

VNUHCM - UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY



Report Lab 03

Lab 03 - Spark Streaming

Course name: Introduction to Big Data

Students:

Nguyen Thi Minh Minh - 21127528
Nguyen Van Dang Huynh - 21127063
Nguyen Tuan Kiet - 21127089
Pham Phu Toan - 21127183

Instructors:

Dr. Nguyen Ngoc Thao
Ta. Do Trong Le
Ta. Bui Huynh Trung Nam

22nd May 2024

Contents

Tasks on Lab 03 2

1 Requirement of Lab 03 2

2 Report on tasks done: 3

2.1 Task 1 3

2.1.1 Idea [1] [3] 3

2.1.2 Code review 4

2.1.3 Result: 5

2.2 Task 2 5

2.2.1 Ideas [2] 5

2.2.2 Code review 5

2.2.3 Result 9

2.3 Task 3 10

2.3.1 Ideas [2] 10

2.3.2 Code review 10

2.3.3 Result 12

2.4 Task 4 15

2.4.1 Ideas [3] 15

2.4.2 Code review 16

2.4.3 Result 16

References 18

Tasks on Lab 03

No.	Task title	Percent of completion
1	Task 1	100%
2	Task 2	100%
3	Task 3	100%
4	Task 4	100%

1 Requirement of Lab 03

Dataset: `taxi-data.zip`

Yellow taxi schema:

Data Fields:
 type, VendorID, tpep_pickup_datetime, tpep_dropoff_datetime, passenger_count, trip_distance, pickup_longitude, pickup_latitude, RatecodeID, store_and_fwd_flag, dropoff_longitude, dropoff_latitude, payment_type, fare_amount, extra, mta_tax, tip_amount, tolls_amount, improvement_surcharge, total_amount.

Green taxi schema:

Data Fields:
 type, VendorID, lpep_pickup_datetime, lpep_dropoff_datetime, Store_and_fwd_flag, RatecodeID, Pickup_longitude, Pickup_latitude, Dropoff_longitude, Dropoff_latitude, Passenger_count, Trip_distance, Fare_amount, Extra, Mta_tax, Tip_amount, Tolls_amount, Ehail_fee, improvement_surcharge, Total_amount, Payment_type, Trip_type.

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

Task 2: Let's start with an `EventCount(.py/.scala/.java)` query that aggregates the number of trips by drop-off datetime for each hour. The output of each aggregation window is stored in a directory named `output-xxxxx` where `xxxxx` is the timestamp.

Task 3: Create a query called `RegionEventCount(.py/.scala/.java)` that counts the number of taxi trips each hour that drop off at either the *Goldman Sachs* headquarters or the *Citigroup* headquarters.

Task 4: Let's build a simple "trend detector" to find out when there are lots of arrivals at either *Goldman Sachs* or *Citigroup* headquarters, defined in terms of the bounding boxes, exactly as above. We'll consider intervals of 10 minutes. The trend detector should "go off" when there are at least twice as many arrivals in the current interval as there are 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. 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.

2 Report on tasks done:

2.1 Task 1

2.1.1 Idea [1] [3]

This report details the approach used to simulate a data stream by processing data sourced from files. The provided code demonstrates this concept using Apache Spark and its Structured Streaming capabilities.

Data Source and Schema

The code assumes a data directory (`inputPath`) containing CSV files with taxi trip data. Two types of taxi trips are expected: yellow and green. Each type has its own schema defined using `StructType`. The yellow trip schema includes fields like `tpep_pickup_datetime`, `passenger_count`, and `fare_amount`, while the green trip schema has corresponding fields like `lpep_pickup_datetime` and `trip_distance`.

A `default_schema` (`default_schema`) is used for initial data ingestion due to the presence of a header row (`header="false"`). This schema consists of generic string type columns (`_cx`).

Stream Processing

Data Ingestion

- The code reads streaming data from the CSV files using `spark.readStream.format(csv)`.
- The `default_schema` is applied to handle the initial data without a header.

Data Filtering and Schema Enforcement

- The default `DataFrame` (`default_df`) is filtered to separate yellow (`_c0` equals `yellow`) and green (`_c0` equals `green`) taxi trips using separate `DataFrames` (`yellow_trips` and `green_trips`).
- Unnecessary columns from the default schema are dropped using `drop`.
- Each filtered `DataFrame` is transformed to its respective schema (`schema_yellow` or `schema_green`) using `toDF`.

Data Type Conversion

- Specific columns within the yellow and green `DataFrames` are cast to their appropriate data types (e.g., `TimestampType`, `DoubleType`) using `withColumn`. This ensures proper data representation.

NOTE: The code snippet on my Task 1 focuses on data preparation and schema enforcement. It doesn't showcase the actual streaming processing logic. However, the processed DataFrames (`yellow_df` and `green_df`) are used further for downstream operations like windowing, aggregation, or visualization in a streaming fashion.

2.1.2 Code review

Spark Session Initialization This block creates a Spark Session, which is the entry point for interacting with Spark. Key configurations include:

- `master("local")`: Specifies running Spark in local mode (single machine). For distributed processing, you'd adjust this configuration.
- `appName("Task 1 - Lab 03")`: Sets the application name for identification within the Spark cluster.
- `config("spark.some.config.option", "some-value")`: While commented out, this line demonstrates how you can set custom Spark configurations if needed.
- `.getOrCreate()`: Creates a new Spark Session or retrieves an existing one if it already exists.

Schema Definition This section defines the schemas for yellow and green taxi trips using `StructType`. Each schema consists of multiple `StructField` objects, each representing a column in the schema. For example, the yellow taxi schema includes fields like `tpep_pickup_datetime`, `passenger_count`, and `fare_amount`.

Default Schema and DataFrame The default schema is defined to handle the initial data ingestion, which lacks a header row. It consists of generic string type columns (`_cx`). The code reads streaming data from the CSV files using this default schema to create the initial DataFrame (`default_df`).

Data Filtering and Schema Enforcement

This section separates the data for yellow and green trips:

- `default_df.filter(col("_c0") == yellow)`: Filters the `default_df` to keep rows where the first column (`_c0`) equals `yellow`.
- `default_df.filter(col("_c0") == green)`: Filters the `default_df` to keep rows where the first column (`_c0`) equals `green`.
- `drop("_c20", "_c21")`: Drops unnecessary columns (likely specific to the data) from the filtered DataFrames.

- `toDF(*schema_yellow.names)` and `toDF(*schema_green.names)`: Transforms the filtered DataFrames to their respective schemas (`schema_yellow` and `schema_green`). This ensures the data conforms to the expected structure.

Data Type Conversion These lines convert specific columns within the yellow and green DataFrames to their appropriate data types. This is crucial for proper data processing and analysis.

2.1.3 Result:

Successfully implemented the data preparation and schema enforcement steps for simulating a data stream using Apache Spark. The code demonstrates how to read streaming data from CSV files, filter the data based on specific criteria, enforce schemas, and convert data types for further processing. This foundational work sets the stage for more advanced streaming operations in subsequent tasks.

2.2 Task 2

2.2.1 Ideas [2]

- Count the trips
 - To calculate the number of trips by drop-off datetime for each hour, we simply add another column with the value = the hour of that datetime + 1.
(Ex: 2015-12-01 00:01:29 => 0 + 1 = 1)
 - Then we can groupBy the above column and count.
- Aggregation problem:
 - How do we keep track of the previous counts of each hour to the aggregation?
 - We could achieve that by having a global variable to store all the previous values of the hours.

2.2.2 Code review

- First, start a Spark Session, and initialize the schema for the data.
- We notice that there are 2 schemas for 2 types of taxi. Despite that, we could use any of those 2 schemas in this situation because we only work with the `dropoff_datetime` column and it appears at the same location in both of the schemas.
- After having the schema defined, we can create a streaming DataFrame.

```
# Create a SparkSession
spark = SparkSession.builder \
    .appName("EventCount") \
    .getOrCreate()

# set the path to the data
data_path = "taxi-data"

# Get active streaming queries
active_queries = spark.streams.active

# Stop each active streaming query
for query in active_queries:
    query.stop()

# Define the directory path
directory = "output"

# Create the directory if it does not exist
if os.path.exists(directory):
    shutil.rmtree(directory)
os.makedirs(directory)
```

Figure 1: Task 2 code snippet

```
# Define the schema
schema = StructType([
    StructField("type", StringType()),
    StructField("VendorID", IntegerType()),
    StructField("pickup_datetime", TimestampType()),
    StructField("dropoff_datetime", TimestampType()),
    StructField("passenger_count", IntegerType()),
    StructField("trip_distance", DoubleType()),
    StructField("pickup_longitude", DoubleType()),
    StructField("pickup_latitude", DoubleType()),
    StructField("RatecodeID", IntegerType()),
    StructField("store_and_fwd_flag", StringType()),
    StructField("dropoff_longitude", DoubleType()),
    StructField("dropoff_latitude", DoubleType()),
    StructField("payment_type", IntegerType()),
    StructField("fare_amount", DoubleType()),
    StructField("extra", DoubleType()),
    StructField("mta_tax", DoubleType()),
    StructField("tip_amount", DoubleType()),
    StructField("tolls_amount", DoubleType()),
    StructField("improvement_surcharge", DoubleType()),
    StructField("total_amount", DoubleType())
])

# Define the streaming DataFrame
streaming_df = (
    spark.readStream
    .format("csv")
    .option("header", "false")
    .schema(schema)
    .load(data_path)
)
```

Figure 2: Task 2 code snippet

- To achieve the aggregated results mentioned above in the Ideas, we initialize a global variable to store the results for each hour up until now.
- After that, we can go on and define a function to handle and process the streaming data.
- The name of the output files is created as follows: (hour + 1) * 360000. Where 360000 is the number of milliseconds in an hour.


```
def process_streaming_data(df, epoch_id):
    global hour_counts

    # Calculate counts for each hour in the current batch
    batch_hour_counts = df \
        .withColumn("hour", f.hour(df["dropoff_datetime"].cast("timestamp"))) \
        .groupBy("hour") \
        .count() \
        .collect()

    # Update running totals with counts from the current batch
    for row in batch_hour_counts:
        hour = row["hour"]
        count = row["count"]
        if hour in hour_counts:
            hour_counts[hour] += count
        else:
            hour_counts[hour] = count

    # Write the updated counts to the corresponding output directories
    for hour, count in hour_counts.items():
        output_directory = f"output/output-{{(hour + 1) * 3600000}}"
        row_df = spark.createDataFrame([(hour, count)], ["hour", "count"])
        # Repartition the DataFrame before writing
        row_df = row_df.repartition(1)
        row_df.write.csv(output_directory, mode="overwrite", header="true")

    # Checkpoint location
    checkpoint_location = "checkpoint"
```

Figure 3: Task 2 code snippet

- Finally, start a streaming query.

```
# Checkpoint location
checkpoint_location = "checkpoint"

# Start the new streaming query
query = streaming_df \
    .writeStream \
    .outputMode("update") \
    .foreachBatch(process_streaming_data) \
    .option("checkpointLocation", checkpoint_location) # Add checkpoint location

query.start().awaitTermination()
```

Figure 4: Task 2 code snippet

- What is checkpoint and why do we need it? In Spark Streaming, a checkpoint is like a save point. It allows the program to pick up where it left off if it crashes or needs to be restarted, avoiding reprocessing everything from the beginning. This improves fault tolerance and helps when scaling or resuming interrupted jobs.

2.2.3 Result

- The output is stored in directories following the format described in the assignment (24 directories, one for each hour).

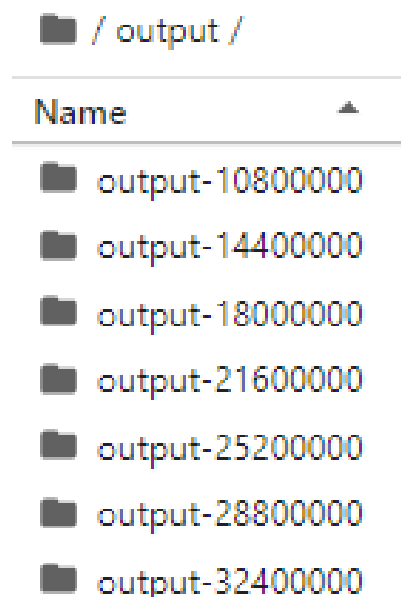
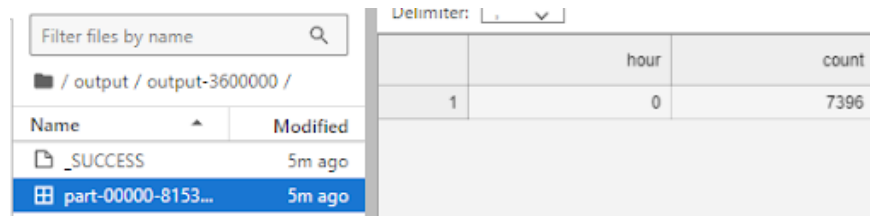


Figure 5: Task 2 output

- Each directory contains 2 files:
 - `_SUCCESS`: indicates the calculation was successful.
 - `part-00000`: contains the aggregated results.



	hour	count
1	0	7396

Figure 6: Task 2 output files in output directory

2.3 Task 3

2.3.1 Ideas [2]

- The reading process and writing to file process is exactly like task 1 and task 2.
- What we need to do is add a function to filter rows that have `dropoff_location` inside *Goldman Sachs* or *Citigroup*.
- Since *Goldman Sachs* and *Citigroup* each of them includes 4 points so we can draw a polygon with these 4 points.
- Then using `Point` and `Polygon` from `shapely.geometry` we can check whether the `dropoff_location` is within *Goldman Sachs* region or *Citigroup* region by using the `longitude` and `latitude` of the `dropoff_location`.
- The filter process of yellow and green taxis has to be done in separate data frames since the position of `dropoff_longitude` and `dropoff_latitude` of each data frame is different.

2.3.2 Code review

- First by using the longitudes and latitudes of *Goldman Sachs* and *Citigroup* we can create 2 polygons for each of them using the `Polygon` function from `shapely geometry`.
- Then we create a function `inside_polygon()` with 3 arguments:
 - `longitude`: Longitude of the point to be checked.
 - `latitude`: Latitude of the point to be checked.
 - `polygon`: *Goldman Sachs* polygon or *Citigroup* polygon.
- The `inside_polygon()` function uses longitude and latitude to create a point and we then use this `Point` to check whether it is in the given polygon.

- The `get_dropoff_location()` simply returns the area that the `dropoff_location` is in.

```
# Create polygons from coordinates
goldman_polygon = Polygon(goldman_coords)
citigroup_polygon = Polygon(citigroup_coords)

# Define a UDF to check if a point is inside the Goldman Sachs or Citigroup polygon
def inside_polygon(longitude, latitude, polygon):
    point = Point(longitude, latitude)
    return point.within(polygon)

# Register UDFs for both locations
inside_goldman = udf(lambda lon, lat: inside_polygon(lon, lat, goldman_polygon), BooleanType())
inside_citigroup = udf(lambda lon, lat: inside_polygon(lon, lat, citigroup_polygon), BooleanType())

# Adding a column to indicate the drop-off location
def get_dropoff_location(longitude, latitude):
    point = Point(longitude, latitude)
    if point.within(goldman_polygon):
        return "Goldman Sachs"
    elif point.within(citigroup_polygon):
        return "Citigroup"
    else:
        return None

get_dropoff_location_udf = udf(get_dropoff_location, StringType())
```

Figure 7: Task 3 code snippet

- Then we will filter the yellow and green df using the two functions that we have created.

```
# Filter dataframe for trips that end within the specified bounding boxes
yellow_trips = yellow_df.filter(
    inside_goldman(col("dropoff_longitude"), col("dropoff_latitude")) |
    inside_citigroup(col("dropoff_longitude"), col("dropoff_latitude"))
)

yellow_trips = yellow_trips.withColumn("dropoff_location", get_dropoff_location_udf(col("dropoff_longitude"), col("dropoff_latitude")))

green_trips = green_df.filter(
    inside_goldman(col("dropoff_longitude"), col("dropoff_latitude")) |
    inside_citigroup(col("dropoff_longitude"), col("dropoff_latitude"))
)

green_trips = green_trips.withColumn("dropoff_location", get_dropoff_location_udf(col("dropoff_longitude"), col("dropoff_latitude")))
```

Figure 8: Task 3 code snippet

- Then we count the number of trips of each df using group by the `textttdropoff_location` and the time. Here we use window function and set the time as 1 hour.

```
# Assuming dropoff_datetime is the column based on which windowed aggregation is done
yellow_agg_df = (
    yellow_trips
        .groupBy(
            yellow_trips.dropoff_location,
            window(col("tpep_dropoff_datetime"), "1 hour"),
        )
        .count()
        .orderBy("window")
)
```

```
# Assuming dropoff_datetime is the column based on which windowed aggregation is done
green_agg_df = (
    green_trips
        .groupBy(
            green_trips.dropoff_location,
            window(col("lpep_dropoff_datetime"), "1 hour"),
        )
        .count()
        .orderBy("window")
)
```

Figure 9: Task 3 code snippet

- Finally, we union 2 df and using group by to count for the sum of two types of taxi.
- The writing to file process is similar to task 2.

```
# Cast the 'count' column to integer if necessary
yellow_agg_df = yellow_agg_df.withColumn("count", when(col("count").cast("integer").isNull(), 0).otherwise(col("count").cast("integer")))
green_agg_df = green_agg_df.withColumn("count", when(col("count").cast("integer").isNull(), 0).otherwise(col("count").cast("integer")))

combined_df = yellow_agg_df.union(green_agg_df).orderBy("window")
final_agg_df = combined_df.groupBy("dropoff_location", "window").sum("count").orderBy("window")
final_agg_df = final_agg_df.withColumnRenamed("sum(count)", "count")

## Format the path string to include the timestamp as milliseconds
final_agg_df = final_agg_df.withColumn("path", format_string("output-%d", (f.hour(col("window.start")) + 1) * 360000))
```

Figure 10: Task 3 code snippet

2.3.3 Result

- The number of output folders is 23 so we are missing 1 folder. This is because at this hour, there are no taxis in the Goldman Sachs or Citigroup area.




















Name	Date modified	Type	Size
 .ipynb_checkpoints	5/21/2024 7:17 PM	File folder	
 output-360000	5/20/2024 10:48 PM	File folder	
 output-720000	5/20/2024 10:48 PM	File folder	
 output-1080000	5/20/2024 10:48 PM	File folder	
 output-1800000	5/20/2024 10:48 PM	File folder	
 output-2160000	5/20/2024 10:48 PM	File folder	
 output-2520000	5/20/2024 10:48 PM	File folder	
 output-2880000	5/20/2024 10:48 PM	File folder	
 output-3240000	5/20/2024 10:48 PM	File folder	
 output-3600000	5/20/2024 10:48 PM	File folder	
 output-3960000	5/20/2024 10:48 PM	File folder	
 output-4320000	5/20/2024 10:48 PM	File folder	
 output-4680000	5/20/2024 10:48 PM	File folder	
 output-5040000	5/20/2024 10:48 PM	File folder	
 output-5400000	5/20/2024 10:48 PM	File folder	
 output-5760000	5/20/2024 10:48 PM	File folder	
 output-6120000	5/20/2024 10:48 PM	File folder	
 output-6480000	5/20/2024 10:48 PM	File folder	
 output-6840000	5/20/2024 10:48 PM	File folder	

Figure 11: Task 3 output

- Some sample output files:

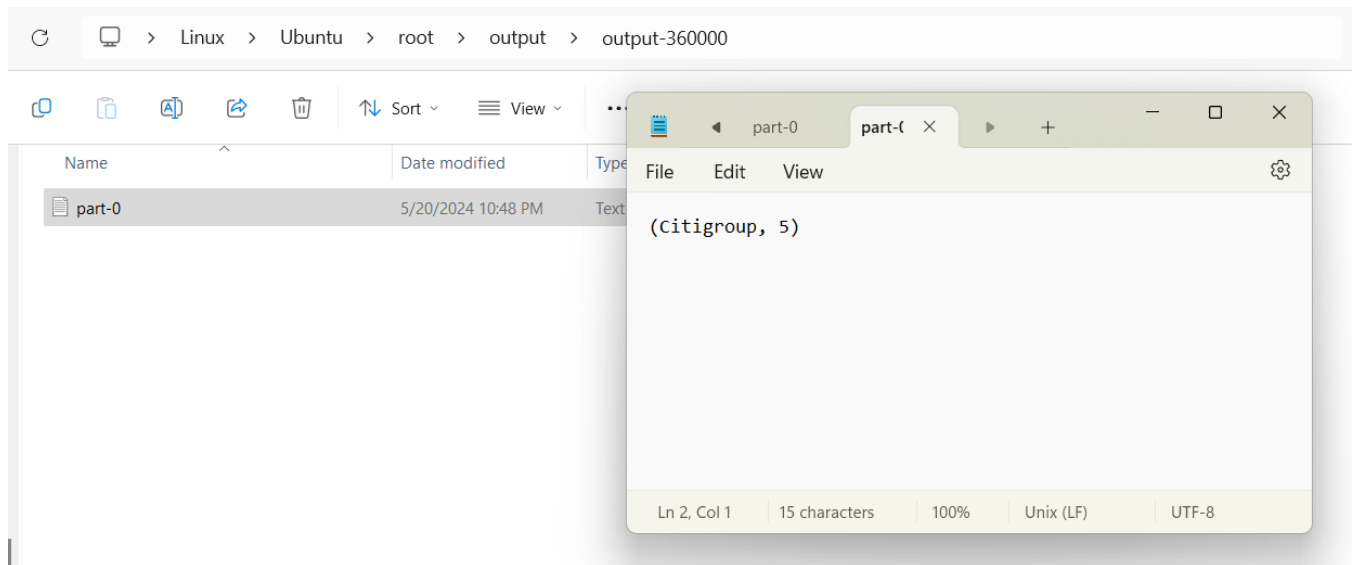


Figure 12: Task 3 output

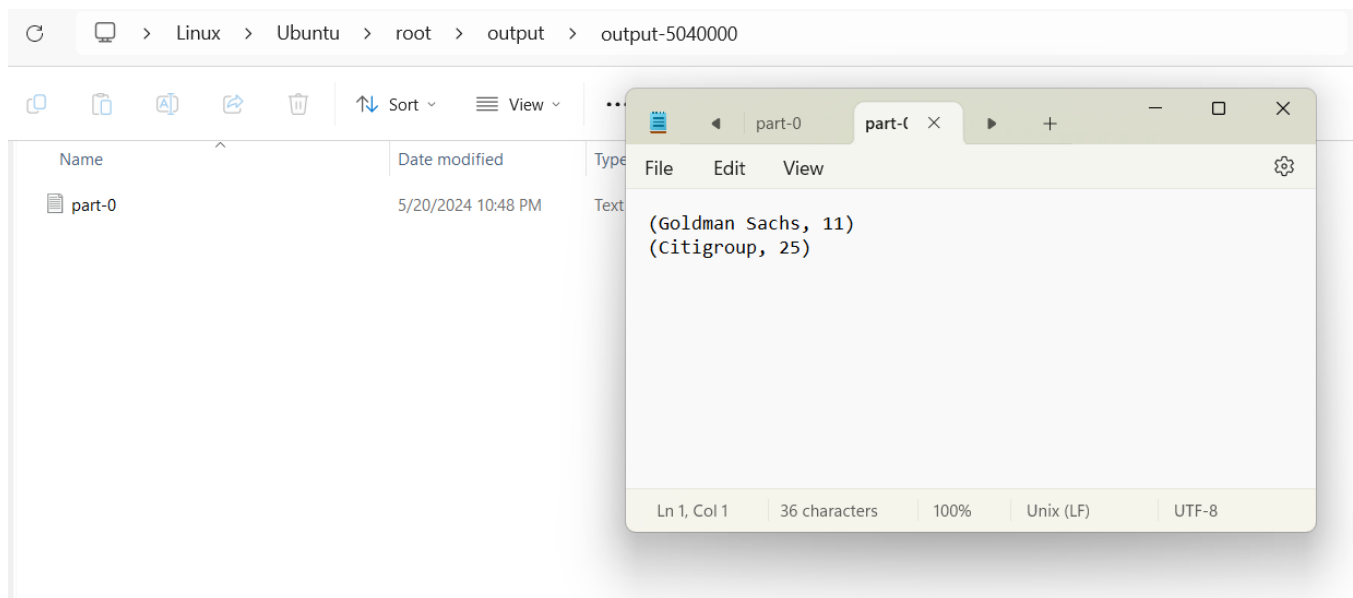


Figure 13: Task 3 output

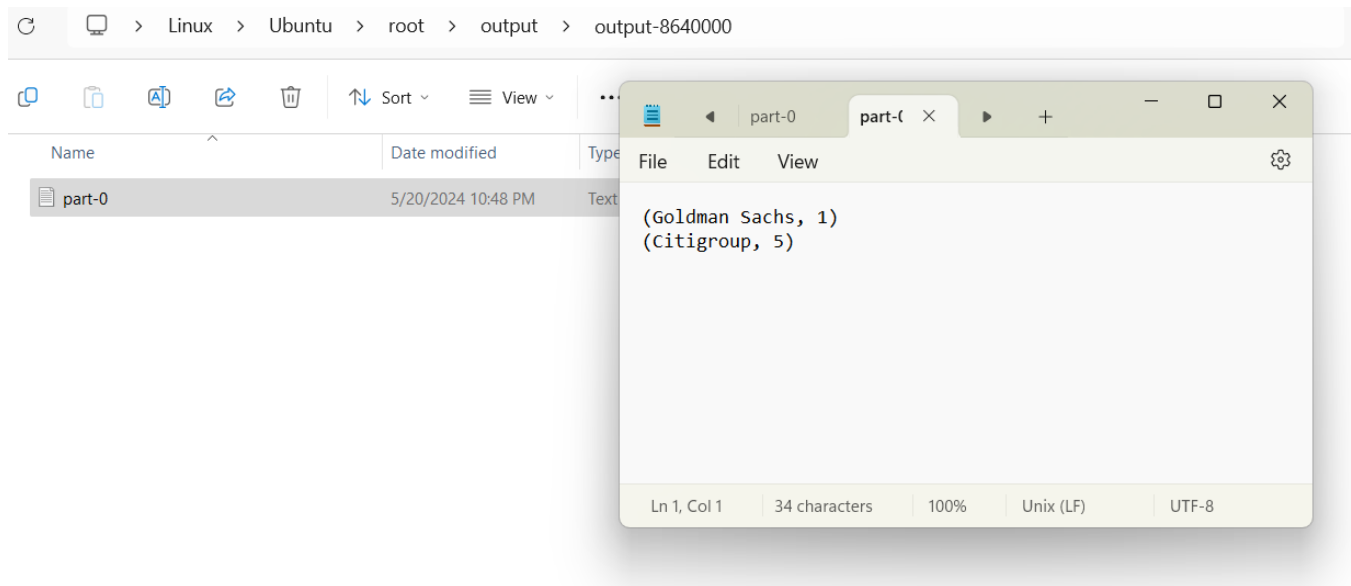


Figure 14: Task 3 output

2.4 Task 4

2.4.1 Ideas [3]

- Using the coding logic of task 3 to filter trips which have drop-off locations are either Goldman or Citigroup.
- Adding timestamp calculated by formula: `(windows.end - windows.start)*100` for later process.
- Declaring a function used in `foreachBatch` function:
 - Transfer the result dataframe to 2 lists, one representing a trip coming from *Goldman Sachs*, and the other representing for *Citigroup*.
 - Traverse through the 2 lists: Check if the current element, the current ten-minutes interval, is matched with the requirements then, print out that element to console and write out to the output file with the format as required in the Lab paper.

We also considered joining the data frame itself or using the lag window function for processing but in spark streaming, the non-time based window functions are not allowed and the joining is also not allowed in complete output mode which is the output mode of our spark streaming code.

2.4.2 Code review

- As the same logic with task 3, we won't discuss again about the filtering code part in this section.
- With the function used in `foreachBatchFunction` the logic code is the same as the description in the Ideas part. The schema of the lists transformed from dataframe are: (Row({start time}, {end time}), {location}, {current_count}, {timestamp})

```
def foreach_batch_function(df, epoch_id):
    goldman_df = df.filter(col("location") == "goldman").rdd.map(tuple).collect()
    citigroup_df = df.filter(col("location") == "citigroup").rdd.map(tuple).collect()

    path = f"output/"
    os.makedirs(path, exist_ok=True)

    for i in range(1, len(goldman_df)):
        if goldman_df[i][2] >= 10:
            if goldman_df[i][3] - 60000 == goldman_df[i-1][3]:
                if goldman_df[i][2] >= goldman_df[i-1][2]*2:
                    print(f"The number of arrivals to Goldman Sachs has doubled from {goldman_df[i-1][2]} to {goldman_df[i][2]} at {goldman_df[i][3]}!")
                    display(f"The number of arrivals to Goldman Sachs has doubled from {goldman_df[i-1][2]} to {goldman_df[i][2]} at {goldman_df[i][3]}!")
                    with open(os.path.join(path, f"part-{goldman_df[i][3]}.txt"), 'a') as file:
                        file.write(f"(goldman,({goldman_df[i][2]}, {goldman_df[i][3]}, {goldman_df[i-1][2]}))\n")
                else:
                    display(f"The number of arrivals to Goldman Sachs has doubled from 0 to {goldman_df[i][2]} at {goldman_df[i][3]}!")
                    print(f"The number of arrivals to Goldman Sachs has doubled from 0 to {goldman_df[i][2]} at {goldman_df[i][3]}!")
                    with open(os.path.join(path, f"part-{goldman_df[i][3]}.txt"), 'a') as file:
                        file.write(f"(goldman,({goldman_df[i][2]}, {goldman_df[i][3]}, 0))\n")

    for i in range(1, len(citigroup_df)):
        if citigroup_df[i][2] >= 10:
            if citigroup_df[i][3] - 60000 == citigroup_df[i-1][3]:
                if citigroup_df[i][2] >= citigroup_df[i-1][2]*2:
                    print(f"The number of arrivals to Citigroup has doubled from {citigroup_df[i-1][2]} to {citigroup_df[i][2]} at {citigroup_df[i][3]}!")
                    display(f"The number of arrivals to Citigroup has doubled from {citigroup_df[i-1][2]} to {citigroup_df[i][2]} at {citigroup_df[i][3]}!")
                    with open(os.path.join(path, f"part-{citigroup_df[i][3]}.txt"), 'a') as file:
                        file.write(f"(citigroup,({citigroup_df[i][2]}, {citigroup_df[i][3]}, {citigroup_df[i-1][2]}))\n")
                else:
                    print(f"The number of arrivals to Citigroup has doubled from {citigroup_df[i-1][2]} to {citigroup_df[i][2]} at {citigroup_df[i][3]}!")
                    display(f"The number of arrivals to Citigroup has doubled from 0 to {citigroup_df[i][2]} at {citigroup_df[i][3]}!")
                    with open(os.path.join(path, f"part-{citigroup_df[i][3]}.txt"), 'a') as file:
                        file.write(f"(citigroup,({citigroup_df[i][2]}, {citigroup_df[i][3]}, 0))\n")
```

Figure 15: Task 4 code snippet

2.4.3 Result

There are 2 ten-minute intervals that satisfy the requirements: 3240000 and 5100000.

```
'The number of arrivals to Citigroup has doubled from 3 to 12 at 3240000!'
```

```
'The number of arrivals to Citigroup has doubled from 3 to 10 at 5100000!'
```

Figure 16: Task 4 output

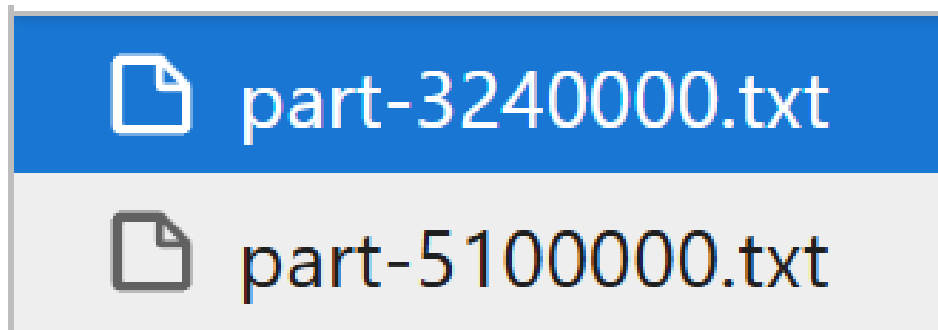


Figure 17: Task 4 output files in output directory

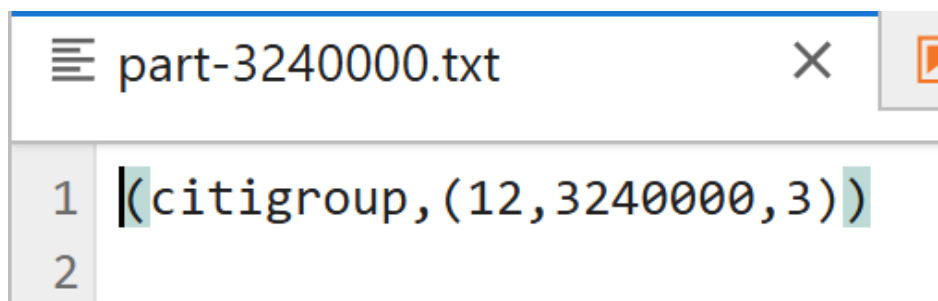


Figure 18: The content of output file part-3240000

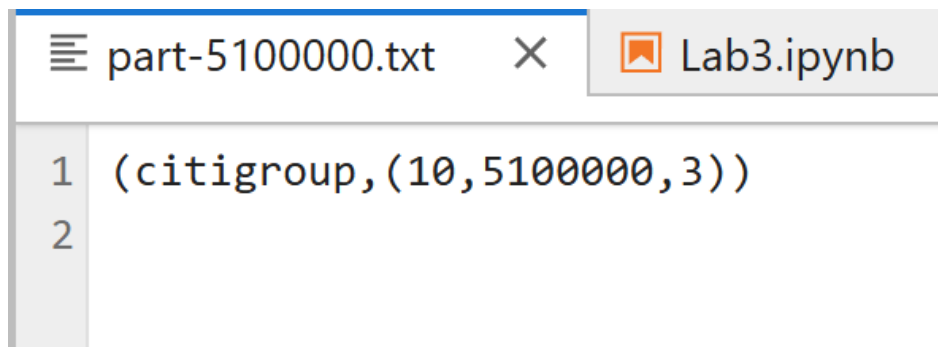


Figure 19: The content of output file part-5100000

References

- [1] Databricks. Title of the Databricks Notebook. [URLdatabricksnotebookONdocs.databricks.com](https://databricksnotebookONdocs.databricks.com), 2024.
- [2] Heliumind. My solution to uw cs 451/651: Data-intensive distributed computing. <https://github.com/heliumind/uw-cs451>, 2024.
- [3] Apache Spark. *Structured Streaming Programming Guide*. The Apache Software Foundation, 2024. Accessed: May 22, 2024.