# Project 2 DESB GRAND CHALLENGE 2015

Juan Gonzalo Quiroz Cadavid[1], Priit Peterson[1], Shivam Maheshwari[1], and Venkata Narayana Bommanaboina[1]

[1]University of Tartu
juangonzalo@ut.ee, shivammahe21@gmail.com, bvnarayana515739@gmail.com, priit.petersonest@gmail.com

March 30, 2025

## Overall architecture 1

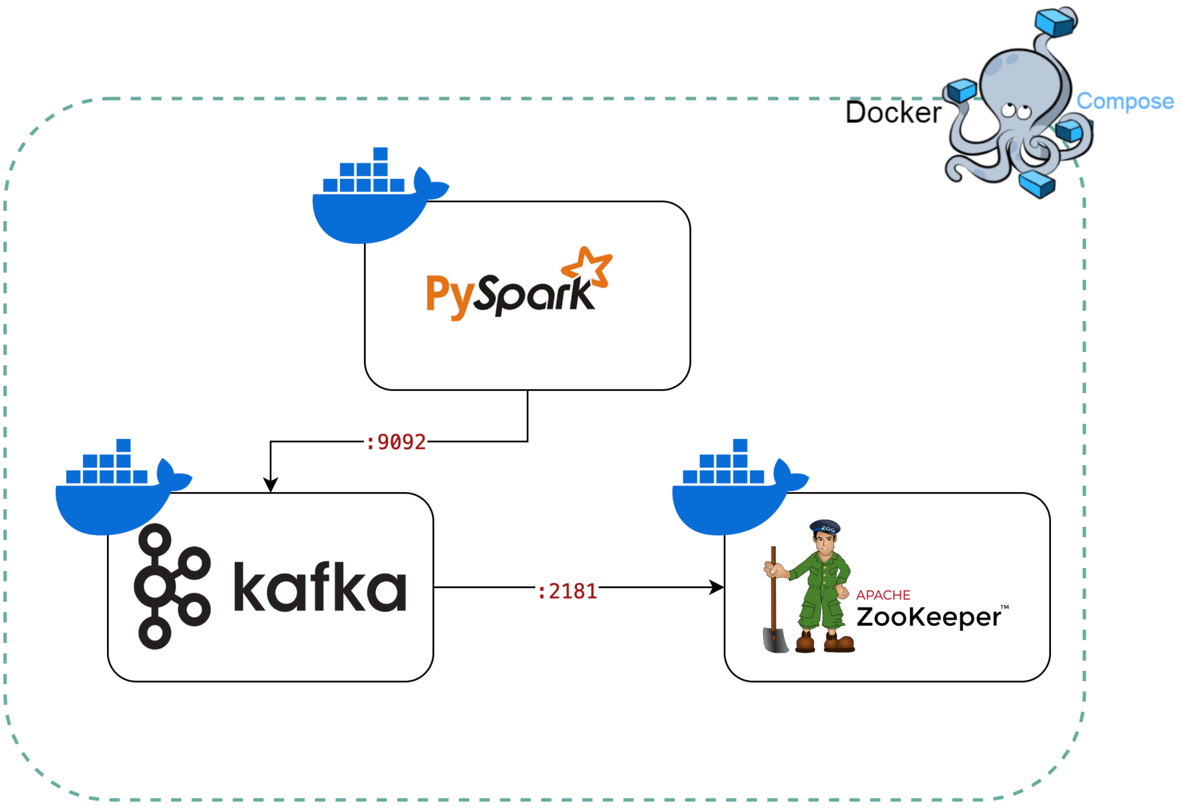

Figure 1: Overall architecture

For this project, the next system architecture were used over docker compose, Kafka, zooKeeper and PySpak as independent containers, communication goes through docker compose networking. Container to container communication was made possible by docker compose networking DNS.

