Data Integration Final Project Report

NYC Yellow Taxi Data Integration with Apache Spark on Databricks

Student Name: Tymur Lukianov
Student ID: GH1026500
GitHub Repository:

https://github.com/xyrp1x/B142-Data-Integration

Video Demonstration:

1. Introduction

Big data does play a crucial important role in optimizing urban transport systems. The **NYC Yellow Taxi dataset** which i will investigate **provides millions of trip records that capture passenger demand, trip distances, fares, tips, and location metadata. Processing such large-scale data requires distributed computing tools like Apache Spark.**

This project demonstrates how to design and implement a **big data management pipeline** on **Databricks** using Spark for taxi trip data. The work focuses on:

- **Ingestion:** Reading raw trip records and zone lookup data.
- **Data cleaning & transformation**
- **Data integration:** Enriching trip records with zone metadata for pickup and drop-off locations.
- **ETL Pipeline:** Trying to persist data and aggregated insights into managed Delta tables.
- **Analytics & Visualization:** Running SQL queries for insights and visualizing trends such as hourly demand and traffic speed patterns.

This work will demonstrate how Spark's distributed capabilities support the end-to-end ETL lifecycle for big data integration.

2. System Design

The system follows a structured ETL pipeline:

1. **Data Ingestion**

Trip data (January 2023) from `yellow_tripdata_2023_01`.

Taxi zone lookup table `taxi_zones`.

2. **Data Cleaning and Transformation**
   Standardized timestamps using `coalesce` into `pickup_ts` and `dropoff_ts`.

   Derived features:

   Trip distance in kilometers, trip duration in minutes, average speed in km/h.

Appliying filters to remove invalid records (e.g., negative fares, zero duration).

3. **Data Integration**

   Saved curated fact table `trips_yellow`.

   Aggregated results into:

   trips and speed by hour, revenue and demand by pickup zones

4. **Analytics & Visualization**

   Spark SQL queries for borough-level revenue, tip percentages, and speed percentiles.

   Matplotlib charts for Trips by Hour and Average Speed by Hour.

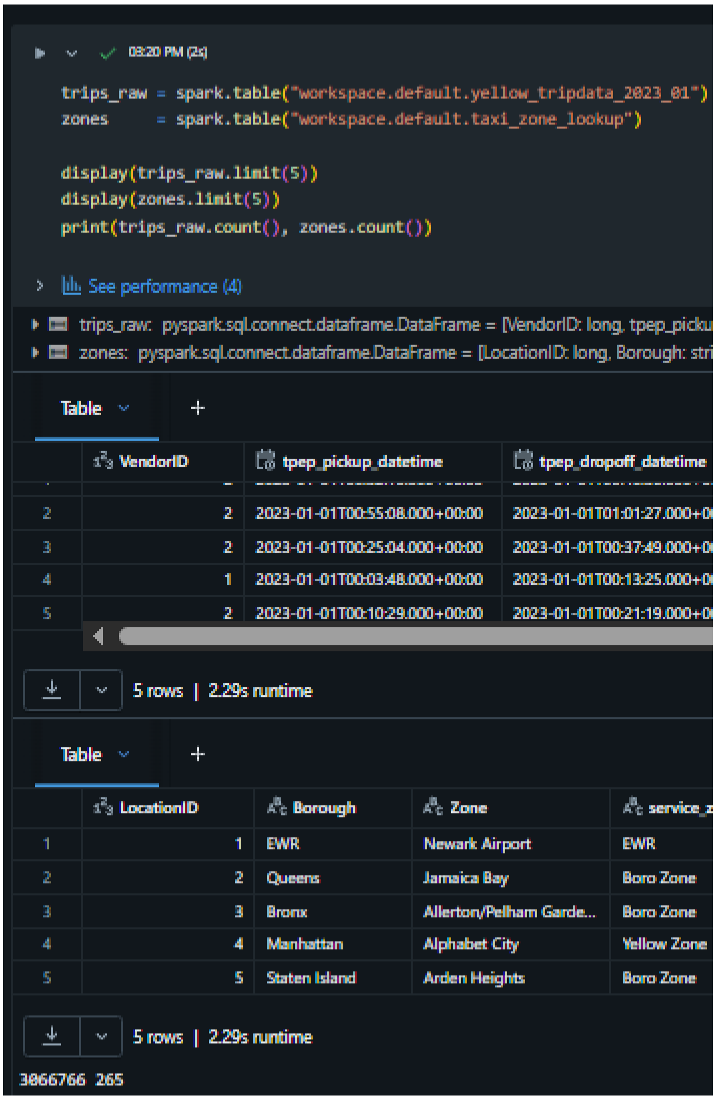
Pipeline Overview:

```
Raw Tables (yellow_tripdata, taxi_zones) --->
Data Cleaning & Transformation (timestamps, km, duration) --->
Integration with Zone Metadata --->
Curated Fact Table (trips_yellow) --->
Aggregated Tables (agg_hourly, agg_top_zones)
SQL Queries & Visualizations --->
```

