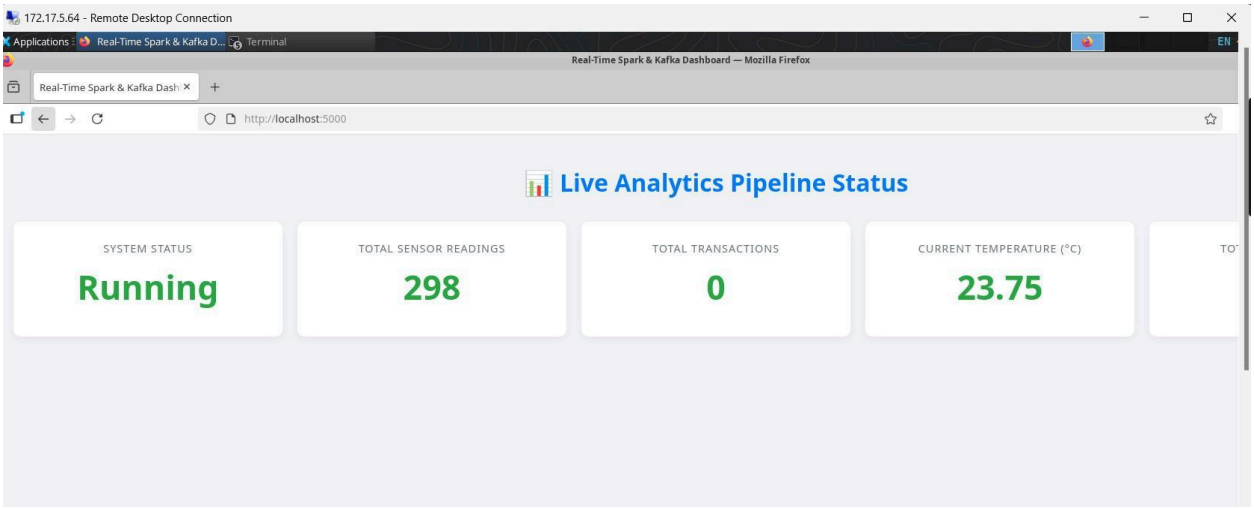


Activity 4:

```
|{2025-10-27 00:22:40, 2025-10-27 00:22:45}|warning |5    |
|{2025-10-27 00:22:40, 2025-10-27 00:22:45}|critical|2    |
+------------------------------------------+--------+-----+

-------------------------------------------
Batch: 29
-------------------------------------------
+------------------------------------------+--------+-----+
|window                                    |status  |count|
+------------------------------------------+--------+-----+
|{2025-10-27 00:22:40, 2025-10-27 00:22:45}|normal  |9    |
|{2025-10-27 00:22:40, 2025-10-27 00:22:45}|warning |12   |
|{2025-10-27 00:22:40, 2025-10-27 00:22:45}|critical|6    |
+------------------------------------------+--------+-----+

-------------------------------------------
Batch: 30
-------------------------------------------
+------------------------------------------+--------+-----+
|window                                    |status  |count|
+------------------------------------------+--------+-----+
|{2025-10-27 00:22:40, 2025-10-27 00:22:45}|warning |20   |
|{2025-10-27 00:22:40, 2025-10-27 00:22:45}|critical|9    |
+------------------------------------------+--------+-----+

-------------------------------------------
Batch: 31
-------------------------------------------
```

