UNIVERSITY OF SCIENCE – VNUHCM

FACULTY OF INFORMATION TECHNOLOGY

LAB 3 REPORT

SPARK STREAMING

Introduction to Big Data

Class: 21KHMT1

Group ID: KHMT1-02

Table of contents

1. General information

1.1 Group information

Student's name	Student's ID	Assigned tasks	Contribution
Lê Anh Thư	21127175	Task 1, 2	25%
Nguyễn Nhật Truyền	21127191	Task 3	25%
Nguyễn Minh Đạt	21127592	Task 4	25%
Huỳnh Đức Thiện	21127693	Task 4	25%

1.2 Task completion

Task	Description	Assessment	Completion
1	Discover a method to simulate a stream by utilizing data sourced from files.	Done	100%
2	Create an EventCount query that aggregates the number of trips by dropoff datetime for each hour.	Done	100%
	The output of each aggregation window is stored in a directory named output-xxxxx where xxxxx is the timestamp.	Done	
3	Create a RegionEventCount query that counts the number of taxi trips each hour that drops off at either the Goldman	Done	100%

	Sachs headquarters or the Citigroup headquarters.		
	Write to the output-xxxx a key, value tuple (headquarters, count) with headquarters in {'goldman', 'citigroup'}.	Done	
4	Build a simple "trend detector" to find out when there are lots of arrivals at either Goldman Sachs or Citigroup headquarters. The trend detector should "go off" when there are at least twice as many arrivals in the current interval as there were in the past interval. To reduce "spurious" detections, we want to make sure the detector only "trips" if there are ten or more arrivals in the current interval.	Done	70%
	The detector should output its results to stdout in the form of the following: “The number of arrivals to Goldman Sachs has doubled from X to Y at Z!” or “The number of arrivals to Citigroup has doubled from X to Y at Z!”.	Done	
	Also, the program should output the status for each batch to the directory specified by the output argument. Each	Haven’t done	

	status is stored in a separate file with the name of the format part-${timestamp}.		

2. Implementation

2.1 Setup

- Data
    - The data is contained in the **taxi-data.zip** file. After extracting the contents, there is a folder with 1,440 CSV files.
    - Each of these CSV files represents the taxi trip information for a single minute on December 1, 2015.
    - So collectively, the 1,440 files cover the entire day's worth of taxi trips, with one file for each minute of the day.
- Runtime
    - We use Google Colab for development.
    - The entire data folder is stored in Google Drive.
    - We then mount Google Drive to Google Colab to access the data.

2.2 Task 1

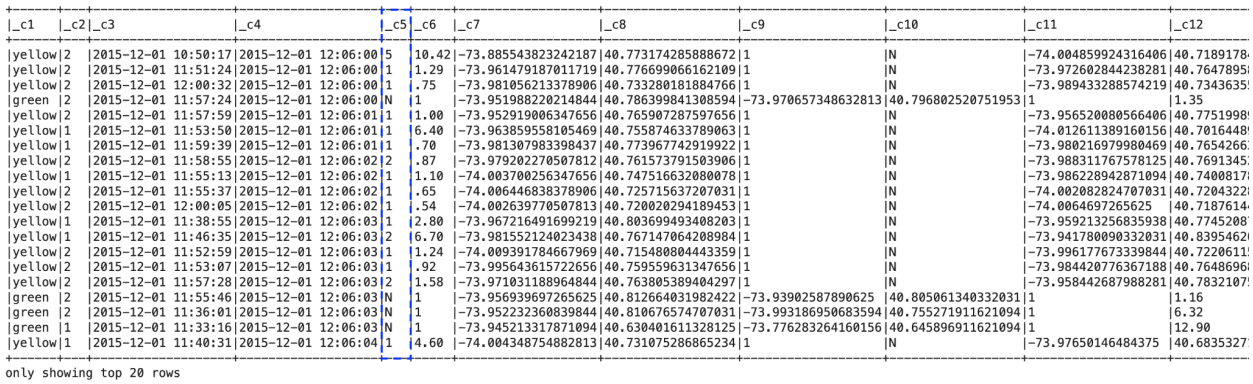
- Requirement:
    - Discover a method to simulate a stream by utilizing data sourced from files.
- Challenge:
    - There are 2 types of taxi records, Yellow and Green, each with a different schema.
    - However, both Yellow and Green records are stored within the same CSV files.

- Solution:
    - We first find the largest number of columns between the two schemas. Since the Yellow schema has 20 columns, and the Green schema has 22 columns, the largest number of columns is 22.
    - We define a **temporary** schema with 22 columns, namely _c1, _c2, …, _c22. Our goal is to use this temporary schema as an unified schema to read the data of both taxi types.
        - We do not set meaningful names for the columns because of the difference between Yellow and Green records. For example, the 5-th column in the Yellow schema is “passenger_count”, while the 5-th column in the Green schema is “Store_and_fwd_flag”.
        - We also have to define all columns as **StringType** for the same reason. For instance, the 5-th column of the Yellow schema is “passenger_count” of type Integer. However, the 5-th column of the Green schema is “Store_and_fwd_flag” of type String.
        - Note that this temporary schema is used solely to read the streaming data. Our goal at this stage is to read the data successfully without losing any columns or records. For further data processing later on, we will need to select specific columns and cast the data type to the correct format.
    - We can then store all the data of both Yellow and Green taxi trips using the temporary schema. Since Yellow taxi has only 20 columns, the remaining 2 columns are filled with **NULL** values.
    - An implementation detail related to Structured Streaming is to specify the value for **maxFilesPerTrigger**. We have tried different values such as 1, 2, 4, … and eventually decided that 8 is an appropriate choice. Our insight is that:

- The larger, the faster. Therefore, it can be tempting to set a very large value for this parameter. However, larger values such as 16, 32, can lead to “OutOfMemoryError” when too many files are processed at once.
- Therefore, we propose 8 as a safe choice that satisfies both fast processing and memory constraints.

```
temp_schema = StructType(
    [
        StructField("_c1", StringType(), True),
        StructField("_c2", StringType(), True),
        ...
        StructField("_c22", StringType(), True),
    ]
)
# using `readStream` instead of `read`
streamingInputDF = (
    spark
        .readStream
        .schema(temp_schema)                    # Set the schema
        .option("maxFilesPerTrigger", 8)
        .csv(inputPath)
)
```