3. Implementation

Step 1: Ingestion

Raw data was ingested from managed tables in the Databricks catalog:



```
trips_raw = spark.table("workspace.default.yellow_tripdata_2023_01")
zones     = spark.table("workspace.default.taxi_zone_lookup")

display(trips_raw.limit(5))
display(zones.limit(5))
print(trips_raw.count(), zones.count())
```

See performance (4)

trips_raw: pyspark.sql.connect.dataframe.DataFrame = [VendorID: long, tpep_picku

zones: pyspark.sql.connect.dataframe.DataFrame = [LocationID: long, Borough: stri

	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime
2	2	2023-01-01T00:55:08.000+00:00	2023-01-01T01:01:27.000+0
3	2	2023-01-01T00:25:04.000+00:00	2023-01-01T00:37:49.000+0
4	1	2023-01-01T00:03:48.000+00:00	2023-01-01T00:13:25.000+0
5	2	2023-01-01T00:10:29.000+00:00	2023-01-01T00:21:19.000+0

5 rows | 2.29s runtime

	LocationID	Borough	Zone	service_z
1	1	EWR	Newark Airport	EWR
2	2	Queens	Jamaica Bay	Boro Zone
3	3	Bronx	Allerton/Pelham Garde...	Boro Zone
4	4	Manhattan	Alphabet City	Yellow Zone
5	5	Staten Island	Arden Heights	Boro Zone

5 rows | 2.29s runtime

3066766 265

The trips table contained around

30.6 million rows, and the zone table contained 265 rows.

Step 2: Data Cleaning & Feature Engineering

We clean up data for out further investigation, provide valuses

Timestamps unified with `coalesce`.

Distance converted to kilometers.

Duration computed in minutes.

Speed calculated as km/h.

Filters are sort of applied to drop unrealistic trips.

03:20 PM (1s)

```python
from pyspark.sql import functions as F

def coalesce_col(df, names, alias):
    cols = [F.col(n) for n in names if n in df.columns]
    return df.withColumn(alias, F.coalesce(*cols)) if cols else df

df = trips_raw
df = coalesce_col(df, ["tpep_pickup_datetime","pickup_datetime","lpep_pickup_datetime"], "pickup_ts")
df = coalesce_col(df, ["tpep_dropoff_datetime","dropoff_datetime","lpep_dropoff_datetime"], "dropoff_ts")

km = F.col("trip_distance") * F.lit(1.60934)
dur_min = (F.unix_timestamp("dropoff_ts") - F.unix_timestamp("pickup_ts"))/60.0
speed_kmh = (km / (dur_min/60.0))

trips_clean = (df
    .filter(F.col("pickup_ts").isNotNull() & F.col("dropoff_ts").isNotNull())
    .filter(F.col("trip_distance").isNotNull())
    .withColumn("trip_distance_km", km)
    .withColumn("trip_duration_min", dur_min)
    .withColumn("avg_speed_kmh", speed_kmh)
    .filter(F.col("trip_duration_min") > 0.5)
    .filter(F.col("trip_distance_km").between(0.05, 200.0))
    .filter(F.col("fare_amount") >= 0.0)
    .withColumn("year",  F.year("pickup_ts"))
    .withColumn("month", F.month("pickup_ts"))
    .withColumn("hour",  F.hour("pickup_ts"))
)
display(trips_clean.limit(10))
```

Step 3: Integration with Zone Metadata

Each trip record was enriched with borough and zone names for pickup and drop-off:

03:20 PM (1s) 9

```python
zones_sel = (zones
             .withColumn("LocationID", F.col("LocationID").cast("int"))
             .select("LocationID","Borough","Zone","service_zone"))

trips_int = (trips_clean
    .join(zones_sel.withColumnRenamed("Borough","pu_borough")
                   .withColumnRenamed("Zone","pu_zone")
                   .withColumnRenamed("service_zone","pu_service_zone"),
          trips_clean.PULocationID == zones_sel.LocationID, "left")
    .drop(zones_sel.LocationID)
)

trips_int = (trips_int
    .join(zones_sel.withColumnRenamed("Borough","do_borough")
                   .withColumnRenamed("Zone","do_zone")
                   .withColumnRenamed("service_zone","do_service_zone"),
          trips_int.DOLocationID == zones_sel.LocationID, "left")
    .drop(zones_sel.LocationID)
)

display(trips_int.limit(10))
```

See performance (1)

trips_int: pyspark.sql.connect.dataframe.DataFrame = [VendorID: long, tpep_pickup_datetime: timestamp ... 31 more fields]

zones_sel: pyspark.sql.connect.dataframe.DataFrame = [LocationID: integer, Borough: string ... 2 more fields]

Table

	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	RatecodeID	store_and_fwd_flag	PU
1	2	2023-01-01T00:32:10.000+00:00	2023-01-01T00:40:36.000+00:00	1	0.97	1	N	
2	2	2023-01-01T00:55:08.000+00:00	2023-01-01T01:01:27.000+00:00	1	1.1	1	N	
3	2	2023-01-01T00:25:04.000+00:00	2023-01-01T00:37:49.000+00:00	1	2.51	1	N	
4	1	2023-01-01T00:03:48.000+00:00	2023-01-01T00:13:25.000+00:00	0	1.9	1	N	
5	2	2023-01-01T00:10:29.000+00:00	2023-01-01T00:21:19.000+00:00	1	1.43	1	N	
6	2	2023-01-01T00:50:34.000+00:00	2023-01-01T01:02:52.000+00:00	1	1.84	1	N	
7	2	2023-01-01T00:09:22.000+00:00	2023-01-01T00:19:49.000+00:00	1	1.66	1	N	
8	2	2023-01-01T00:27:12.000+00:00	2023-01-01T00:49:56.000+00:00	1	11.7	1	N	
9								