Activity 5:

```
(venv) mhzaman@bdacourse:~/spark-kafka-lab$ python3 activity5.py
Starting Flask server on http://localhost:5000
 * Serving Flask app 'activity5'
 * Debug mode: off
Consumer started for topic: sensor-data
Consumer started for topic: transactions
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://172.17.5.64:5000
Press CTRL+C to quit
127.0.0.1 - - [27/Oct/2025 02:51:09] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [27/Oct/2025 02:51:10] "GET /socket.io/?EIO=4&transport=polling&t=PeYWFyO HTTP/1.1" 200 -
127.0.0.1 - - [27/Oct/2025 02:51:10] "GET /favicon.ico HTTP/1.1" 404 -
Client connected to /data_stream
127.0.0.1 - - [27/Oct/2025 02:51:10] "POST /socket.io/?EIO=4&transport=polling&t=PeYWFzQ&sid=e-0EKaV-J-7Xrq6eAAAA HTTP/1.1" 200 -
127.0.0.1 - - [27/Oct/2025 02:51:10] "GET /socket.io/?EIO=4&transport=polling&t=PeYWFzT&sid=e-0EKaV-J-7Xrq6eAAAA HTTP/1.1" 200 -
127.0.0.1 - - [27/Oct/2025 02:51:10] "GET /socket.io/?EIO=4&transport=polling&t=PeYWFzx&sid=e-0EKaV-J-7Xrq6eAAAA HTTP/1.1" 200 -
```



Show of Real time update on a html based analytical dashboard.

```
Files removed: 398 (2411.2 MB)
Installing six...
Collecting six
  Downloading six-1.17.0-py2.py3-none-any.whl.metadata (1.7 kB)
Downloading six-1.17.0-py2.py3-none-any.whl (11 kB)
Installing collected packages: six
Successfully installed six-1.17.0
Installing kafka-python...
Collecting kafka-python
  Downloading kafka_python-2.2.15-py2.py3-none-any.whl.metadata (10.0 kB)
Downloading kafka_python-2.2.15-py2.py3-none-any.whl (309 kB)
Installing collected packages: kafka-python
Successfully installed kafka-python-2.2.15
(venv) mhzaman@bdacourse:~/spark-kafka-lab$ python3 activity3_kafka_producer.py
============================================================
Kafka Producer - Activity 3
============================================================

Sending 500 messages to topic: sensor-data
Sent 100/500 messages...
Sent 200/500 messages...
Sent 300/500 messages...
Sent 400/500 messages...
Sent 500/500 messages...

Completed:
  Success: 500
  Errors: 0
```

Activity 6:

```
+------------+-----------+------------+-----------+-----------+
|sepal_length|sepal_width|petal_length|petal_width|    species|
+------------+-----------+------------+-----------+-----------+
|         5.1|        3.5|         1.4|        0.2|Iris-setosa|
|         4.9|        3.0|         1.4|        0.2|Iris-setosa|
|         4.7|        3.2|         1.3|        0.2|Iris-setosa|
|         4.6|        3.1|         1.5|        0.2|Iris-setosa|
|         5.0|        3.6|         1.4|        0.2|Iris-setosa|
+------------+-----------+------------+-----------+-----------+
only showing top 5 rows

Training Data Count: 104
Test Data Count: 46

=== Training Decision Tree Model... ===
Training Complete.
+-----------------+-----+----------+---------------+
|         features|label|prediction|        species|
+-----------------+-----+----------+---------------+
|[4.4,3.0,1.3,0.2]|  2.0|       2.0|    Iris-setosa|
|[4.6,3.2,1.4,0.2]|  2.0|       2.0|    Iris-setosa|
|[4.6,3.6,1.0,0.2]|  2.0|       2.0|    Iris-setosa|
|[4.7,3.2,1.3,0.2]|  2.0|       2.0|    Iris-setosa|
|[4.8,3.1,1.6,0.2]|  2.0|       2.0|    Iris-setosa|
|[4.8,3.4,1.6,0.2]|  2.0|       2.0|    Iris-setosa|
|[4.8,3.4,1.9,0.2]|  2.0|       2.0|    Iris-setosa|
|[4.9,3.1,1.5,0.1]|  2.0|       2.0|    Iris-setosa|
|[4.9,3.1,1.5,0.1]|  2.0|       2.0|    Iris-setosa|
|[5.0,2.3,3.3,1.0]|  1.0|       1.0|Iris-versicolor|
+-----------------+-----+----------+---------------+
only showing top 10 rows


============================================================
Model Training Time: DecisionTreeClassifier_fc78ceb5e25f
Test Set Accuracy = 0.9783
============================================================

=== Activity 6 Complete ===
(venv) mhzaman@bdacourse:~/spark-kafka-lab$
```