National University of Vietnam - HCM city

University of Science

Department of Information Technology

Report Lab 03: Spark Streaming

Subject: Big Data

Students:
Nguyễn Thế Thiện - 21127170
Châu Tấn Kiệt – 21127329
Trịnh Minh Long - 21127642

Guidance teachers:
Ms. Nguyễn Ngọc Thảo
Mr. Bùi Huỳnh Trung Nam
MSc. Đỗ Trọng Lễ

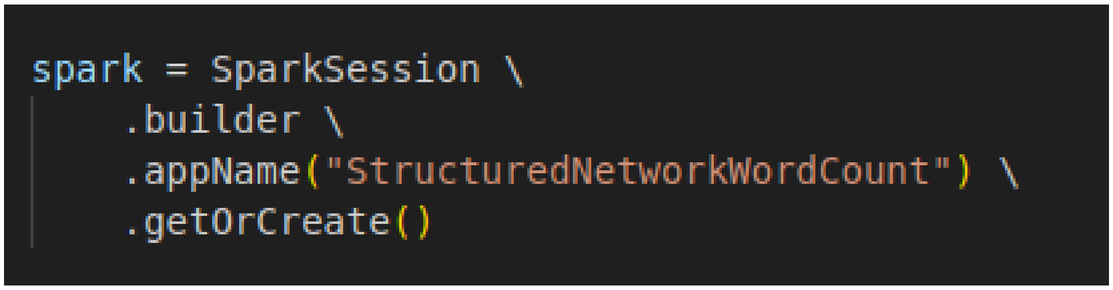
Ngày 22 tháng 5 năm 2024

Mục lục

1 Task 1: Discover a method to simulate a stream by utilizing data sourced from files.

1.1 Implementation

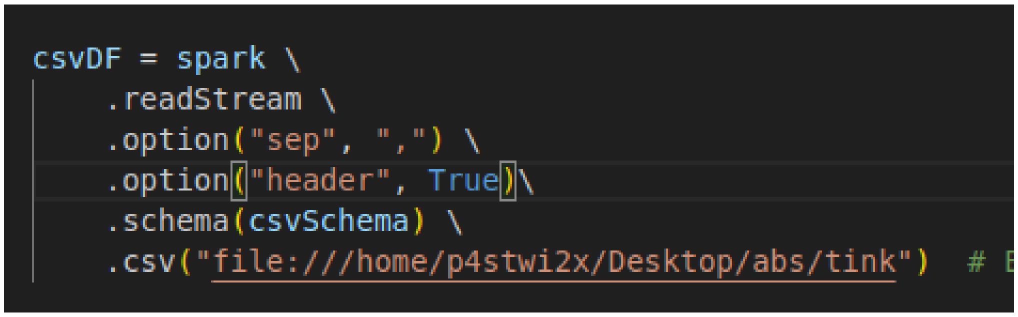
The method we have implemented is guided by Apache Spark guide on Structured Streaming [5]. We have created a data stream simulation from a local directory of csv files.
Here are the steps of the implementation:
- **Create a SparkSession object.** This is the most basic step in order to use Apache Spark, we shall name the object *spark* for easy understanding.



```
spark = SparkSession \
    .builder \
    .appName("StructuredNetworkWordCount") \
    .getOrCreate()
```

Hình 1: Spark Session object

- **Read data stream.** We call *spark.readStream* with additional configurations stated below:

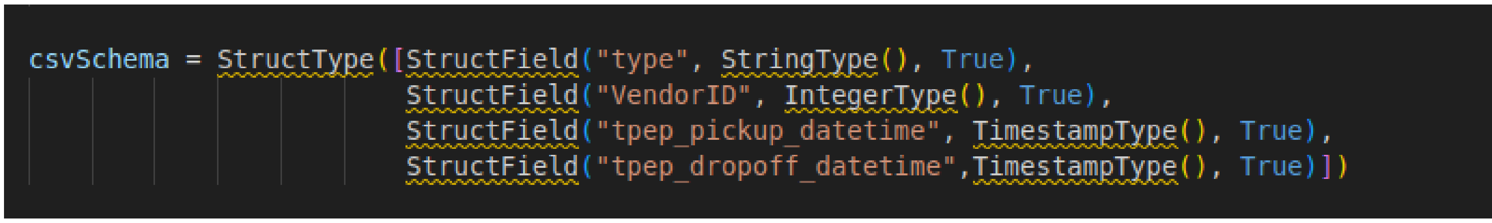


```
csvDF = spark \
    .readStream \
    .option("sep", ",") \
    .option("header", True)\
    .schema(csvSchema) \
    .csv("file:///home/p4stwi2x/Desktop/abs/tink")  # E
```

Hình 2: Read data stream from local files

- **schema()**. This function is optional though recommended for easier usage and more fault tolerance. From Apache Spark guide on Structured Streaming, "schema of the DataFrame is not checked at compile time, only checked at runtime when the query is submitted", that if not for the function, Spark gets the data untyped, which is inconvenient for future usage, especially in selecting columns.

  Additionally, the preset schema is shown below, in which the structure and data types are from *pyspark.sql.types*. Note that by this method, we could temporarily assign column names we shall be using during queries, and we could preset a schema of fewer columns than it should be in the files in order to extract enough data for usage instead of all, the example of this shall be covered in the **1.2 Running** section.



```
csvSchema = StructType([StructField("type", StringType(), True),
                        StructField("VendorID", IntegerType(), True),
                        StructField("tpep_pickup_datetime", TimestampType(), True),
                        StructField("tpep_dropoff_datetime",TimestampType(), True)])
```

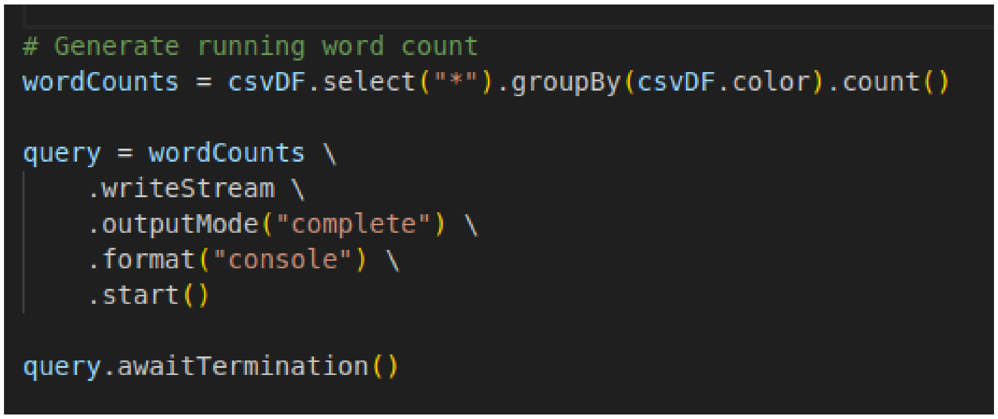
Hình 3: Preset schema

- **csv()**. This function is used to indicate the source directory of csv files, it is a replacement of *format("csv").load("/path/to/directory")*.

  It is worth noting that in order to access the local files of the machine, the user should have the prefix *file://*, else it is defaulted as a path in Hadoop Filesystem, which is an another method of data streaming from HDFS. But for now we use the local files.

- **Additional configurations**. This is implemented in the *option()* function, which we use *option("sep", ",")* to indicate the separator character(s) in the source data, and *option("header", True)* to indicate if there exists headers in the source data files to skip

spark.readStream returns a streaming DataFrame object that we shall use for querying.
- **Query**. The query session for streaming mode is almost the same as batch mode since most basic operators work in both modes. However, showing results of query is different, as in this case we need to write from the streaming DataFrame by *spark.writeStream*.



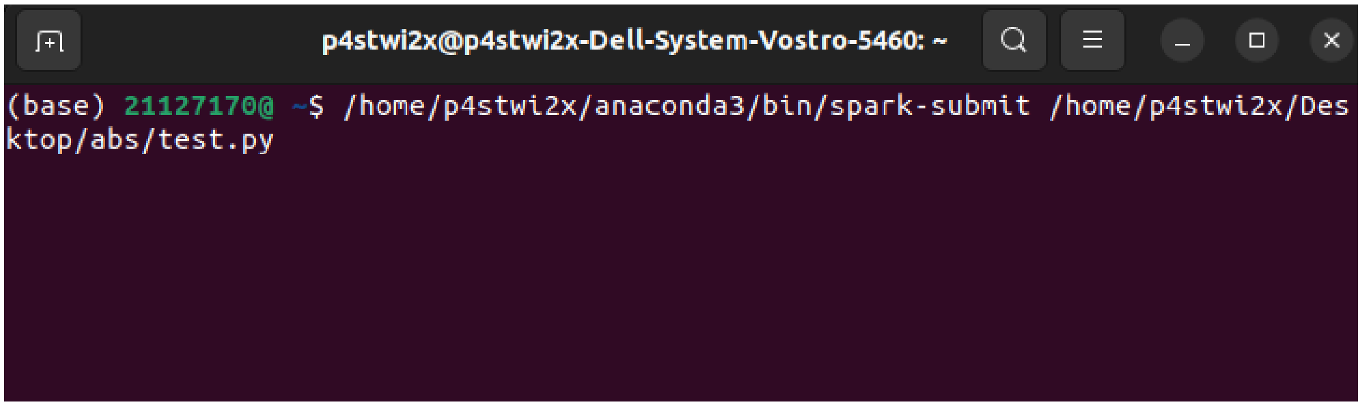
```
# Generate running word count
wordCounts = csvDF.select("*").groupBy(csvDF.color).count()

query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

Hình 4: Query in streaming mode

- **outputMode("complete")**. Setting this output mode shall update the query results when data is changed, in real time.
- **format("console")**. Setting this output format shall output query results in the console window.
- **start()**. This function calls for starting query computations.
- **awaitTermination()**. This function sets the active streaming query to wait for termination (usually by Ctrl+C). User can input an integer as the timer (in seconds) to terminate the streaming query, else leaving blank means running indefinitely.

1.2 Running

The process could be started in console by using the command *spark-submit*.

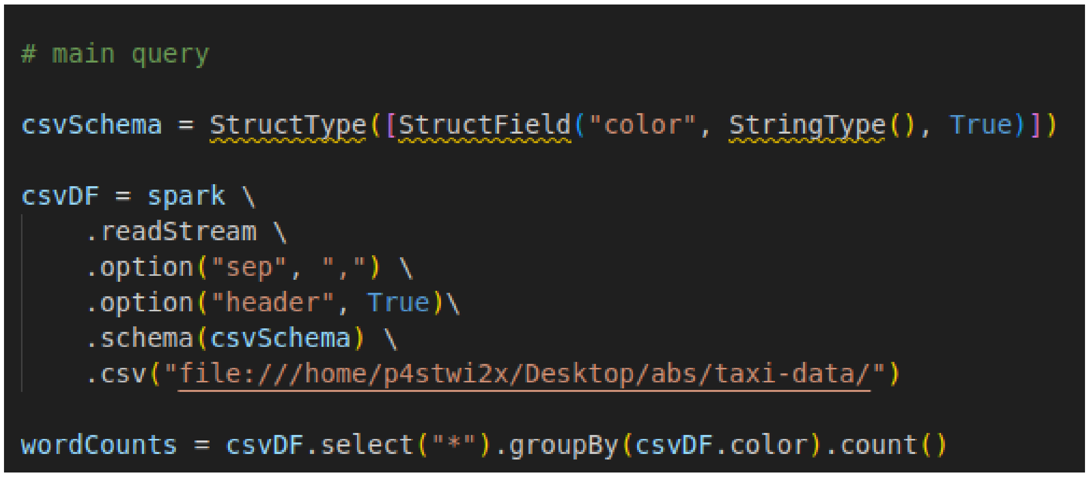


p4stwi2x@p4stwi2x-Dell-System-Vostro-5460: ~

```
(base) 21127170@ ~$ /home/p4stwi2x/anaconda3/bin/spark-submit /home/p4stwi2x/Des
ktop/abs/test.py
```

Hình 5: Running in console

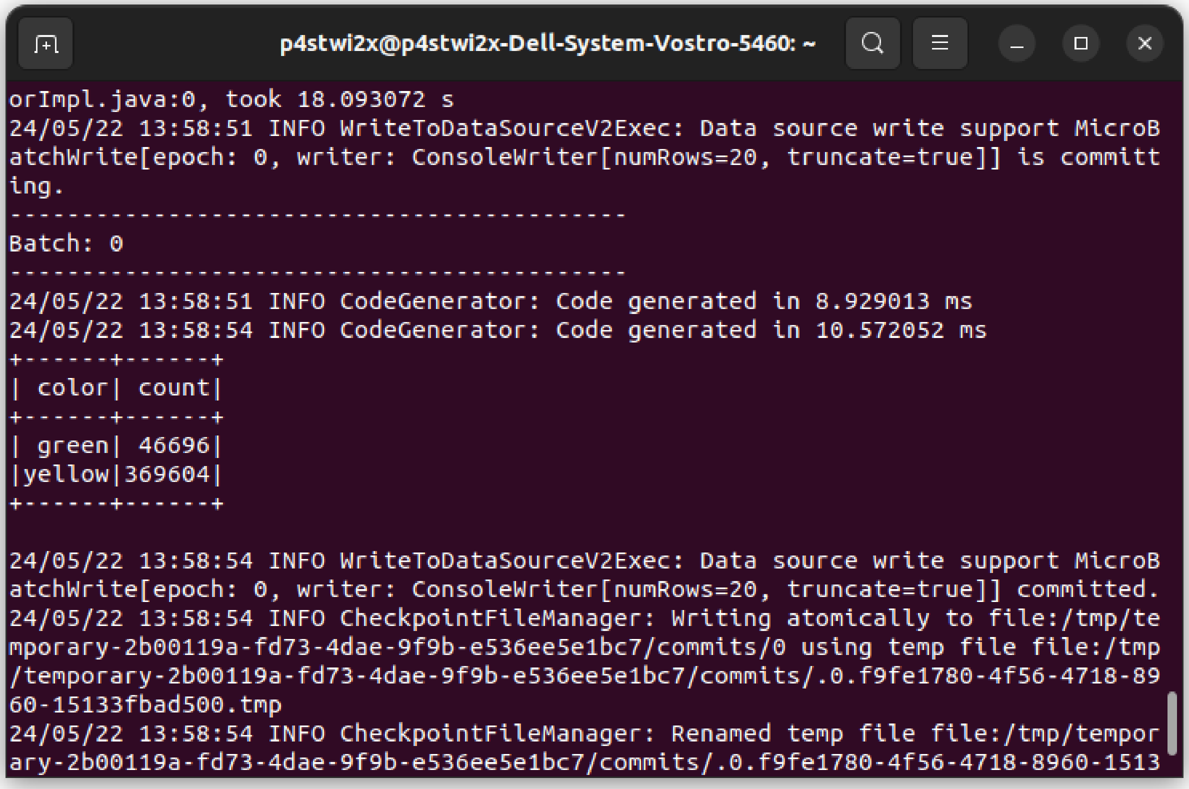
In this case we will be running *test.py* which will be counting the number of samples with different colors for the *taxi-data* dataset.



```
# main query

csvSchema = StructType([StructField("color", StringType(), True)])

csvDF = spark \
    .readStream \
    .option("sep", ",") \
    .option("header", True)\
    .schema(csvSchema) \
    .csv("file:///home/p4stwi2x/Desktop/abs/taxi-data/")

wordCounts = csvDF.select("*").groupBy(csvDF.color).count()
```

Hình 6: Query: counting colors

And this is the results:

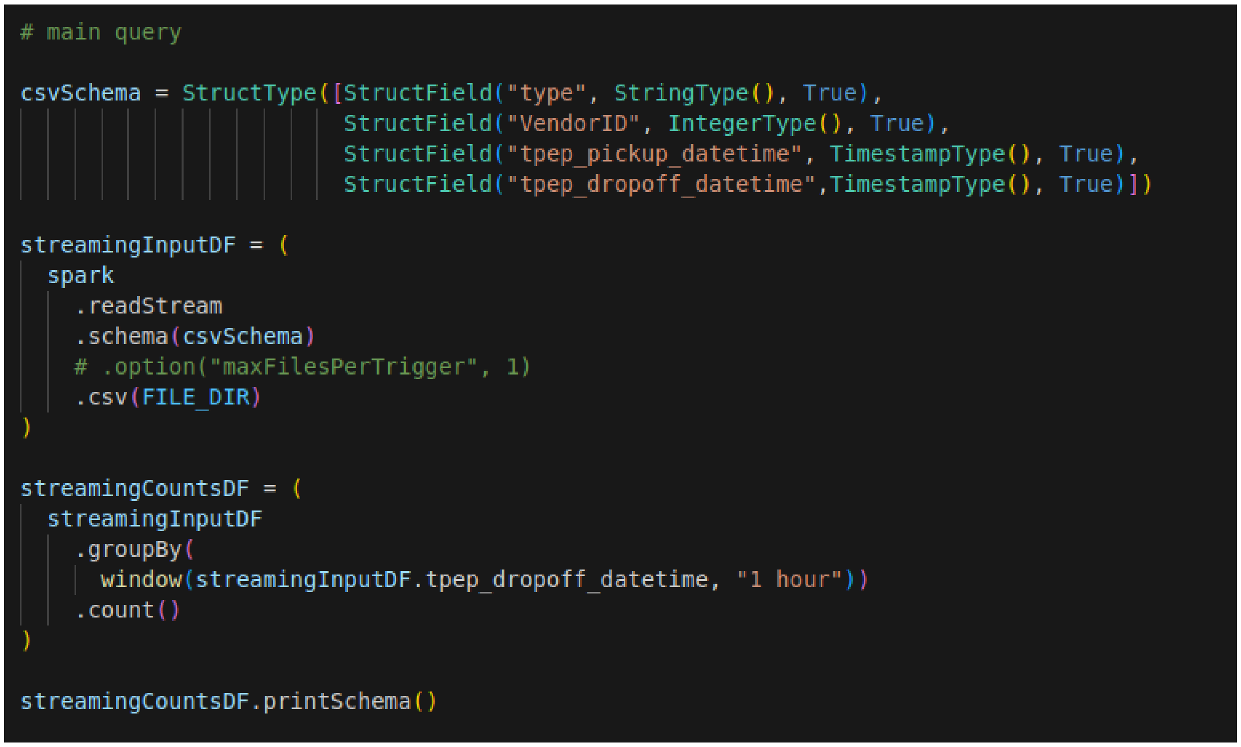


```
orImpl.java:0, took 18.093072 s
24/05/22 13:58:51 INFO WriteToDataSourceV2Exec: Data source write support MicroB
atchWrite[epoch: 0, writer: ConsoleWriter[numRows=20, truncate=true]] is committ
ing.
-------------------------------------------
Batch: 0
-------------------------------------------
24/05/22 13:58:51 INFO CodeGenerator: Code generated in 8.929013 ms
24/05/22 13:58:54 INFO CodeGenerator: Code generated in 10.572052 ms
+------+------+
| color| count|
+------+------+
| green| 46696|
|yellow|369604|
+------+------+

24/05/22 13:58:54 INFO WriteToDataSourceV2Exec: Data source write support MicroB
atchWrite[epoch: 0, writer: ConsoleWriter[numRows=20, truncate=true]] committed.
24/05/22 13:58:54 INFO CheckpointFileManager: Writing atomically to file:/tmp/te
mporary-2b00119a-fd73-4dae-9f9b-e536ee5e1bc7/commits/0 using temp file file:/tmp
/temporary-2b00119a-fd73-4dae-9f9b-e536ee5e1bc7/commits/.0.f9fe1780-4f56-4718-89
60-15133fbad500.tmp
24/05/22 13:58:54 INFO CheckpointFileManager: Renamed temp file file:/tmp/tempor
ary-2b00119a-fd73-4dae-9f9b-e536ee5e1bc7/commits/.0.f9fe1780-4f56-4718-8960-1513
```

Hình 7: Console results

2 Task 2: A query that aggregates the number of trips by drop-off datetime for each hour.

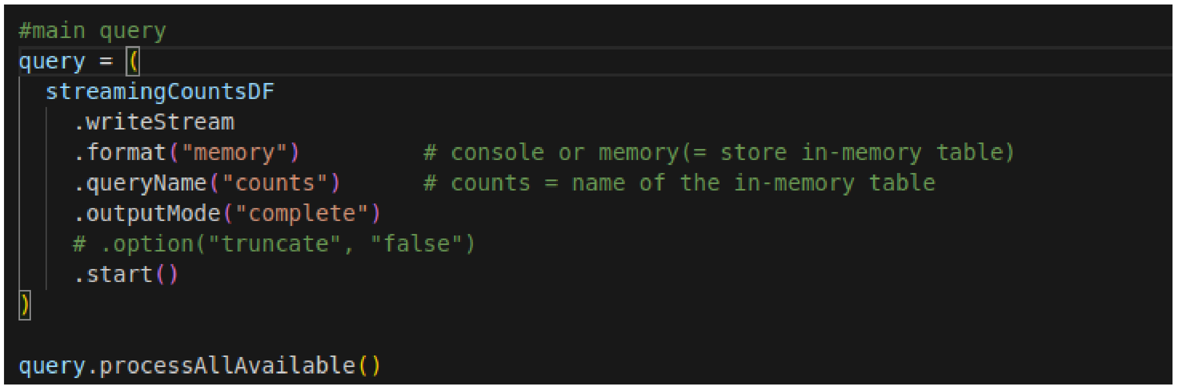
Starting with the query, since the *drop-off datetime* column appears at 4th, we shall preset a schema of 4 columns. By that, we input the data files into a streaming DataFrame *streamingInputDF*. The query session is by grouping the dataframe by the dropoff column, using Spark function *window*, and grouping the dataframe by each time window of 1 hour.



```
# main query

csvSchema = StructType([StructField("type", StringType(), True),
                        StructField("VendorID", IntegerType(), True),
                        StructField("tpep_pickup_datetime", TimestampType(), True),
                        StructField("tpep_dropoff_datetime",TimestampType(), True)])

streamingInputDF = (
  spark
    .readStream
    .schema(csvSchema)
    # .option("maxFilesPerTrigger", 1)
    .csv(FILE_DIR)
)

streamingCountsDF = (
  streamingInputDF
    .groupBy(
      window(streamingInputDF.tpep_dropoff_datetime, "1 hour"))
    .count()
)

streamingCountsDF.printSchema()
```

Hình 8: Setup and input data stream

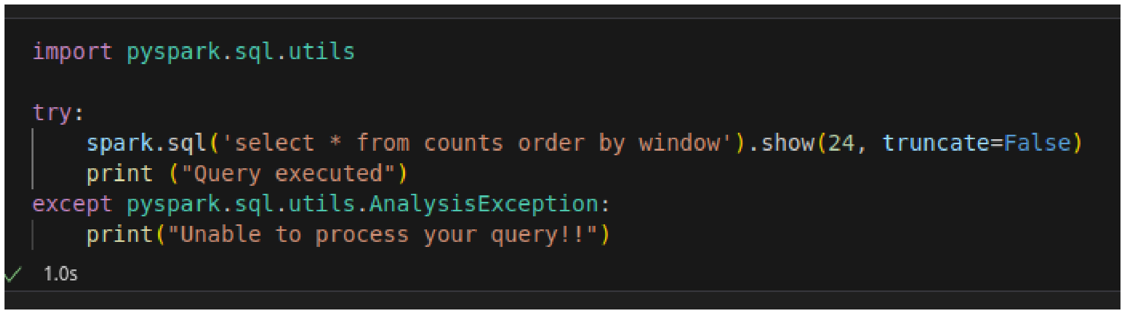
Then we shall start the streaming query as said above in task 1, with *format("memory")* instead in order to store the in-memory query result for later usage. Also we do not use *query.awaitTermination()* but *query.processAllAvailable()* as it blocks arriving data until all available in the source has been processed and committed. That way could we escape the live query process and continue with other commands while the streaming continues in the background.



```python
#main query
query = (
  streamingCountsDF
    .writeStream
    .format("memory")          # console or memory(= store in-memory table)
    .queryName("counts")       # counts = name of the in-memory table
    .outputMode("complete")
    # .option("truncate", "false")
    .start()
)

query.processAllAvailable()
```

Hình 9: Query

Hence by that, we are able to extract the static in-memory query result.

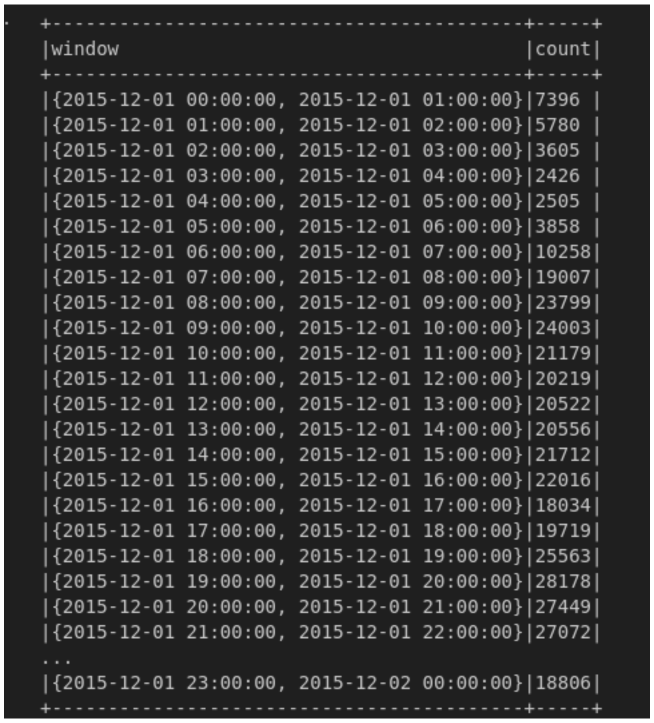


```python
import pyspark.sql.utils

try:
    spark.sql('select * from counts order by window').show(24, truncate=False)
    print ("Query executed")
except pyspark.sql.utils.AnalysisException:
    print("Unable to process your query!!")
```

```
✓ 1.0s
```

Hình 10: Extract query result as static dataframe

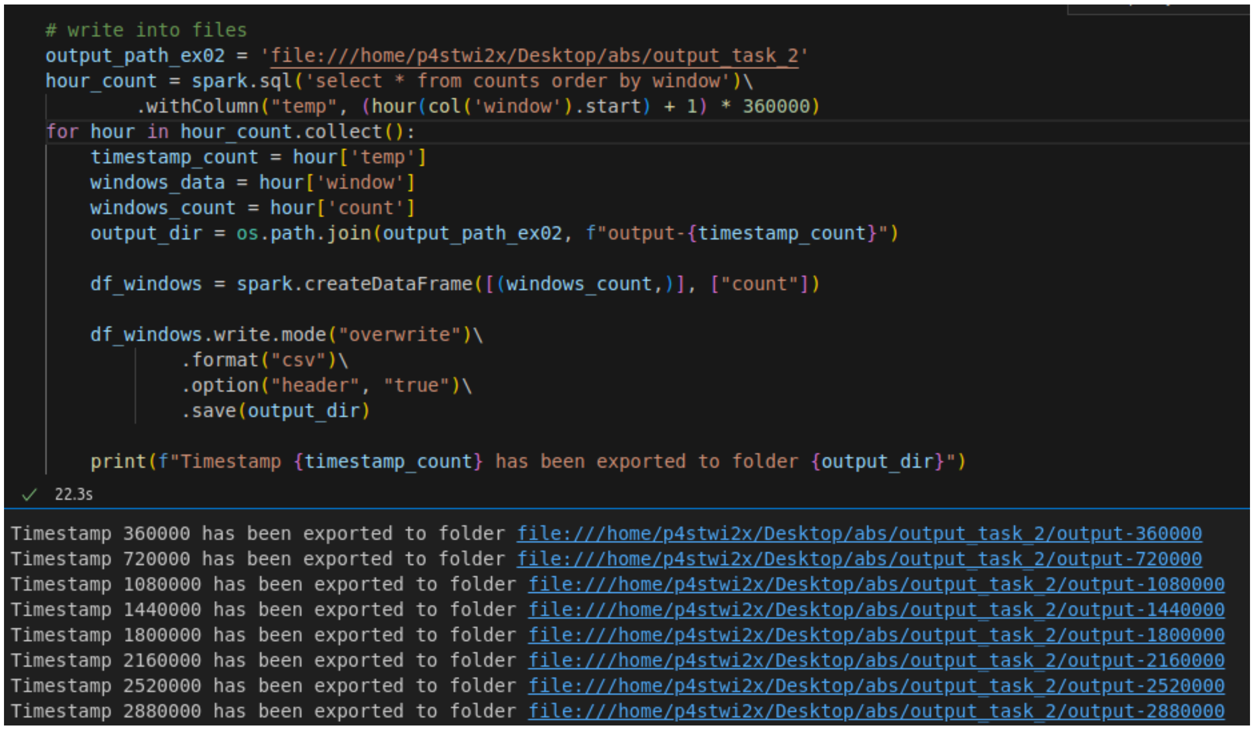
...which gives the result we are looking for



```
+----------------------------------------+-----+
|window                                  |count|
+----------------------------------------+-----+
|{2015-12-01 00:00:00, 2015-12-01 01:00:00}|7396 |
|{2015-12-01 01:00:00, 2015-12-01 02:00:00}|5780 |
|{2015-12-01 02:00:00, 2015-12-01 03:00:00}|3605 |
|{2015-12-01 03:00:00, 2015-12-01 04:00:00}|2426 |
|{2015-12-01 04:00:00, 2015-12-01 05:00:00}|2505 |
|{2015-12-01 05:00:00, 2015-12-01 06:00:00}|3858 |
|{2015-12-01 06:00:00, 2015-12-01 07:00:00}|10258|
|{2015-12-01 07:00:00, 2015-12-01 08:00:00}|19007|
|{2015-12-01 08:00:00, 2015-12-01 09:00:00}|23799|
|{2015-12-01 09:00:00, 2015-12-01 10:00:00}|24003|
|{2015-12-01 10:00:00, 2015-12-01 11:00:00}|21179|
|{2015-12-01 11:00:00, 2015-12-01 12:00:00}|20219|
|{2015-12-01 12:00:00, 2015-12-01 13:00:00}|20522|
|{2015-12-01 13:00:00, 2015-12-01 14:00:00}|20556|
|{2015-12-01 14:00:00, 2015-12-01 15:00:00}|21712|
|{2015-12-01 15:00:00, 2015-12-01 16:00:00}|22016|
|{2015-12-01 16:00:00, 2015-12-01 17:00:00}|18034|
|{2015-12-01 17:00:00, 2015-12-01 18:00:00}|19719|
|{2015-12-01 18:00:00, 2015-12-01 19:00:00}|25563|
|{2015-12-01 19:00:00, 2015-12-01 20:00:00}|28178|
|{2015-12-01 20:00:00, 2015-12-01 21:00:00}|27449|
|{2015-12-01 21:00:00, 2015-12-01 22:00:00}|27072|
...
|{2015-12-01 23:00:00, 2015-12-02 00:00:00}|18806|
+----------------------------------------+-----+
```

Hình 11: Output

Which, now we extract the data frame, with column *temp* storing according directory name, then for each row we export into each named directory as csv files, all in *output_task_2*.



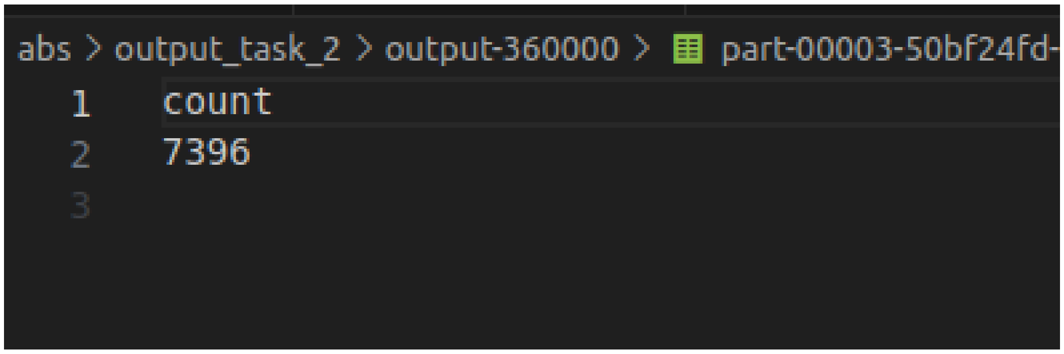
```python
# write into files
output_path_ex02 = 'file:///home/p4stwi2x/Desktop/abs/output_task_2'
hour_count = spark.sql('select * from counts order by window')\
        .withColumn("temp", (hour(col('window').start) + 1) * 360000)
for hour in hour_count.collect():
    timestamp_count = hour['temp']
    windows_data = hour['window']
    windows_count = hour['count']
    output_dir = os.path.join(output_path_ex02, f"output-{timestamp_count}")

    df_windows = spark.createDataFrame([(windows_count,)], ["count"])

    df_windows.write.mode("overwrite")\
            .format("csv")\
            .option("header", "true")\
            .save(output_dir)

    print(f"Timestamp {timestamp_count} has been exported to folder {output_dir}")
```

✓ 22.3s

```
Timestamp 360000 has been exported to folder file:///home/p4stwi2x/Desktop/abs/output_task_2/output-360000
Timestamp 720000 has been exported to folder file:///home/p4stwi2x/Desktop/abs/output_task_2/output-720000
Timestamp 1080000 has been exported to folder file:///home/p4stwi2x/Desktop/abs/output_task_2/output-1080000
Timestamp 1440000 has been exported to folder file:///home/p4stwi2x/Desktop/abs/output_task_2/output-1440000
Timestamp 1800000 has been exported to folder file:///home/p4stwi2x/Desktop/abs/output_task_2/output-1800000
Timestamp 2160000 has been exported to folder file:///home/p4stwi2x/Desktop/abs/output_task_2/output-2160000
Timestamp 2520000 has been exported to folder file:///home/p4stwi2x/Desktop/abs/output_task_2/output-2520000
Timestamp 2880000 has been exported to folder file:///home/p4stwi2x/Desktop/abs/output_task_2/output-2880000
```

Hình 12: Write into files

For example, the number of drop-offs at the first hour is 7396.

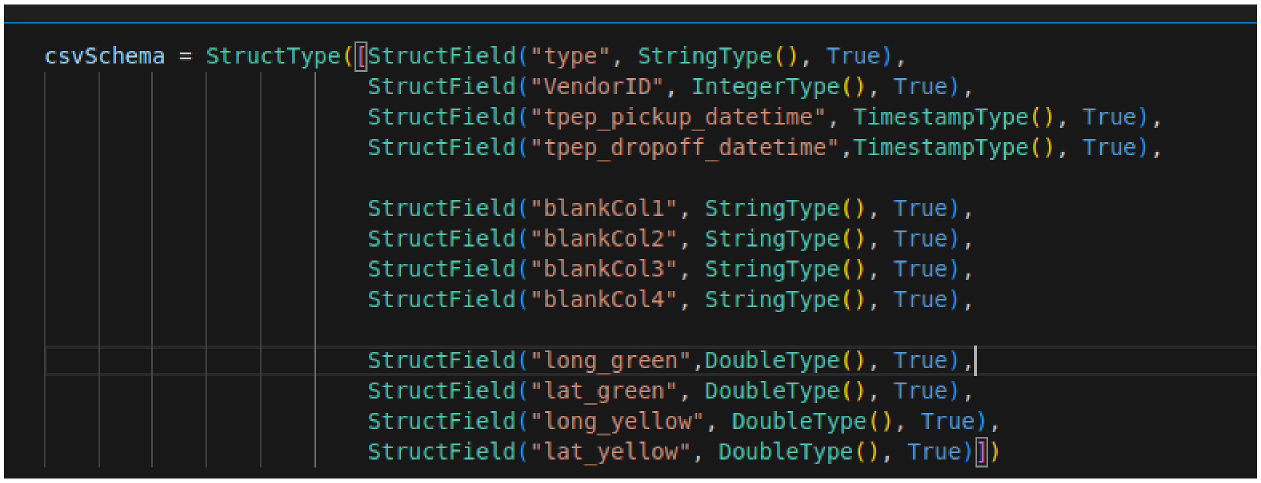


```
abs > output_task_2 > output-360000 > part-00003-50bf24fd-
count
7396
```

Hình 13: Write into files

3 Task 3: Count the number of taxi trips each hour that drop off at either the Goldman Sachs headquarters or the Citigroup headquarters.

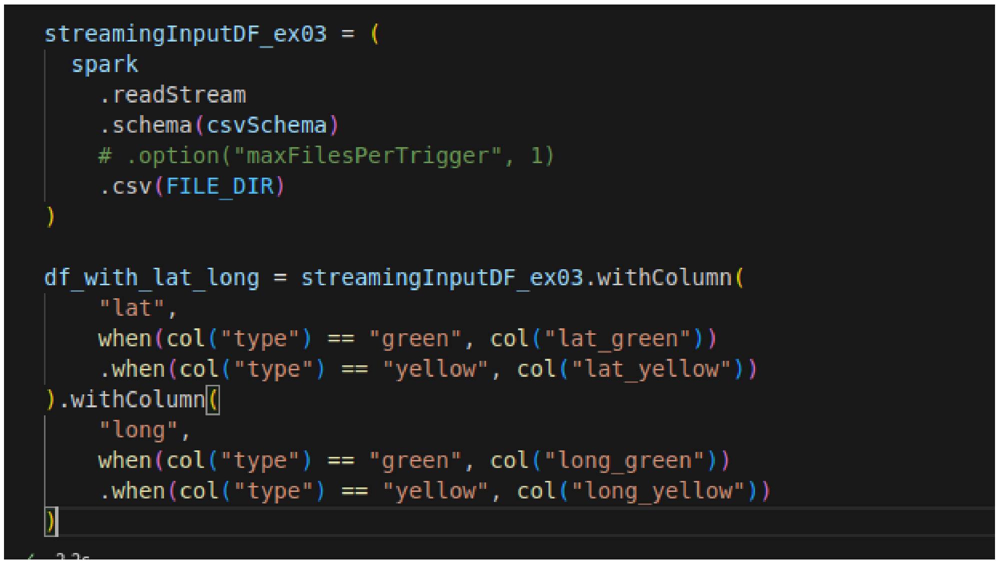
It shall be the similar implementations as task 2, except for that the data retrieved from the two types of taxis are in 4 different columns from 9th to 12th. So firstly we would input the first 12 columns.



```
csvSchema = StructType([StructField("type", StringType(), True),
                        StructField("VendorID", IntegerType(), True),
                        StructField("tpep_pickup_datetime", TimestampType(), True),
                        StructField("tpep_dropoff_datetime",TimestampType(), True),

                        StructField("blankCol1", StringType(), True),
                        StructField("blankCol2", StringType(), True),
                        StructField("blankCol3", StringType(), True),
                        StructField("blankCol4", StringType(), True),

                        StructField("long_green",DoubleType(), True),
                        StructField("lat_green", DoubleType(), True),
                        StructField("long_yellow", DoubleType(), True),
                        StructField("lat_yellow", DoubleType(), True)])
```

Hình 14: Preset schema

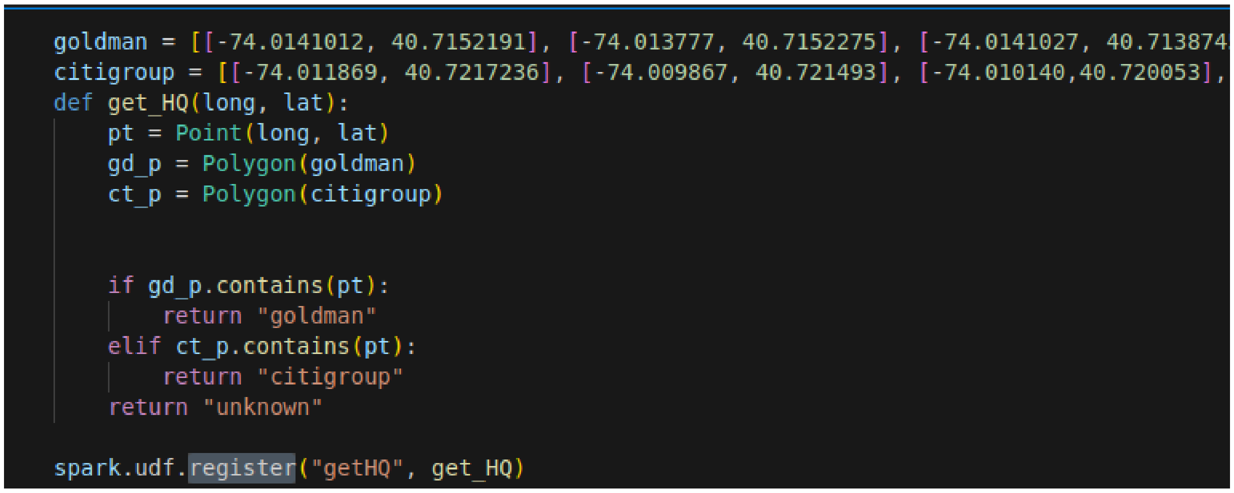
Then we create another two columns storing both longitudes and latitudes of both taxi types, for easier future working, using Spark function *when* in order to extract the values.



```
streamingInputDF_ex03 = (
  spark
    .readStream
    .schema(csvSchema)
    # .option("maxFilesPerTrigger", 1)
    .csv(FILE_DIR)
)

df_with_lat_long = streamingInputDF_ex03.withColumn(
    "lat",
    when(col("type") == "green", col("lat_green"))
    .when(col("type") == "yellow", col("lat_yellow"))
).withColumn(
    "long",
    when(col("type") == "green", col("long_green"))
    .when(col("type") == "yellow", col("long_yellow"))
)
```

Hình 15: Preset schema

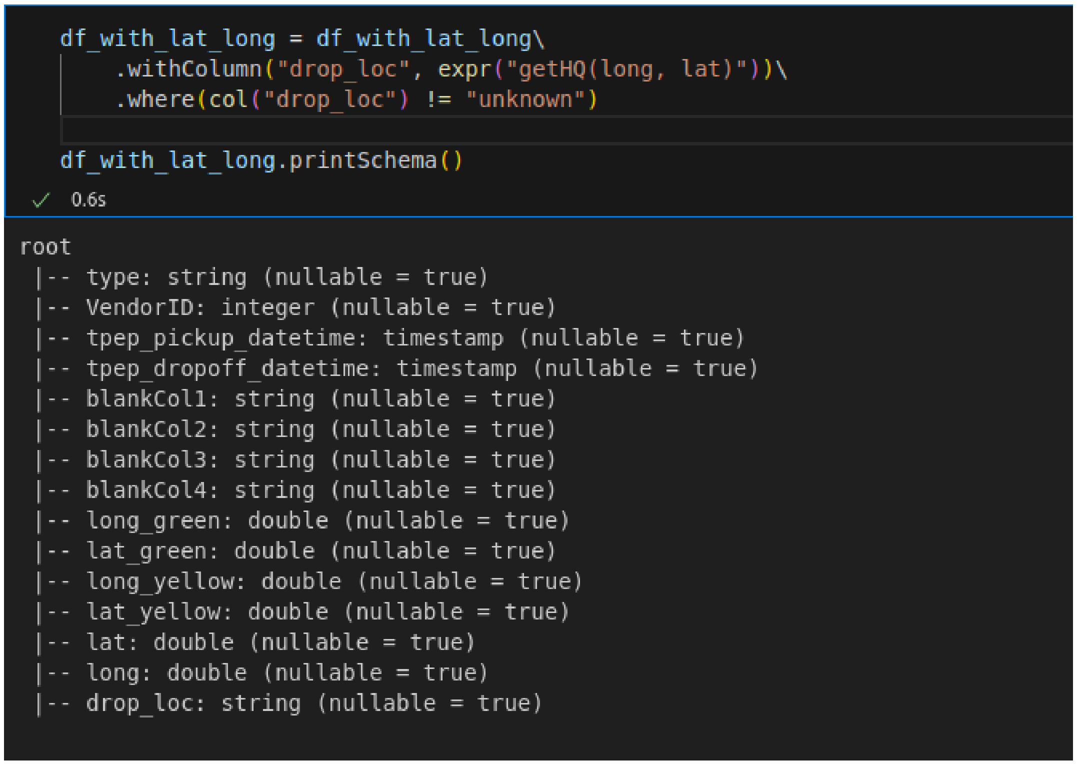
From then on, for simplicity, we implement a user-defined function *get_HQ()* that checks a point in which region and returns the region name, with assistance from a Python library *Shapely*, and we use *spark.udf.register()* for implementation into Spark.



```
goldman = [[-74.0141012, 40.7152191], [-74.013777, 40.7152275], [-74.0141027, 40.713874
citigroup = [[-74.011869, 40.7217236], [-74.009867, 40.721493], [-74.010140,40.720053],
def get_HQ(long, lat):
    pt = Point(long, lat)
    gd_p = Polygon(goldman)
    ct_p = Polygon(citigroup)


    if gd_p.contains(pt):
        return "goldman"
    elif ct_p.contains(pt):
        return "citigroup"
    return "unknown"

spark.udf.register("getHQ", get_HQ)
```

Hình 16: Preset schema

Then we call the usage of such function using *expr()* and store the region name assignments into a new column *drop_loc*.

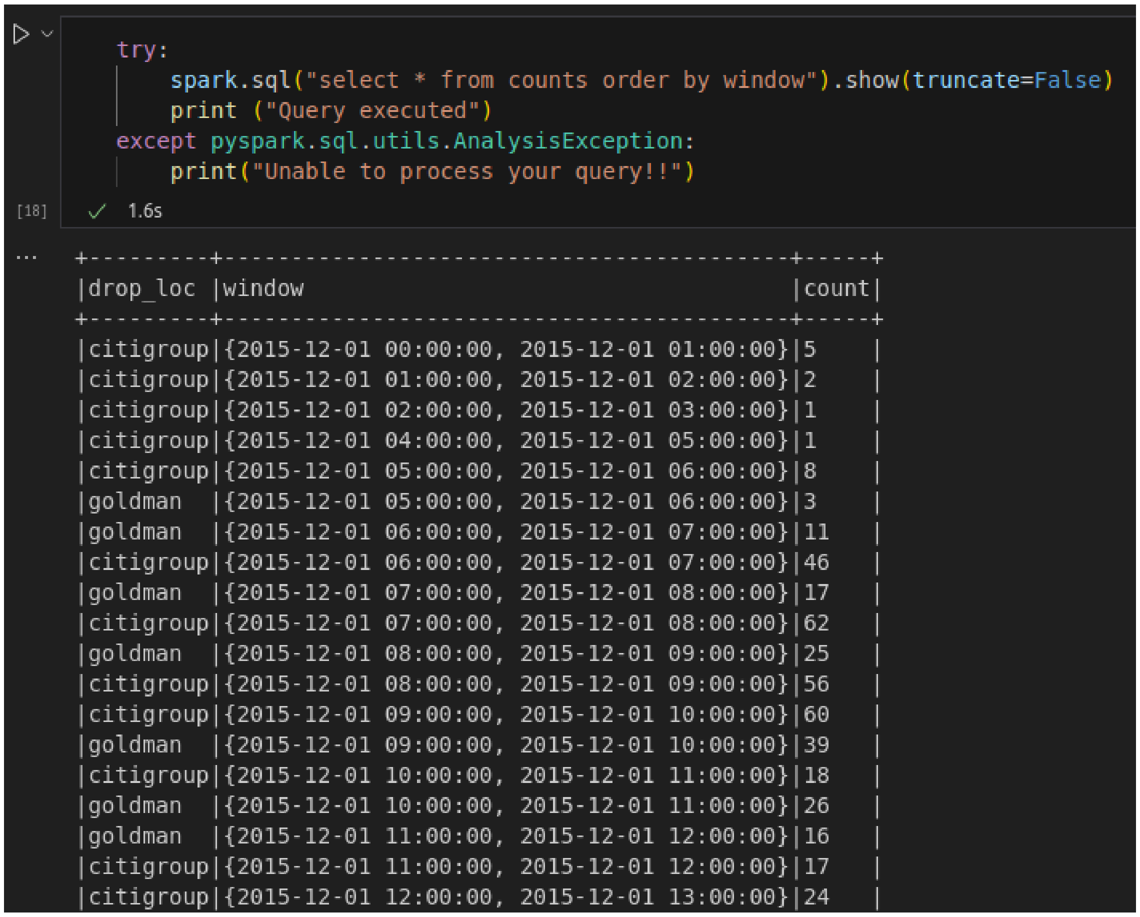


```
df_with_lat_long = df_with_lat_long\
    .withColumn("drop_loc", expr("getHQ(long, lat)"))\
    .where(col("drop_loc") != "unknown")

df_with_lat_long.printSchema()
✓ 0.6s
```

```
root
 |-- type: string (nullable = true)
 |-- VendorID: integer (nullable = true)
 |-- tpep_pickup_datetime: timestamp (nullable = true)
 |-- tpep_dropoff_datetime: timestamp (nullable = true)
 |-- blankCol1: string (nullable = true)
 |-- blankCol2: string (nullable = true)
 |-- blankCol3: string (nullable = true)
 |-- blankCol4: string (nullable = true)
 |-- long_green: double (nullable = true)
 |-- lat_green: double (nullable = true)
 |-- long_yellow: double (nullable = true)
 |-- lat_yellow: double (nullable = true)
 |-- lat: double (nullable = true)
 |-- long: double (nullable = true)
 |-- drop_loc: string (nullable = true)
```

Hình 17: Preset schema

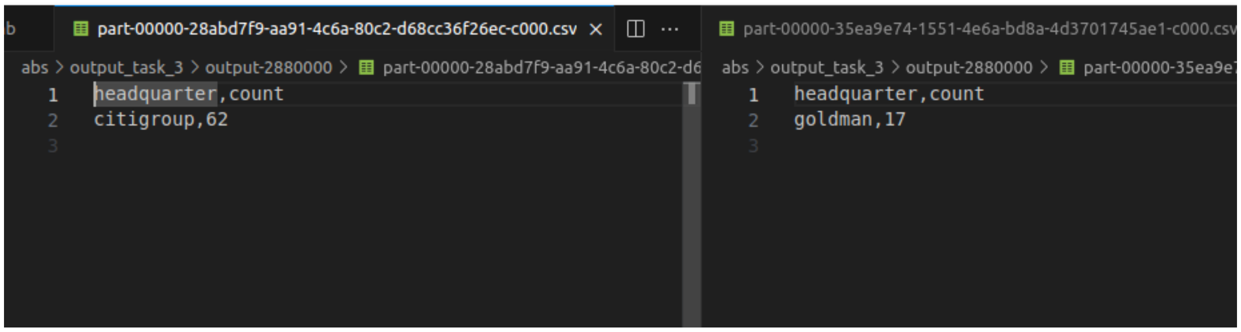
Hence, we get the results from querying:



```
try:
    spark.sql("select * from counts order by window").show(truncate=False)
    print ("Query executed")
except pyspark.sql.utils.AnalysisException:
    print("Unable to process your query!!")
[18]  ✓ 1.6s

+---------+------------------------------------------+-----+
|drop_loc |window                                    |count|
+---------+------------------------------------------+-----+
|citigroup|{2015-12-01 00:00:00, 2015-12-01 01:00:00}|5    |
|citigroup|{2015-12-01 01:00:00, 2015-12-01 02:00:00}|2    |
|citigroup|{2015-12-01 02:00:00, 2015-12-01 03:00:00}|1    |
|citigroup|{2015-12-01 04:00:00, 2015-12-01 05:00:00}|1    |
|citigroup|{2015-12-01 05:00:00, 2015-12-01 06:00:00}|8    |
|goldman  |{2015-12-01 05:00:00, 2015-12-01 06:00:00}|3    |
|goldman  |{2015-12-01 06:00:00, 2015-12-01 07:00:00}|11   |
|citigroup|{2015-12-01 06:00:00, 2015-12-01 07:00:00}|46   |
|goldman  |{2015-12-01 07:00:00, 2015-12-01 08:00:00}|17   |
|citigroup|{2015-12-01 07:00:00, 2015-12-01 08:00:00}|62   |
|goldman  |{2015-12-01 08:00:00, 2015-12-01 09:00:00}|25   |
|citigroup|{2015-12-01 08:00:00, 2015-12-01 09:00:00}|56   |
|citigroup|{2015-12-01 09:00:00, 2015-12-01 10:00:00}|60   |
|goldman  |{2015-12-01 09:00:00, 2015-12-01 10:00:00}|39   |
|citigroup|{2015-12-01 10:00:00, 2015-12-01 11:00:00}|18   |
|goldman  |{2015-12-01 10:00:00, 2015-12-01 11:00:00}|26   |
|goldman  |{2015-12-01 11:00:00, 2015-12-01 12:00:00}|16   |
|citigroup|{2015-12-01 11:00:00, 2015-12-01 12:00:00}|17   |
|citigroup|{2015-12-01 12:00:00, 2015-12-01 13:00:00}|24   |
```

Hình 18: Preset schema

And then we write the results into output folders, same as task 2, we sort the dataframe by time windows and write the results line-by-line. That means there is a good chance that some time windows have two csv files, representing both regions in counting.
For example, here is the result for time window hour 7 *output-2880000*:



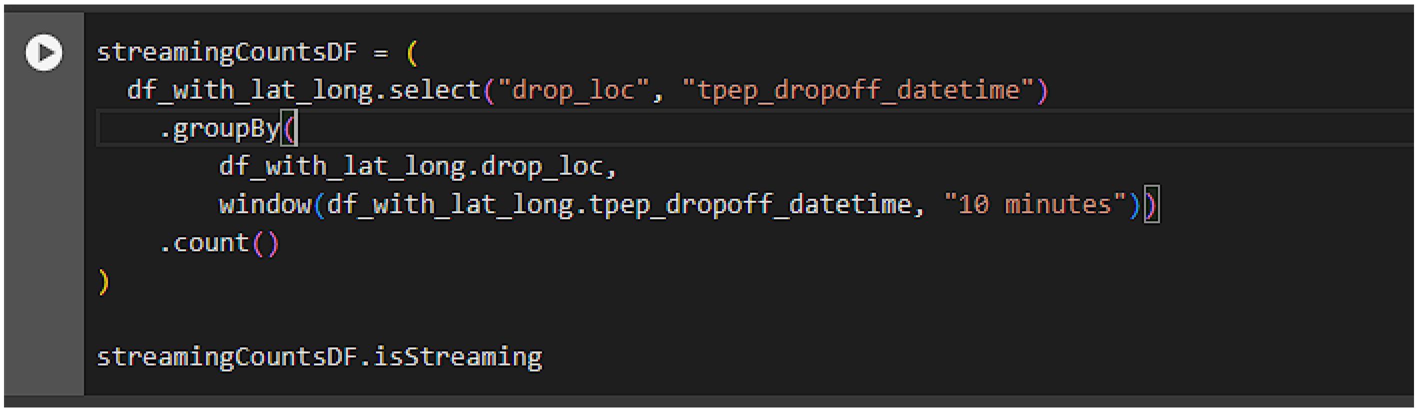
```
part-00000-28abd7f9-aa91-4c6a-80c2-d68cc36f26ec-c000.csv
abs > output_task_3 > output-2880000 > part-00000-28abd7f9-aa91-4c6a-80c2-d6
headquarter,count
citigroup,62
```

```
part-00000-35ea9e74-1551-4e6a-bd8a-4d3701745ae1-c000.csv
abs > output_task_3 > output-2880000 > part-00000-35ea9e7
headquarter,count
goldman,17
```

Hình 19: Preset schema

4 Task 4: Build a simple "trend detector" to find out when there are lots of arrivals at either Goldman Sachs or Citigroup headquarters

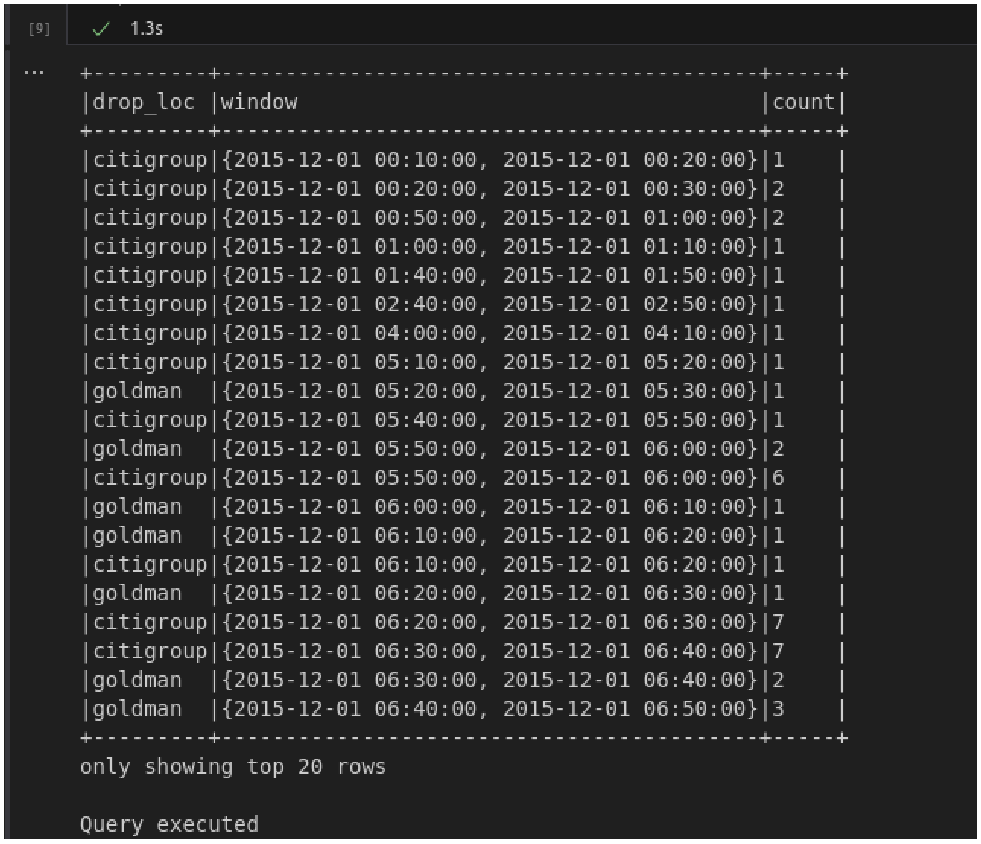
The query session is by grouping the dataframe by the dropoff column, using Spark function *window*, and grouping the dataframe by each time window of 10 minutes.



```
streamingCountsDF = (
  df_with_lat_long.select("drop_loc", "tpep_dropoff_datetime")
    .groupBy(
        df_with_lat_long.drop_loc,
        window(df_with_lat_long.tpep_dropoff_datetime, "10 minutes"))
    .count()
)

streamingCountsDF.isStreaming
```

Hình 20: Changing the windows to 10 minutes

Executing the query similar to Task 3, we'll have the following results:



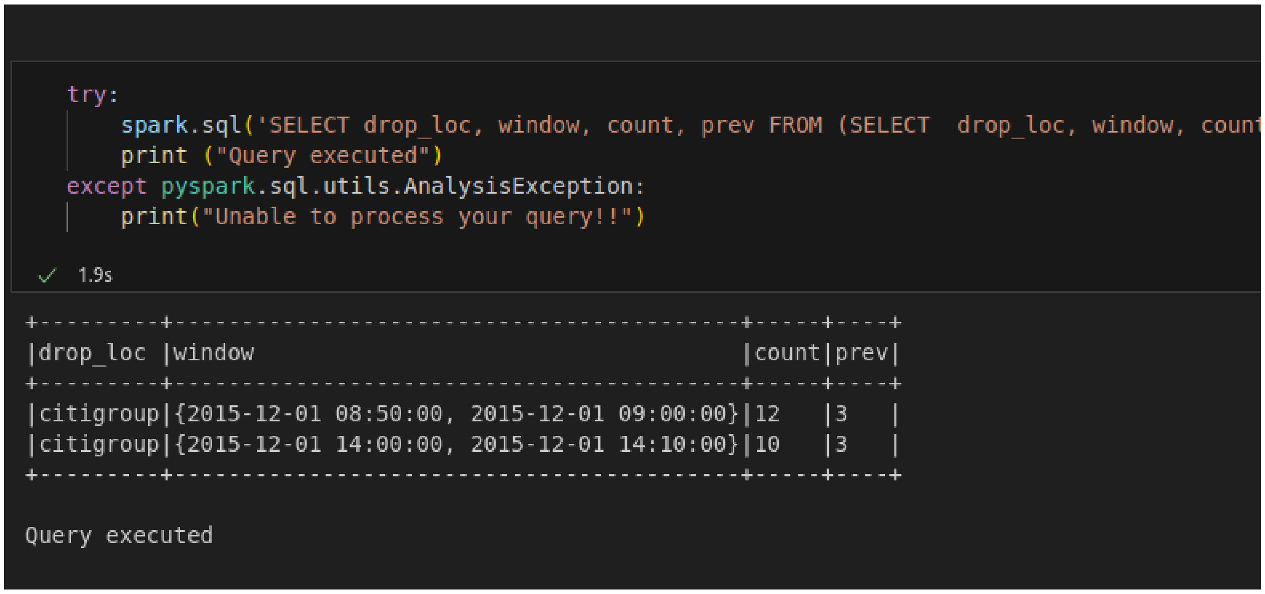
```
[9]  ✓ 1.3s
...
+---------+------------------------------------------+-----+
|drop_loc |window                                    |count|
+---------+------------------------------------------+-----+
|citigroup|{2015-12-01 00:10:00, 2015-12-01 00:20:00}|1    |
|citigroup|{2015-12-01 00:20:00, 2015-12-01 00:30:00}|2    |
|citigroup|{2015-12-01 00:50:00, 2015-12-01 01:00:00}|2    |
|citigroup|{2015-12-01 01:00:00, 2015-12-01 01:10:00}|1    |
|citigroup|{2015-12-01 01:40:00, 2015-12-01 01:50:00}|1    |
|citigroup|{2015-12-01 02:40:00, 2015-12-01 02:50:00}|1    |
|citigroup|{2015-12-01 04:00:00, 2015-12-01 04:10:00}|1    |
|citigroup|{2015-12-01 05:10:00, 2015-12-01 05:20:00}|1    |
|goldman  |{2015-12-01 05:20:00, 2015-12-01 05:30:00}|1    |
|citigroup|{2015-12-01 05:40:00, 2015-12-01 05:50:00}|1    |
|goldman  |{2015-12-01 05:50:00, 2015-12-01 06:00:00}|2    |
|citigroup|{2015-12-01 05:50:00, 2015-12-01 06:00:00}|6    |
|goldman  |{2015-12-01 06:00:00, 2015-12-01 06:10:00}|1    |
|goldman  |{2015-12-01 06:10:00, 2015-12-01 06:20:00}|1    |
|citigroup|{2015-12-01 06:10:00, 2015-12-01 06:20:00}|1    |
|goldman  |{2015-12-01 06:20:00, 2015-12-01 06:30:00}|1    |
|citigroup|{2015-12-01 06:20:00, 2015-12-01 06:30:00}|7    |
|citigroup|{2015-12-01 06:30:00, 2015-12-01 06:40:00}|7    |
|goldman  |{2015-12-01 06:30:00, 2015-12-01 06:40:00}|2    |
|goldman  |{2015-12-01 06:40:00, 2015-12-01 06:50:00}|3    |
+---------+------------------------------------------+-----+
only showing top 20 rows

Query executed
```

Hình 21: Query

In order to reduce "spurious" detections, we want to make sure the detector only "trips" if there are ten or more arrivals in the current interval. That is, if there are two arrivals in the last ten minute interval and four arrivals in the current ten-minute interval, that's not particularly interesting (although the number of arrivals has indeed doubled), so we want to suppress such results).
In short, we want to find the timestamp that has the count that is larger than 10 and double than that of the previous timestamp, of the same headquarters.
Here is the query that we used in spark.sql:

```
try:
    spark.sql('SELECT drop_loc, window, count, prev FROM (SELECT  drop_loc, window,
    count, lag(count, 1, NULL) OVER (ORDER BY drop_loc, window)
    AS prev FROM counts)
    WHERE count >= 10 and count >= 2 * prev AND prev IS NOT NULL')
    .show(100,truncate=False)
    print ("Query executed")
except pyspark.sql.utils.AnalysisException:
    print("Unable to process your query!!")
```

Hence by that, we are able to extract the static in-memory query result.



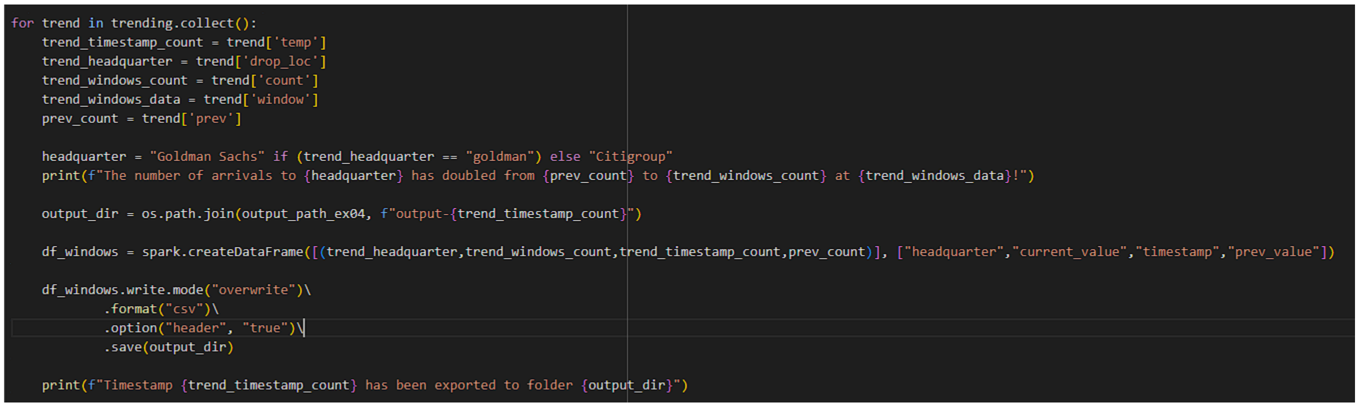
```
try:
    spark.sql('SELECT drop_loc, window, count, prev FROM (SELECT  drop_loc, window, count
    print ("Query executed")
except pyspark.sql.utils.AnalysisException:
    print("Unable to process your query!!")
```

✓ 1.9s

```
+---------+------------------------------------------+-----+----+
|drop_loc |window                                    |count|prev|
+---------+------------------------------------------+-----+----+
|citigroup|{2015-12-01 08:50:00, 2015-12-01 09:00:00}|12   |3   |
|citigroup|{2015-12-01 14:00:00, 2015-12-01 14:10:00}|10   |3   |
+---------+------------------------------------------+-----+----+

Query executed
```

Hình 22: Extract query result as static dataframe

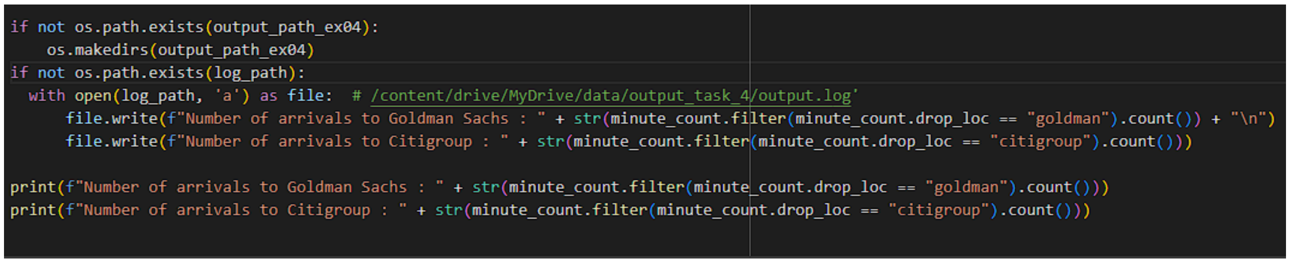
Which, now we extract the data frame, with column *temp* storing according directory name, then for each row we export into each named directory as csv files, all in *output_task_4*.



```python
for trend in trending.collect():
    trend_timestamp_count = trend['temp']
    trend_headquarter = trend['drop_loc']
    trend_windows_count = trend['count']
    trend_windows_data = trend['window']
    prev_count = trend['prev']

    headquarter = "Goldman Sachs" if (trend_headquarter == "goldman") else "Citigroup"
    print(f"The number of arrivals to {headquarter} has doubled from {prev_count} to {trend_windows_count} at {trend_windows_data}!")

    output_dir = os.path.join(output_path_ex04, f"output-{trend_timestamp_count}")

    df_windows = spark.createDataFrame([(trend_headquarter,trend_windows_count,trend_timestamp_count,prev_count)], ["headquarter","current_value","timestamp","prev_value"])

    df_windows.write.mode("overwrite")\
            .format("csv")\
            .option("header", "true")\
            .save(output_dir)

    print(f"Timestamp {trend_timestamp_count} has been exported to folder {output_dir}")
```

Hình 23: Write into files

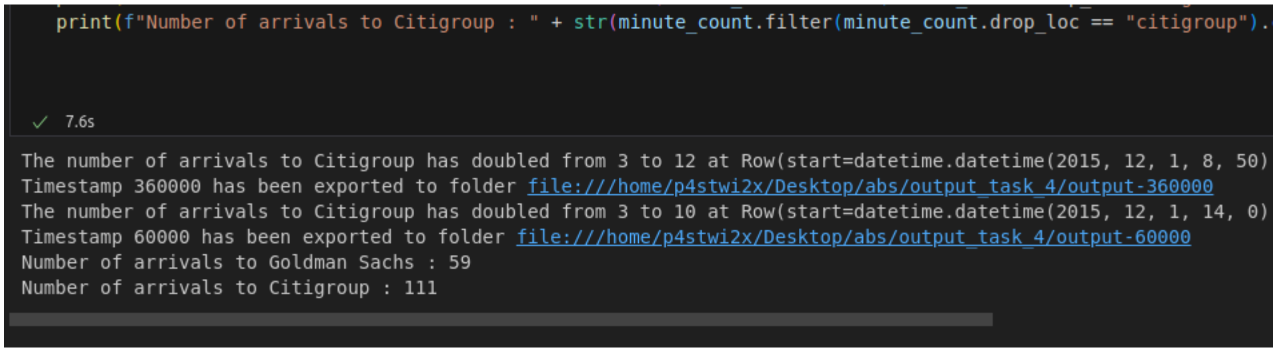
Printing the results



```python
if not os.path.exists(output_path_ex04):
    os.makedirs(output_path_ex04)
if not os.path.exists(log_path):
  with open(log_path, 'a') as file:  # /content/drive/MyDrive/data/output_task_4/output.log'
      file.write(f"Number of arrivals to Goldman Sachs : " + str(minute_count.filter(minute_count.drop_loc == "goldman").count()) + "\n")
      file.write(f"Number of arrivals to Citigroup : " + str(minute_count.filter(minute_count.drop_loc == "citigroup").count()))

print(f"Number of arrivals to Goldman Sachs : " + str(minute_count.filter(minute_count.drop_loc == "goldman").count()))
print(f"Number of arrivals to Citigroup : " + str(minute_count.filter(minute_count.drop_loc == "citigroup").count()))
```

Hình 24: Write into files at output task 4

Here are the final results of task 4



```
    print(f"Number of arrivals to Citigroup : " + str(minute_count.filter(minute_count.drop_loc == "citigroup").

  ✓ 7.6s

The number of arrivals to Citigroup has doubled from 3 to 12 at Row(start=datetime.datetime(2015, 12, 1, 8, 50)
Timestamp 360000 has been exported to folder file:///home/p4stwi2x/Desktop/abs/output_task_4/output-360000
The number of arrivals to Citigroup has doubled from 3 to 10 at Row(start=datetime.datetime(2015, 12, 1, 14, 0)
Timestamp 60000 has been exported to folder file:///home/p4stwi2x/Desktop/abs/output_task_4/output-60000
Number of arrivals to Goldman Sachs : 59
Number of arrivals to Citigroup : 111
```

Hình 25: Results of output task 4

5 Self-assessment of completeness

We believe we have accomplished all the tasks and corresponding requirements needed to be done, as well as having done providing our instructions and approaches, step-by-step, along this report.

Generally we did not have many problems surfaced during the work on these lab exercises, even though it took longer than we thought in order to finish researching the necessary functionalities, debugging and finding solutions.

We have referenced through similar exercises and solutions from other universities and people, especially [3], [4], [2]. Also we have referenced through Apache Spark guide on Structured Streaming [5] and Shapely library [1] for assistance in mathematical computations.

Tài liệu

[1] Sean Gilles. Shapely 2.0.4: Python package for set-theoretic analysis and manipulation of planar features. *https://shapely.readthedocs.io/en/stable/index.html*, 2024.

[2] Armen Jeddi. Data-intensive-distributed-computing: Big data processing frameworks (spark, hadoop), programming in java and scala, and numerous practical and well-known algorithms. *https://github.com/ArmenJeddi/Data-Intensive-Distributed-Computing/tree/master*, 2020.

[3] Jimmy Lin. bespin: Reference implementations of data-intensive algorithms in mapreduce and spark. *https://github.com/lintool/bespin/tree/master*, 2018.

[4] University of Waterloo. Cs 451/651 431/631 data-intensive distribute computing (winter 2018). *https://github.com/lintool/bigdata-2018w/tree/master*, 2018.

[5] Apache Spark. Structured streaming programming guide. *https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html*.