# 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



### 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")

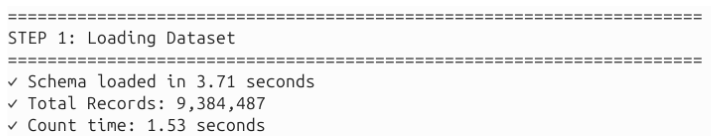
print("

" + "=" \* 70)

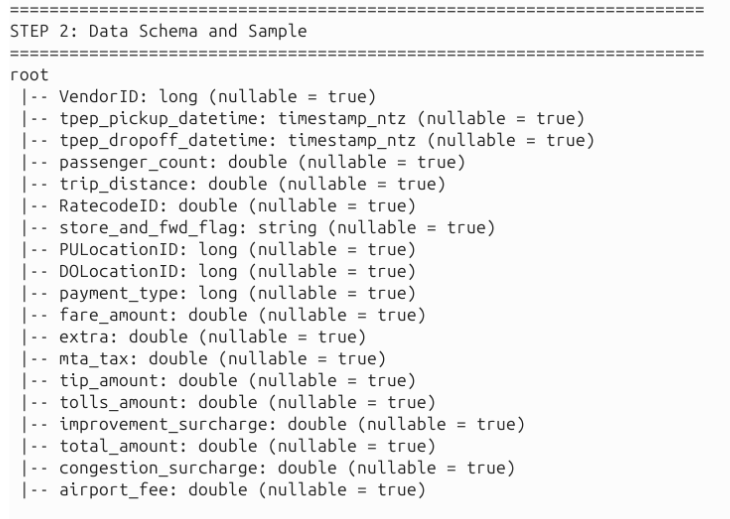
print("STEP 2: Data Schema and Sample") print("=" \* 70)

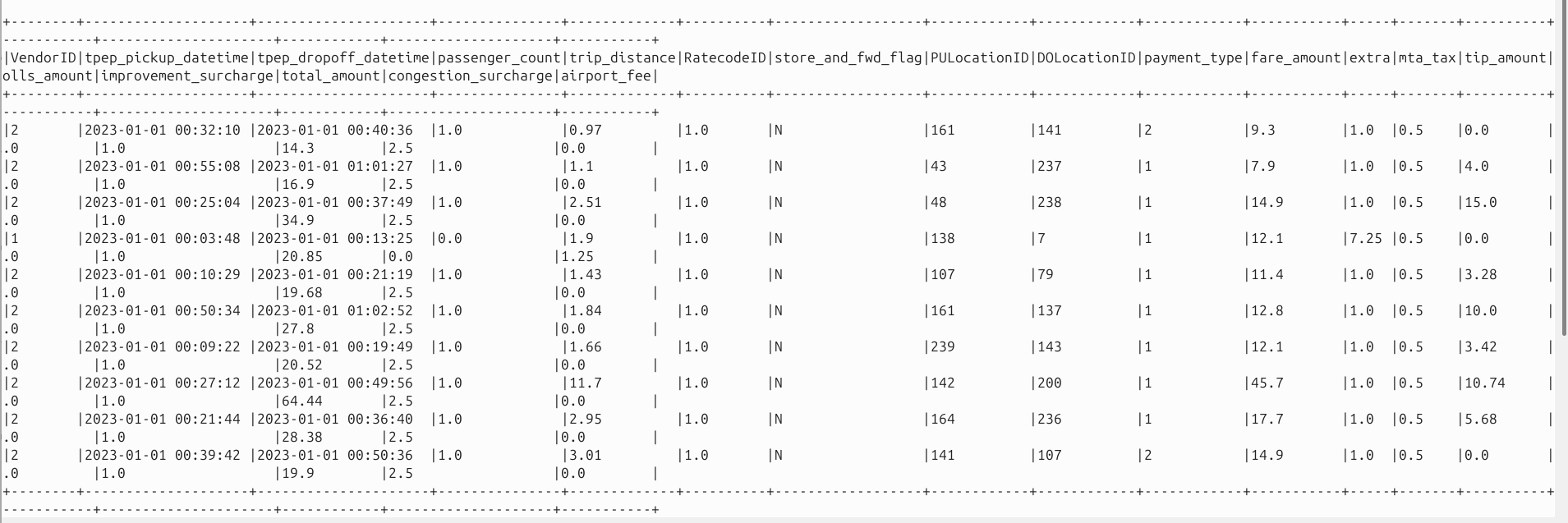
df.printSchema() df.show(10, truncate=False) PYEOF

**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:**

****

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

****



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")

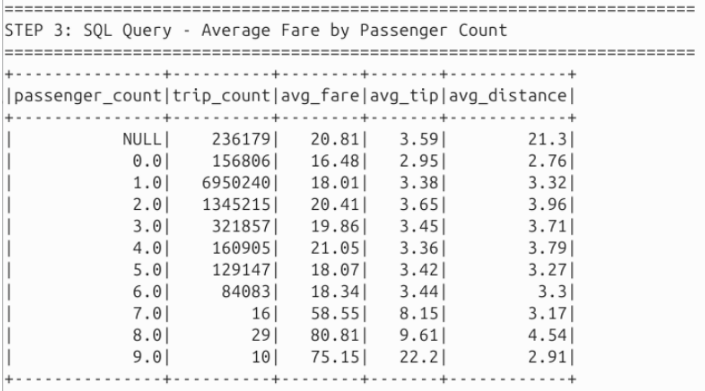
PYEOF

**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.**

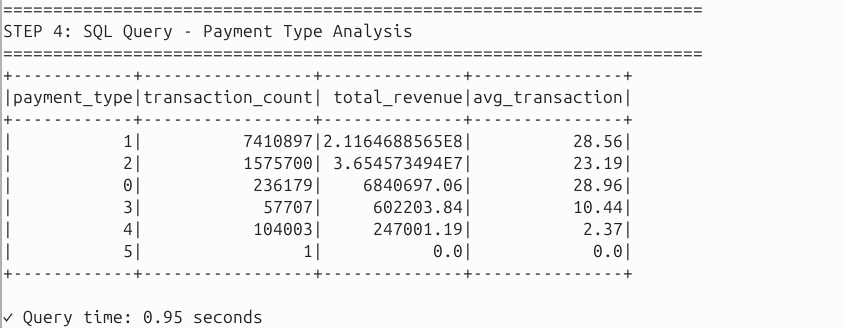
**Query1:** The first query is used to evaluate the total trips, average fair amount, average trip amount, and average trip distance for each passenger count. The results are ordered by passenger count.

**Output:**

****

**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:**



### 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

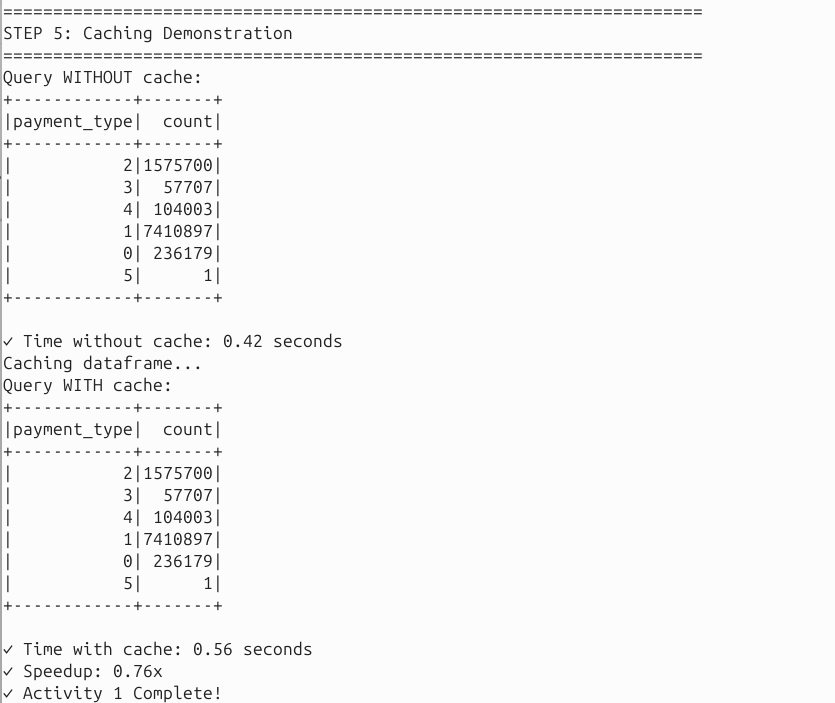
This part of the script compares the computation times for a DataFrame **without caching and with caching**. It runs a simple query:

