CASE STUDY 4

Title: Smart City Traffic Sensor Analytics using PySpark

A city government has installed traffic sensors across major roads.

Every sensor sends data every few seconds about traffic density, speed, and congestion levels.

The data is stored daily in a CSV file:

```
traffic_data.csv
```

Columns:

```
sensor_id
location
road_name
vehicle_count
avg_speed
temperature
timestamp
status
```

Sample (students will get a bigger file):

```
S101,Hyderabad,Hitech City Rd,45,32.5,29,2026-01-12 08:15:00,ACTIVE
S102,Hyderabad,Madhapur Rd,invalid,28.0,30,12/01/2026 08:16:00,ACTIVE
S103,Bangalore,Outer Ring Rd,60,,31,2026/01/12 08:17:00,ACTIVE
S104,Delhi,Ring Road,70,40.2,27,invalid_time,INACTIVE
S105,Mumbai,Western Express,55,35.1,28,2026-01-12 08:18:00,ACTIVE
```

This data looks structured but is unreliable:

- vehicle_count has invalid values
- avg_speed is missing in some rows
- timestamps are in different formats
- INACTIVE sensors still generate rows
- temperature is irrelevant for analytics but present

Your job is to create a **Traffic Intelligence Pipeline** using PySpark.

PHASE 1 – Ingestion

1. Read traffic_data.csv as StringType.

2. Print schema and count records.
3. Identify data quality issues by inspection.

PHASE 2 – Cleaning

1. Trim all string columns.

2. Clean vehicle_count:

    - Replace invalid and empty with null
    - Cast to IntegerType

3. Clean avg_speed:

    - Replace empty with null
    - Cast to DoubleType

4. Parse timestamp into:

```
event_time
```

with TimestampType supporting:

- yyyy-MM-dd HH:mm:ss
- dd/MM/yyyy HH:mm:ss
- yyyy/MM/dd HH:mm:ss

5. Keep original timestamp for audit.

PHASE 3 – Validation

1. Count invalid vehicle_count rows.
2. Count invalid timestamp rows.
3. Remove rows where:

```
status != "ACTIVE"
```

4. Validate row counts.

PHASE 4 – Traffic Metrics

1. Average speed per location.
2. Total vehicle count per road.

3. Peak traffic time per location.
4. Roads with lowest average speed (most congestion).

PHASE 5 – Window Functions

1. Rank roads by congestion (lowest speed).
2. For each location, rank roads by vehicle_count.
3. Identify top 3 congested roads per location.

PHASE 6 – Anomaly Detection

1. Detect sudden drop in avg_speed.
2. Detect sudden spikes in vehicle_count.
3. Use:

```
lag()
```

window function to compare with previous event.

PHASE 7 – Performance Engineering

1. Check number of partitions.
2. Use explain(True) on congestion queries.
3. Repartition by location.
4. Cache cleaned DataFrame.
5. Compare execution plans.

PHASE 8 – RDD

1. Convert cleaned DataFrame to RDD.

2. Compute:

    - Total vehicle count using reduce.
    - Count of records per location using map-reduce.

3. Explain why DataFrames are better for this case.

PHASE 9 – Sorting & Set Operations

1. Sort roads by highest congestion.

2. Create two sets:

- Roads with avg_speed < 25
- Roads with vehicle_count > 60

3. Find:

- Roads in both sets
- Roads in only one set

PHASE 10 – Storage

1. Write cleaned traffic data to:

```
Parquet (partitioned by location)
```

2. Write congestion analytics to:

```
ORC
```

3. Read back and validate.