10 rows | 1.39s runtime

Step 4: Persisting Fact and Aggregate Tables

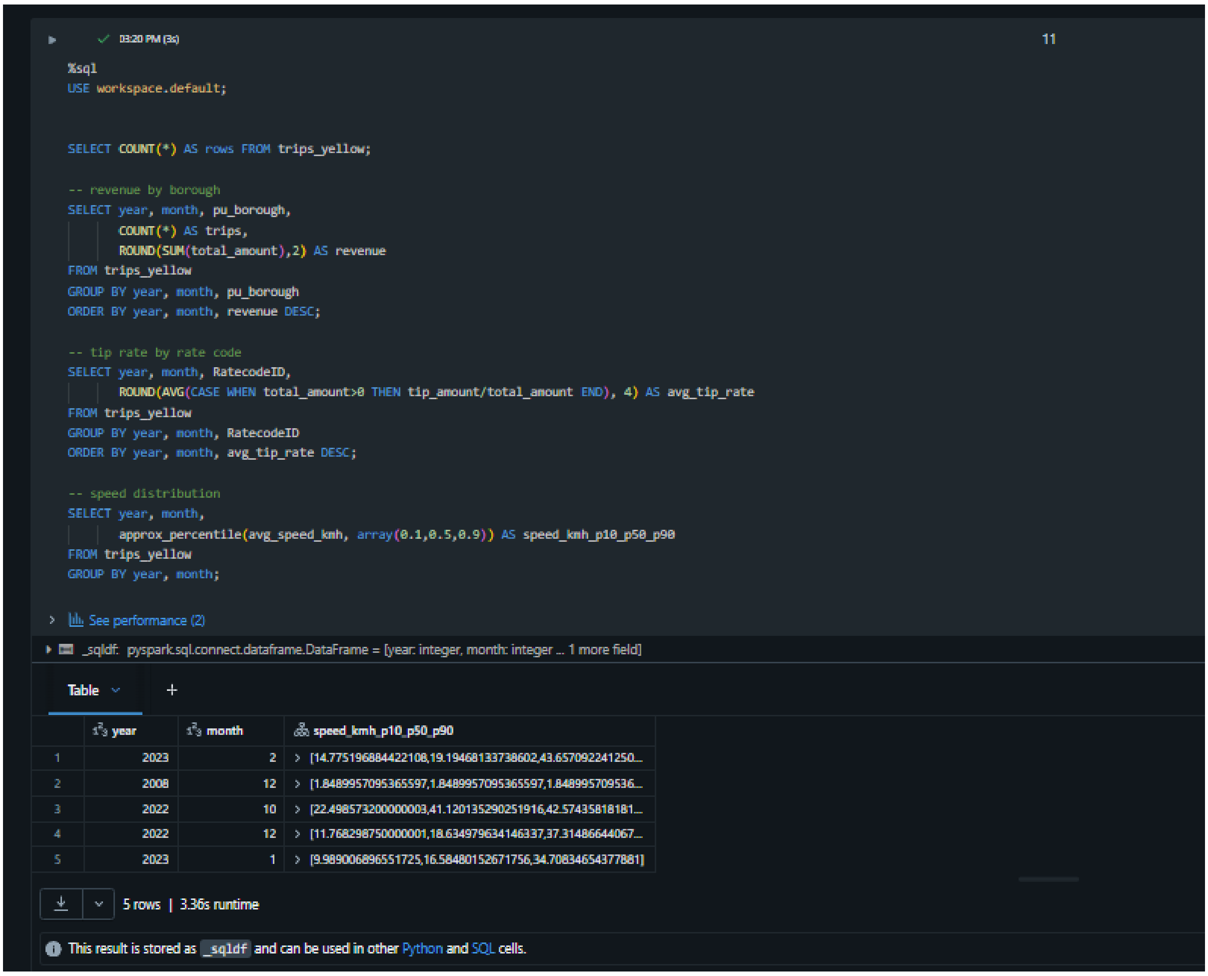
Curated fact and aggregate tables were saved as managed Delta tables:

```
03:20 PM (13s)                                                    10

# refresh facat table
(trips_int
 .write
 .format("delta")
 .mode("overwrite")
 .saveAsTable("workspace.default.trips_yellow"))

# aggregations
hourly = (trips_int.groupBy("year","month","hour")
          .agg(F.count("*").alias("trips"),
               F.avg("trip_distance_km").alias("avg_km"),
               F.avg("trip_duration_min").alias("avg_min"),
               F.avg("avg_speed_kmh").alias("avg_speed_kmh"),
               F.avg("tip_amount").alias("avg_tip"),
               F.avg("total_amount").alias("avg_total")))
(hourly.write.format("delta").mode("overwrite")
       .saveAsTable("workspace.default.agg_hourly"))

top_zones = (trips_int.groupBy("year","month","pu_borough","pu_zone")
             .agg(F.count("*").alias("trips"),
                  F.sum("total_amount").alias("sum_revenue"),
                  F.avg("tip_amount").alias("avg_tip")))
(top_zones.write.format("delta").mode("overwrite")
          .saveAsTable("workspace.default.agg_top_zones"))

See performance (3)
hourly: pyspark.sql.connect.dataframe.DataFrame = [year: integer, month: integer ... 7 more fields]
top_zones: pyspark.sql.connect.dataframe.DataFrame = [year: integer, month: integer ... 5 more fields]
```

Outcome: Three reusable tables (`trips_yellow`, `agg_hourly`, `agg_top_zones`) are now available in the Databricks Catalog for SQL queries, BI dashboards, or ML pipelines.