## The master plan. 2

- **Data Ingestion & Cleansing**
  - Load a portion (e.g., 1 GB) of the NYC Taxi trip dataset.
  - Remove invalid rows (e.g., null or zero coordinates, unknown driver IDs) and cast columns to appropriate types.
- **Grid Cell Computation**
  - Map each latitude/longitude coordinate to a cell ID.
  - For Query 1, use a 500 m grid (cell IDs range from `1.1` to `300.300`).
  - For Query 2, use a 250 m grid (cell IDs range from `1.1` to `600.600`).
- **Query 1: Frequent Routes**
  - Compute the 10 most frequent routes in the last 30 minutes, where a route is identified by `(start_cell, end_cell)`.
  - Update results only when the top-10 changes, and output one row containing:
    - `pickup_datetime`
    - `dropoff_datetime`
    - `start_cell_id_n`
    - `end_cell_id_n`
    - `delay`
- **Query 2: Profitable Areas**
  - Calculate profitability for each area as:

$$\frac{\text{median fare + tip in last 15 minutes}}{\text{number of empty taxis in last 30 minutes}}$$

  - Report the 10 most profitable areas. If fewer than 10 exist, fill in `NULL` for missing cells. Output one row for each update that includes:
    - `pickup_datetime`
    - `dropoff_datetime`
    - `profitable_cell_id_n`
    - `empty_taxies_in_cell_id_n`
    - `median_profit_in_cell_id_n`
    - `profitability_of_cell_n`
    - `delay`
- **Streaming Setup & Output**
  - Use PySpark Structured Streaming with `foreachBatch` to process micro-batches.
  - Maintain global state to detect changes in top-10 routes or profitable areas.
  - Write final results to a table or console for each batch, including the delay metric (system time minus ingest time).

## 2.1 Setup

First, we defined the spark session. We decided to configure spark to work with Delta lake by:

- Adding its JARS package into Spark runtime *[Line 6-7]*.
- Registering Delta SQL commands into SQL extensions *[Line 10-11]*.
- Making it the default Catalog for all tables. *[Line 13-14]*.

Listing 1: Spark session

```
1  builder = SparkSession.builder
2      .appName(
3          "project2_debs_grand_challenge"
4      )
5      .config(
6          "spark.jars.packages",
7          "io.delta:delta-core_2.12:2.4.0"
8      )
9      .config(
10         "spark.sql.extensions",
11         "io.delta.sql.DeltaSparkSessionExtension")
12     .config(
13         "spark.sql.catalog.spark_catalog",
14         "org.apache.spark.sql.delta.catalog.DeltaCatalog"
15     )
16 spark = configure_spark_with_delta_pip(builder).getOrCreate()
```

## 2.2 Reading and cleaning

We decided to read the CSV without headers, in order to inject the headers as we want to name then on **line 9**, Then, we took 10% of the data, (Around 1Gb) **line 11**

Listing 2: Reading

```
1  columns = [
2      "medallion", "hack_license", "pickup_datetime", "dropoff_datetime",
3      ....
4  ]
5
6  df = spark.read.option("header", "false").csv("data/sorted_data.csv")
7  df = df.toDF(*columns)
8
9  df_sample = df.sample(withReplacement=False, fraction=0.1)
```

After reading, we clean the data by the next operations, which leave us with **13.4 Million elements to analyze**, a before cleaning it was **17.3 Million data**, which means almost **4 Million data was corrupted**

- Changing format of pickup and dropoff date time to timestamp Colum type. 3
- Removing null or 0.0 columns and unknown licenses or drivers. 4
- Convert relevant columns to appropriate data types for numerical computations. 5

Listing 3: Timestamp

```
df_sample = df_sample
    .withColumn(
        "pickup_datetime",
        to_timestamp(
            col("pickup_datetime"),
            "yyyy-MM-dd-HH:mm:ss")
    )
    .withColumn(
        "dropoff_datetime",
        to_timestamp(
            col("dropoff_datetime"),
            "yyyy-MM-dd-HH:mm:ss")
    )
```

Listing 4: Filtering empty fields

```

df_clean = df_sample.filter(
    (col("medallion").isNotNull())
        & (col("medallion") != "0")
        & (col("medallion") != "UNKNOWN") &
    (col("hack_license").isNotNull())
        & (col("hack_license") != "0")
        & (col("hack_license") != "UNKNOWN") &
    (col("pickup_datetime").isNotNull()) &
    (col("dropoff_datetime").isNotNull()) &
    (col("trip_time_in_secs").isNotNull())
        & (col("trip_time_in_secs") != 0) &
    (col("trip_distance").isNotNull())
        & (col("trip_distance") != 0) &
    (col("pickup_longitude").isNotNull())
        & (col("pickup_longitude") != 0.0) &
    (col("pickup_latitude").isNotNull())
        & (col("pickup_latitude") != 0.0) &
    (col("dropoff_longitude").isNotNull())
        & (col("dropoff_longitude") != 0.0) &
    (col("dropoff_latitude").isNotNull())
        & (col("dropoff_latitude") != 0.0) &
    (col("trip_distance").cast("float") > 0) &
    (col("fare_amount").cast("float") > 0)
)
```