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:**



### Part 5: Run the Script

mkdir -p output

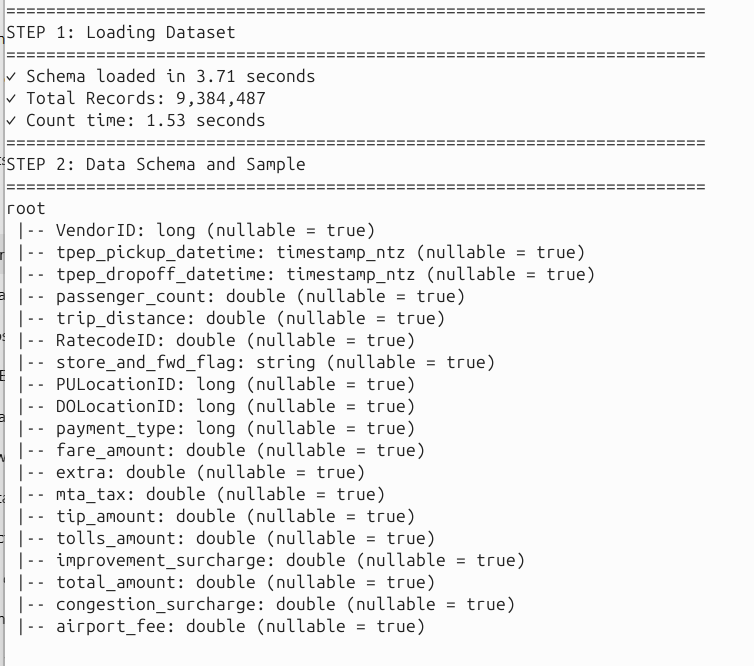
python3 activity1\_spark\_sql.py

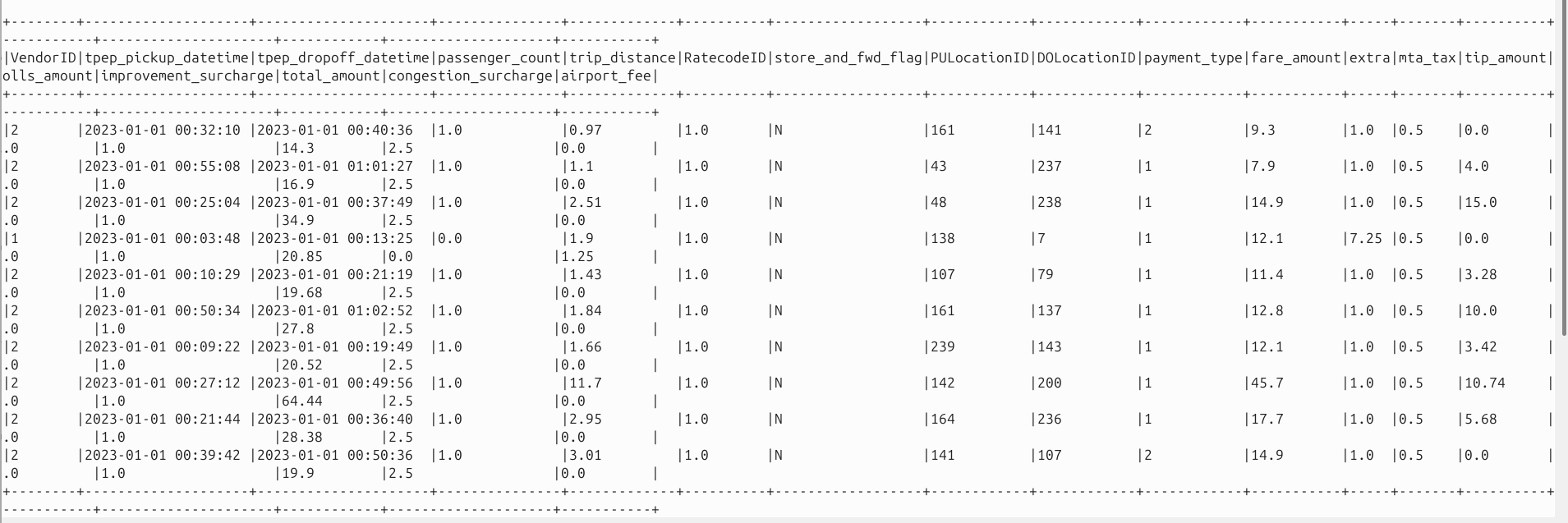
### 

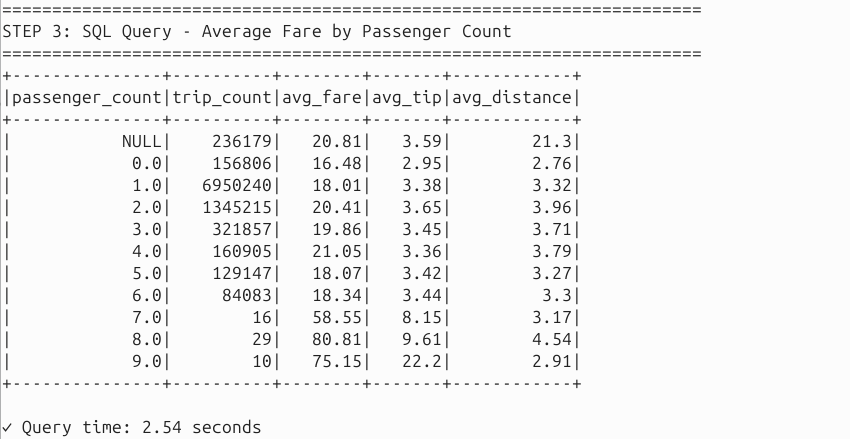
### 

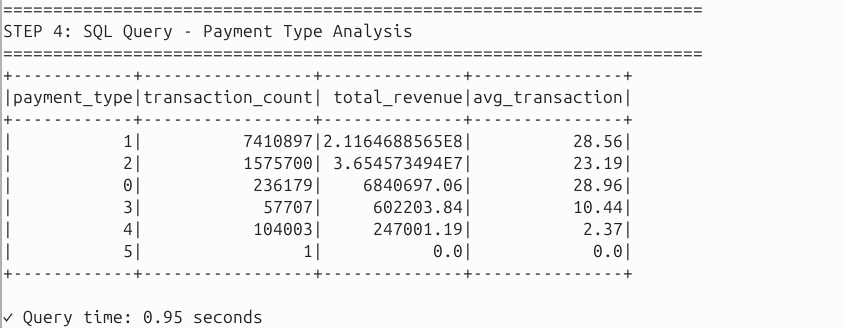
### REQUIRED SCREENSHOTS

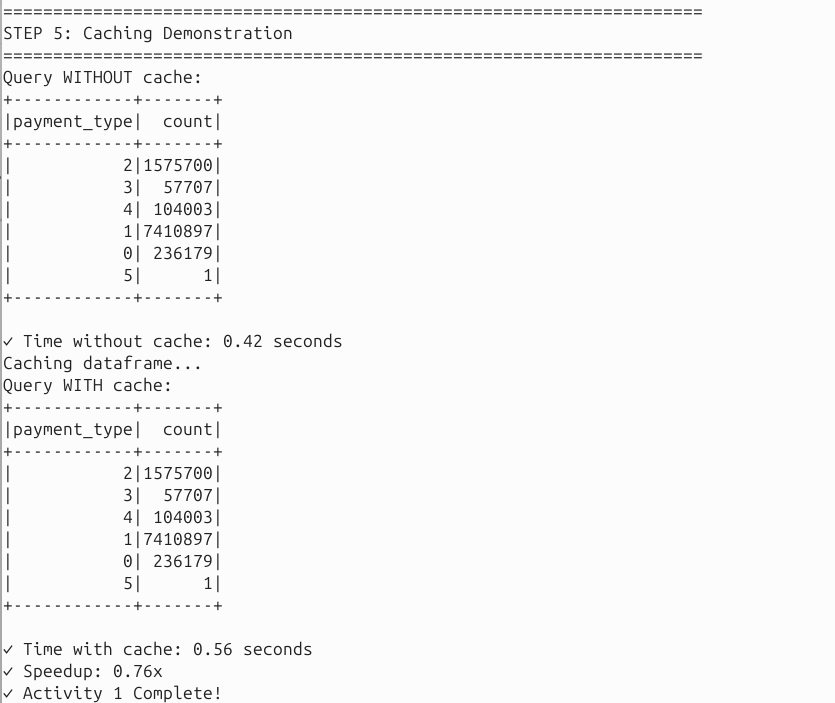
**Screenshot 1:** Complete terminal output showing all 5 steps executed successfully











## 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



### 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



$ 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

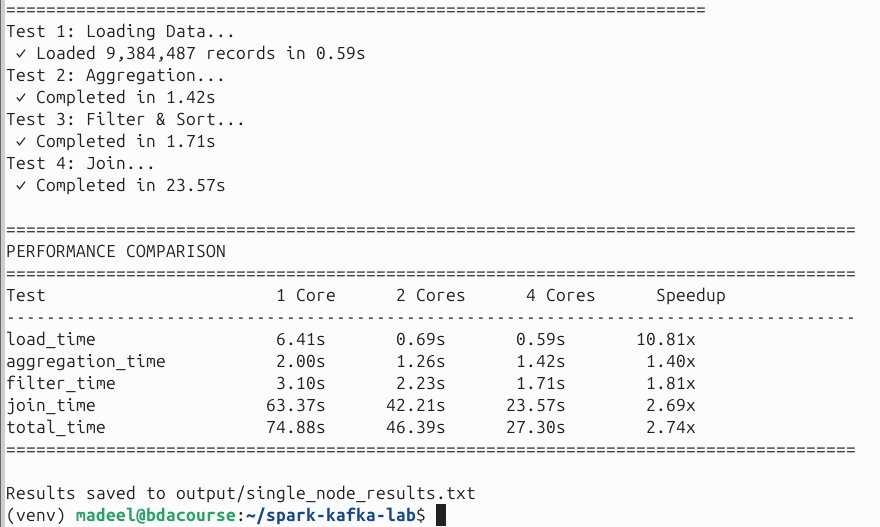
### Part 6: View Results File

cat output/single\_node\_results.txt

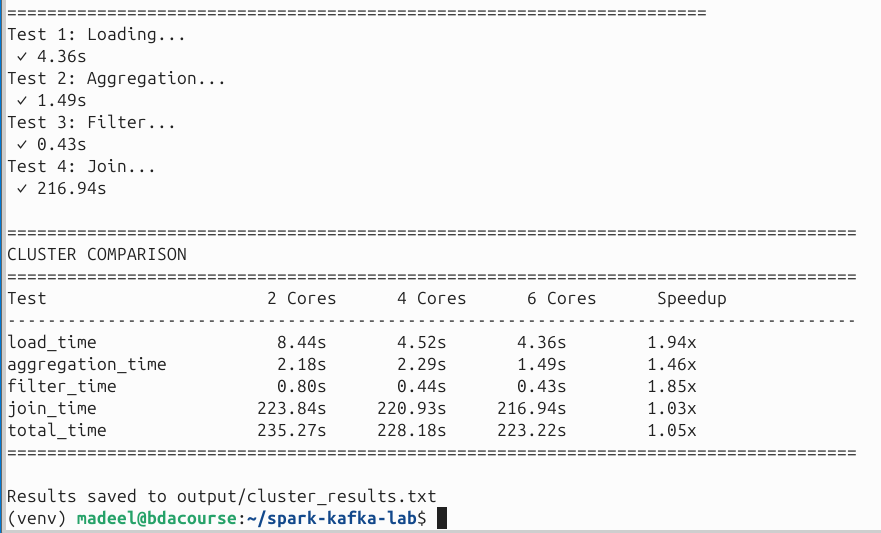
cat output/cluster\_results.txt

## REQUIRED SCREENSHOTS

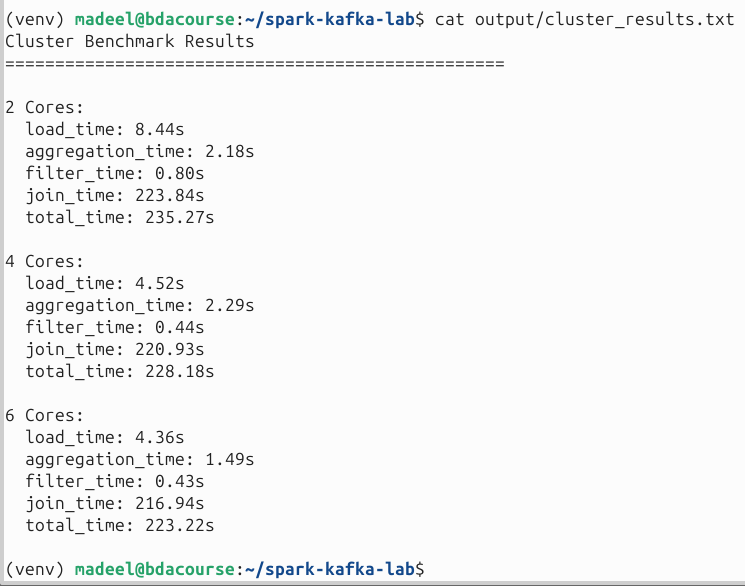
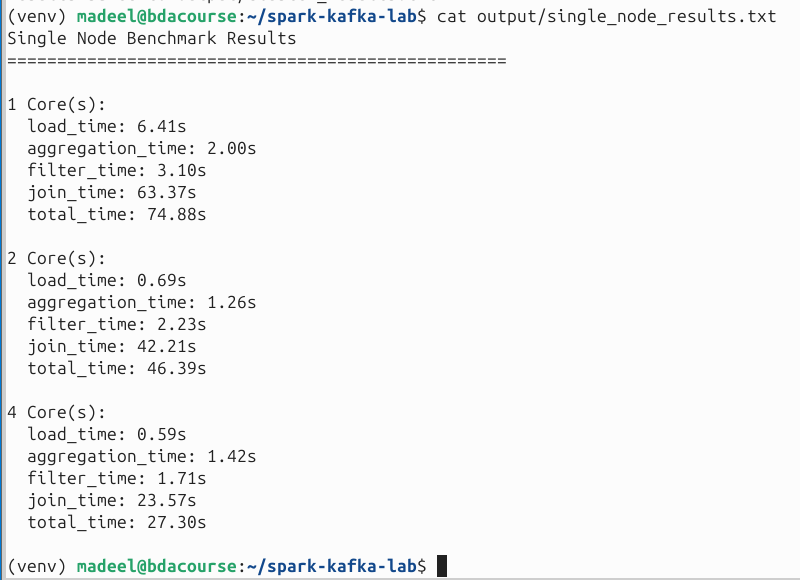
**Screenshot 2.1:** Terminal showing single-node benchmark comparison table (1, 2, 4 cores)