Step 5: SQL Analytics

Example Spark SQL queries:



```sql
%sql
USE workspace.default;


SELECT COUNT(*) AS rows FROM trips_yellow;

-- revenue by borough
SELECT year, month, pu_borough,
       COUNT(*) AS trips,
       ROUND(SUM(total_amount),2) AS revenue
FROM trips_yellow
GROUP BY year, month, pu_borough
ORDER BY year, month, revenue DESC;

-- tip rate by rate code
SELECT year, month, RatecodeID,
       ROUND(AVG(CASE WHEN total_amount>0 THEN tip_amount/total_amount END), 4) AS avg_tip_rate
FROM trips_yellow
GROUP BY year, month, RatecodeID
ORDER BY year, month, avg_tip_rate DESC;

-- speed distribution
SELECT year, month,
       approx_percentile(avg_speed_kmh, array(0.1,0.5,0.9)) AS speed_kmh_p10_p50_p90
FROM trips_yellow
GROUP BY year, month;
```

See performance (2)

_sqldf: pyspark.sql.connect.dataframe.DataFrame = [year: integer, month: integer ... 1 more field]

	year	month	speed_kmh_p10_p50_p90
1	2023	2	[14.775196884422108,19.19468133738602,43.657092241250...
2	2008	12	[1.8489957095365597,1.8489957095365597,1.848995709536...
3	2022	10	[22.498573200000003,41.120135290251916,42.57435818181...
4	2022	12	[11.768298750000001,18.634979634146337,37.31486644067...
5	2023	1	[9.989006896551725,16.58480152671756,34.70834654377881]

5 rows | 3.36s runtime

This result is stored as _sqldf and can be used in other Python and SQL cells.

Once the Delta tables were created, we ran SQL queries to extract business insights

Revenue by Borough **Tip Percentage by Ratecode and correlation with tips**

Step 6: Visualization

Two charts were plotted from the agg_hourly table:

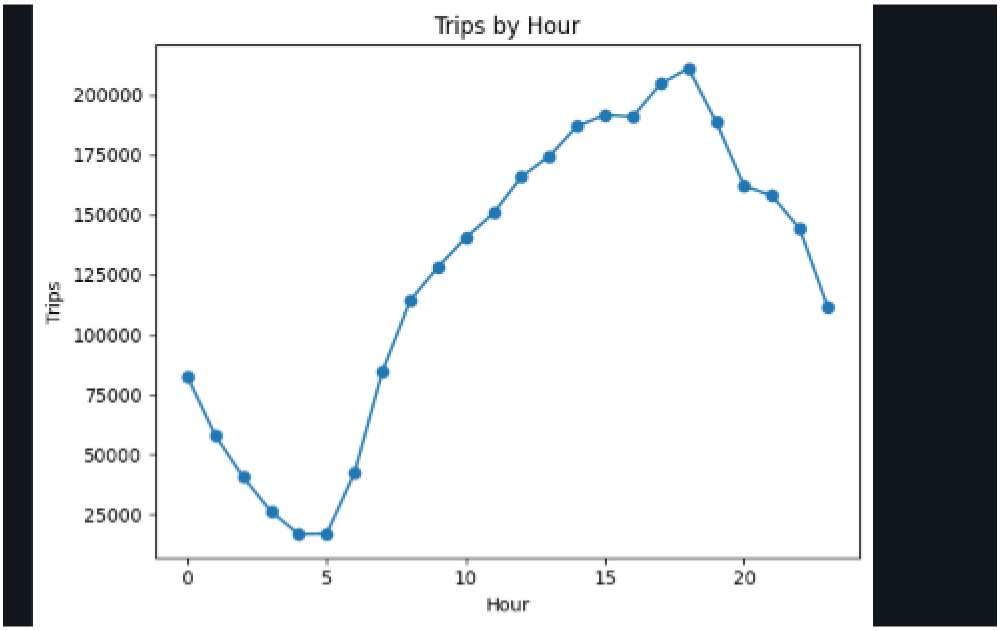
03:20 PM (1s) 13

```python
import matplotlib.pyplot as plt

hourly = spark.table("workspace.default.agg_hourly").toPandas().sort_values(["year","month","hour"])

plt.figure()
hourly.groupby("hour")["trips"].sum().plot(kind="line", marker="o", title="Trips by Hour")
plt.xlabel("Hour"); plt.ylabel("Trips"); plt.tight_layout()
display(plt.gcf())

plt.figure()
hourly.groupby("hour")["avg_speed_kmh"].mean().plot(kind="line", marker="o", title="Average Speed by Hour (km/h)")
plt.xlabel("Hour"); plt.ylabel("km/h"); plt.tight_layout()
display(plt.gcf())
```

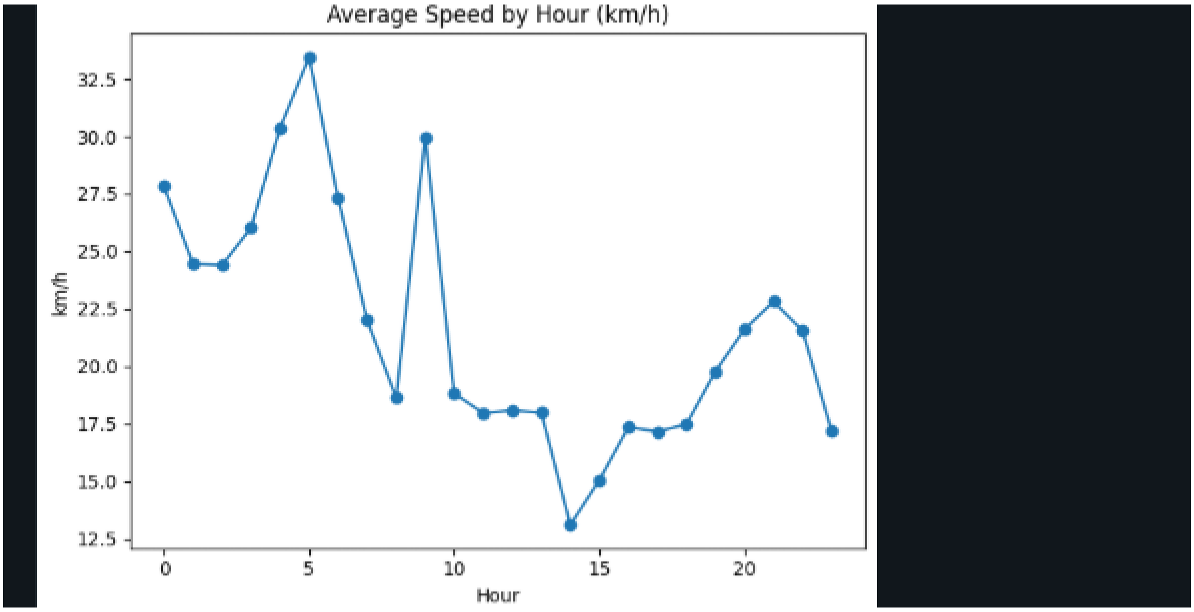
See performance (1)

hourly: pandas.core.frame.DataFrame = [year: int32, month: int32 ... 7 more fields]

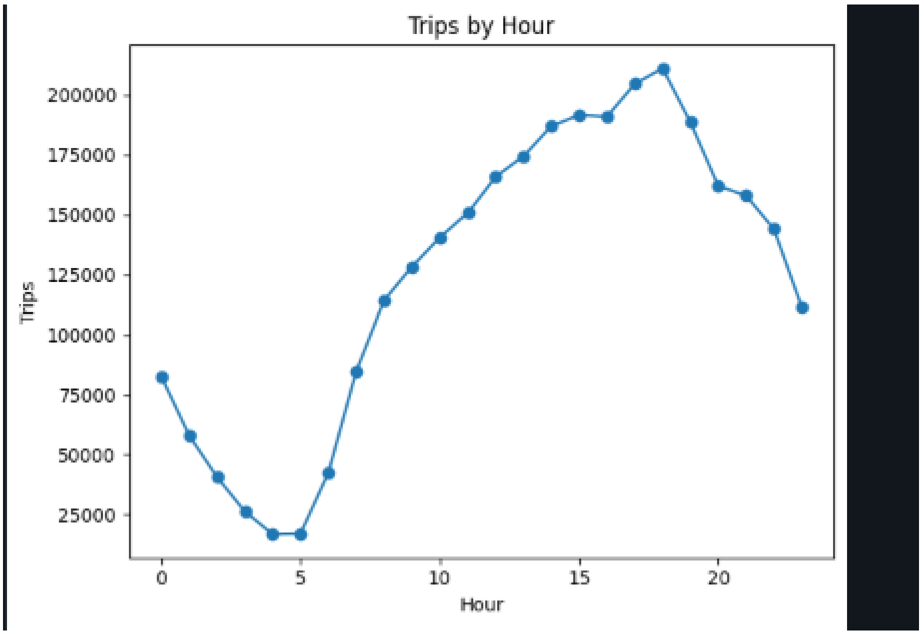
Trips by Hour – showing demand peaks during evening rush hours.