Listing 5: Casting Columns

```
df_clean = df_clean
    .withColumn("trip_time_in_secs", col("trip_time_in_secs").cast("int"))
    .withColumn("trip_distance", col("trip_distance").cast("float"))
    .withColumn("fare_amount", col("fare_amount").cast("float"))
    .withColumn("surcharge", col("surcharge").cast("float"))
    .withColumn("mta_tax", col("mta_tax").cast("float"))
    .withColumn("tip_amount", col("tip_amount").cast("float"))
    .withColumn("tolls_amount", col("tolls_amount").cast("float"))
```

## 2.3 Query 1

### 2.3.1 Defining cells and computing

In order to get the cells for each Lat/Lon point, we:

1. Moved the center of the firs cell to the corner of the cell (by 0.250 Km to the North, 0.250Km to the West). See Code 6.
2. Shared the new P0.0 across all nodes using Spark context. See Code 6.
3. For upcoming lat/lon attributes, we will only need to calculate the distance to the Point 0 and then divide it by 0.5 Km on both lat and long distances to get the cell location. See Code 7.
4. Defined a new UDF function for calculating the cells locations. See Code 8.
5. Applied the UDF as a new column for all elements in the DF. See Code 9.
6. Filtered by -1_-1, which means the (Lat, Lon) pair is located outside the boundaries. **After cleaning, 54K rows were removed**. See Code 10.
7. Grouped them by tumbling time window of 30 mins, start and end cell and Ordered then by count. See Code 11.
8. **Ta-da**! The resulting top 10 is presented on table 1, Fig 2 shows their distribution as a pie chart.

Listing 6: Moving

```
pRoot = [41.474937, −74.913585]
p0 = move_point(pRoot[0], pRoot[1], 0.250 , 0.250)
broadcast_center = spark.sparkContext.broadcast(p0)
```

Listing 7: getCells

```
def getCells(center, lat, lon, upTo=300):
    distLat, distLon = latlon_distance(center[0], center[1], lat, lon)
    if distLat == -1 or distLon == -1:
        return -1, -1

    cellLat, cellLon = int(distLat//0.5), int(distLon//0.5)

    if distLat > upTo or distLon > upTo:
        return -1, -1

    return cellLat, cellLon
```

Listing 8: UDF

```
def getCellString(lat, lon):
    center = broadcast_center.value
    cellLat, cellLon = getCells(center, lat, lon)
    return f"{cellLat}_{cellLon}"

getCellUDF = F.udf(getCellString, StringType())
```

Listing 9: Adding cells

```
df_routes = df_clean.withColumn(
    "start_cell",
    getCellUDF(col("pickup_latitude"),col("pickup_longitude"))

).withColumn(
    "end_cell",
    getCellUDF(col("dropoff_latitude"),col("dropoff_longitude"))
)
```

Listing 10: filterign

```
df_routes_cleaned = df_routes.filter(
    (col("end_cell") != "-1_-1") &
    (col("start_cell") != "-1_-1")
)
```

| | Time_window | start_cell | end_cell | Number_of_Rides |
|---|---|---|---|---|
| 0 | (('2013-01-15 09:00:00'), ('2013-01-15 09:30:00')) | 159_154 | 159_151 | 15 |
| 1 | (('2013-01-14 08:00:00'), ('2013-01-14 08:30:00')) | 159_154 | 160_151 | 13 |
| 2 | (('2013-01-15 06:30:00'), ('2013-01-15 07:00:00')) | 160_153 | 159_154 | 13 |
| 3 | (('2013-06-20 07:00:00'), ('2013-06-20 07:30:00')) | 160_153 | 160_155 | 12 |
| 4 | (('2013-03-07 06:30:00'), ('2013-03-07 07:00:00')) | 160_153 | 159_154 | 11 |
| 5 | (('2013-04-15 07:00:00'), ('2013-04-15 07:30:00')) | 160_153 | 160_155 | 11 |
| 6 | (('2013-05-07 06:30:00'), ('2013-05-07 07:00:00')) | 160_153 | 159_155 | 11 |
| 7 | (('2013-11-14 06:30:00'), ('2013-11-14 07:00:00')) | 160_153 | 160_155 | 11 |
| 8 | (('2013-01-14 08:00:00'), ('2013-01-14 08:30:00')) | 159_154 | 159_151 | 11 |
| 9 | (('2013-03-13 08:30:00'), ('2013-03-13 09:00:00')) | 161_155 | 164_153 | 11 |