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



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



## 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



### 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

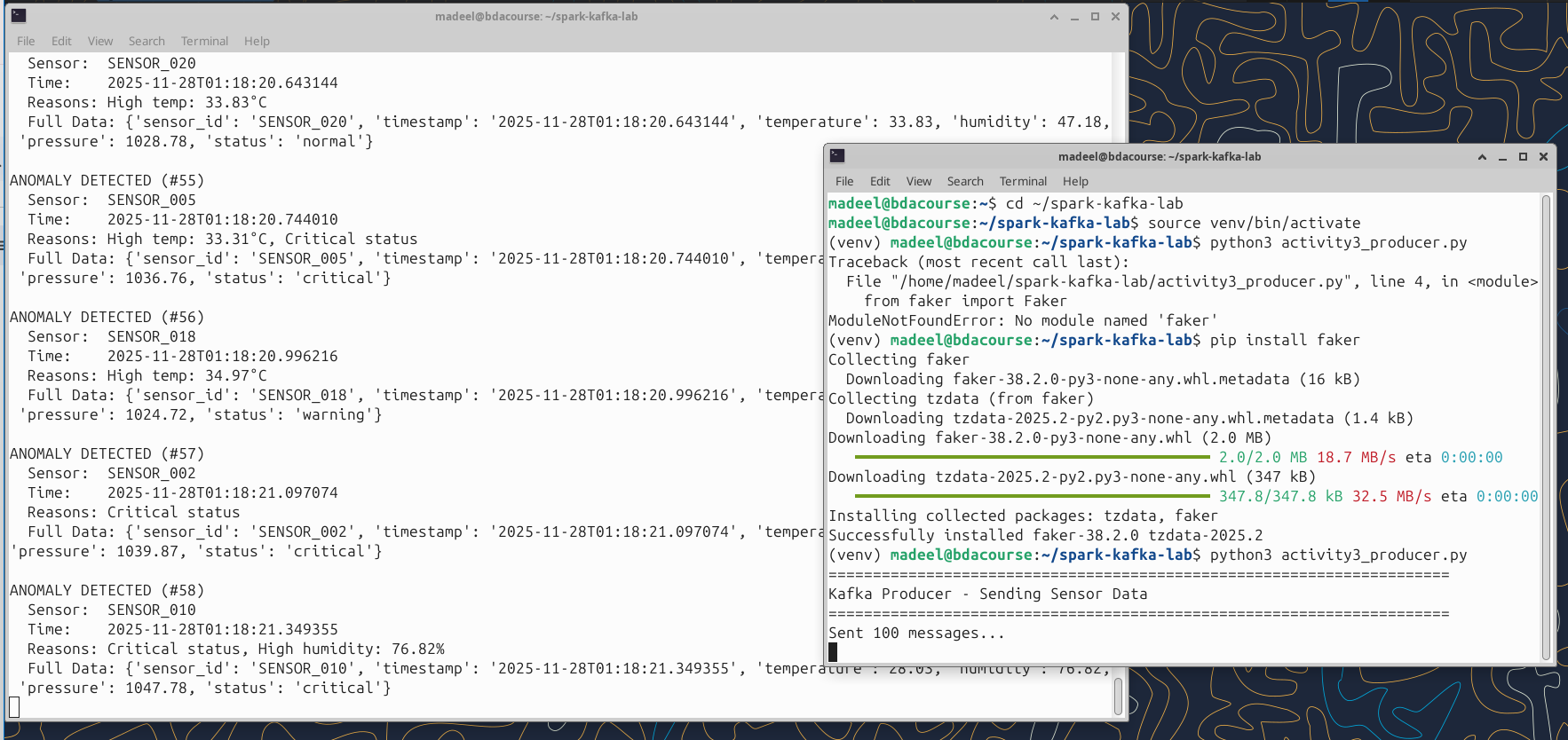
##### Terminal 1 - Consumer (Start First):

python3 activity3\_consumer.py

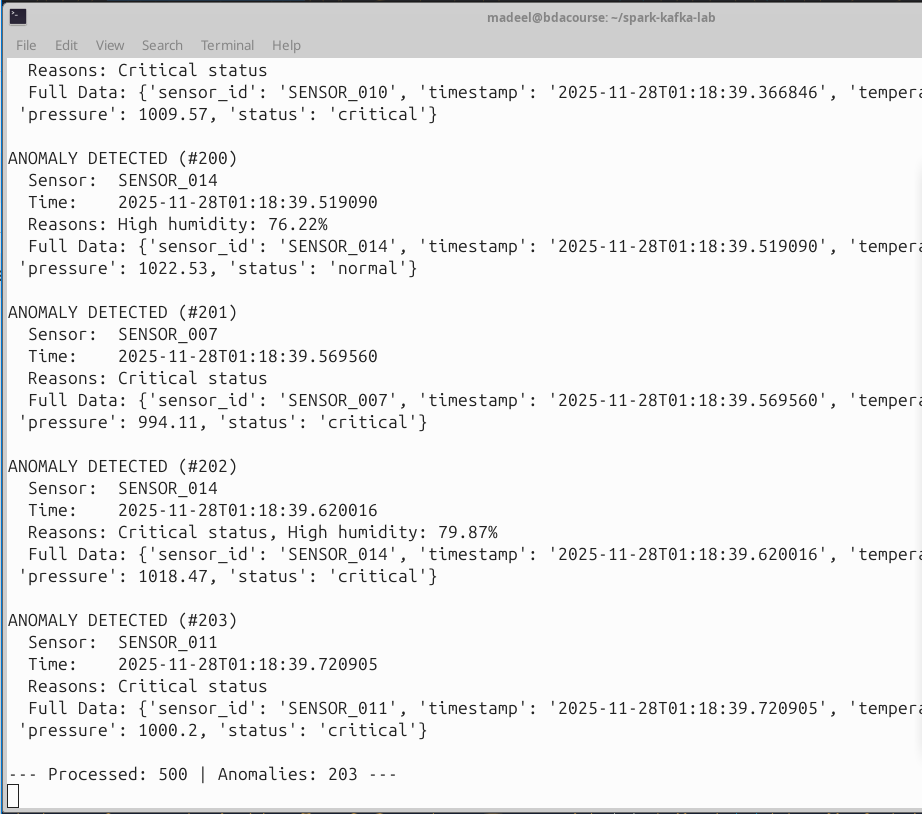
##### Terminal 2 - Producer:

python3 activity3\_producer.py

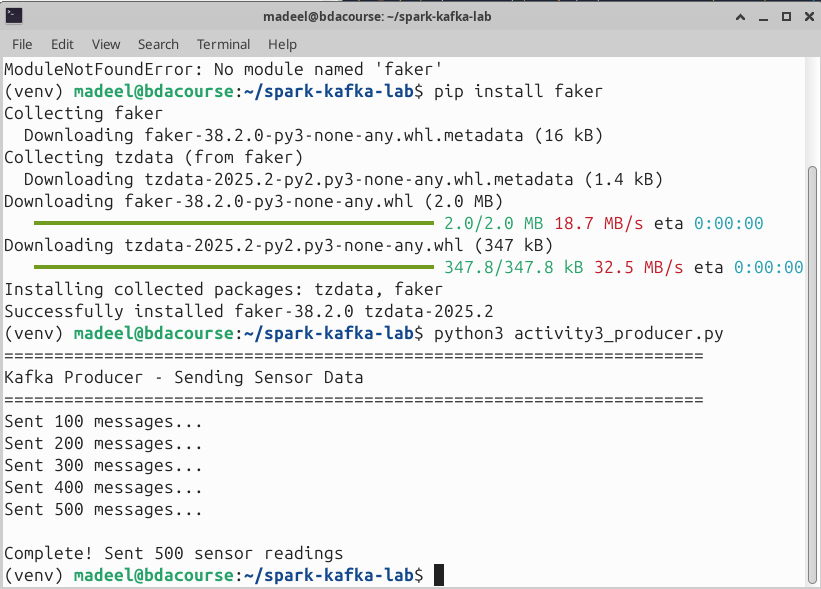
## REQUIRED SCREENSHOTS



**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()

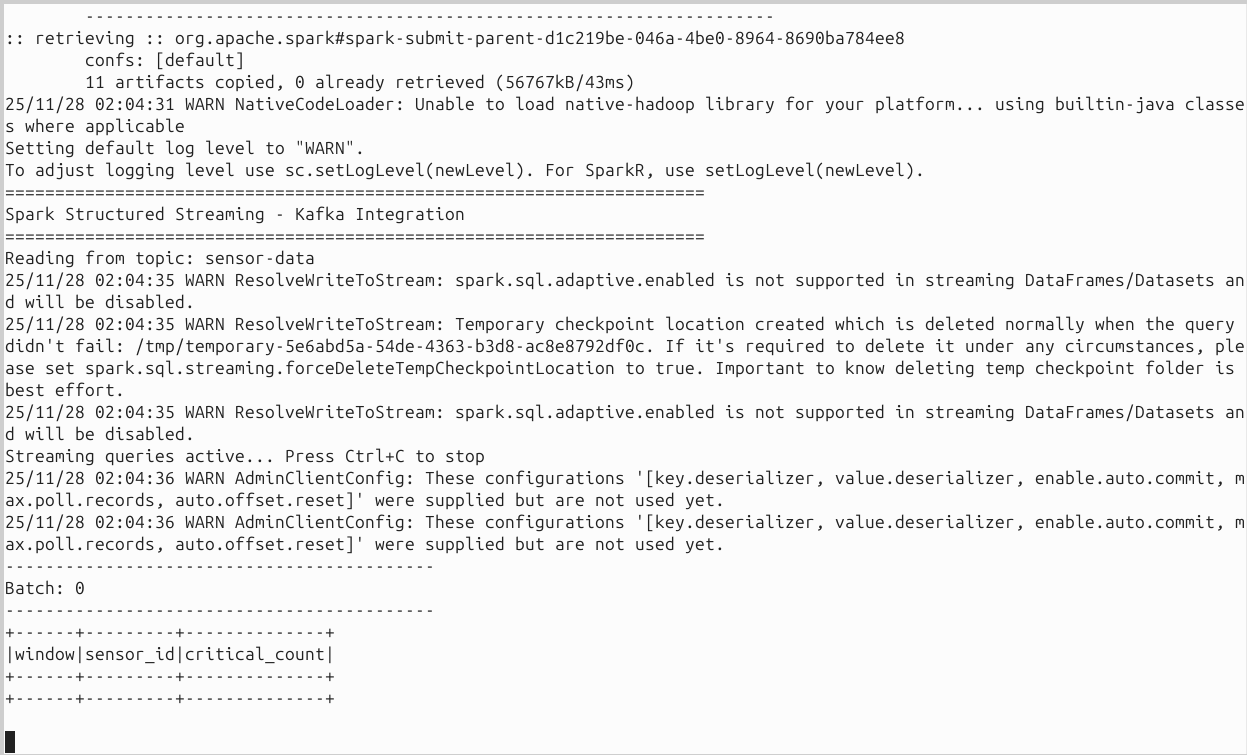
PYEOF

### Part 2: Run Streaming Pipeline (3 Terminals)

##### Terminal 1 - Start Spark Streaming:

rm -rf checkpoint/

python3 activity4\_streaming.py



##### 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)

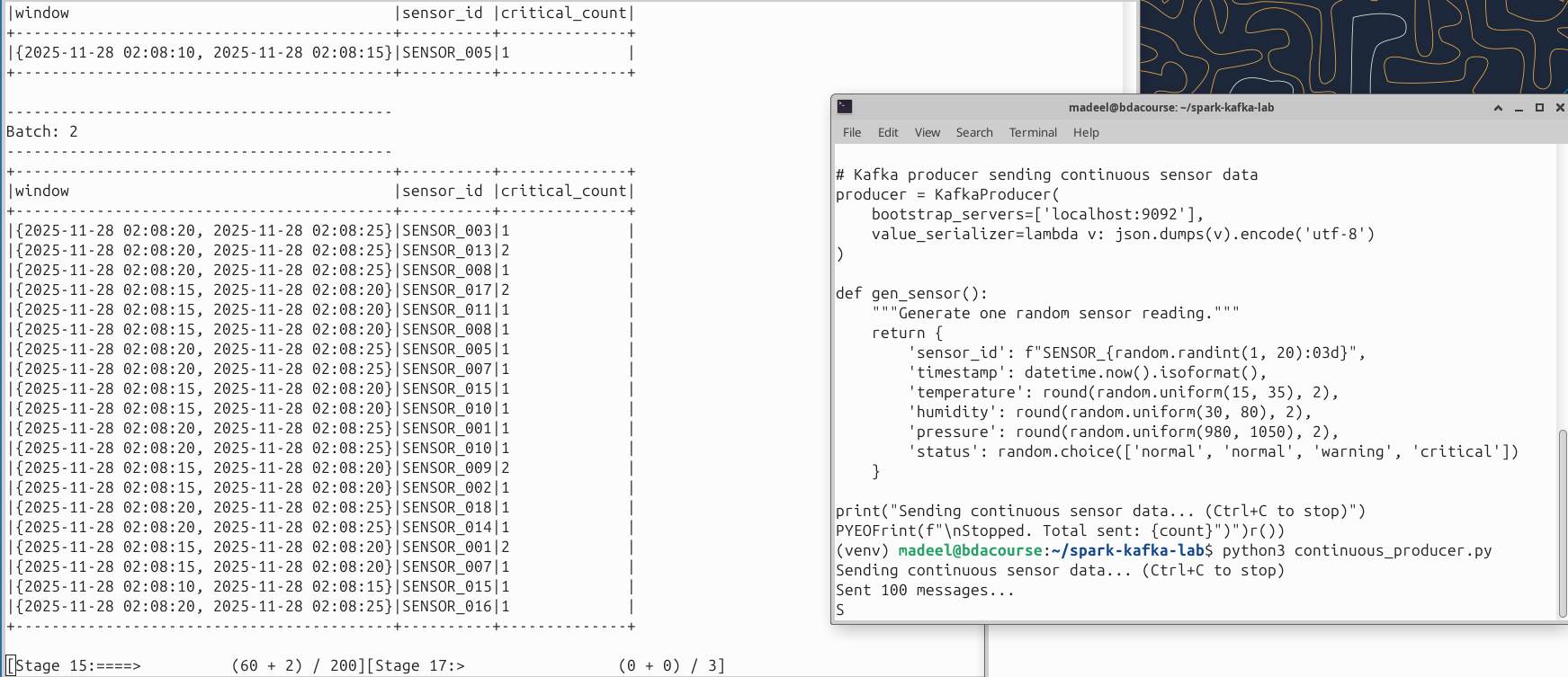
except KeyboardInterrupt:

producer.close()

print(f"\nStopped. Total sent: {count}")

PYEOF

python3 continuous\_producer.py



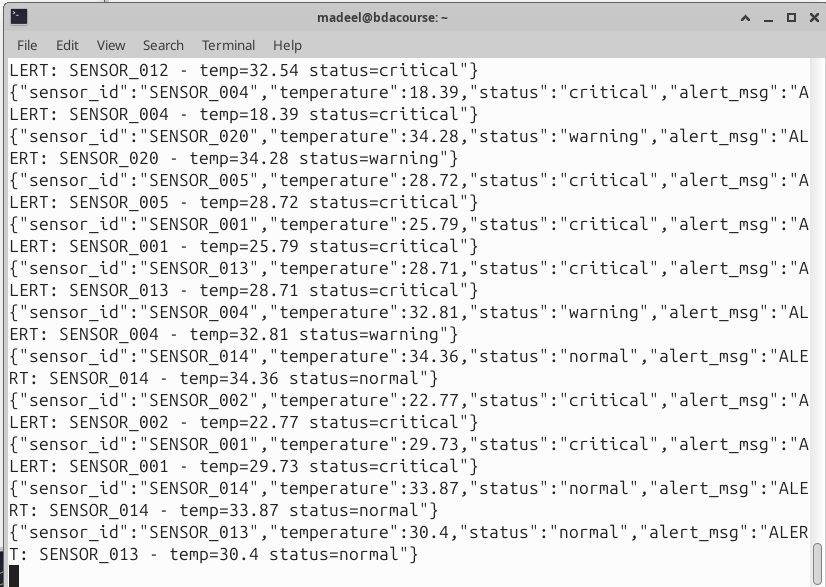
##### Terminal 3 - Monitor Alerts Topic:

kafka-console-consumer.sh \

--bootstrap-server localhost:9092 \

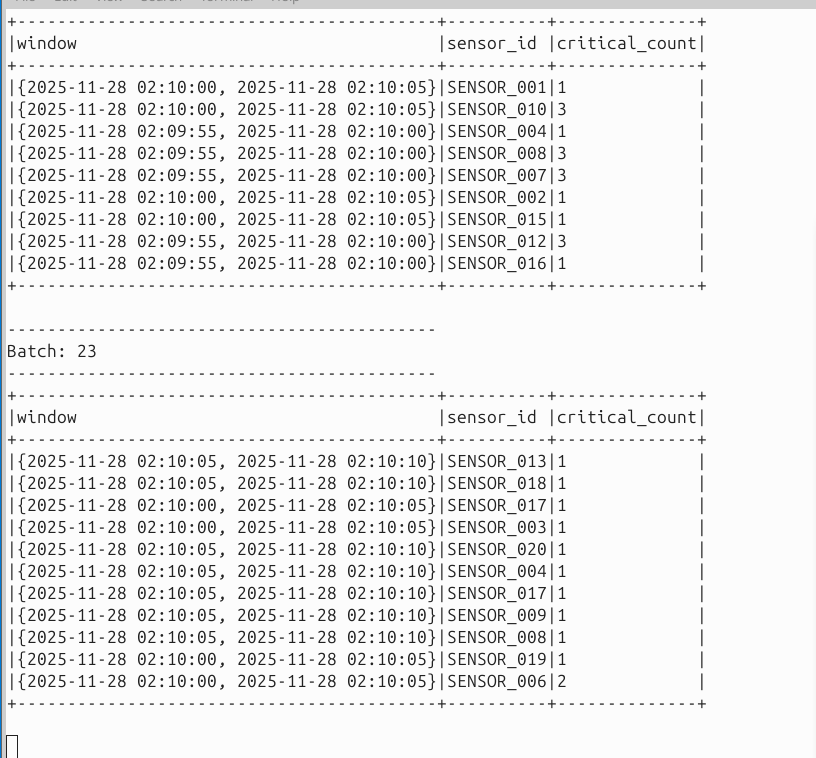
--topic alerts \

--from-beginning

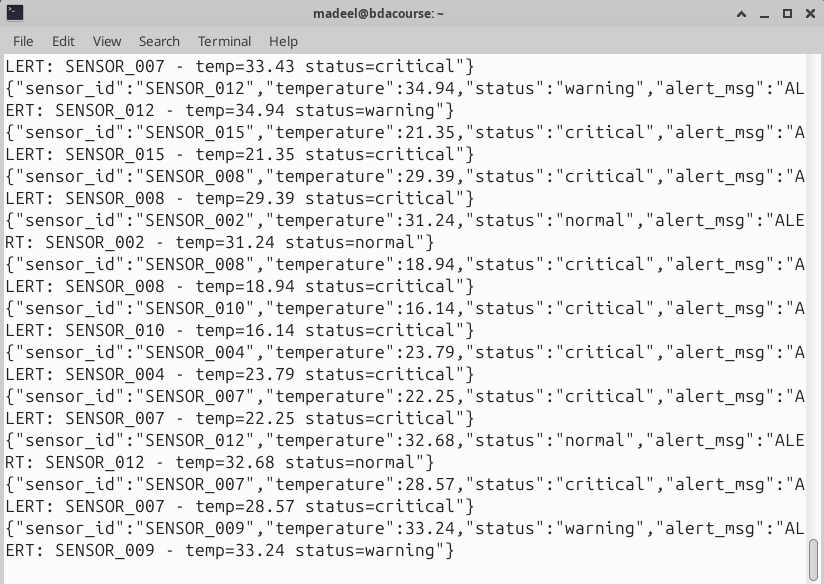


## REQUIRED SCREENSHOTS

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



**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](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}")

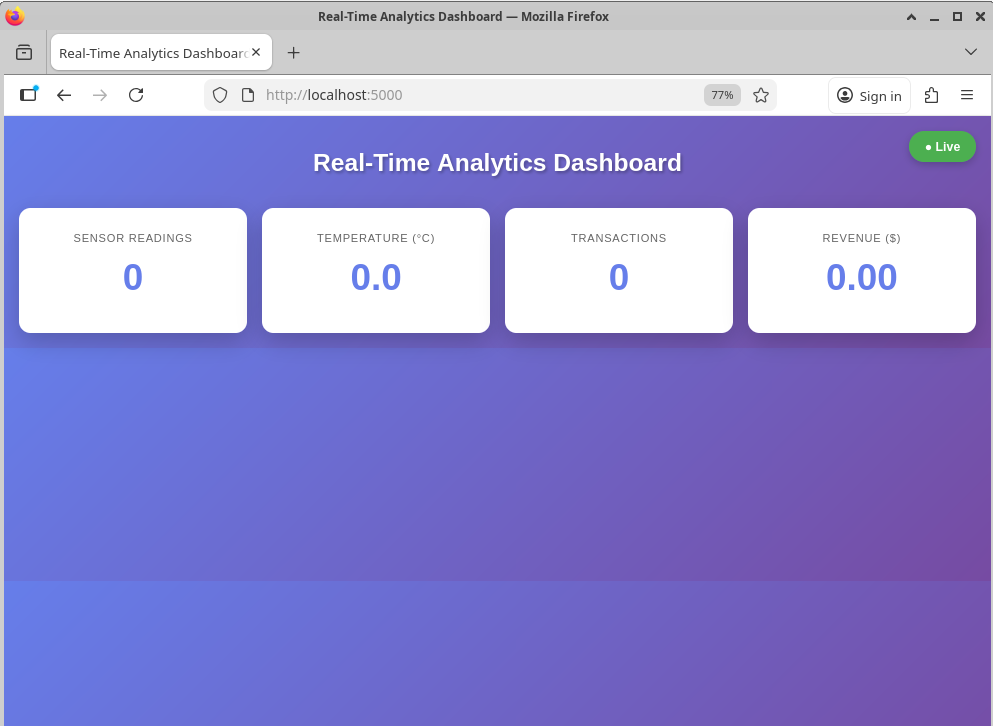
PYEOF

### Part 4: Run Dashboard (3 Terminals)

##### Terminal 1 - Dashboard Server:

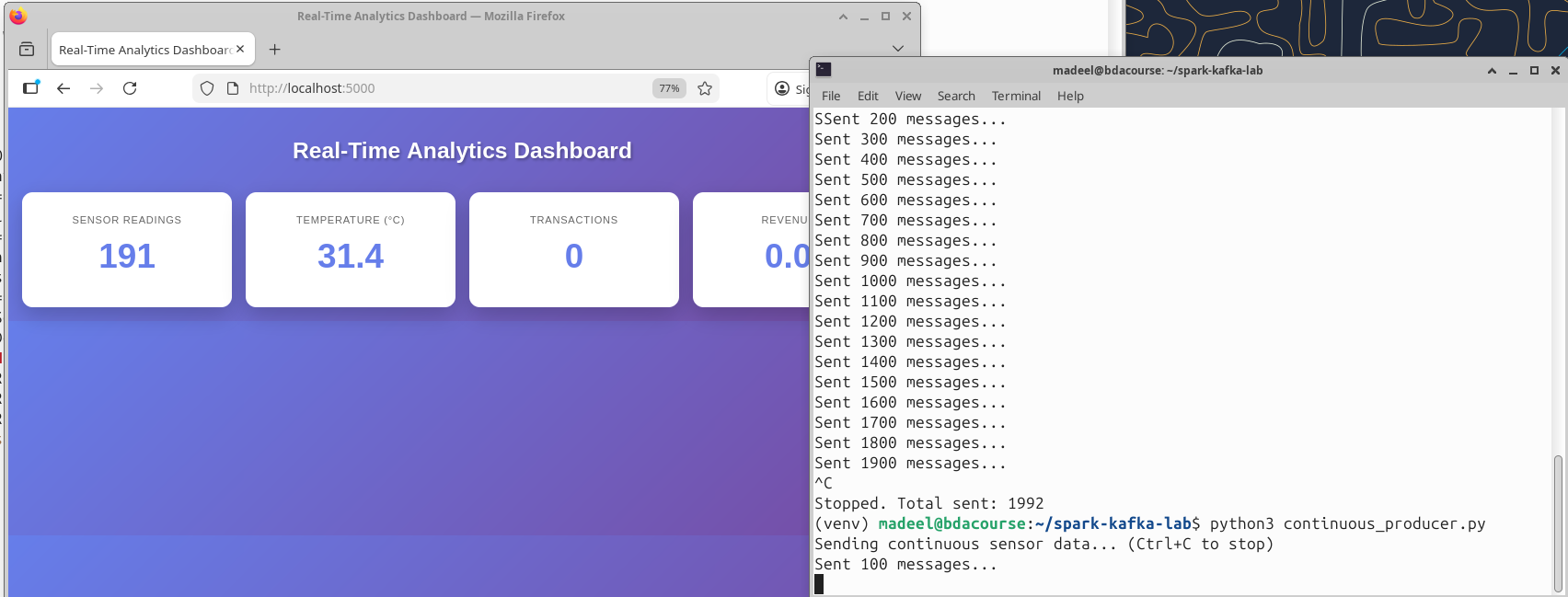
python3 activity5\_dashboard.py

# Open browser: [http://localhost:5000](http://localhost:5000/)



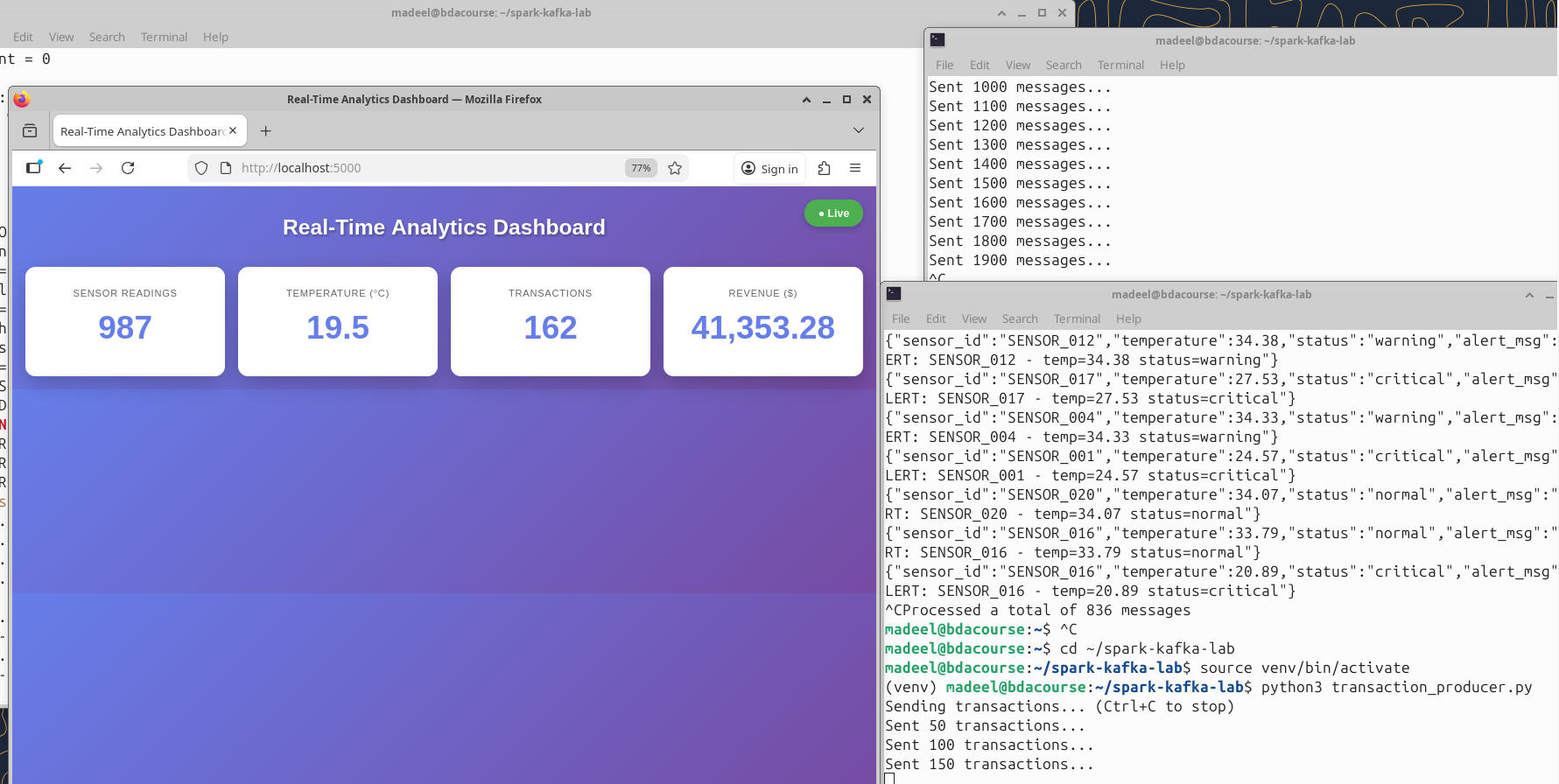
##### Terminal 2 - Sensor Data Producer:

python3 continuous\_producer.py



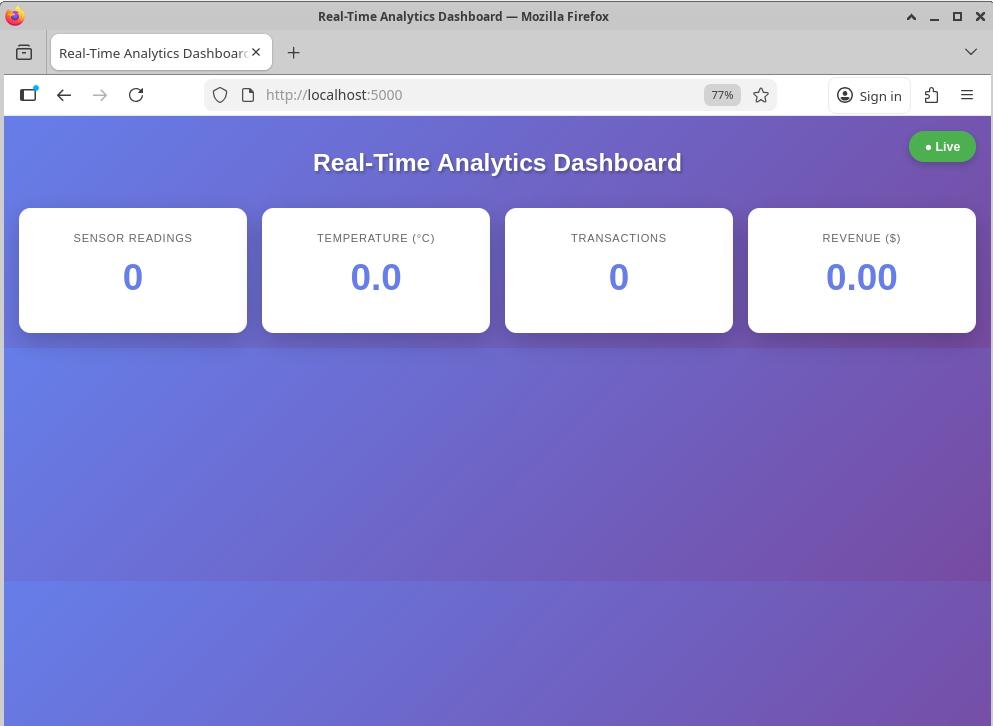
##### Terminal 3 - Transaction Producer:

python3 transaction\_producer.py

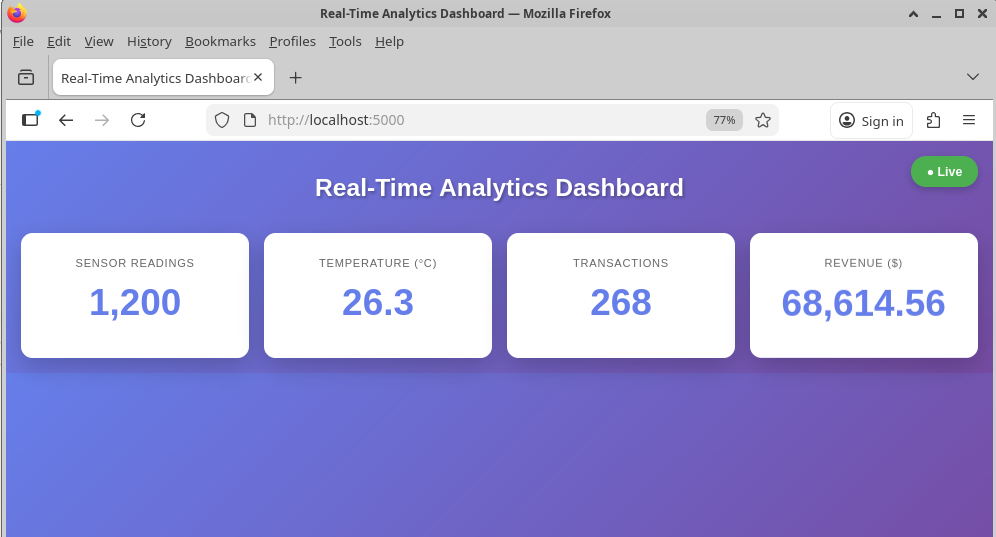


## REQUIRED SCREENSHOTS

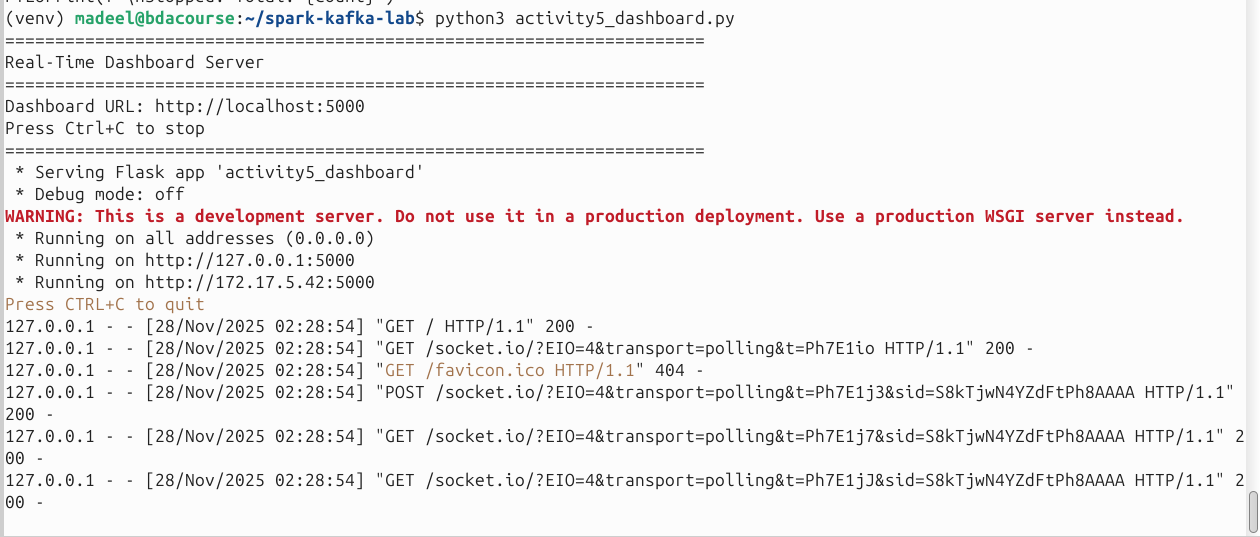
**Screenshot 5.1:** Web browser showing the live dashboard with all 4 metrics updating in real-time (initial state)



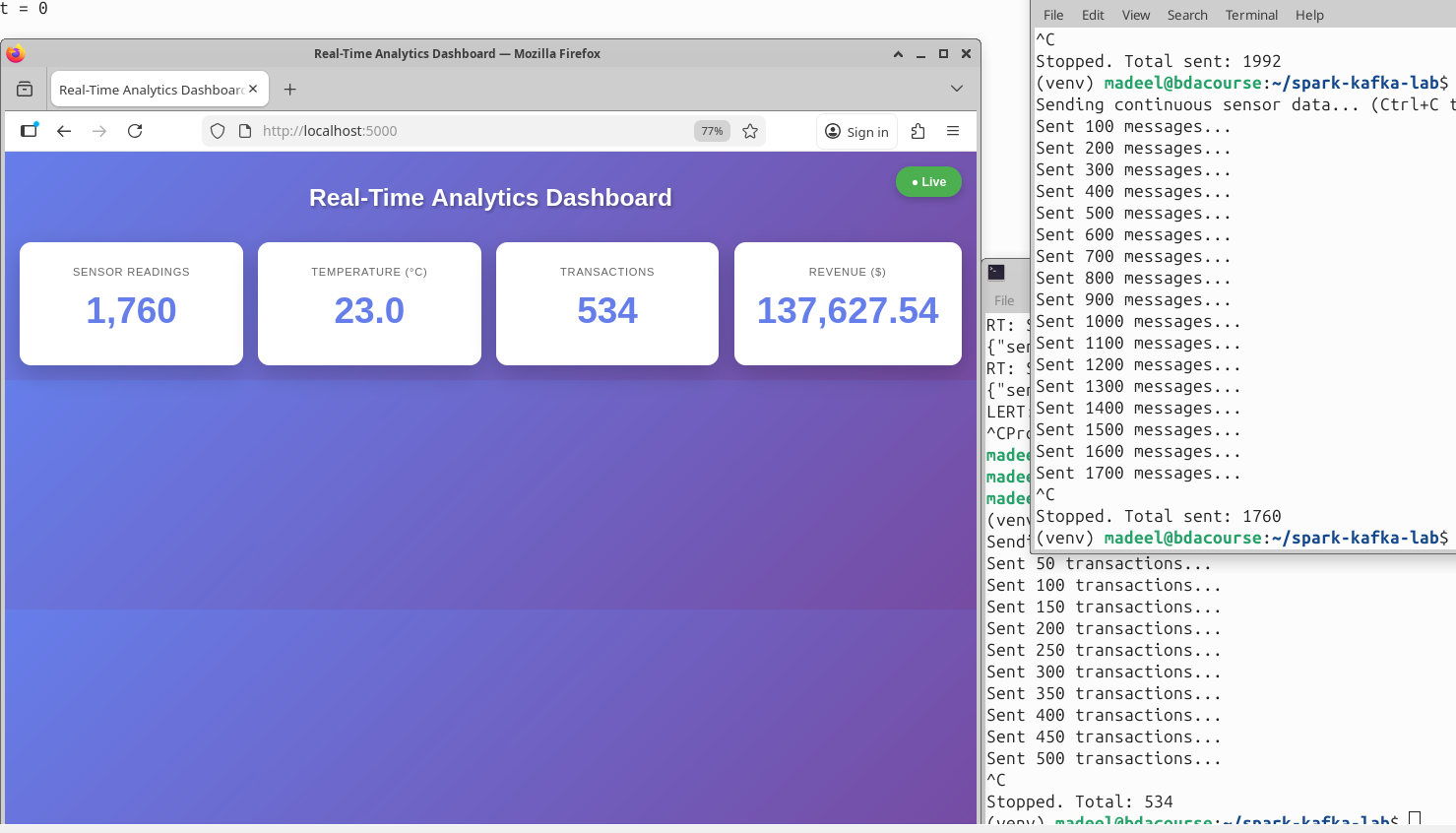
**Screenshot 5.2:** Same dashboard after 1–2 minutes showing updated metrics (sensor count, temperature, transactions, revenue)



**Screenshot 5.3:** Terminal showing dashboard server logs with connection messages



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.