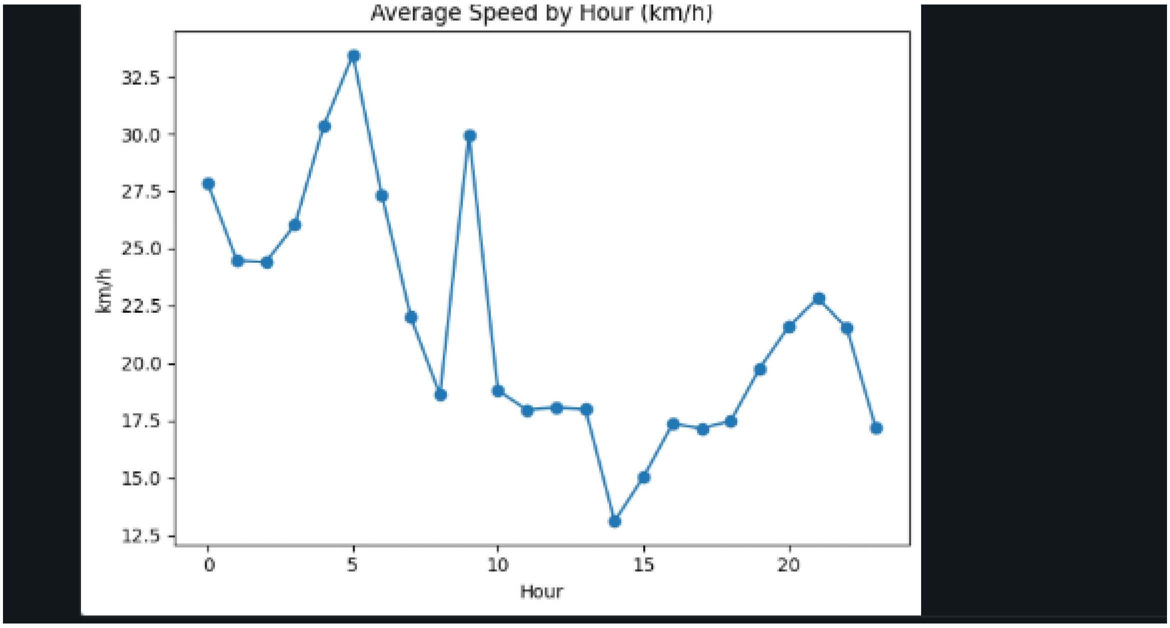
Average Speed by Hour (km/h) – showing congestion during peak times.



Trips by Hour



Average speed/hour (kh/m)



Challenges & Solutions

I did face couple challenges during this project, and here is the solution:

DBFS was disabled /filestore ---> I managed tables in *workspace.default*

Data quality issues (negative fares, 0 distance trips) ---> applying strict filters helped me out to drop not usable information

Handling large dataset (30M+ rows) ---> Simply saved curated outputs as Delta tables for optimized queries.

Results

From the curated dataset and aggregates:

- **Trips by Hour:** Peak demand occurs around **5–8 PM**, while lowest demand occurs around **3–5 AM**.
- **Average Speed:** Speeds drop significantly during daytime traffic, averaging **12–15 km/h** in Manhattan, but rise above **30 km/h** overnight.
- **Revenue by Borough:** Manhattan accounts for the majority of both trips and revenue.
- **Tip Rates:** Highest tip rates are associated with certain RatecodeIDs, showing different passenger behaviors.

6. Conclusion

This project made me successfully implement a full **data integration pipeline, i** used Spark on Databricks. It demonstrated how to ingest, clean, transform, integrate, and analyze millions of taxi trips efficiently.

7. References

NYC Taxi & Limousine Commission (TLC). (2023) *Yellow Taxi Trip Records, January 2023*. New York City Open Data. Available at: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

NYC Taxi & Limousine Commission (TLC). (2023) *Yellow Taxi Trip Records, January 2023*. New York City Open Data. Available at: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page