
# BigData 2025 Group 17

Project Big Data is provided by University of Tartu.

**Students:** Andres Caceres, Fidan Karimova, Moiz Ahmad, Siddiga Gadirova

# Query 0

## Data pipeline

Our data cleaning pipeline employs a modern architecture using Kafka for data ingestion and Delta Lake for reliable storage.

```
Raw CSV Files → Kafka Topics → Spark Processing → Delta Tables
```

As an initial step, we sampled the data, taking just 4% of the available information using distributed sampling; we enforced the data left to be distributed among all the records; the starting CSV had 33 GB.

```
sample_df = full_sdf.sample(fraction=0.04, seed=42)
```

### 3.1 Data Ingestion: CSV to Kafka

The pipeline begins with ingesting CSV data into Kafka topics:

```
def ingest_csv_to_kafka(csv_path, batch_size=1000):
    """
    Read a CSV file and publish records to Kafka using Spark's partitioning
    """
    print(f"Ingesting data from {csv_path} to Kafka topic 'raw-taxi-data'")

    # Load the data
    taxi_data = spark.read.csv(csv_path, header=False,
schema=raw_taxi_schema)

    # Get total count for reporting
    total_count = taxi_data.count()
    print(f"Total records to process: {total_count}")

    # Add partition key
    taxi_data = taxi_data.withColumn("kafka_key", col("medallion"))

    # Create the JSON structure
    kafka_batch = taxi_data.select(
        col("kafka_key").cast("string"),
        to_json(
            struct(*[col(c) for c in taxi_data.columns if c !=
"kafka_key"])
        ).alias("value")
    )

    # Write to Kafka in one go (Spark will handle the batching internally)
    kafka_batch.write.format("kafka") \
        .option("kafka.bootstrap.servers", "kafka:9092") \
        .option("topic", "raw-taxi-data") \
        .option("maxOffsetsPerTrigger", batch_size) \
        .save()

    print(f"Finished ingesting data to Kafka topic 'raw-taxi-data'")
```

The pipeline continues with a streaming consumer that reads from Kafka and writes to a Delta table:

```python
def run_consumer(skip_table_creation=False):
    # 1. Read from Kafka
    raw_stream = spark \
        .readStream \
        .format("kafka") \
        .option("kafka.bootstrap.servers", "kafka:9092") \
        .option("subscribe", "raw-taxi-data") \
        .option("startingOffsets", "earliest") \
        .load()

    # 2. Parse JSON data
    filtered_kafka_stream = raw_stream.select(
        col("key").cast("string").alias("kafka_key"),
        col("value").cast("string").alias("json_data"),
        col("timestamp").alias("kafka_timestamp")
    )

    raw_data_df = filtered_kafka_stream.select(
        "kafka_key",
        from_json("json_data", raw_taxi_schema).alias("data"),
        "kafka_timestamp"
    ).select("kafka_key", "kafka_timestamp", "data.*")

    # 3. Write raw data to Delta
    raw_query = (raw_data_df.writeStream
        .outputMode("append")
        .format("delta")
        .queryName("raw_taxi_query")
        .trigger(processingTime="10 second")
        .option("checkpointLocation", RAW_CHECKPOINT_PATH)
        .start(RAW_OUTPUT_PATH)
    )

    # Wait for some data to be written
    print("Waiting for raw data to be processed...")
    time.sleep(60)

    # 4. Create table in the metastore
    if not skip_table_creation:
        create_table_if_exists(RAW_OUTPUT_PATH, RAW_TABLE_NAME)

    return raw_query
```

Then, we applied clean-up to the records we had persisted in the delta table.

**Coordinate validation**: Ensures latitude and longitude values are within valid ranges
**Speed validation**: Filters out trips with impossibly high speeds (>100 mph)
**Distance validation**: Removes zero or negative trip distances
**Time validation**: Eliminates zero or negative trip durations
**Null handling**: Filters out records without a medallion identifier

| Namespace | Table Name | Is Temporary |
|---|---|---|
| default | clean_taxi_data | false |
| default | raw_taxi_data | false |

Both tables are stored in Delta Lake format and contain:

- **raw_taxi_data**: Original taxi trip data ingested from Kafka
- **clean_taxi_data**: Validated and cleaned taxi trip data after applying data quality rules

Before the processing, we had 6927726 records; after the clean-up, 6871525 records were left.

# Query 1

This query analyzed trip data using PySpark to uncover frequent taxi routes and peak traffic periods. The process involved mapping geographical locations to grid cells, identifying high-traffic time windows, and ranking the most common routes.

Instead of inferring the schema, we used the same schema previously used in the clean-up stage.

```
def create_raw_taxi_schema():
    """
    Create the complete schema for NYC taxi data based on the DEBS Grand
Challenge 2015 specification
```

```
    This schema includes all fields from the original dataset:
    - Basic identifiers (medallion, hack_license)
    - Time data (pickup_datetime, dropoff_datetime, trip_time_in_secs)
    - Trip information (trip_distance)
    - Location coordinates (pickup/dropoff longitude/latitude)
    - Payment information (payment_type, fare_amount, surcharge, mta_tax,
tip_amount, tolls_amount)
    """
    return StructType([
        # Taxi and driver identifiers
        StructField("medallion", StringType(), True),         # Taxi
vehicle identifier (md5sum)
        StructField("hack_license", StringType(), True),      # Taxi
license identifier (md5sum)

        # Trip time information
        StructField("pickup_datetime", TimestampType(), True),   # Time of
passenger pickup
        StructField("dropoff_datetime", TimestampType(), True),  # Time of
passenger dropoff
        StructField("trip_time_in_secs", IntegerType(), True),   # Duration
of the trip in seconds

        # Trip distance
        StructField("trip_distance", DoubleType(), True),     # Trip
distance in miles

        # Pickup coordinates
        StructField("pickup_longitude", DoubleType(), True),  # Longitude
coordinate of pickup
        StructField("pickup_latitude", DoubleType(), True),   # Latitude
coordinate of pickup

        # Dropoff coordinates
        StructField("dropoff_longitude", DoubleType(), True), # Longitude
coordinate of dropoff
        StructField("dropoff_latitude", DoubleType(), True),  # Latitude
coordinate of dropoff

        # Payment information
        StructField("payment_type", StringType(), True),      # Payment
method (credit card or cash)
```

```
        StructField("fare_amount", DoubleType(), True),       # Fare amount
in dollars
        StructField("surcharge", DoubleType(), True),         # Surcharge
in dollars
        StructField("mta_tax", DoubleType(), True),           # Tax in
dollars
        StructField("tip_amount", DoubleType(), True),        # Tip in
dollars
        StructField("tolls_amount", DoubleType(), True),      # Bridge and
tunnel tolls in dollars

        # Additional fields that may be present
        StructField("total_amount", DoubleType(), True)       # Total
amount paid (calculated field)
    ])
```

Unix_timestamp was used to converted the drop-off time to a timestamp format to facilitate time-based analysis.

```
df = df.withColumn("dropoff_time", unix_timestamp(col("dropoff_datetime")))
```

## Mapping Coordinates to Grid Cells

To analyze trip patterns, atitude and longitude coordinates were mapped to grid cells based on a fixed reference point and cell size:

```
# Helper functions to convert lat/lon to cell coordinates
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

# Function to compute grid cell ID
def get_cell_id(lat, lon):
    base_lat, base_lon = 41.474937, -74.913585  # NYC reference point
    cell_size = 0.0045  # Approx. 500m in degrees

    cell_x = int((lon - base_lon) / cell_size) + 1
    cell_y = int((base_lat - lat) / cell_size) + 1
    return f"{cell_x}.{cell_y}"

# Register UDFs
```

```
spark.udf.register("get_cell_id", get_cell_id, StringType())
```

Then we use the previous udf called "get_cell_id." defined to add two new columns—"start_cell" and "end_cell"—which represent geospatial grid cell identifiers calculated from the pickup and dropoff coordinates.

```
routes_df = df.withColumn(
        "start_cell",
        F.expr("get_cell_id(pickup_latitude, pickup_longitude)")
    ).withColumn(
        "end_cell",
        F.expr("get_cell_id(dropoff_latitude, dropoff_longitude)")
    )
```

## Peak Traffic Period

Then, we compute taxi trips into 30-minute windows based on dropoff time, group them by origin and destination grid cells, count the number of rides between each cell pair in each time window, and rename the count column to "Number_of_Rides" for the next steps.

```
    windowed_routes = routes_df.withWatermark("dropoff_datetime", "30
minutes") \
        .groupBy(
            F.window("dropoff_datetime", "30 minutes"),
            "start_cell",
            "end_cell"
        ) \
        .count() \
        .withColumnRenamed("count", "Number_of_Rides")
```

then identified the window with the highest number of rides:

```
max_rides_window =
windowed_counts.orderBy(col("num_rides").desc()).limit(1).collect()[0][0]
```

Extracting the start and end times of this peak window, I filtered the dataset to include only trips within this period:

```
window_start, window_end = max_rides_window["start"], max_rides_window["end"]
peak_time_df = df.filter((col("dropoff_time") >= window_start) &
(col("dropoff_time") < window_end))
```

The result after occurrences are counted, cut is in 10 as seen here.

```
top_routes = filtered_routes.orderBy(col("num_rides").desc()).limit(10)
top_routes.show()
```

```
+----------+--------+---------+
|start_cell|end_cell|num_rides|
+----------+--------+---------+
|   206.165| 208.163|        3|
|   206.165| 209.162|        3|
|   207.162| 208.166|        3|
|   206.165| 205.162|        2|
|   204.168| 204.164|        2|
|   206.168| 202.171|        2|
|   204.168| 206.162|        2|
|   205.161| 207.164|        2|
|   206.168| 207.165|        2|
|   205.161| 207.157|        2|
+----------+--------+---------+
```

All the results were then stored in a delta table

```
query1_part1_stream = (part1_results.writeStream
    .outputMode("complete")
    .format("delta")
    .queryName("frequent_routes")
    .trigger(processingTime="3 second")
    .option("checkpointLocation", FREQ_CHECKPOINT_PATH)
    .start(FREQ_OUTPUT_PATH))
```

Later on it was easier to check the delta table for checking the distribution of rides inside each time window created.

```
max_rides_window = freq_batch_df.orderBy(col("window.end").desc(),
col("Number_Of_Rides").desc()).limit(1).collect()[0][0]
```

| | window_start | window_end | start_cell | end_cell | Number_of_Rides |
|---|---|---|---|---|---|
| **0** | 2014-01-01 | 2014-01-01 00:30:00 | 211.160 | 210.158 | 1 |
| **1** | 2014-01-01 | 2014-01-01 00:30:00 | 252.185 | 210.177 | 1 |
| **2** | 2014-01-01 | 2014-01-01 00:30:00 | 205.161 | 216.144 | 1 |
| **3** | 2014-01-01 | 2014-01-01 00:30:00 | 202.168 | 209.153 | 1 |
| **4** | 2014-01-01 | 2014-01-01 00:30:00 | 211.159 | 212.155 | 1 |
| **5** | 2014-01-01 | 2014-01-01 00:30:00 | 207.157 | 214.156 | 1 |
| **6** | 2014-01-01 | 2014-01-01 00:30:00 | 207.158 | 205.162 | 1 |
| **7** | 2014-01-01 | 2014-01-01 00:30:00 | 207.178 | 213.179 | 1 |
| **8** | 2014-01-01 | 2014-01-01 00:30:00 | 207.156 | 214.153 | 1 |
| **9** | 2014-01-01 | 2014-01-01 00:30:00 | 216.155 | 233.137 | 1 |

## Part 2:

For the part 2 micro-batches were used to compute the data.

This part is based on what was used in part 1. However we added new field called **ingestion_time** that will be lazy computed later on when the batch is being processed.

```
df = spark.readStream.format("delta").load(SOURCE_DELTA_TABLE)

# ingestion time in first read
```

```
df = df.withColumn("ingestion_time", F.current_timestamp())
```

We created the route identifiers for the start cell and the end cell.

```
# Create route identifiers in X.Y format
cell_routes_df = valid_routes_df.withColumn( "start_cell",
get_cell_id(F.col("start_cell_x"), F.col("start_cell_y")) ).withColumn(
"end_cell", get_cell_id(F.col("end_cell_x"), F.col("end_cell_y")) )
```

Then, we create a time-based tumbling window over the data stream, grouping taxi trips by 30-minute intervals and origin-destination cell pairs while calculating ride counts and tracking the latest event timestamps for latency analysis.

```
windowed_routes = cell_routes_df.withWatermark("dropoff_datetime", "30
minutes") \ .groupBy( F.window("dropoff_datetime", "30 minutes"),
"start_cell", "end_cell" ) \ .agg( F.count("*").alias("Number_of_Rides"),
F.max("pickup_datetime").alias("latest_pickup"),
F.max("dropoff_datetime").alias("latest_dropoff"),
F.max("ingestion_time").alias("latest_ingestion") )
```

This foreachBatch processor function analyzes each micro-batch of data to identify changes in the top 10 most frequent routes, comparing the current set against the previous one and outputting results when changes occur.

```
def detect_changes_and_output(batch_df, batch_id):
# Function implementation...
```

Since we want to use a reference timestamp to process batches against, we selected the 75th percentile of all available windows (magic date); any timestamp that belonged to the data could be used.

```
# Get the latest 30-minute window from the batch
if latest_window is None:
    window_ends = batch_df.select("window.end").distinct().collect()
    window_ends = [row["end"] for row in window_ends]

    if not window_ends:
```

```
        return

    # Sort the window ends
    window_ends.sort()

    # Calculate the 75th percentile index
    percentile_idx = int(len(window_ends) * 0.75)
    if percentile_idx >= len(window_ends):
        percentile_idx = len(window_ends) - 1

    # Get the window at the 75th percentile
    latest_window = window_ends[percentile_idx]
```

We then get the top routes again from the window we are in.

```
top_routes_df =
window_data.orderBy(F.col("Number_of_Rides").desc()).limit(10)
```

We select only the top 10 most frequent ones for change detection and reporting.

```
# Check if top routes have changed
routes_changed = False

if previous_top_routes is None:
    routes_changed = True
else:
    # Compare current routes with previous ones
    current_route_set = set((r[0], r[1]) for r in current_top_routes)
    previous_route_set = set((r[0], r[1]) for r in previous_top_routes)

    if current_route_set != previous_route_set:
        routes_changed = True
```

Then as required we measure the delay between ingestion and processing

```
# Calculate delay between ingestion time and current processing time
output_time = time.time()
delay = output_time - ingestion_time.timestamp()
```

Then, we output directly in the delta table that has the records with the updated records for the time window previously selected.

```
# Create output DataFrame
```

```
output_df = spark.createDataFrame([output_data], schema=output_schema)

# Write to Delta table
output_df.write.format("delta").mode("append").save(OUTPUT_DELTA_TABLE)
```

We apply the streaming query configuration using the "complete" output mode with foreachBatch processing to detect route popularity changes, while checkpoint management ensures exactly-once processing semantics and fault tolerance.

```
query = windowed_routes.writeStream \
    .outputMode("complete") \
    .foreachBatch(detect_changes_and_output) \
    .option("checkpointLocation", CHECKPOINT_LOCATION) \
    .start()
```

The results were as follows:

| me | dropoff_datetime | delay | start_cell_id_1 | end_cell_id_1 | start_cell_id_2 | end_cell_id_2 | start_cell_id_3 |
|---|---|---|---|---|---|---|---|
| -01 :00 | 2013-10-01 19:27:27 | 76.60893 | 158.152 | 157.155 | 160.156 | 159.158 | 158.152 |
| s | | | | | | | |

```
[Row(pickup_datetime=datetime.datetime(2013, 10, 1, 21, 41),
dropoff_datetime=datetime.datetime(2013, 10, 1, 22, 27, 27),
delay=76.60892987251282, start_cell_id_1='158.152', end_cell_id_1='157.155',
start_cell_id_2='160.156', end_cell_id_2='159.158', start_cell_id_3='158.152',
end_cell_id_3='161.156', start_cell_id_4='160.157', end_cell_id_4='156.159',
start_cell_id_5='162.155', end_cell_id_5='162.156', start_cell_id_6='152.169',
end_cell_id_6='157.156', start_cell_id_7='158.161', end_cell_id_7='160.160',
start_cell_id_8='154.162', end_cell_id_8='154.167', start_cell_id_9='159.158',
end_cell_id_9='162.155', start_cell_id_10='157.158', end_cell_id_10='155.158')]
```

# Query 2

## Part 1

The goal of this implementation is to identify **the 10 most profitable areas (250m x 250m grid cells)** for taxi drivers in real-time. This is achieved by:

1. Calculating the **median fare + tip** for trips that started in a cell and ended within the last 15 minutes.

2. Counting the **number of empty taxis** (taxis that have dropped off but not picked up anyone for the last 30 minutes).
3. Computing **profitability = (median fare + tip) / number of empty taxis**.
4. Reporting the top 10 profitable areas.

We configure a custom Spark session with:

- **Delta Lake integration** (extensions and catalog support),
- **Optimized memory settings** for HPC environment,
- **Hive support** for potential future cataloging.

*Delta is used for efficient streaming reads and writes, especially suited for continuously updating parquet-like datasets.*

We define a detailed schema for the cleaned NYC taxi dataset. We read the data using streaming from a Delta table, which allows us to simulate incoming trip data in real time.

```
raw_schema = create_raw_taxi_schema()
CLEAN_OUTPUT_PATH = "clean_taxi_data"
batch_df = spark.read.format("delta").load(CLEAN_OUTPUT_PATH)
df = spark.readStream.format("delta").load(CLEAN_OUTPUT_PATH)
```

We implement a function called get_250m_cell, which uses geographic approximation to convert GPS coordinates into grid cells for both pickup and drop-off points. To ensure accuracy, we filter out any invalid or out-of-bound cells. This mapping process is essential for effectively grouping taxis by area.

```
df_cells = (
    df
    # Add columns for start_cell_x, start_cell_y, end_cell_x, end_cell_y
    .withColumn("start_cell",
        udf_get_250m_cell(col("pickup_latitude"), col("pickup_longitude")))
    .withColumn("start_cell_x", expr("start_cell[0]"))
    .withColumn("start_cell_y", expr("start_cell[1]"))
    .withColumn("end_cell",
        udf_get_250m_cell(col("dropoff_latitude"), col("dropoff_longitude")))
    .withColumn("end_cell_x", expr("end_cell[0]"))
    .withColumn("end_cell_y", expr("end_cell[1]"))
    # Keep only rows with valid cells
    .filter(col("start_cell_x").isNotNull() & col("end_cell_x").isNotNull())
)
```

We use Spark's window() and percentile_approx() functions to calculate the median fare and tip over 15-minute intervals for each cell. The trigger_pickup and trigger_dropoff timestamps provide a reference for reporting time.

```
profit_agg = (
    df_cells
    .withColumn("fare_plus_tip", col("fare_amount") + col("tip_amount"))
    .withWatermark("dropoff_datetime", "20 minutes")
    .groupBy(
        window("dropoff_datetime", "15 minutes"),
        col("start_cell_x"),
        col("start_cell_y")
    )
    .agg(
        percentile_approx("fare_plus_tip", 0.5).alias("median_fare_tip"),
        spark_max("pickup_datetime").alias("trigger_pickup"),
        spark_max("dropoff_datetime").alias("trigger_dropoff")
    )
)
```

In our process for managing empty taxis, we utilize an in-memory Python dictionary named empty_state to monitor each taxi's last dropoff time and location. This allows us to update dropoff locations as needed efficiently. Additionally, if a taxi has a later pickup scheduled, we remove it from our state, as it is no longer considered empty. We define a taxi as empty if no pickup has occurred within 30 minutes following the dropoff. This state tracking mechanism effectively simulates live taxi availability, which would be quite challenging to achieve with a solely SQL-based approach.

```
def process_empty_taxis(batch_df, batch_id):
    global empty_state
    rows = batch_df.collect()  # gather micro-batch
    if not rows:
        return

    # Update ephemeral dictionary
    now_ts = datetime.datetime.utcnow()

    for r in rows:
        med = r["medallion"]
        pick_t = r["pickup_datetime"]
        drop_t = r["dropoff_datetime"]
        end_x = r["end_cell_x"]
        end_y = r["end_cell_y"]

        # update dropoff
        if drop_t:
            empty_state[med] = (drop_t, end_x, end_y)

        # if there's a subsequent pickup after the last dropoff, remove it
        old_val = empty_state.get(med, None)
        if old_val and pick_t:
            (old_drop_t, old_cellx, old_celly) = old_val
            if pick_t > old_drop_t:
                # not empty
                del empty_state[med]
```