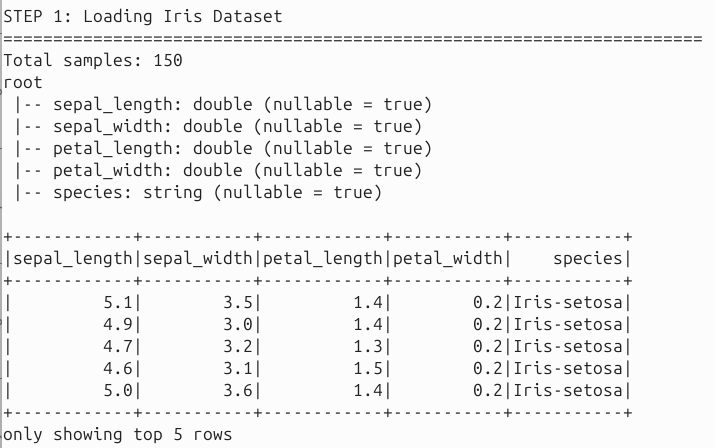
### Part 4: Run ML Pipeline

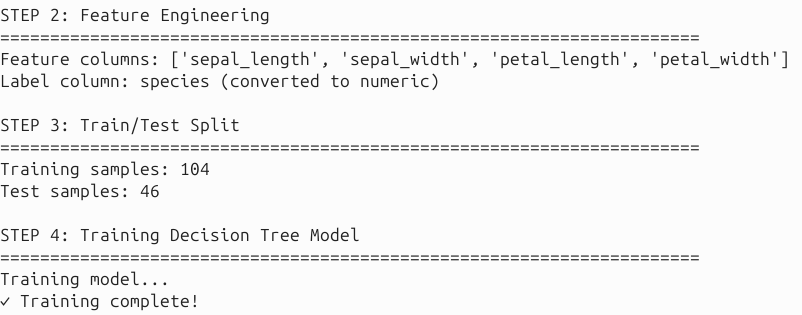
bash

python3 activity6\_ml.py

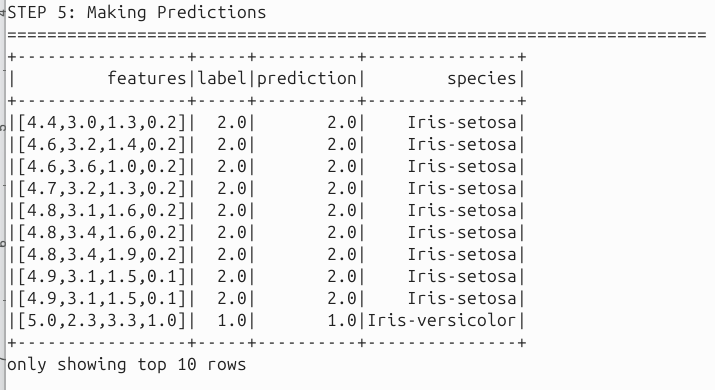
## REQUIRED SCREENSHOTS

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

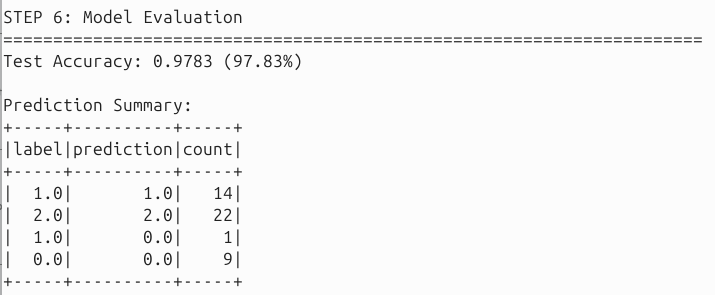




**Screenshot 6.2:** Terminal showing predictions with features, actual label, and predicted label (Step 5)



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



## 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

#### 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

#### 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

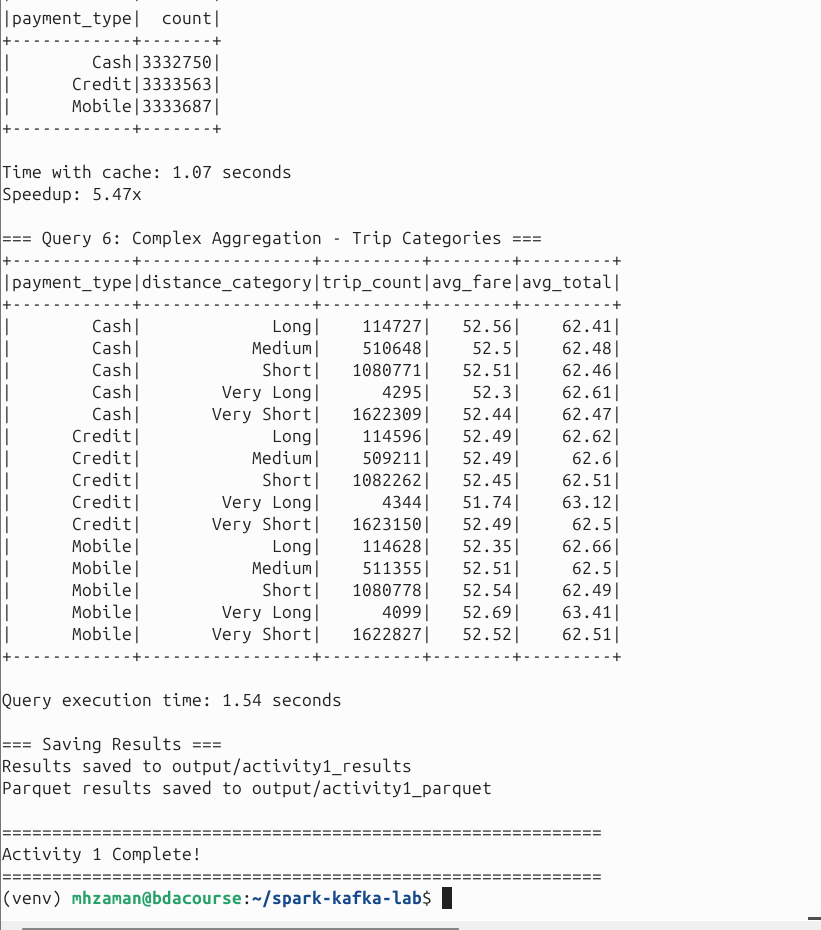
└── output/

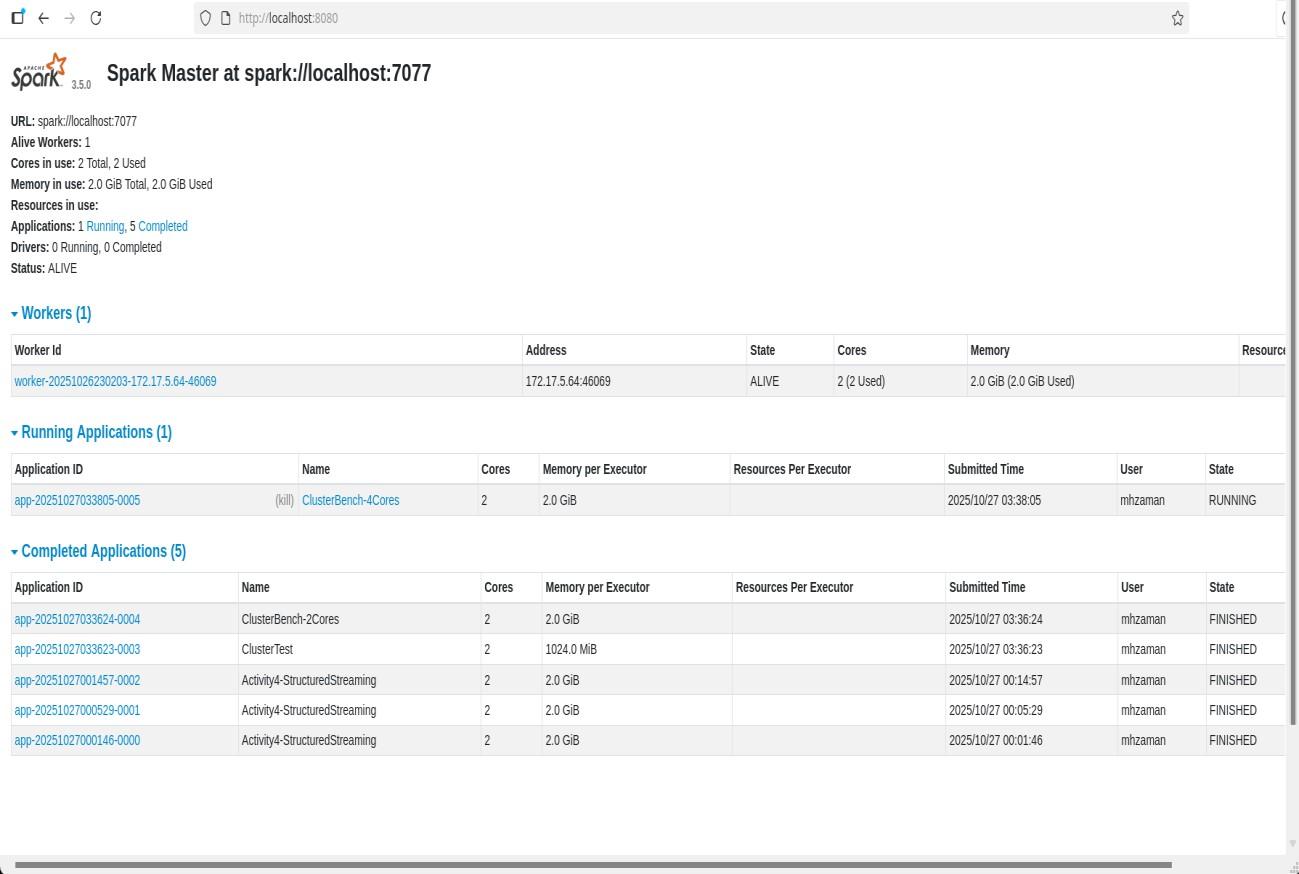
├── single\_node\_results.txt

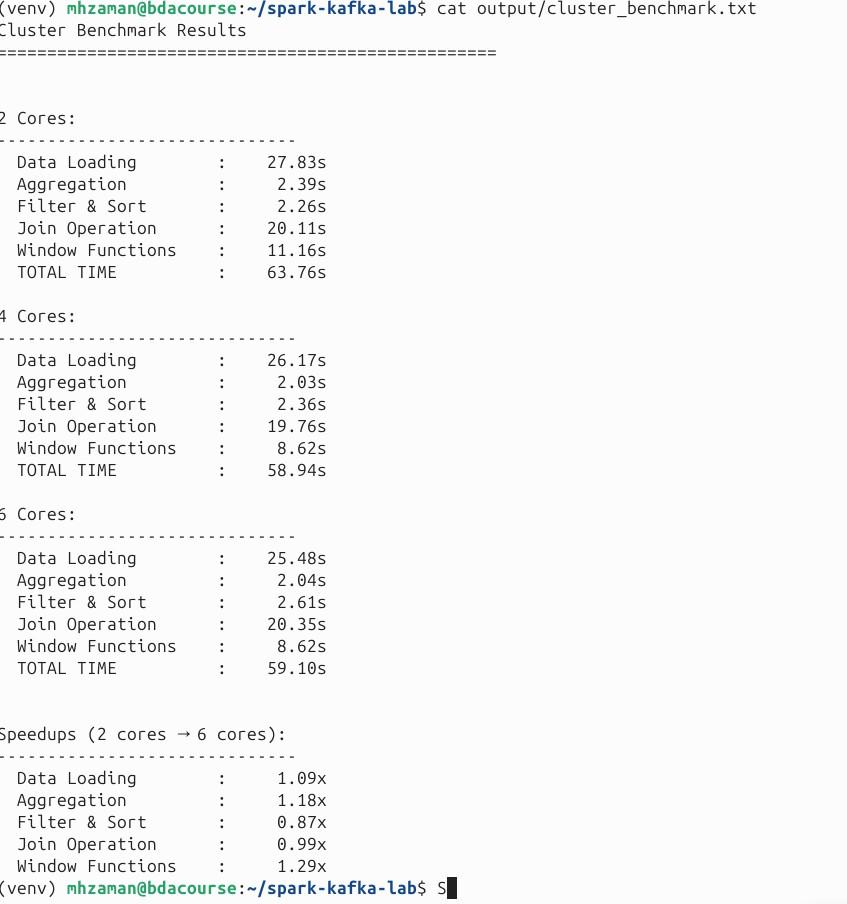
└── cluster\_results.txt

### Outcome & Achievements (Concise)

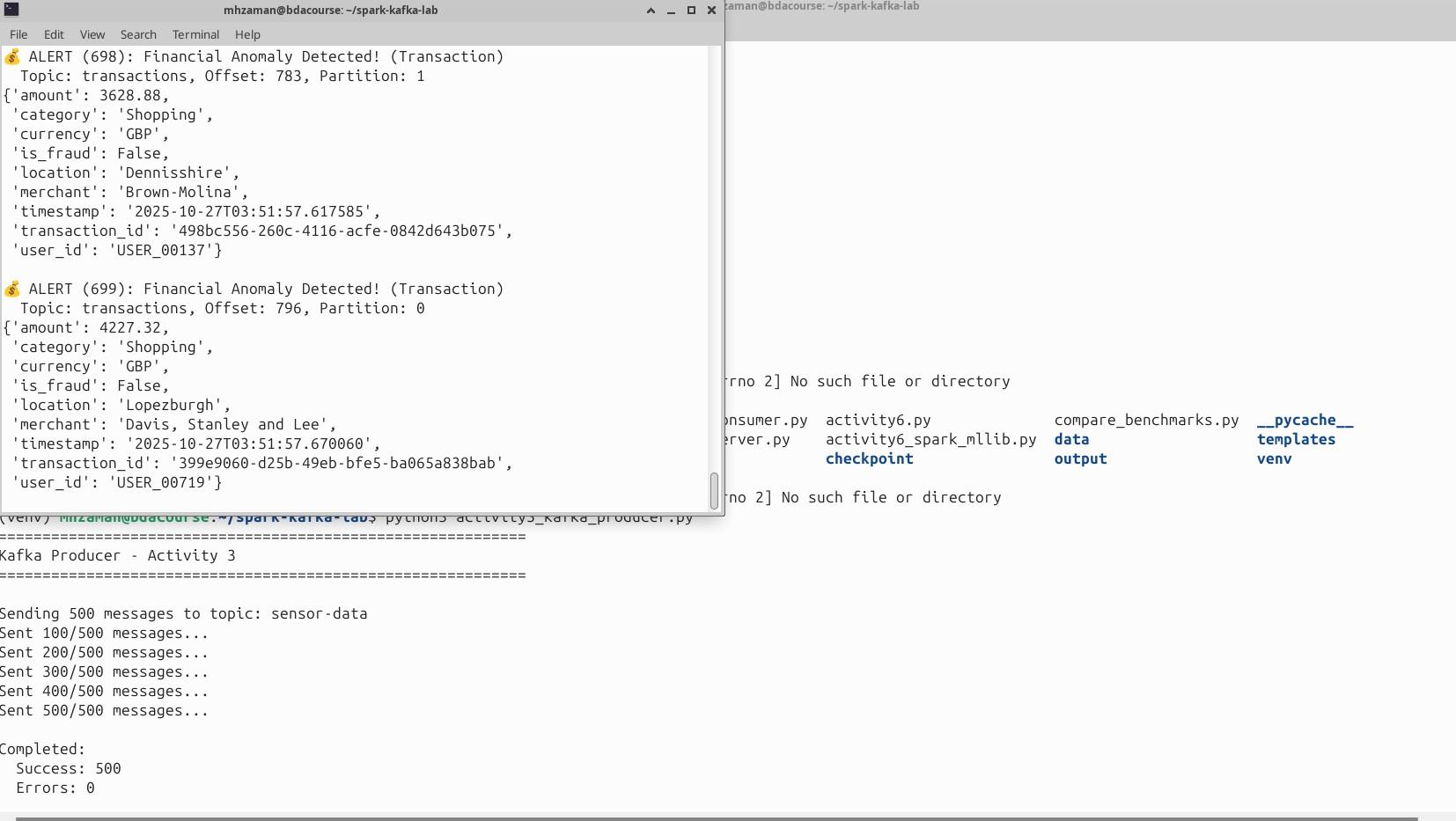
* **Dataset:** Implemented use of NYC Yellow Taxi Parquet (Jan–Mar 2023) as the primary large dataset; fallback sample created if download fails. Parquet improves IO and preserves schema.
* **Activity 1:** Loaded taxi data, ran SQL aggregations (avg fare by passenger\_count, payment-type revenue), demonstrated 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.