Table 1: Top 10 rides

Listing 11: Group by Time and cells

```
df_frequent_routes_by_time_window = (
    df_routes_cleaned
        .groupBy(
            F.window("pickup_datetime", "30 minutes"),
            "start_cell",
            "end_cell"
        )
        .count()
        .withColumnRenamed("count", "Number_of_Rides")
        .withColumnRenamed("window", "Time_window")
)
top10_routes_by_time_window = df_frequent_routes_by_time_window
    .orderBy(
        col("Number_of_Rides").desc()
    )
    .limit(10)
```

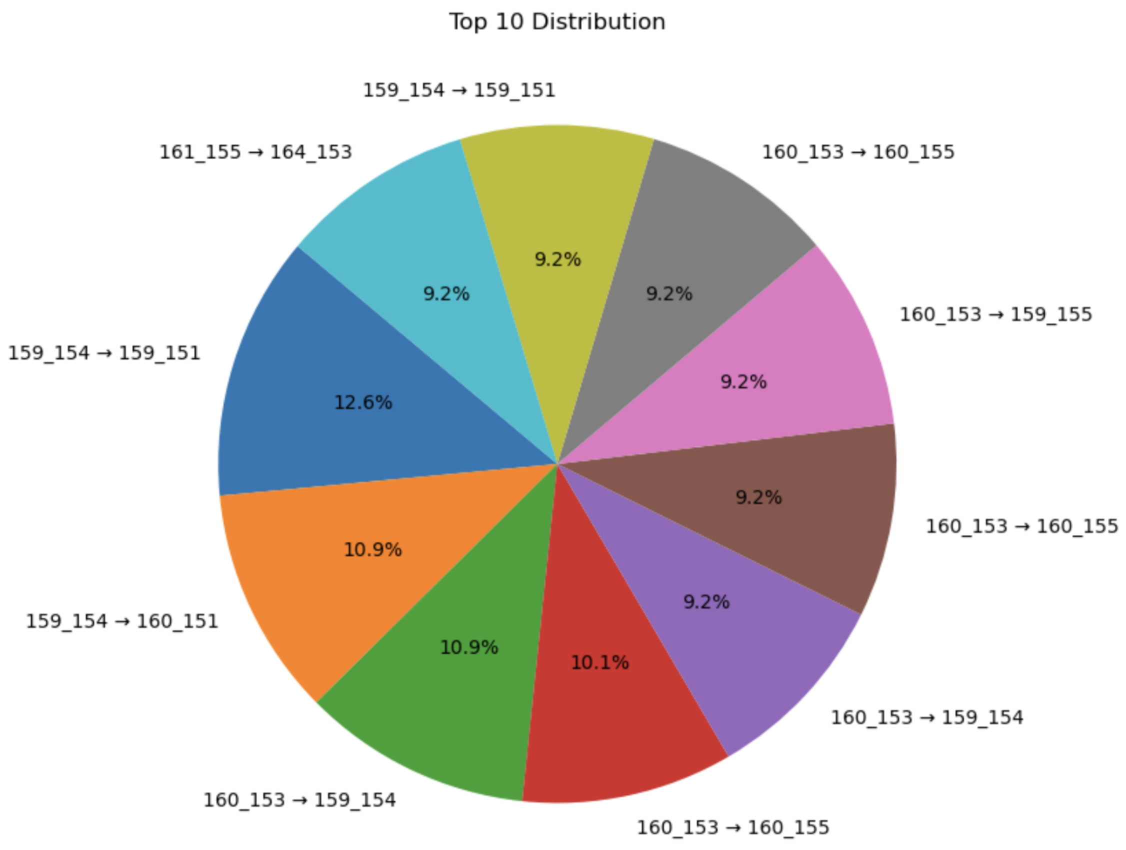

Figure 2: Top 10 rides distribution

### 2.4 Part 2

To address query 2, we:

- Created a Stream dataframe by writing into disk and loading it as a Stream. See code 12.
- Using the stream, we run a writeStream process on batches. See code 13.
- On every batch, we filter the data for only the last 30 mins, base on their drop off time. See code 14.
- aggregated the output into a single table. See code 15.
- Tables 2, and 3 shows the final result.

Listing 12: Transforming into stream

```
df_routes_cleaned
    .write
    .format("delta")
    .mode("overwrite")
    .save("./output/delta/taxi_data")

df_stream = spark.readStream
    .format("delta")
    .load("./output/delta/taxi_data")
```

Listing 13: Stream

```

df_stream = df_stream.withColumn(
    "pickup_datetime",
    to_timestamp(
        col("pickup_datetime"),
        "yyyy-MM-dd HH:mm:ss"
    )
)
df_stream = df_stream.withColumn(
    "dropoff_datetime",
    to_timestamp(
        col("dropoff_datetime"),
        "yyyy-MM-dd HH:mm:ss"
    )
)
df_stream = df_stream.withColumn("ingest_time", current_timestamp())

query = (
    df_stream.writeStream
    .trigger(once=True)
    .foreachBatch(process_batch)
    .outputMode("append")
    .start()
)
query.awaitTermination()
```

Listing 14: Bacth operation

```python

def process_batch(batch_df, batch_id):
    # Skip empty batches
    if batch_df.rdd.isEmpty():
        return

    max_dropoff = batch_df.agg({"dropoff_datetime": "max"}).collect()[0][0]
    if max_dropoff is None:
        return
    ref_time = max_dropoff - timedelta(minutes=30)

    # Compute grid cell IDs for pickup and dropoff using the 500m grid
    ...
    Calculating cells ids
    ...

    # Filter for trips with dropoff_datetime >= ref_time (last 30 minutes)
    df_last30 = batch_df.filter(col("dropoff_datetime") >= F.lit(ref_time))
    print(f"Window filter: dropoff_datetime >= {ref_time}")
    print("df_last30 count =", df_last30.count())
    df_last30.show(5)

    # Aggregate routes and get top 10 most frequent
    df_frequent_routes = df_last30.groupBy("start_cell", "end_cell") \
        .count() \
        .withColumnRenamed("count", "Number_of_Rides")
    top10_routes = df_frequent_routes
        .orderBy(
            col("Number_of_Rides").desc()
        )
        .limit(10)
    top10_list = top10_routes.collect()
    ...
    ...
    for i in range(10):
        if i < len(top10_list):
            route = top10_list[i]
            output_row[f"start_cell_id_{i+1}"] = route["start_cell"]
            output_row[f"end_cell_id_{i+1}"] = route["end_cell"]
        else:
            output_row[f"start_cell_id_{i+1}"] = None
            output_row[f"end_cell_id_{i+1}"] = None
```

Listing 15: Output

```
# This is your foreachBatch function to process each micro-batch
def process_batch(batch_df, batch_id):
    ...
    ...
    ...
    # Define the output schema explicitly
    output_schema = StructType([
        StructField("pickup_datetime", TimestampType(), True),
        StructField("dropoff_datetime", TimestampType(), True),
        StructField("start_cell_id_1", StringType(), True),
        ...
        ...
        ...
    ])
    result_df = spark.createDataFrame([output_row], schema=output_schema)
    result_df.write.mode("append").saveAsTable("frequent_routes")
```

| **Pickup Datetime** | **Dropoff Datetime** | **Delay (s)** |
|---|---|---:|
| 2013-12-31 23:54:08 | 2014-01-01 00:36:35 | 58.90 |

Table 2: Trip Timing Information

| **Start Cell** | **End Cell** | **Start Cell** | **End Cell** |
|---:|---:|---:|---:|
| 156.160 | 152.167 | 154.159 | 172.160 |
| 158.151 | 166.149 | 153.163 | 171.135 |
| 161.145 | 156.162 | 157.159 | 158.153 |
| 176.157 | 157.180 | 153.165 | 155.158 |
| 153.158 | 154.162 | 156.161 | 154.159 |

Table 3: Top 10 Frequent Routes

# Query 2: Profitable Areas 3

In this query, we compute which cells (or areas) offer the highest current profit opportunity. Profitability is defined as

$$\text{Profitability} \;=\; \frac{\text{median(fare + tip in last 15 min)}}{\text{number of empty taxis in last 30 min}}.$$

A taxi is considered *empty* in a cell if its last dropoff happened there in the preceding 30 minutes and there has been no subsequent pickup. The system reports the top ten areas at each update, placing them in columns such as `profitable_cell_id_1` through `profitable_cell_id_10`. If fewer than ten cells meet the criteria, the remaining columns are `NULL`.

## 3.1 Part 1: 500 m Grid

Listing 16 shows how each micro-batch of streaming data is processed to find the most profitable 500 m cells. The code begins by determining the latest `dropoff_datetime` in the batch (`max_dropoff`) and setting up two time windows: one of 15 minutes to calculate the median profit, and another of 30 minutes to count how many taxis remain empty in each cell. In the snippet, `profit_df` uses the `pickup_longitude` and `pickup_latitude` columns to assign each trip to a pickup cell, which is then grouped to find the median of `fare_amount` + `tip_amount`. Meanwhile, a second aggregation locates each taxi's most recent dropoff cell (within 30 minutes) in order to calculate the number of currently empty taxis. After a join merges these values, the system computes `profitability` = `median_profit`/`empty_taxis`, sorts in descending order, then extracts the top ten cells as a single output row. A `delay` is appended to measure the processing latency between reading the triggering event and writing the table row.

Listing 16: Query 2, Part 1: 500 m Grid

```python
grid_origin_lat = 41.474937
grid_origin_lon = -74.913585
delta_lat = 0.0045    # about 500m in latitude
delta_lon = 0.0060    # about 500m in longitude

def process_batch_query2_part1(batch_df, batch_id):
    if batch_df.rdd.isEmpty():
        print(f"Batch {batch_id}: empty, skipping.")
        return

    max_dropoff = batch_df.agg({"dropoff_datetime": "max"}).collect()[0][0]
    if not max_dropoff:
        return

    # Past 15 minutes for profit, past 30 for empty taxis
    ref_time_profit = max_dropoff - timedelta(minutes=15)
    ref_time_empty  = max_dropoff - timedelta(minutes=30)

    # Build a DF for profit computations
    profit_df = (
        batch_df
        .filter(F.col("dropoff_datetime") >= ref_time_profit)
        .withColumn("profit", F.col("fare_amount") + F.col("tip_amount"))
        .withColumn(
            "pickup_cell_east",
            F.floor((F.col("pickup_longitude") - grid_origin_lon) / delta_lon) + 1
        )
        .withColumn(
            "pickup_cell_south",
            F.floor((grid_origin_lat - F.col("pickup_latitude")) / delta_lat) + 1
        )
        .withColumn(
            "pickup_cell",
            F.concat(
                F.col("pickup_cell_east").cast("int"),
                F.lit("."),
                F.col("pickup_cell_south").cast("int")
            )
        )
    )
    profit_agg = profit_df.groupBy("pickup_cell").agg(
        F.expr("approx_percentile(profit, 0.5) as median_profit")
    )

    # For each taxi, identify the cell of its most recent dropoff
    w = Window.partitionBy("medallion").orderBy(F.col("dropoff_datetime").desc())
    last_dropoff_df = (
        batch_df
        .withColumn("rn", F.row_number().over(w))
        .filter(F.col("rn") == 1)
        .filter(F.col("dropoff_datetime") >= ref_time_empty)
        .withColumn(
            "dropoff_cell_east",
            F.floor((F.col("dropoff_longitude") - grid_origin_lon) / delta_lon) +
        )
        .withColumn(
            "dropoff_cell_south",
            F.floor((grid_origin_lat - F.col("dropoff_latitude")) / delta_lat) + 1
        )
        .withColumn(
            "dropoff_cell",
            F.concat(
                F.col("dropoff_cell_east").cast("int"),
                F.lit("."),
                F.col("dropoff_cell_south").cast("int")
            )
        )
    )
    empty_agg = last_dropoff_df.groupBy("dropoff_cell").agg(
        F.countDistinct("medallion").alias("empty_taxis")
    )

    area_df = (
        profit_agg
        .join(empty_agg, profit_agg.pickup_cell == empty_agg.dropoff_cell, "inner"
        .select(profit_agg.pickup_cell.alias("cell_id"), "median_profit", "empty_
        .filter(F.col("empty_taxis") > 0)
        .withColumn("profitability", F.col("median_profit") / F.col("empty_taxis")
        .orderBy(F.col("profitability").desc())
    )
    top10_rows = area_df.limit(10).collect()

    # Determine the event that triggered this output
    trigger_row = batch_df.orderBy(F.col("dropoff_datetime").desc()).limit(1).coll
    pickup_ts   = trigger_row["pickup_datetime"]
    dropoff_ts  = trigger_row["dropoff_datetime"]
    ingest_time = trigger_row.get("ingest_time", dropoff_ts)
    delay       = (datetime.now() - ingest_time).total_seconds()

    # Build a single output row with up to 10 columns for the profitable cells
    result_data = {
        "pickup_datetime": pickup_ts,
        "dropoff_datetime": dropoff_ts,
        "delay": delay
    }
    for i in range(10):
        if i < len(top10_rows):
            info = top10_rows[i]
            result_data[f"profitable_cell_id_{i+1}"]        = info["cell_id"]
            result_data[f"empty_taxies_in_cell_id_{i+1}"]   = info["empty_taxis"]
            result_data[f"median_profit_in_cell_id_{i+1}"]  = info["median_profit"]
            result_data[f"profitability_of_cell_{i+1}"]     = info["profitability"]
        else:
            result_data[f"profitable_cell_id_{i+1}"]        = None
            result_data[f"empty_taxies_in_cell_id_{i+1}"]   = None
            result_data[f"median_profit_in_cell_id_{i+1}"]  = None
            result_data[f"profitability_of_cell_{i+1}"]     = None

    # (Remaining code: building a schema, writing result_data to a Spark table, e

df_stream = df_stream.withColumn("ingest_time", F.current_timestamp())

query2_part1 = (
    df_stream
    .writeStream
    .trigger(once=True)
    .foreachBatch(process_batch_query2_part1)
    .outputMode("append")
    .start()
)

query2_part1.awaitTermination()
```

Whenever a batch includes rides for multiple cells, several columns can be filled. If only one cell meets the criteria, the rest appear as `NULL`. This behavior is normal because the specification requires that unoccupied columns be blank.

## 3.2 Part 2: 250 m Grid

Part 2 uses the same reasoning, but cell coordinates are calculated with half the deltas so that each cell measures about 250 m. Listing 17 shows the difference. The same time windows apply: 15 minutes for median profit, 30 minutes for empty taxi counts, and top-ten output columns that fill with `NULL` if fewer than ten exist. Table 4 illustrates an example row, where `profitable_cell_id_1` indicates the highest profitability and `profitable_cell_id_2` the second highest, etc.

Listing 17: Query 2, Part 2: 250 m Grid

```python
new_delta_lat = 0.0045 / 2  # 0.00225
new_delta_lon = 0.0060 / 2  # 0.0030

def process_batch_query2_part2(batch_df, batch_id):
    if batch_df.rdd.isEmpty():
        print(f"Batch {batch_id}: empty, skipping.")
        return

    max_dropoff = batch_df.agg({"dropoff_datetime": "max"}).collect()[0][0]
    if not max_dropoff:
        return

    ref_time_profit = max_dropoff - timedelta(minutes=15)
    ref_time_empty  = max_dropoff - timedelta(minutes=30)

    # The rest is analogous to Part 1, but with new_delta_lat/new_delta_lon
    ...
```

| Column | Example Value |
|---|---|
| `pickup_datetime` | 2013-12-31 23:54:08 |
| `dropoff_datetime` | 2014-01-01 00:36:35 |
| `profitable_cell_id_1` | 317.319 |
| `empty_taxies_in_cell_id_1` | 1 |
| `median_profit_in_cell_id_1` | 16.0 |
| `profitability_of_cell_1` | 16.0 |
| `profitable_cell_id_2` | `NULL` |
| ... | ... |
| `delay` | 42.4519 |

Table 4: Sample row from the Query 2 (Part 2) output using a 250 m grid.

When only a single cell qualifies, the remaining nine columns are `NULL`. This is the intended behavior whenever fewer than ten distinct profitable cells are found in the batch and we only used 10% of the whole dataset.