We begin by joining the profit data with the information regarding empty taxis within a foreachBatch context. For each batch, we calculate the profitability by dividing the median fare by the number of empty taxis. Once we have these calculations, we sort the list based on profitability. Finally, we print out the top ten most profitable cells.

```
def process_profit(batch_df, batch_id):
    global empty_state, last_top10

    # This micro-batch has the windowed aggregator for median_fare_tip
    # We'll build a local aggregator for empties: (cell_x, cell_y) -> count
    now_ts = datetime.datetime.utcnow()

    # 4.1. Build an ephemeral aggregator for empties
    empty_map = {}
    for med, (drop_time, cx, cy) in empty_state.items():
        # if drop_time < 30 min
        if drop_time is not None:
            delta_sec = (now_ts - drop_time).total_seconds()
            if delta_sec <= 1800:
                key = (cx, cy)
                empty_map[key] = empty_map.get(key, 0) + 1

    # 4.2. Convert batch df to a local list
```

This shows the results of the **first batch (batch=0)** of streaming data processed.

```
=== Top 10 Profitable Areas (batch=0) ===
{'pickup_datetime': '2013-01-01 02:03:00', 'dropoff_datetime': '2013-01-01 02:08:00', 'profitable_cell_id_1': '309.334', 'empty_taxies_in_cell_id_1': 0, 'median_profit_in
_cell_id_1': 8.25, 'profitability_of_cell_1': None, 'profitable_cell_id_2': '327.342', 'empty_taxies_in_cell_id_2': 0, 'median_profit_in_cell_id_2': 5.0, 'profitability_o
f_cell_2': None, 'profitable_cell_id_3': '310.333', 'empty_taxies_in_cell_id_3': 0, 'median_profit_in_cell_id_3': 9.1, 'profitability_of_cell_3': None, 'profitable_cell_i
d_4': '317.301', 'empty_taxies_in_cell_id_4': 0, 'median_profit_in_cell_id_4': 9.5, 'profitability_of_cell_4': None, 'profitable_cell_id_5': '317.316', 'empty_taxies_in_c
ell_id_5': 0, 'median_profit_in_cell_id_5': 4.5, 'profitability_of_cell_5': None, 'profitable_cell_id_6': '307.320', 'empty_taxies_in_cell_id_6': 0, 'median_profit_in_cel
l_id_6': 9.5, 'profitability_of_cell_6': None, 'profitable_cell_id_7': '309.331', 'empty_taxies_in_cell_id_7': 0, 'median_profit_in_cell_id_7': 7.5, 'profitability_of_cel
l_7': None, 'profitable_cell_id_8': '314.324', 'empty_taxies_in_cell_id_8': 0, 'median_profit_in_cell_id_8': 10.0, 'profitability_of_cell_8': None, 'profitable_cell_id_
9': '309.330', 'empty_taxies_in_cell_id_9': 0, 'median_profit_in_cell_id_9': 12.0, 'profitability_of_cell_9': None, 'profitable_cell_id_10': '311.336', 'empty_taxies_in_c
ell_id_10': 0, 'median_profit_in_cell_id_10': 11.5, 'profitability_of_cell_10': None, 'delay': 1.0}
25/03/30 17:45:59 WARN ProcessingTimeExecutor: Current batch is falling behind. The trigger interval is 10000 milliseconds, but spent 139107 milliseconds
[Stage 38:=============================================================> (96 + 3) / 99]
```

These timestamps originate from the trip that triggered this window. They represent the latest pickup and drop-off within the 15-minute profit window. This information serves as a temporal anchor for the profitability snapshot. In later batches (when more data has been streamed in), the empty_state will be populated and updated, so profitability values will start appearing.