Code: Define schema and read the data stream



```
+------+---+-------------------+-------------------+---+-----+------------------+------------------+---+-----------------+------------------+-----------
|_c1   |_c2|_c3                |_c4                |_c5|_c6  |_c7               |_c8               |_c9|_c10              |_c11              |_c12
+------+---+-------------------+-------------------+---+-----+------------------+------------------+---+-----------------+------------------+-----------
|yellow|2  |2015-12-01 10:50:17|2015-12-01 12:06:00|5  |10.42|-73.885543823242187|40.773174285888672|1  |N                |-74.0048599243164 |40.7189178
|yellow|2  |2015-12-01 11:51:24|2015-12-01 12:06:00|1  |1.29 |-73.961479187011719|40.776699066162109|1  |N                |-73.972602844238281|40.7647895
|yellow|2  |2015-12-01 12:00:32|2015-12-01 12:06:00|1  |.75  |-73.981056213378906|40.733280181884766|1  |N                |-73.989433288574219|40.7343635
|green |2  |2015-12-01 11:57:24|2015-12-01 12:06:00|N  |1    |-73.951988220214844|40.786399841308594|-73.970657348632813|40.796802520751953|1                 |1.35
|yellow|2  |2015-12-01 11:57:59|2015-12-01 12:06:01|1  |1.00 |-73.952919006347656|40.765907287597656|1  |N                |-73.956520080566406|40.7751998
|yellow|1  |2015-12-01 11:53:50|2015-12-01 12:06:01|1  |6.40 |-73.963859558105469|40.755874633789063|1  |N                |-74.012611389160156|40.7016448
|yellow|1  |2015-12-01 11:59:39|2015-12-01 12:06:01|1  |.70  |-73.981307983398437|40.773967742919922|1  |N                |-73.980216979980469|40.7654266
|yellow|2  |2015-12-01 11:58:55|2015-12-01 12:06:02|2  |.87  |-73.979202270507812|40.761573791503906|1  |N                |-73.988311767578125|40.7691345
|yellow|1  |2015-12-01 11:55:13|2015-12-01 12:06:02|1  |1.10 |-74.003700256347656|40.747516632080078|1  |N                |-73.986228942871094|40.7400817
|yellow|2  |2015-12-01 11:55:37|2015-12-01 12:06:02|1  |.65  |-74.006446838378906|40.725715637207031|1  |N                |-74.002082824707031|40.7204322
|yellow|2  |2015-12-01 12:00:05|2015-12-01 12:06:02|1  |.54  |-74.002639770507813|40.720020294189453|1  |N                |-74.0064697265625 |40.7187614
|yellow|1  |2015-12-01 11:38:55|2015-12-01 12:06:03|1  |2.80 |-73.967216491699219|40.803699493408203|1  |N                |-73.959213256835938|40.7745208
|yellow|1  |2015-12-01 11:46:35|2015-12-01 12:06:03|2  |6.70 |-73.981552124023438|40.767147064208984|1  |N                |-73.941780090332031|40.8395462
|yellow|2  |2015-12-01 11:52:59|2015-12-01 12:06:03|1  |1.24 |-74.009391784667969|40.715480804443359|1  |N                |-73.996177673339844|40.7220615
|yellow|2  |2015-12-01 11:53:07|2015-12-01 12:06:03|1  |.92  |-73.995643615722656|40.759559631347656|1  |N                |-73.984420776367188|40.7648696
|yellow|2  |2015-12-01 11:57:28|2015-12-01 12:06:03|2  |1.58 |-73.971031188964844|40.763805389404297|1  |N                |-73.958442687988281|40.7832107
|green |2  |2015-12-01 11:55:46|2015-12-01 12:06:03|N  |1    |-73.956939697265625|40.812664031982422|-73.93902587890625 |40.805061340332031|1                 |1.16
|green |2  |2015-12-01 11:36:01|2015-12-01 12:06:03|N  |1    |-73.952232360839844|40.810676574707031|-73.993186950683594|40.755271911621094|1                 |6.32
|green |1  |2015-12-01 11:33:16|2015-12-01 12:06:03|N  |1    |-73.945213317871094|40.630401611328125|-73.776283264160156|40.645896911621094|1                 |12.90
|yellow|1  |2015-12-01 11:40:31|2015-12-01 12:06:04|1  |4.60 |-74.004348754882813|40.731075268865234|1  |N                |-73.97650146484375|40.6835327
+------+---+-------------------+-------------------+---+-----+------------------+------------------+---+-----------------+------------------+-----------
only showing top 20 rows
```

Figure 1: A fraction of the result table. The dotted area is the 5-th column in which there is a type difference between the Yellow and Green records

2.3 Task 2

- Requirement:
  - Create an EventCount(.py/.scala/.java) query that aggregates the number of trips by dropoff datetime for each hour.
  - The output of each aggregation window is stored in a directory named output-xxxxx where xxxxx is the timestamp.
- Solution:
  - First, we select the dropoff datetime column from the dataframe obtained in Task 1.
    - Conveniently, both Yellow and Green schemas have dropoff datetime information at the 4-th column.
    - As a result, we simply select the **_c4** column without having to distinguish between Yellow and Green taxis.
  - Because we have read all columns as StringType in Task 1, we now need to cast the dropoff datetime datatype to Timestamp.

```
# cast the Timestamp type
dropoff_datetime_DF = (
   streamingInputDF
        .select(streamingInputDF['_c4']
        .cast("timestamp")
        .alias("dropoff_datetime"))
)
```

Code: Select column and cast type

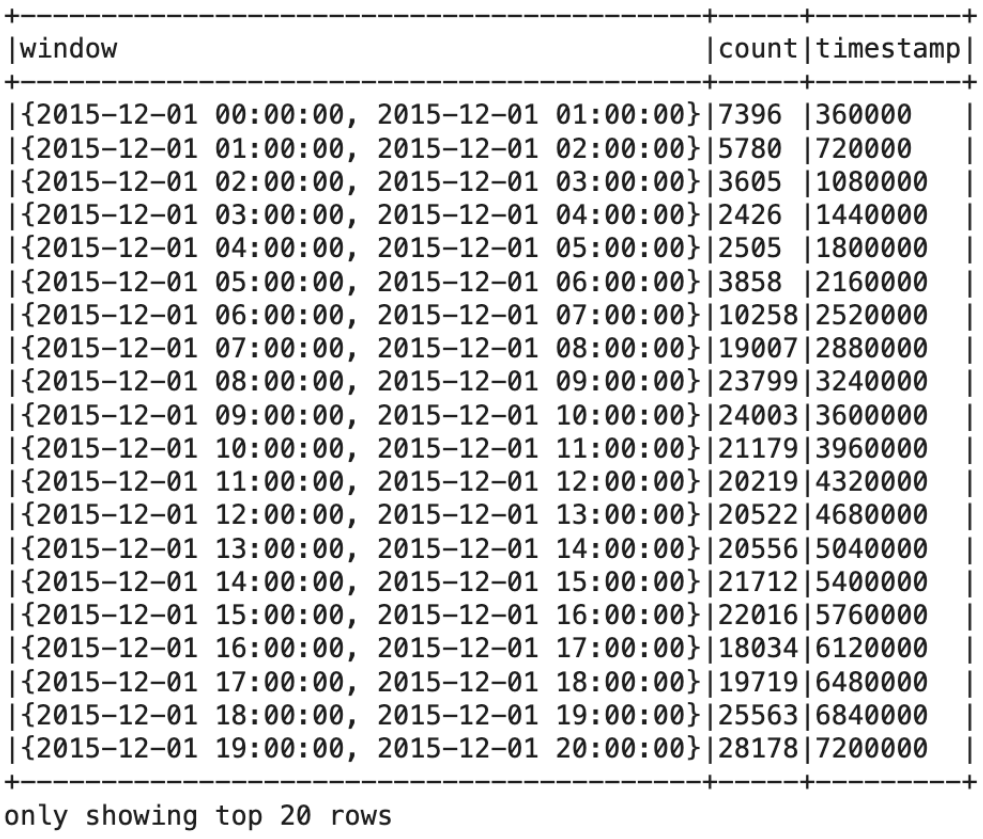
  - Finally, we can count the number of trips grouped by dropoff datetime for each hour.
  - To handle late data, we define the **watermark** on the value of the column “dropoff_datetime”, and define “5 minutes” as the threshold of how late the data is allowed to be.

- Because of the groupBy count aggregation, this query runs in the **Complete** output mode. The whole Result Table will be outputted to the sink after every trigger.
- To create the timestamp information which is required in the expected output, we add a new column named `timestamp`. Specifically, we extract the hour from the end time of each window, and multiply it by 360,000. Note that in the last interval, the end time is 00:00 (the beginning of the next day), therefore, we need to apply a condition to catch this case and change it to 24.

```
# groupBy count
count_dropoff_datetime_DF = (
   dropoff_datetime_DF
        .withWatermark("dropoff_datetime", "5 minutes")
        .groupBy(window("dropoff_datetime", "1 hour"))
        .count()
        .withColumn("timestamp",
                    when(hour(col("window")["end"]) == 0, 24 * 360_000)
                    .otherwise(hour(col("window")["end"]) * 360_000))
)
```

Code: Add watermark, perform group by - count, and add `timestamp` column

- We can then use the `timestamp` column in the **partitionBy** method to write each window into a separate directory. To satisfy the expected directory name format, we also have to rename the `timestamp` column to `output`.



```
+------------------------------------------+-----+---------+
|window                                    |count|timestamp|
+------------------------------------------+-----+---------+
|{2015-12-01 00:00:00, 2015-12-01 01:00:00}|7396 |360000   |
|{2015-12-01 01:00:00, 2015-12-01 02:00:00}|5780 |720000   |
|{2015-12-01 02:00:00, 2015-12-01 03:00:00}|3605 |1080000  |
|{2015-12-01 03:00:00, 2015-12-01 04:00:00}|2426 |1440000  |
|{2015-12-01 04:00:00, 2015-12-01 05:00:00}|2505 |1800000  |
|{2015-12-01 05:00:00, 2015-12-01 06:00:00}|3858 |2160000  |
|{2015-12-01 06:00:00, 2015-12-01 07:00:00}|10258|2520000  |
|{2015-12-01 07:00:00, 2015-12-01 08:00:00}|19007|2880000  |
|{2015-12-01 08:00:00, 2015-12-01 09:00:00}|23799|3240000  |
|{2015-12-01 09:00:00, 2015-12-01 10:00:00}|24003|3600000  |
|{2015-12-01 10:00:00, 2015-12-01 11:00:00}|21179|3960000  |
|{2015-12-01 11:00:00, 2015-12-01 12:00:00}|20219|4320000  |
|{2015-12-01 12:00:00, 2015-12-01 13:00:00}|20522|4680000  |
|{2015-12-01 13:00:00, 2015-12-01 14:00:00}|20556|5040000  |
|{2015-12-01 14:00:00, 2015-12-01 15:00:00}|21712|5400000  |
|{2015-12-01 15:00:00, 2015-12-01 16:00:00}|22016|5760000  |
|{2015-12-01 16:00:00, 2015-12-01 17:00:00}|18034|6120000  |
|{2015-12-01 17:00:00, 2015-12-01 18:00:00}|19719|6480000  |
|{2015-12-01 18:00:00, 2015-12-01 19:00:00}|25563|6840000  |
|{2015-12-01 19:00:00, 2015-12-01 20:00:00}|28178|7200000  |
+------------------------------------------+-----+---------+
only showing top 20 rows
```

Figure 2: A fraction of the result table after running the EventCount query that aggregates the number of trips by dropoff datetime for each hour

2.4 Task 3

- Requirement:
  - Create a RegionEventCount query that counts the number of taxi trips each hour that drop off at either the Goldman Sachs headquarters or the Citigroup headquarters.
  - Write to the output-xxxx a key, value tuple (headquarters, count) with headquarters in {'goldman', 'citigroup'}.
- Solution:
  - We first need to select the specific columns related to the dropoff location and datetime. Specifically, we select 3 columns including: dropoff_longitude, dropoff_latitude, dropoff_datetime.

        - For the Yellow records, the dropoff_longitude and dropoff_latitude columns are **_c11** and **_c12**. However, for the Green records, the dropoff_longitude and dropoff_latitude columns are **_c9** and **_c10**.
        - Therefore, we have to select these columns based on the type of taxi records. In implementation, we do that by combining the **select** method with the **when** and **otherwise** methods.
    - Next, we need to know whether or not a dropoff location is within one of the headquarters. We add a new column named "**state**" to store that information. The value of this new column can be either: "goldman", "citigroup", or "None".
        - Looking at the bounding boxes of the two headquarters, we can see that they are 4-sided **polygons**. Therefore, checking whether or not a dropoff location is within one of these bounding boxes is essentially the task of checking if a point is inside a polygon.

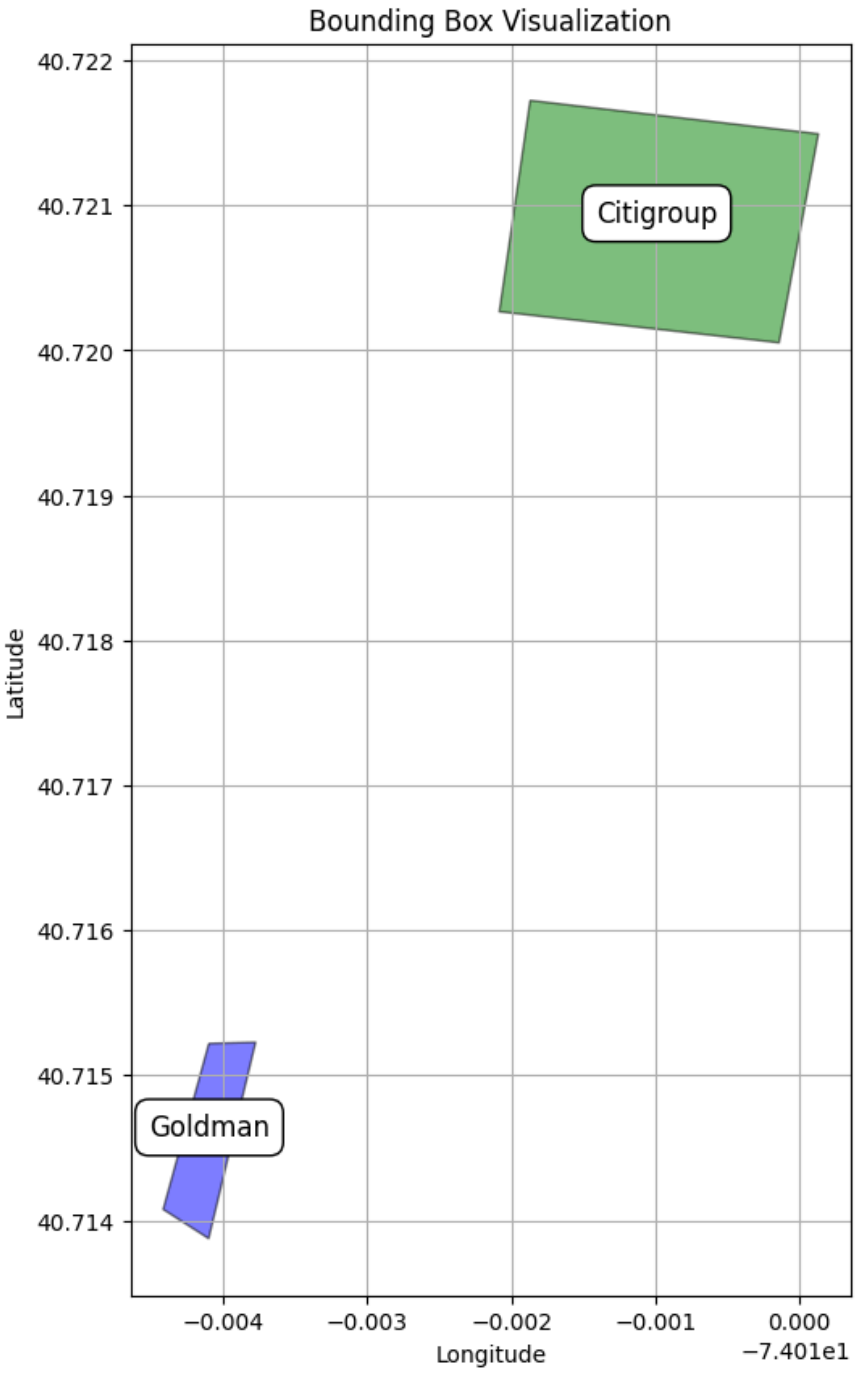


Figure 3: Visualization of the Goldman and Citigroup headquarters

- In this lab, we use the **Ray casting** algorithm to test if a point is inside a polygon. Basically, we track the number of times the point intersects the edges of the polygon. If the count of intersections is even, the point is outside the polygon.

Conversely, if the count is odd, the point is either inside the polygon or on its boundary.

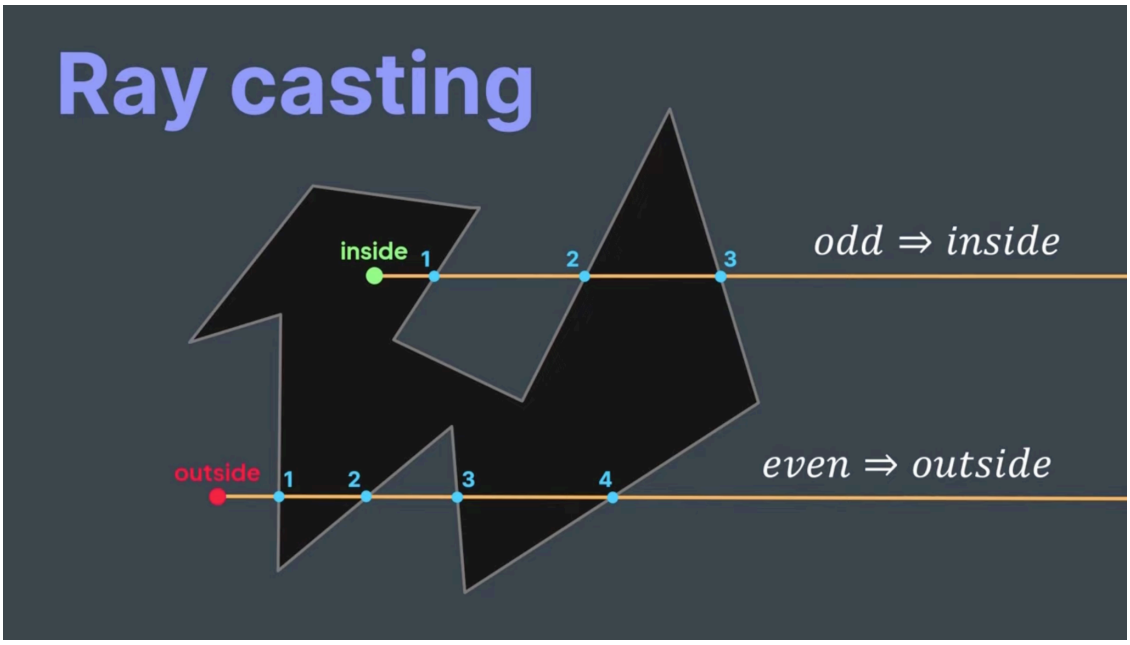


Figure 4: The intuition behind the Ray casting algorithm (Source)

- Specifically, we refer to the point as **P**, the polygon as **Q**. We refer to the ray, starting at P and going right to infinity, as **R**.
- Next, the algorithm will loop through all the edges in Q and test whether R intersects with each edge. For each edge, we set two variables $A(x_1, y_1)$ and $B(x_2, y_2)$ corresponding to the edge's points such that $y_1 < y_2$.

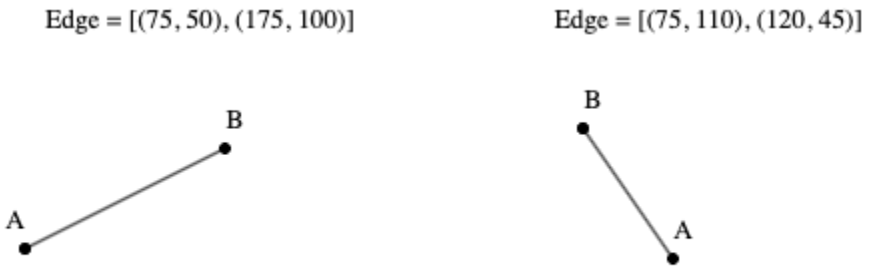


Figure 5: Edge's points setting such that A always has a y-value less than B (Source)

- The next step is to check if P is above or below each edge. To check this we can check the truth of the following statement:

$$y_1 < y_p < y_2$$

- If this statement is false, then P is above or below the edge shown in Figure 6 and R does not intersect with the edge.

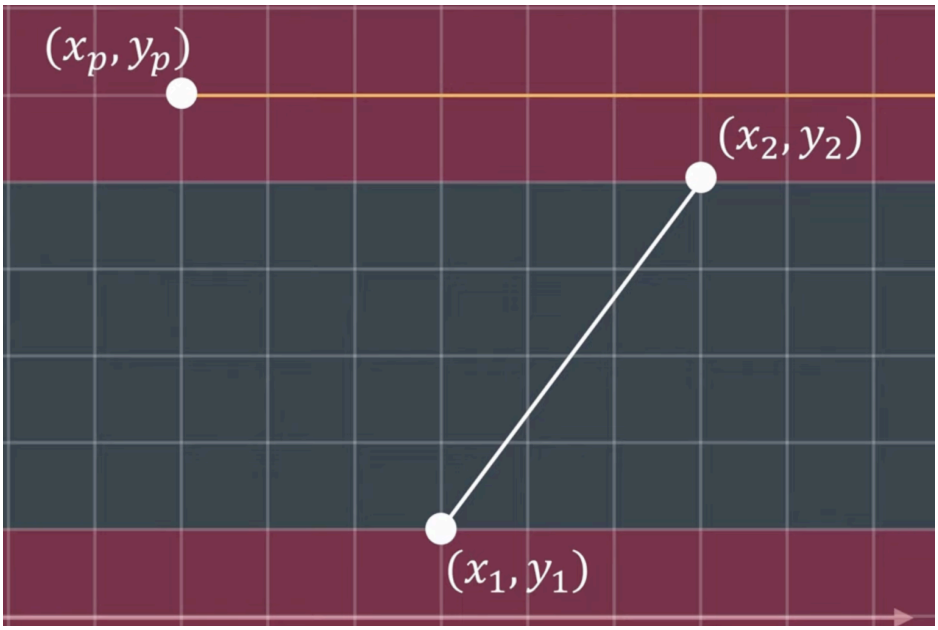


Figure 6: Points above or below edge (Source)

- We also check to see if P is to the right of the edge using the following statement:

$$x_p > max(x_1, x_2)$$

- If this statement is true then P is to the right of the edge as shown in Figure 7 and, R does not intersect with the edge.

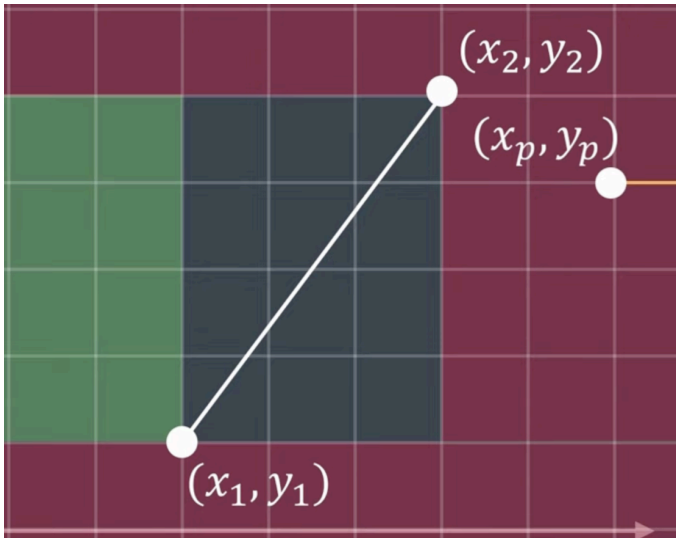


Figure 7: Point to the right of the furthest edge point (Source)

- We can now be sure that P is somewhere in the **dotted** section in Figure 8. If P is in the green section, R intersects with the edge. If P is in the red section, R does not intersect with the edge.

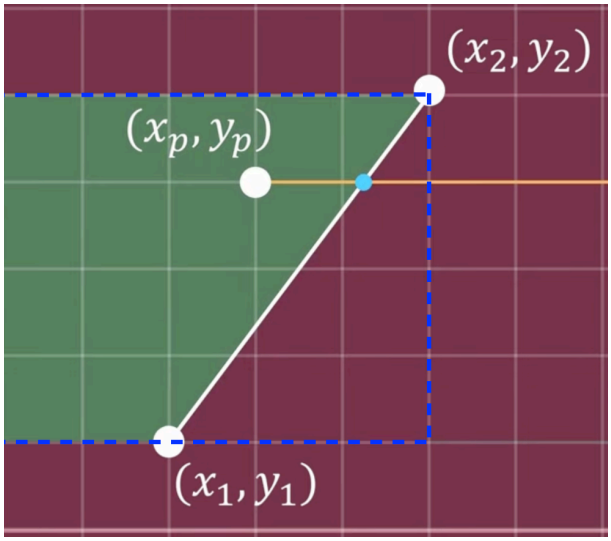


Figure 8: Remaining area to check (Source)

- The first step to check if P is inside the green section is to find the intersection point x_0.

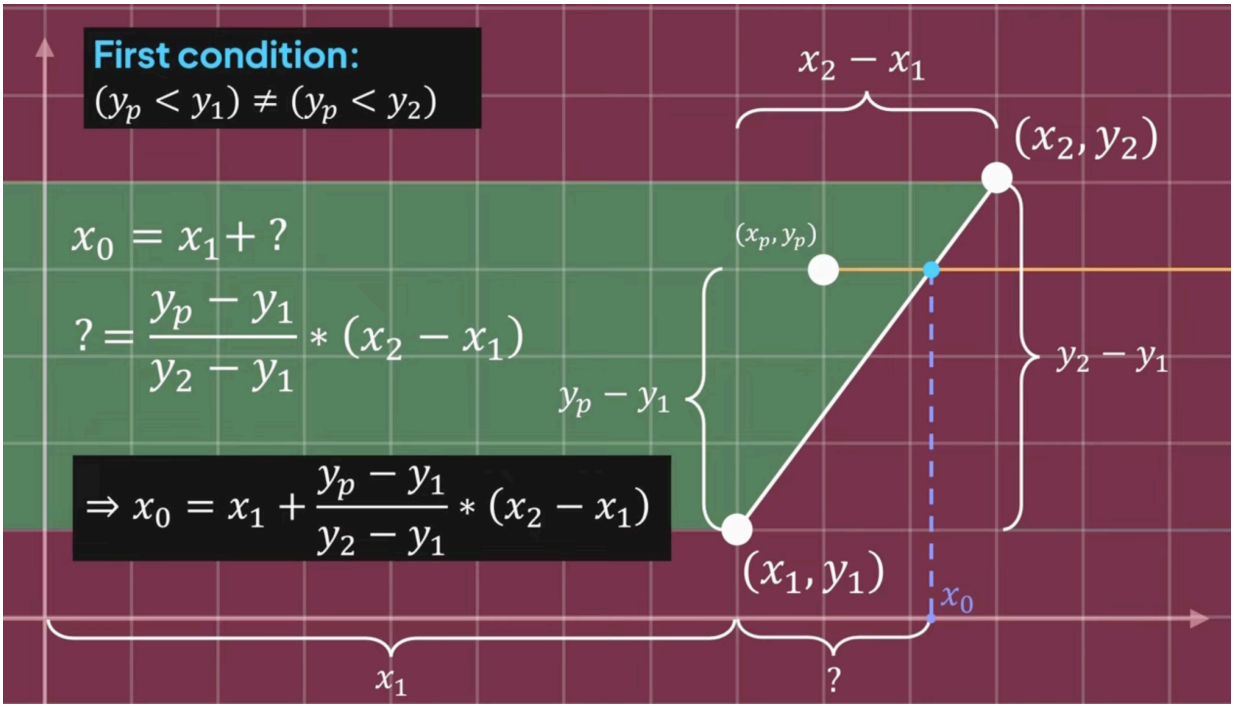
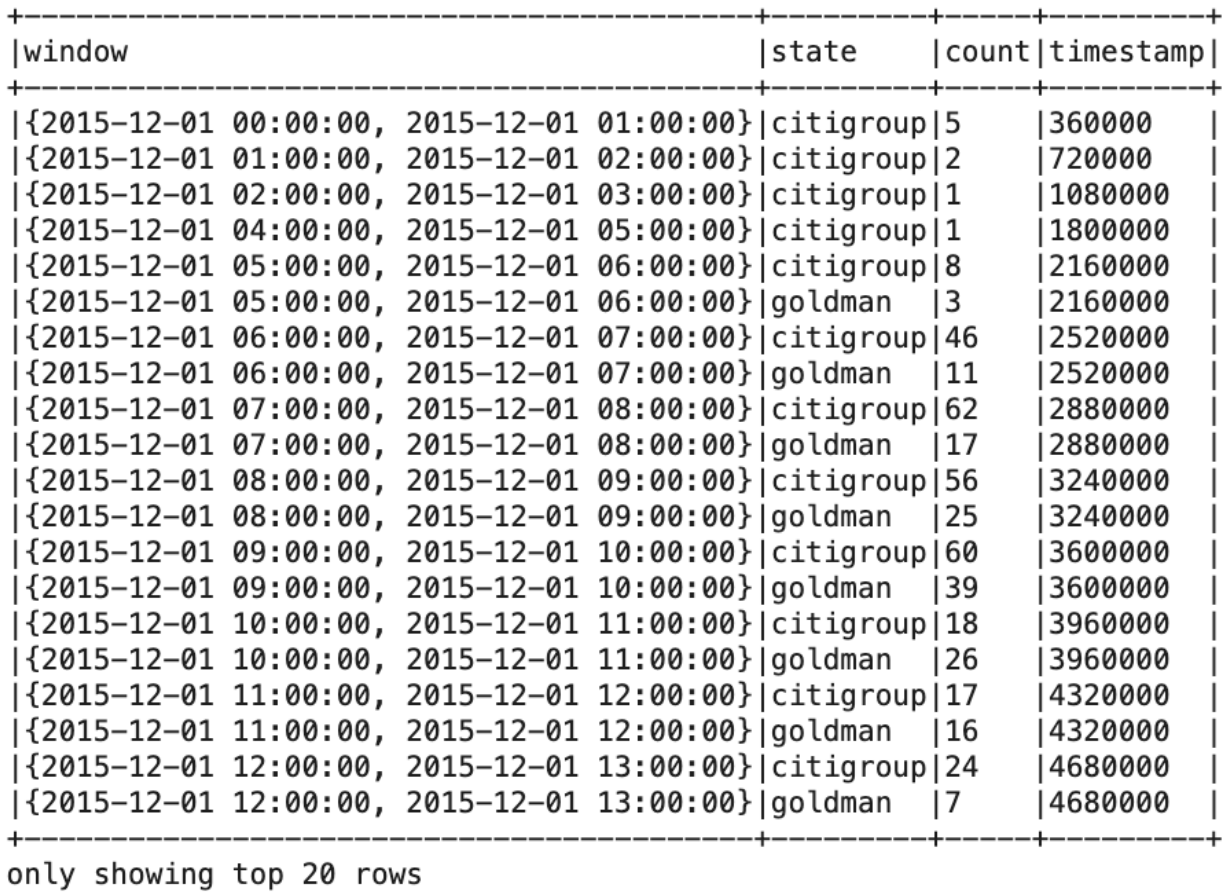


Figure 9: Math formula to find the x-value of the intersection point (Source)

- Finally, we can check if x_p is less than x_0. If this is the case, then P is to the left of the intersection point, thus crossing the edge.

- Overall, to add a new column “state” to each record, we first need to build a user-defined function. This function implements the Ray casting algorithm to check whether or not the dropoff location, which is just a 2D point, is inside one of the headquarter polygons, hence, the function name is **point_in_polygon**. We can then use the **withColumn** method with this user-defined function to create the “state” column.
- We can then filter the dataframe to retain only the records whose “state” value is either “goldman” or “citigroup”. This way, we can perform group by and count the number of records at either headquarters for each hour.



window	state	count	timestamp
{2015-12-01 00:00:00, 2015-12-01 01:00:00}	citigroup	5	360000
{2015-12-01 01:00:00, 2015-12-01 02:00:00}	citigroup	2	720000
{2015-12-01 02:00:00, 2015-12-01 03:00:00}	citigroup	1	1080000
{2015-12-01 04:00:00, 2015-12-01 05:00:00}	citigroup	1	1800000
{2015-12-01 05:00:00, 2015-12-01 06:00:00}	citigroup	8	2160000
{2015-12-01 05:00:00, 2015-12-01 06:00:00}	goldman	3	2160000
{2015-12-01 06:00:00, 2015-12-01 07:00:00}	citigroup	46	2520000
{2015-12-01 06:00:00, 2015-12-01 07:00:00}	goldman	11	2520000
{2015-12-01 07:00:00, 2015-12-01 08:00:00}	citigroup	62	2880000
{2015-12-01 07:00:00, 2015-12-01 08:00:00}	goldman	17	2880000
{2015-12-01 08:00:00, 2015-12-01 09:00:00}	citigroup	56	3240000
{2015-12-01 08:00:00, 2015-12-01 09:00:00}	goldman	25	3240000
{2015-12-01 09:00:00, 2015-12-01 10:00:00}	citigroup	60	3600000
{2015-12-01 09:00:00, 2015-12-01 10:00:00}	goldman	39	3600000
{2015-12-01 10:00:00, 2015-12-01 11:00:00}	citigroup	18	3960000
{2015-12-01 10:00:00, 2015-12-01 11:00:00}	goldman	26	3960000
{2015-12-01 11:00:00, 2015-12-01 12:00:00}	citigroup	17	4320000
{2015-12-01 11:00:00, 2015-12-01 12:00:00}	goldman	16	4320000
{2015-12-01 12:00:00, 2015-12-01 13:00:00}	citigroup	24	4680000
{2015-12-01 12:00:00, 2015-12-01 13:00:00}	goldman	7	4680000

only showing top 20 rows

Figure 10: A fraction of the result table after running the RegionEventCount query that counts the number of taxi trips each hour that drop off at either the Goldman Sachs headquarters or the Citigroup headquarters

2.5 Task 4

- Requirement:
    - Build a simple "trend detector" to find out when there are lots of arrivals at either Goldman Sachs or Citigroup headquarters.
        - The trend detector should "go off" when there are at least twice as many arrivals in the current interval as there were in the past interval.
        - To reduce "spurious" detections, we want to make sure the

detector only "trips" if there are ten or more arrivals in the current interval.

- The detector should output its results to stdout in the form of the following: “The number of arrivals to Goldman Sachs has doubled from X to Y at Z!” or “The number of arrivals to Citigroup has doubled from X to Y at Z!”.

- Solution:
  - We perform group by and count the number of records at either headquarters for each 10 minutes. This is similar to Task 3.
  - We separate the Result Table into 2 dataframes, one for Goldman Sach and the other for Citigroup. The reason is that the time interval is different between headquarters.
    - For example, Goldman Sach may have arrivals from 13:10 to 13:20, but Citigroup does not. Such inconsistency causes difficulty for later processing when we want to identify if the number of arrivals to a specific headquarters has doubled.
  - Now each dataframe stores taxi records of 1 headquarter. To detect a doubled number of arrivals, each record needs to know 2 things:
    - First, the arrivals count of the previous record.
    - Second, the time interval of the previous record. This is necessary because we need to check if 2 records have consecutive time intervals.
  - Therefore, we add 2 new columns namely “prev_window” and “prev_count” into each dataframe. To implement this, we use the **lag** function inside the **withColumn** method.

```
# Define a window specification
windowSpec = Window.orderBy("window")

goldmanEventCount10mDF = spark.sql('select (*) from
```

```
goldman_event_count_each_10m order by window')

goldmanEventCount10mDF_with_prev = (
   goldmanEventCount10mDF
       .withColumn("prev_window", lag("window", offset=1,
default=None).over(windowSpec))
       .withColumn("prev_count", lag("count", offset=1,
default=None).over(windowSpec))
)
```

Code: Add the “prev_window” and “prev_count” columns to the dataframe for Goldman Sach headquarter

```
+-------+-----------------------------------------+-----+-----------------------------------------+----------+
|state  |window                                   |count|prev_window                              |prev_count|
+-------+-----------------------------------------+-----+-----------------------------------------+----------+
|goldman|{2015-12-01 05:20:00, 2015-12-01 05:30:00}|1    |NULL                                     |NULL      |
|goldman|{2015-12-01 05:50:00, 2015-12-01 06:00:00}|2    |{2015-12-01 05:20:00, 2015-12-01 05:30:00}|1         |
|goldman|{2015-12-01 06:00:00, 2015-12-01 06:10:00}|1    |{2015-12-01 05:50:00, 2015-12-01 06:00:00}|2         |
|goldman|{2015-12-01 06:10:00, 2015-12-01 06:20:00}|1    |{2015-12-01 06:00:00, 2015-12-01 06:10:00}|1         |
|goldman|{2015-12-01 06:20:00, 2015-12-01 06:30:00}|1    |{2015-12-01 06:10:00, 2015-12-01 06:20:00}|1         |
|goldman|{2015-12-01 06:30:00, 2015-12-01 06:40:00}|2    |{2015-12-01 06:20:00, 2015-12-01 06:30:00}|1         |
|goldman|{2015-12-01 06:40:00, 2015-12-01 06:50:00}|3    |{2015-12-01 06:30:00, 2015-12-01 06:40:00}|2         |
|goldman|{2015-12-01 06:50:00, 2015-12-01 07:00:00}|3    |{2015-12-01 06:40:00, 2015-12-01 06:50:00}|3         |
|goldman|{2015-12-01 07:00:00, 2015-12-01 07:10:00}|3    |{2015-12-01 06:50:00, 2015-12-01 07:00:00}|3         |
|goldman|{2015-12-01 07:20:00, 2015-12-01 07:30:00}|3    |{2015-12-01 07:00:00, 2015-12-01 07:10:00}|3         |
|goldman|{2015-12-01 07:30:00, 2015-12-01 07:40:00}|5    |{2015-12-01 07:20:00, 2015-12-01 07:30:00}|3         |
|goldman|{2015-12-01 07:40:00, 2015-12-01 07:50:00}|4    |{2015-12-01 07:30:00, 2015-12-01 07:40:00}|5         |
|goldman|{2015-12-01 07:50:00, 2015-12-01 08:00:00}|2    |{2015-12-01 07:40:00, 2015-12-01 07:50:00}|4         |
|goldman|{2015-12-01 08:00:00, 2015-12-01 08:10:00}|6    |{2015-12-01 07:50:00, 2015-12-01 08:00:00}|2         |
|goldman|{2015-12-01 08:10:00, 2015-12-01 08:20:00}|6    |{2015-12-01 08:00:00, 2015-12-01 08:10:00}|6         |
|goldman|{2015-12-01 08:20:00, 2015-12-01 08:30:00}|1    |{2015-12-01 08:10:00, 2015-12-01 08:20:00}|6         |
|goldman|{2015-12-01 08:30:00, 2015-12-01 08:40:00}|2    |{2015-12-01 08:20:00, 2015-12-01 08:30:00}|1         |
|goldman|{2015-12-01 08:40:00, 2015-12-01 08:50:00}|3    |{2015-12-01 08:30:00, 2015-12-01 08:40:00}|2         |
|goldman|{2015-12-01 08:50:00, 2015-12-01 09:00:00}|7    |{2015-12-01 08:40:00, 2015-12-01 08:50:00}|3         |
|goldman|{2015-12-01 09:00:00, 2015-12-01 09:10:00}|3    |{2015-12-01 08:50:00, 2015-12-01 09:00:00}|7         |
+-------+-----------------------------------------+-----+-----------------------------------------+----------+
only showing top 20 rows
```

Figure 11: A fraction of the result table after adding 2 new columns (“prev_window” and “prev_count”).

- Then, we can test if a specific time interval has witnessed a doubled number of intervals. Specifically, we add 1 more column named “has_doubled”. If both of the two conditions below are met, the “has_doubled” column will be “yes”, otherwise its value will be “no”.
  - For each record, we check if “count” is greater than or equal to 10, and is at least twice as much as “prev_count”.
  - We also check if the start time of “window” is equal to the end

time of “prev_window”. If this is the case, then the current record and the previous record are consecutive.

- Finally, we can filter for records where the has_doubled column has the value 'yes', and print these records to console.

```
citigroupEventCountEach10mDF_check_doubled = (
   citigroupEventCount10mDF_with_prev
       .withColumn(
           "has_doubled",
           when(
               (col("count") >= 10) &
               (col("count") >= 2 * col("prev_count")) &
               (col("window.start") == col("prev_window.end")),
               "yes")
           .otherwise("no"))
)
```

Code: Add the “has_doubled” column to the dataframe for the Citigroup headquarter

```
+---------+-----------------------------------------+-----+-----------------------------------------+----------+-----------+
|state    |window                                   |count|prev_window                              |prev_count|has_doubled|
+---------+-----------------------------------------+-----+-----------------------------------------+----------+-----------+
|citigroup|{2015-12-01 08:50:00, 2015-12-01 09:00:00}|12   |{2015-12-01 08:40:00, 2015-12-01 08:50:00}|3         |yes        |
|citigroup|{2015-12-01 14:00:00, 2015-12-01 14:10:00}|10   |{2015-12-01 13:50:00, 2015-12-01 14:00:00}|3         |yes        |
+---------+-----------------------------------------+-----+-----------------------------------------+----------+-----------+
```

Figure 12: The result table of records having doubled arrivals count in the Citigroup headquarter

References

1. Structured Streaming Programming Guide (https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html)
2. Simulate Lag Function - Spark structured streaming (https://stackoverflow.com/questions/48725764/simulate-lag-function-spark-structured-streaming)
3. Checking if a point is inside a polygon is RIDICULOUSLY simple (Ray casting algorithm) - Inside code (https://www.youtube.com/watch?v=RSXM9bgqxJM)
4. Ray Casting Algorithm (http://www.philliplemons.com/posts/ray-casting-algorithm)