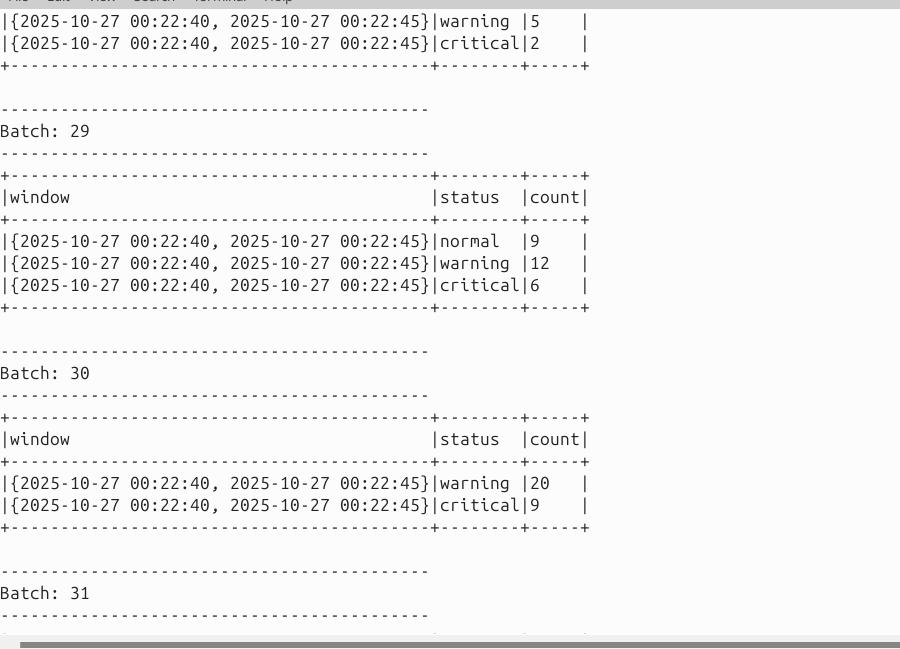
Activity 1:

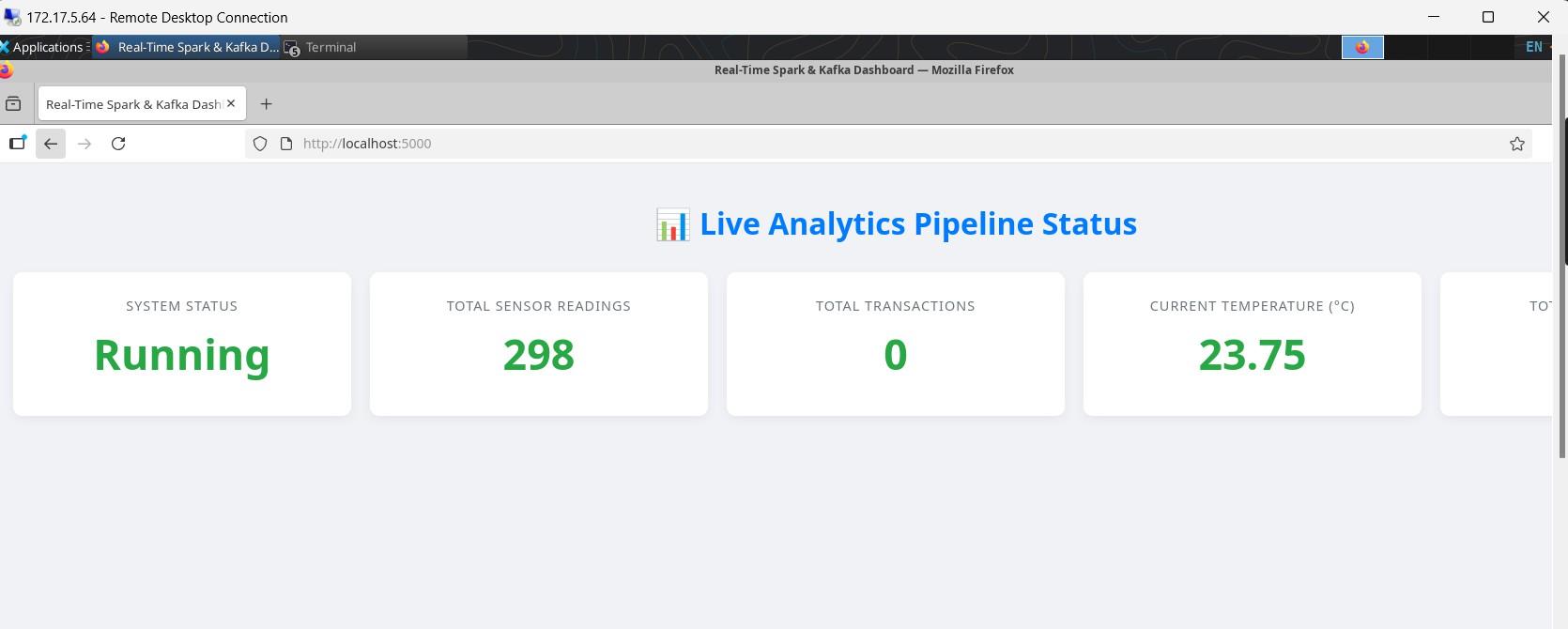
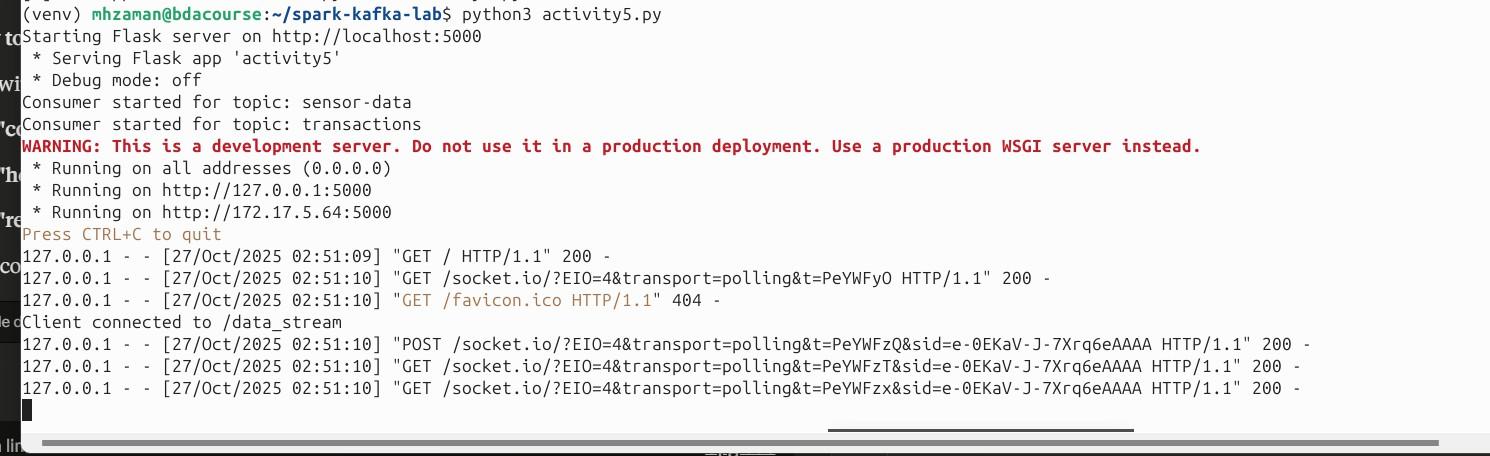
Activity 2:

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:

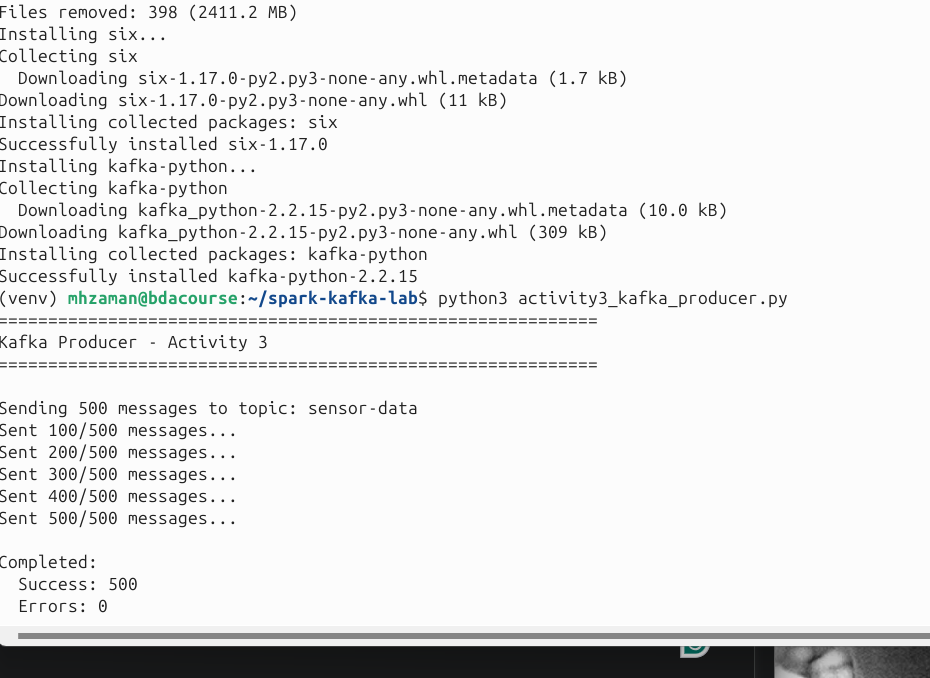
Activity 3:

Detecting Anamolies in real time data stream

Activity 4:

Activity 5:

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



Activity 6: