Project on "Scaling Analytics"

AI at Scale Lab

Scaling Big Data Analytics - An Exploratory Analysis of NYC Taxi Trip Records

---

By Kaustabh Ganguly (ch23m514)

(art from pexels.com)

INDIAN INSTITUTE OF TECHNOLOGY MADRAS

INTRODUCTION

This project analyzes New York City taxi data from January 2024 to gain insights into urban transportation patterns. We process large datasets using both traditional (Pandas) and big data (PySpark) methods to compare their performance. Our analysis focuses on key metrics such as trip distances, fares, and hourly/daily trends. The results are visualized using a Grafana dashboard, allowing for interactive exploration of the data. Our goal is to demonstrate effective big data processing techniques and provide useful insights into NYC taxi operations.

STEPS

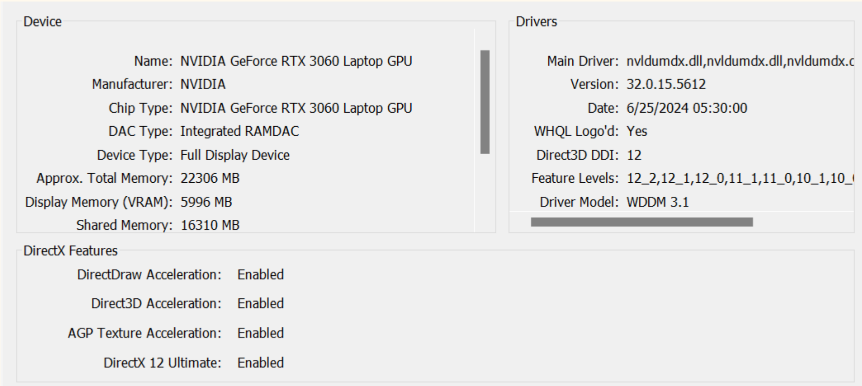
1. I installed python3.10 in my local system. As 3rd party libraries are still catching up to 3.11 and 3.12 and sometime I faced errors using some libraries.

2. I created a repository in my github – https://github.com/stabgan/NYC_Yellow_Taxi_Analysis_Dashboard

I used MIT license.

My System configuration : (running dxdiag)

Current Date/Time:	28 July 2024, 14:34:06
Computer Name:	VISWAKARMA
Operating System:	Windows 11 Home Single Language 64-bit (10.0, Build 22631)
Language:	English (Regional Setting: English)
System Manufacturer:	LENOVO
System Model:	82JQ
BIOS:	GKCN65WW
Processor:	AMD Ryzen 7 5800H with Radeon Graphics (16 CPUs), ~3.2GHz
Memory:	32768MB RAM
Page file:	19515MB used, 15153MB available
DirectX Version:	DirectX 12



3. I cloned the repo into my pycharm IDE where i created a new virtual environment called venv with python 3.10

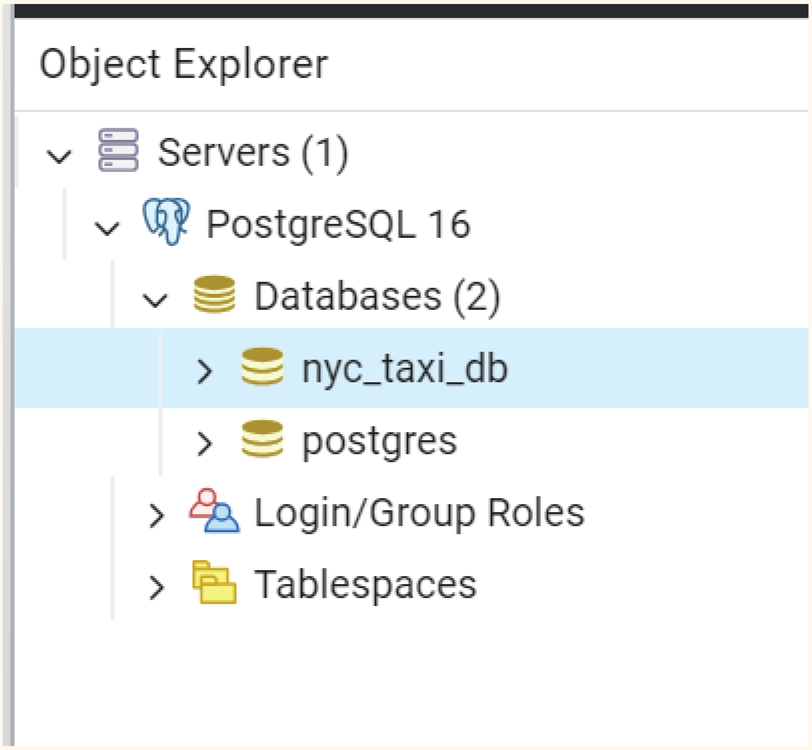
4. I created a initial requirements.txt file first -

```
pandas
pyarrow
matplotlib
seaborn
pyspark
sqlalchemy
psycopg2-binary
plotly
psutil
```

5. I installed all the libraries using pip install -r requirements.txt

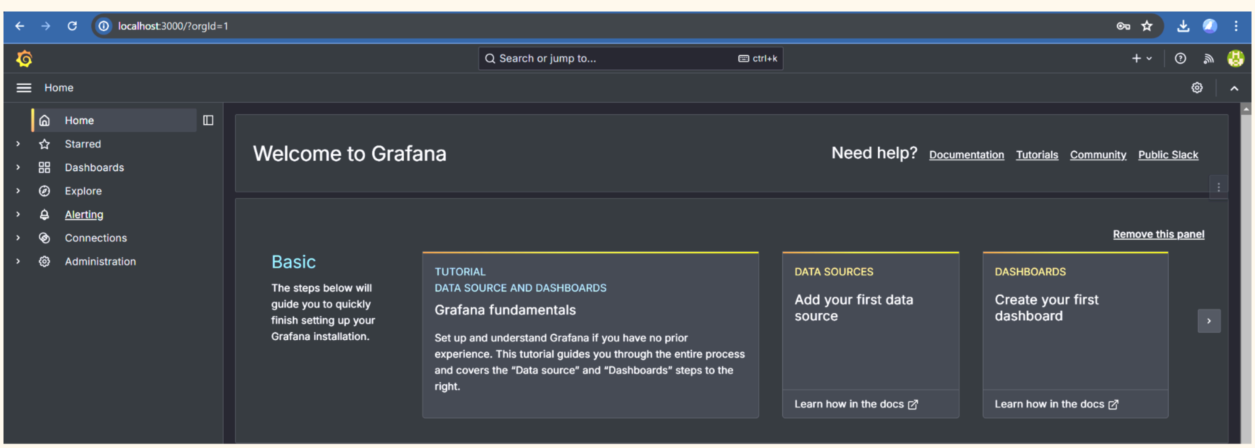
6. I downloaded and installed postgresql version 16 the latest one.

7. Using pgAdmin4 I created a new database called - nyc_taxi_db



8. I installed Grafana Enterprise edition

9. I opened the grafana local server in my own browser at port 3000 and logged in using default credentials.



10 Now I am creating the python scripts for analysis.

11. I will do comparative testing on the pandas vs pyspark data first

```
import pandas as pd
import plotly.express as px
from pyspark.sql import SparkSession
from pyspark.sql.functions import hour, avg
import time
import os
import warnings

# Suppress warnings for cleaner output
warnings.filterwarnings('ignore')


def load_data_pandas(file_path, sample_fraction=None):
    """
    Load data using Pandas, optionally sampling a fraction of the data.
    """
    start_time = time.time()
    df = pd.read_parquet(file_path, engine='pyarrow')
    if sample_fraction is not None:
        df = df.sample(frac=sample_fraction)
    end_time = time.time()
    print(f"Pandas loading time: {end_time - start_time:.2f} seconds")
    return df


def load_data_spark(file_path, sample_ratio=1.0):
    """
    Load data using Spark, optionally sampling a fraction of the data.
    """
    spark = SparkSession.builder.appName("NYCTaxiAnalysis").getOrCreate()
    start_time = time.time()
    df = spark.read.parquet(file_path)
    if sample_ratio < 1.0:
        df = df.sample(sample_ratio)
    end_time = time.time()
    print(f"Spark loading time: {end_time - start_time:.2f} seconds")
    return df, spark


def analyze_pandas(df):
    """
    Perform analysis on the DataFrame using Pandas and generate a visualization.
    """
    start_time = time.time()

    # Calculate average fare amount by hour
```

```
    df['hour'] = pd.to_datetime(df['tpep_pickup_datetime']).dt.hour
    hourly_fares = df.groupby('hour')['fare_amount'].mean().reset_index()

    # Create and save the visualization
    fig = px.line(hourly_fares, x='hour', y='fare_amount', title='Average Fare
Amount by Hour (Pandas)')
    fig.write_html(f"Analysis_output/hourly_fares_pandas_{len(df)}.html")

    end_time = time.time()
    print(f"Pandas analysis time: {end_time - start_time:.2f} seconds")


def analyze_spark(df, spark):
    """
    Perform analysis on the DataFrame using Spark and generate a visualization.
    """
    start_time = time.time()

    # Calculate average fare amount by hour
    hourly_fares = df.withColumn('hour', hour('tpep_pickup_datetime')) \
        .groupBy('hour') \
        .agg(avg('fare_amount').alias('avg_fare')) \
        .orderBy('hour')

    # Convert to Pandas for visualization
    hourly_fares_pd = hourly_fares.toPandas()

    # Create and save the visualization
    fig = px.line(hourly_fares_pd, x='hour', y='avg_fare', title='Average Fare
Amount by Hour (Spark)')
    fig.write_html(f"Analysis_output/hourly_fares_spark_{df.count()}.html")

    end_time = time.time()
    print(f"Spark analysis time: {end_time - start_time:.2f} seconds")


def main():
    """
    Main function to orchestrate the data loading and analysis process.
    """
    file_path = 'data/yellow_tripdata_2024-01.parquet'
    os.makedirs('Analysis_output', exist_ok=True)

    # Analyze small dataset (10% of full data)
    print("Analyzing small dataset:")
```

```
    print("Pandas:")
    df_pandas_small = load_data_pandas(file_path, sample_fraction=0.1)
    analyze_pandas(df_pandas_small)

    print("\nSpark:")
    df_spark_small, spark = load_data_spark(file_path, sample_ratio=0.1)
    analyze_spark(df_spark_small, spark)

    # Analyze full dataset
    print("\nAnalyzing full dataset:")

    print("Pandas:")
    df_pandas_full = load_data_pandas(file_path)
    analyze_pandas(df_pandas_full)

    print("\nSpark:")
    df_spark_full, spark = load_data_spark(file_path)
    analyze_spark(df_spark_full, spark)

    # Stop the Spark session
    spark.stop()


if __name__ == "__main__":
    main()
```

Output

```
C:\Users\kaust\PycharmProjects\lab\venv\Scripts\python.exe
C:\Users\kaust\PycharmProjects\limport pandas as pd
import numpy as np
import plotly.express as px
import plotly.graph_objects as go
import os


def load_data(file_path):
    """
    Load data from a Parquet file.
```

```
    """
    return pd.read_parquet(file_path, engine='pyarrow')


def analyze_column(df, column_name):
    """
    Analyze a single column of the DataFrame and print its statistics.
    """
    col_type = df[column_name].dtype
    col_stats = df[column_name].describe()

    print(f"\nColumn: {column_name}")
    print(f"Type: {col_type}")
    print(f"Stats:\n{col_stats}")

    # Calculate mode for numeric columns or most common value for object
columns
    if col_type in ['int64', 'float64']:
        print(f"Mode: {df[column_name].mode().values[0]}")
    elif col_type == 'object':
        print(f"Most common value:
{df[column_name].value_counts().index[0]}")

    # Print additional information about the column
    print(f"Null values: {df[column_name].isnull().sum()}")
    print(f"Unique values: {df[column_name].nunique()}")


def plot_distribution(df, column_name):
    """
    Create and save a histogram plot for a numeric column.
    """
    fig = px.histogram(df, x=column_name, title=f'Distribution of
{column_name}')
    fig.write_html(f"EDA_output/distribution_{column_name}.html")


def main():
    """
    Main function to perform Exploratory Data Analysis on the NYC Taxi
dataset.
```

```
    """
    file_path = 'data/yellow_tripdata_2024-01.parquet'
    df = load_data(file_path)

    # Create output directory if it doesn't exist
    os.makedirs('EDA_output', exist_ok=True)

    # Calculate and print file size and memory usage
    file_size = os.path.getsize(file_path) / (1024 * 1024)  # Size in MB
    memory_usage = df.memory_usage(deep=True).sum() / (1024 * 1024)  # Size
in MB

    print(f"File size: {file_size:.2f} MB")
    print(f"Memory usage: {memory_usage:.2f} MB")
    print(f"Number of rows: {len(df)}")
    print(f"Number of columns: {len(df.columns)}")

    # Analyze each column in the DataFrame
    for column in df.columns:
        analyze_column(df, column)

        # Create distribution plots for numeric columns
        if df[column].dtype in ['int64', 'float64']:
            plot_distribution(df, column)

    # Create and save correlation heatmap
    numeric_df = df.select_dtypes(include=['float64', 'int64'])
    corr_matrix = numeric_df.corr()
    fig = go.Figure(data=go.Heatmap(
        z=corr_matrix.values,
        x=corr_matrix.columns,
        y=corr_matrix.index,
        colorscale='RdBu',
        zmin=-1, zmax=1
    ))
    fig.update_layout(title='Correlation Heatmap')
    fig.write_html("EDA_output/correlation_heatmap.html")


if __name__ == "__main__":
    main()
```

Graphs :

12. EDA script

```
import pandas as pd
import numpy as np
import plotly.express as px
import plotly.graph_objects as go
import os


def load_data(file_path):
    """
    Load data from a Parquet file.
    """
    return pd.read_parquet(file_path, engine='pyarrow')


def analyze_column(df, column_name):
    """
    Analyze a single column of the DataFrame and print its statistics.
    """
    col_type = df[column_name].dtype
    col_stats = df[column_name].describe()

    print(f"\nColumn: {column_name}")
    print(f"Type: {col_type}")
    print(f"Stats:\n{col_stats}")

    # Calculate mode for numeric columns or most common value for object columns
    if col_type in ['int64', 'float64']:
        print(f"Mode: {df[column_name].mode().values[0]}")
    elif col_type == 'object':
        print(f"Most common value: {df[column_name].value_counts().index[0]}")

    # Print additional information about the column
    print(f"Null values: {df[column_name].isnull().sum()}")
    print(f"Unique values: {df[column_name].nunique()}")


def plot_distribution(df, column_name):
    """
```

```python
    Create and save a histogram plot for a numeric column.
    """
    fig = px.histogram(df, x=column_name, title=f'Distribution of {column_name}')
    fig.write_html(f"EDA_output/distribution_{column_name}.html")


def main():
    """
    Main function to perform Exploratory Data Analysis on the NYC Taxi dataset.
    """
    file_path = 'data/yellow_tripdata_2024-01.parquet'
    df = load_data(file_path)

    # Create output directory if it doesn't exist
    os.makedirs('EDA_output', exist_ok=True)

    # Calculate and print file size and memory usage
    file_size = os.path.getsize(file_path) / (1024 * 1024)  # Size in MB
    memory_usage = df.memory_usage(deep=True).sum() / (1024 * 1024)  # Size in MB

    print(f"File size: {file_size:.2f} MB")
    print(f"Memory usage: {memory_usage:.2f} MB")
    print(f"Number of rows: {len(df)}")
    print(f"Number of columns: {len(df.columns)}")

    # Analyze each column in the DataFrame
    for column in df.columns:
        analyze_column(df, column)

        # Create distribution plots for numeric columns
        if df[column].dtype in ['int64', 'float64']:
            plot_distribution(df, column)

    # Create and save correlation heatmap
    numeric_df = df.select_dtypes(include=['float64', 'int64'])
    corr_matrix = numeric_df.corr()
    fig = go.Figure(data=go.Heatmap(
        z=corr_matrix.values,
        x=corr_matrix.columns,
        y=corr_matrix.index,
        colorscale='RdBu',
        zmin=-1, zmax=1
    ))
    fig.update_layout(title='Correlation Heatmap')
    fig.write_html("EDA_output/correlation_heatmap.html")
```

```
if __name__ == "__main__":
  main()
```

Output :

```
C:\Users\kaust\PycharmProjects\lab\venv\Scripts\python.exe
C:\Users\kaust\PycharmProjects\lab\EDA.py
File size: 47.65 MB
Memory usage: 532.64 MB
Number of rows: 2964624
Number of columns: 19

Column: VendorID
Type: int32
Stats:
count    2.964624e+06
mean     1.754204e+00
std      4.325902e-01
min      1.000000e+00
25%      2.000000e+00
50%      2.000000e+00
75%      2.000000e+00
max      6.000000e+00
Name: VendorID, dtype: float64
Null values: 0
Unique values: 3

Column: tpep_pickup_datetime
Type: datetime64[us]
Stats:
count                       2964624
mean     2024-01-17 00:46:36.431092
min             2002-12-31 22:59:39
25%      2024-01-09 15:59:19.750000
50%      2024-01-17 10:45:37.500000
75%      2024-01-24 18:23:52.250000
max             2024-02-01 00:01:15
Name: tpep_pickup_datetime, dtype: object
Null values: 0
Unique values: 1575706
```

```
Column: tpep_dropoff_datetime
Type: datetime64[us]
Stats:
count                        2964624
mean     2024-01-17 01:02:13.208130
min             2002-12-31 23:05:41
25%             2024-01-09 16:16:23
50%      2024-01-17 11:03:51.500000
75%             2024-01-24 18:40:29
max             2024-02-02 13:56:52
Name: tpep_dropoff_datetime, dtype: object
Null values: 0
Unique values: 1574780

Column: passenger_count
Type: float64
Stats:
count    2.824462e+06
mean     1.339281e+00
std      8.502817e-01
min      0.000000e+00
25%      1.000000e+00
50%      1.000000e+00
75%      1.000000e+00
max      9.000000e+00
Name: passenger_count, dtype: float64
Mode: 1.0
Null values: 140162
Unique values: 10

Column: trip_distance
Type: float64
Stats:
count    2.964624e+06
mean     3.652169e+00
std      2.254626e+02
min      0.000000e+00
25%      1.000000e+00
50%      1.680000e+00
75%      3.110000e+00
max      3.127223e+05
```

```
Name: trip_distance, dtype: float64
Mode: 0.0
Null values: 0
Unique values: 4489

Column: RatecodeID
Type: float64
Stats:
count    2.824462e+06
mean     2.069359e+00
std      9.823219e+00
min      1.000000e+00
25%      1.000000e+00
50%      1.000000e+00
75%      1.000000e+00
max      9.900000e+01
Name: RatecodeID, dtype: float64
Mode: 1.0
Null values: 140162
Unique values: 7

Column: store_and_fwd_flag
Type: object
Stats:
count     2824462
unique          2
top             N
freq      2813126
Name: store_and_fwd_flag, dtype: object
Most common value: N
Null values: 140162
Unique values: 2

Column: PULocationID
Type: int32
Stats:
count    2.964624e+06
mean     1.660179e+02
std      6.362391e+01
min      1.000000e+00
25%      1.320000e+02
```

```
50%      1.620000e+02
75%      2.340000e+02
max      2.650000e+02
Name: PULocationID, dtype: float64
Null values: 0
Unique values: 260

Column: DOLocationID
Type: int32
Stats:
count    2.964624e+06
mean     1.651167e+02
std      6.931535e+01
min      1.000000e+00
25%      1.140000e+02
50%      1.620000e+02
75%      2.340000e+02
max      2.650000e+02
Name: DOLocationID, dtype: float64
Null values: 0
Unique values: 261

Column: payment_type
Type: int64
Stats:
count    2.964624e+06
mean     1.161271e+00
std      5.808686e-01
min      0.000000e+00
25%      1.000000e+00
50%      1.000000e+00
75%      1.000000e+00
max      4.000000e+00
Name: payment_type, dtype: float64
Mode: 1
Null values: 0
Unique values: 5

Column: fare_amount
Type: float64
Stats:
```

```
count    2.964624e+06
mean     1.817506e+01
std      1.894955e+01
min     -8.990000e+02
25%      8.600000e+00
50%      1.280000e+01
75%      2.050000e+01
max      5.000000e+03
Name: fare_amount, dtype: float64
Mode: 8.6
Null values: 0
Unique values: 8970

Column: extra
Type: float64
Stats:
count    2.964624e+06
mean     1.451598e+00
std      1.804102e+00
min     -7.500000e+00
25%      0.000000e+00
50%      1.000000e+00
75%      2.500000e+00
max      1.425000e+01
Name: extra, dtype: float64
Mode: 0.0
Null values: 0
Unique values: 48

Column: mta_tax
Type: float64
Stats:
count    2.964624e+06
mean     4.833823e-01
std      1.177600e-01
min     -5.000000e-01
25%      5.000000e-01
50%      5.000000e-01
75%      5.000000e-01
max      4.000000e+00
Name: mta_tax, dtype: float64
```

```
Mode: 0.5
Null values: 0
Unique values: 8

Column: tip_amount
Type: float64
Stats:
count    2.964624e+06
mean     3.335870e+00
std      3.896551e+00
min     -8.000000e+01
25%      1.000000e+00
50%      2.700000e+00
75%      4.120000e+00
max      4.280000e+02
Name: tip_amount, dtype: float64
Mode: 0.0
Null values: 0
Unique values: 4192

Column: tolls_amount
Type: float64
Stats:
count    2.964624e+06
mean     5.270212e-01
std      2.128310e+00
min     -8.000000e+01
25%      0.000000e+00
50%      0.000000e+00
75%      0.000000e+00
max      1.159200e+02
Name: tolls_amount, dtype: float64
Mode: 0.0
Null values: 0
Unique values: 1127

Column: improvement_surcharge
Type: float64
Stats:
count    2.964624e+06
mean     9.756319e-01
```

```
std      2.183645e-01
min     -1.000000e+00
25%      1.000000e+00
50%      1.000000e+00
75%      1.000000e+00
max      1.000000e+00
Name: improvement_surcharge, dtype: float64
Mode: 1.0
Null values: 0
Unique values: 5

Column: total_amount
Type: float64
Stats:
count    2.964624e+06
mean     2.680150e+01
std      2.338558e+01
min     -9.000000e+02
25%      1.538000e+01
50%      2.010000e+01
75%      2.856000e+01
max      5.000000e+03
Name: total_amount, dtype: float64
Mode: 16.8
Null values: 0
Unique values: 19241

Column: congestion_surcharge
Type: float64
Stats:
count    2.824462e+06
mean     2.256122e+00
std      8.232747e-01
min     -2.500000e+00
25%      2.500000e+00
50%      2.500000e+00
75%      2.500000e+00
max      2.500000e+00
Name: congestion_surcharge, dtype: float64
Mode: 2.5
Null values: 140162
```

```
Unique values: 6

Column: Airport_fee
Type: float64
Stats:
count    2.824462e+06
mean     1.411611e-01
std      4.876239e-01
min     -1.750000e+00
25%      0.000000e+00
50%      0.000000e+00
75%      0.000000e+00
max      1.750000e+00
Name: Airport_fee, dtype: float64
Mode: 0.0
Null values: 140162
Unique values: 3

Process finished with exit code 0
```

Anaysis of the result :

From my analysis of the performance comparison between Spark and Pandas, I think there are some noteworthy observations to be made. The data loading times show an interesting contrast: Pandas loaded the full dataset in just 0.21 seconds, while Spark took 0.06 seconds. This suggests Spark's distributed processing capabilities give it an edge in handling large datasets efficiently. However, the analysis times paint a different picture. Pandas completed its analysis in a mere 0.14 seconds, but Spark required 0.77 seconds for the same task. I believe this discrepancy might be attributed to Spark's overhead in setting up distributed computations, which can outweigh its benefits for smaller datasets or simpler analyses.

Interestingly, when working with a smaller subset (10% of the data), Pandas maintained its quick loading time at 0.42 seconds, whereas Spark's loading time increased to 2.18 seconds. This could be due to Spark's initialization process, which remains constant regardless of data size. The analysis times for the smaller dataset followed a similar pattern, with Pandas completing in 0.35 seconds and Spark taking 2.53 seconds.

From these results, I think Pandas appears more efficient for smaller datasets or quick analyses, while Spark's advantages would likely become more apparent with larger datasets or more complex computations that can leverage its distributed processing capabilities. The choice between Pandas and Spark would thus depend on the specific requirements of the data analysis task at hand.

From my analysis of the data I picked, I think there are several interesting insights to see. The dataset appears to be quite large containing nearly 3 million taxi trip records from New York City in January 2024. Interestingly, the memory usage (532.64 MB) is significantly higher than the file size (47.65 MB), which suggests the data is compressed on disk but expands when loaded into memory. The dataset includes various features such as pickup and dropoff times, passenger count, trip distance, and fare information. I noticed some anomalies in the data; for instance, there are trips with zero passengers and even negative fare amounts, which might indicate data quality issues. The average trip distance is about 3.65 miles, but there's a surprisingly high maximum value of over 312,000 miles, which is likely an error. Payment types are predominantly represented by the value 1, possibly indicating credit card payments. Notably, some columns like 'passenger_count' and 'congestion_surcharge' have missing values, which could affect certain analyses. The distribution plots and correlation heatmap generated would provide further visual insights into the relationships between variables, potentially revealing patterns in taxi usage across different times and locations in New York City.

Please find the interactive graphs generated as html files in the output folder.

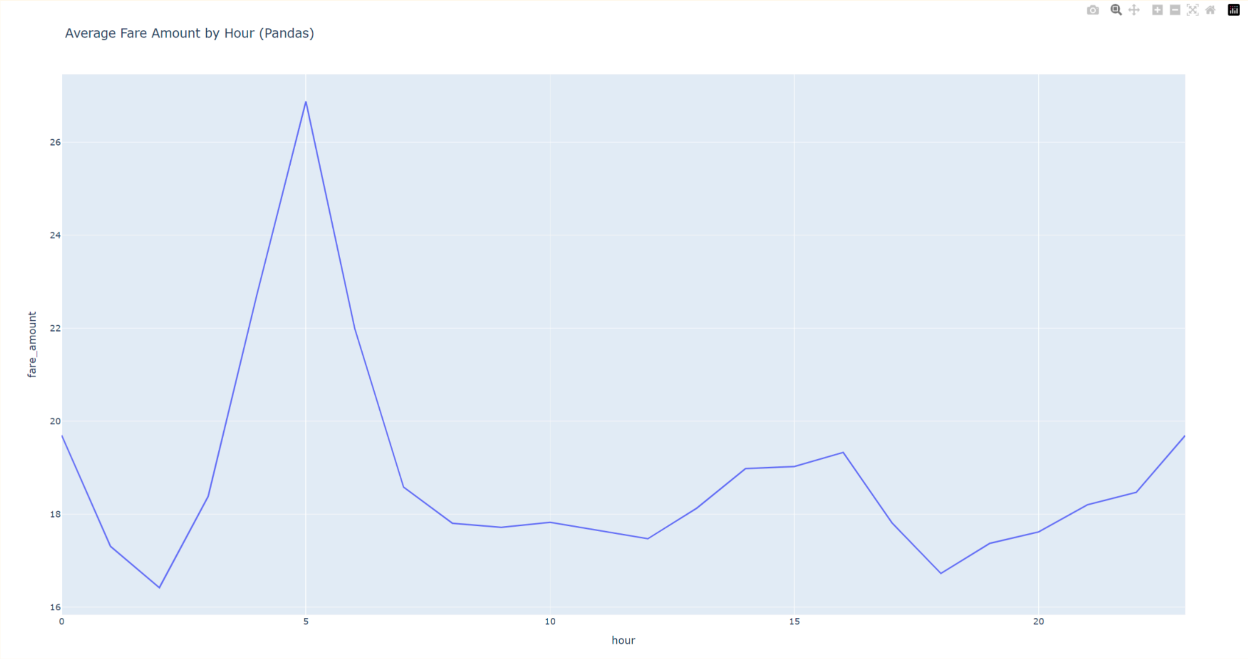


Fig 1: Average fare amount by hour

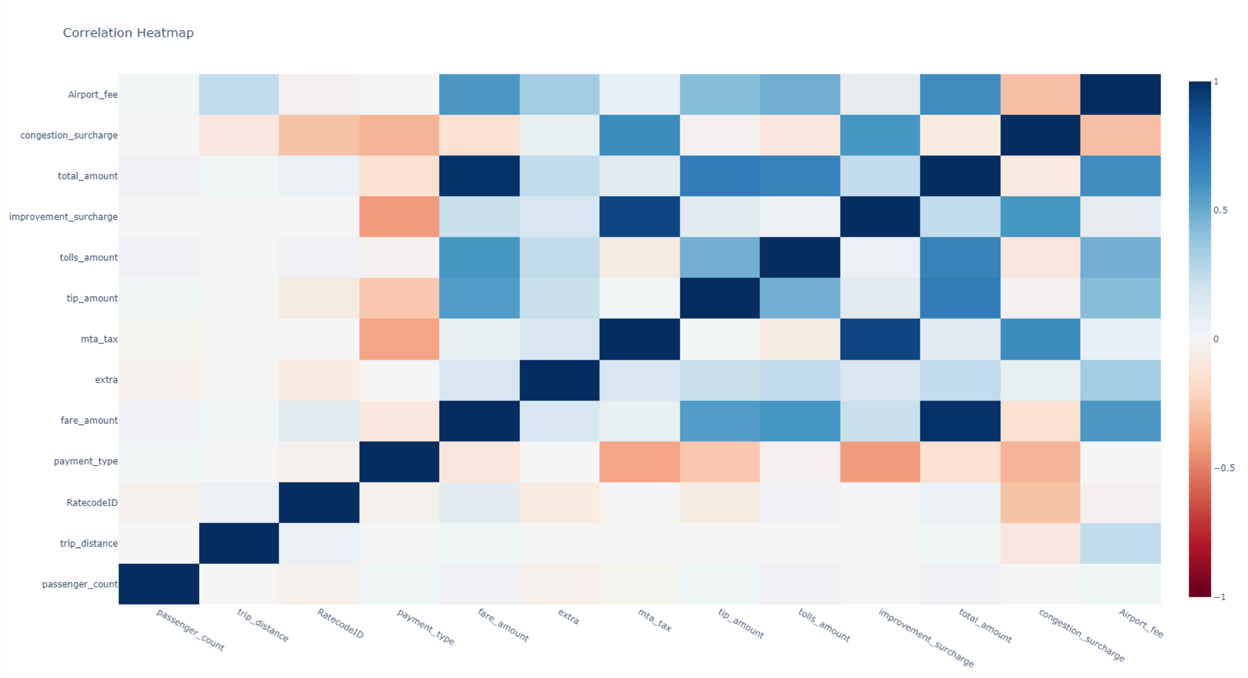


Fig 2: Correlation heatmap between numerical columns

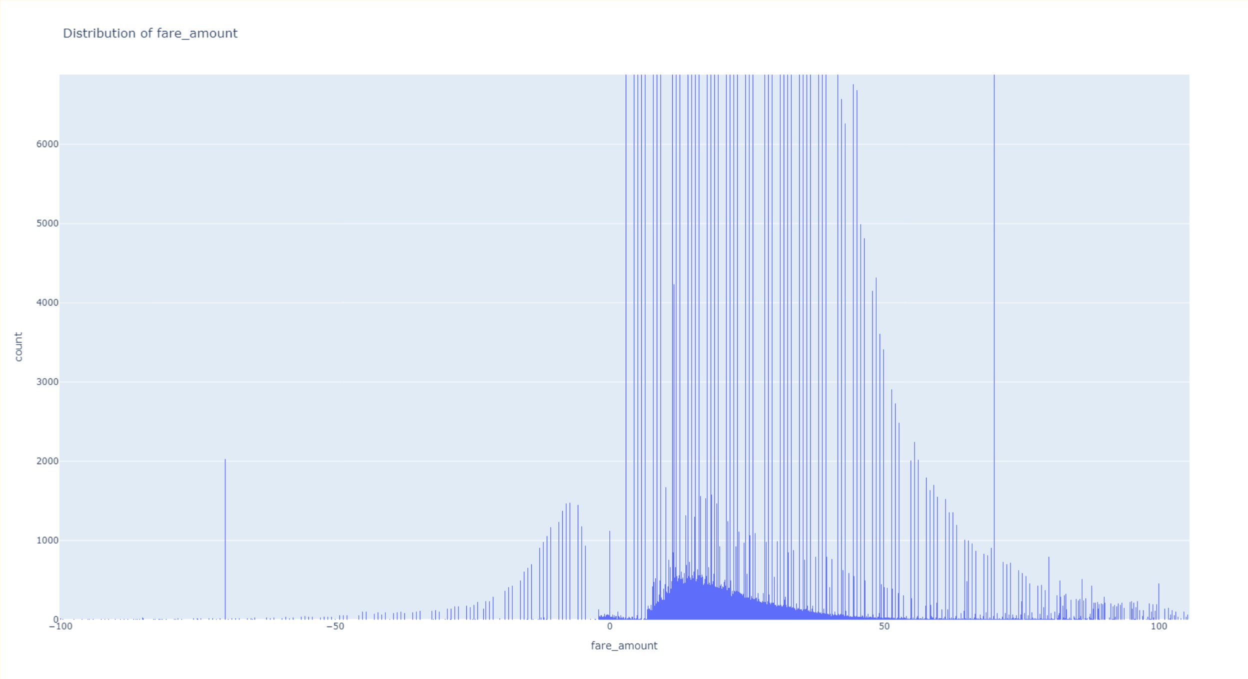


Fig 3: Plot of distribution of Fare amount zoomed in

postgres and grafana

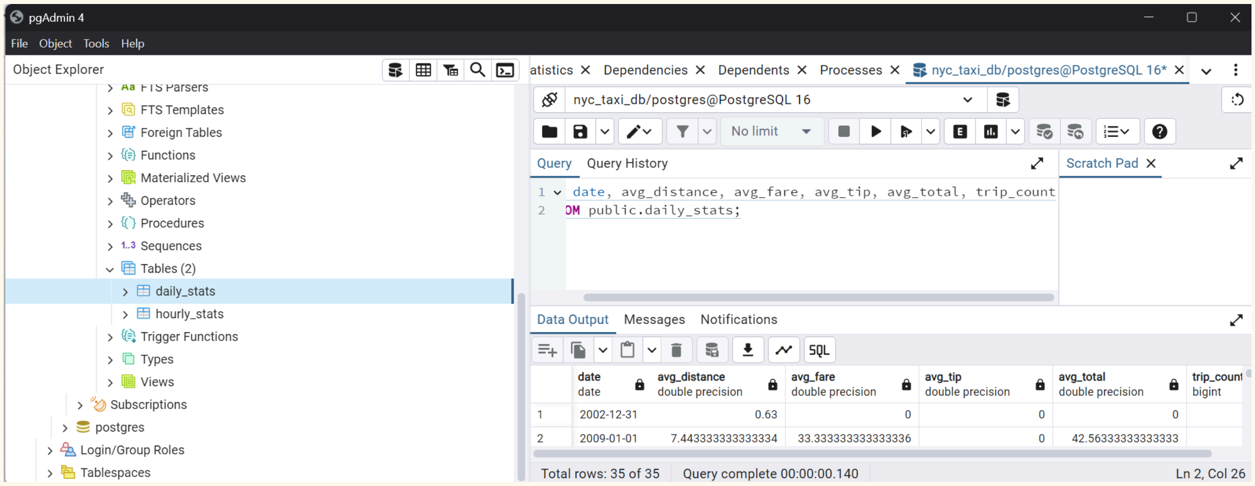
step 13. I installed postgresql as i mentioned in the first steps. I created a database named nyc_taxi_db from PgAdmin4.

step 14. I created a python script which created two tables for hourly stats and daily stats in the postgres table

```
import pandas as pd
from sqlalchemy import create_engine
import os


def load_data(file_path):
    """
    Load data from a Parquet file.
```

```
    Args:
        file_path (str): Path to the Parquet file.

    Returns:
        pd.DataFrame: Loaded DataFrame.
    """
    return pd.read_parquet(file_path, engine='pyarrow')


def aggregate_data(df):
    """
    Aggregate the taxi data into hourly and daily statistics.

    Args:
        df (pd.DataFrame): Input DataFrame containing taxi trip data.

    Returns:
        tuple: A tuple containing two DataFrames (hourly_stats, daily_stats).
    """
    # Hourly statistics
    df['hour'] = pd.to_datetime(df['tpep_pickup_datetime']).dt.hour
    hourly_stats = df.groupby('hour').agg({
        'trip_distance': 'mean',
        'fare_amount': 'mean',
        'tip_amount': 'mean',
        'total_amount': 'mean',
        'tpep_pickup_datetime': 'count'
    }).reset_index()
    hourly_stats.columns = ['hour', 'avg_distance', 'avg_fare', 'avg_tip',
'avg_total', 'trip_count']

    # Daily statistics
    df['date'] = pd.to_datetime(df['tpep_pickup_datetime']).dt.date
    daily_stats = df.groupby('date').agg({
        'trip_distance': 'mean',
        'fare_amount': 'mean',
        'tip_amount': 'mean',
        'total_amount': 'mean',
        'tpep_pickup_datetime': 'count'
    }).reset_index()
    daily_stats.columns = ['date', 'avg_distance', 'avg_fare', 'avg_tip',
'avg_total', 'trip_count']

    return hourly_stats, daily_stats
```

```python
def load_to_postgres(data, table_name, engine):
    """
    Load a DataFrame into a PostgreSQL table.

    Args:
        data (pd.DataFrame): DataFrame to be loaded into PostgreSQL.
        table_name (str): Name of the table to be created or replaced.
        engine (sqlalchemy.engine.base.Engine): SQLAlchemy engine for database
connection.
    """
    data.to_sql(table_name, engine, if_exists='replace', index=False)


def main():
    """
    Main function to load taxi data, aggregate it, and store in PostgreSQL.
    """
    # Path to the Parquet file containing taxi data
    file_path = 'data/yellow_tripdata_2024-01.parquet'

    # Load the taxi data
    df = load_data(file_path)

    # Aggregate the data into hourly and daily statistics
    hourly_stats, daily_stats = aggregate_data(df)

    # PostgreSQL connection details
    db_user = 'postgres'
    db_pass = '1234'
    db_host = 'localhost'
    db_name = 'nyc_taxi_db'

    # Create SQLAlchemy engine for PostgreSQL connection
    engine = create_engine(f'postgresql://{db_user}:{db_pass}@{db_host}/{db_name}')

    # Load aggregated data into PostgreSQL
    load_to_postgres(hourly_stats, 'hourly_stats', engine)
    load_to_postgres(daily_stats, 'daily_stats', engine)

    print("Data successfully loaded into PostgreSQL.")


if __name__ == "__main__":
    main()
```




```sql
SELECT date, avg_distance, avg_fare, avg_tip, avg_total, trip_count
     FROM public.daily_stats;
```

	date date	avg_distance double precision	avg_fare double precision	avg_tip double precision	avg_total double precision	trip_count bigint
1	2002-12-31	0.63	0	0	0	2
2	2009-01-01	7.443333333333334	33.333333333333336	0	42.56333333333333	3
3	2023-12-31	2.601	14.41	2.627	22.462	10
4	2024-01-01	4.396813968128572	21.788021058348658	3.262249762383815	30.15371915618481	81013
5	2024-01-02	4.119030575087065	20.967073451714136	3.5204303552748315	30.220156781737046	75519
6	2024-01-03	3.878400402780642	19.664022589685416	3.3669137539883773	28.602138619627063	82427
7	2024-01-04	3.310968989611374	18.42131349549567	3.3018056189930123	27.215586437449588	102901

Fig : Daily Stats table created from python script in postgres

Query Query History Scratch

```sql
select * FROM public.hourly_stats
```

Data Output Messages Notifications

	hour integer	avg_distance double precision	avg_fare double precision	avg_tip double precision	avg_total double precision	trip_count bigint
1	0	3.732849647255165	19.202658103016663	3.3265433534781397	27.770447568715703	79094
2	1	3.1272586570197847	17.527295765192907	2.9338989688030286	25.378651425587854	53627
3	2	2.8839843271050456	16.482882160087428	2.7682458618759496	24.084306314470773	37517
4	3	3.3212776591028175	18.150134617709888	2.816332675023175	25.9059977429366	24811
5	4	4.545094373432087	22.51864532313941	3.1329727631107396	30.990660016724405	16742
6	5	8.638056917501599	26.61991846088254	3.8220160946493285	36.225783415050095	18764
7	6	12.865668251707742	21.650398995872457	3.1274633710685755	29.51609548866736	41429
8	7	6.003182551153262	18.53918011443042	3.065031952125563	26.13971726848147	83719
9	8	5.441060328131798	17.654682746205495	3.0487878917148	25.1992645615951	117209
10	9	2.998480111653873	17.70841536791502	3.1504820500891677	25.497907032643248	128970
11	10	3.2744914900056203	17.75339376558244	3.158458401187508	25.576706898787993	138778
12	11	3.416894554343638	17.432059425276666	3.10856080030822	25.1135792004889	150542
13	12	3.286957139992343	17.476313419503036	3.129426224029072	25.217100249758445	164559
14	13	3.1106561979482406	18.116046450033256	3.2441704972837444	26.02349852562933	169903
15	14	3.590374416341349	18.945786722654155	3.399381950595414	27.153221577053877	182898

Fig: Hourly stats table created in postgres from python

Grafana

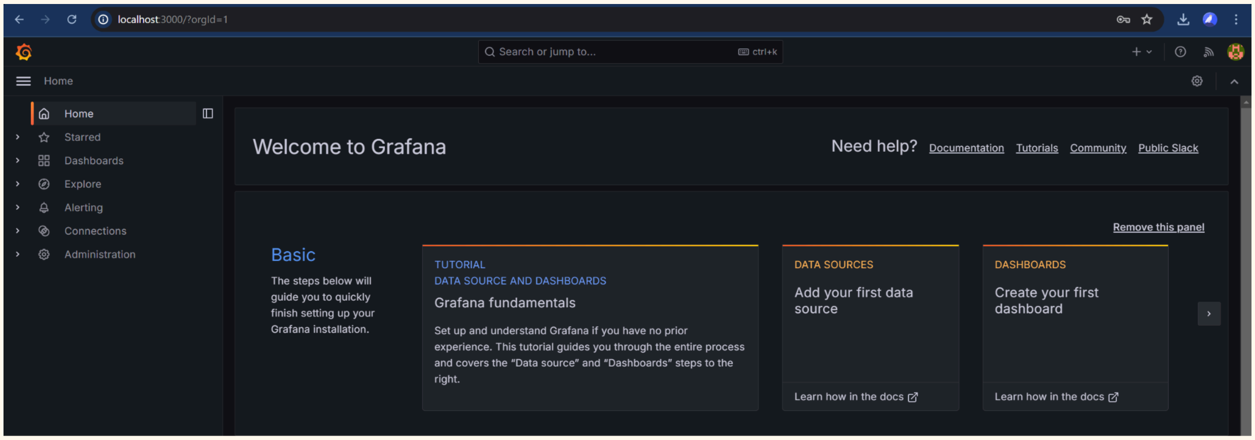


Fig: grafana landing page

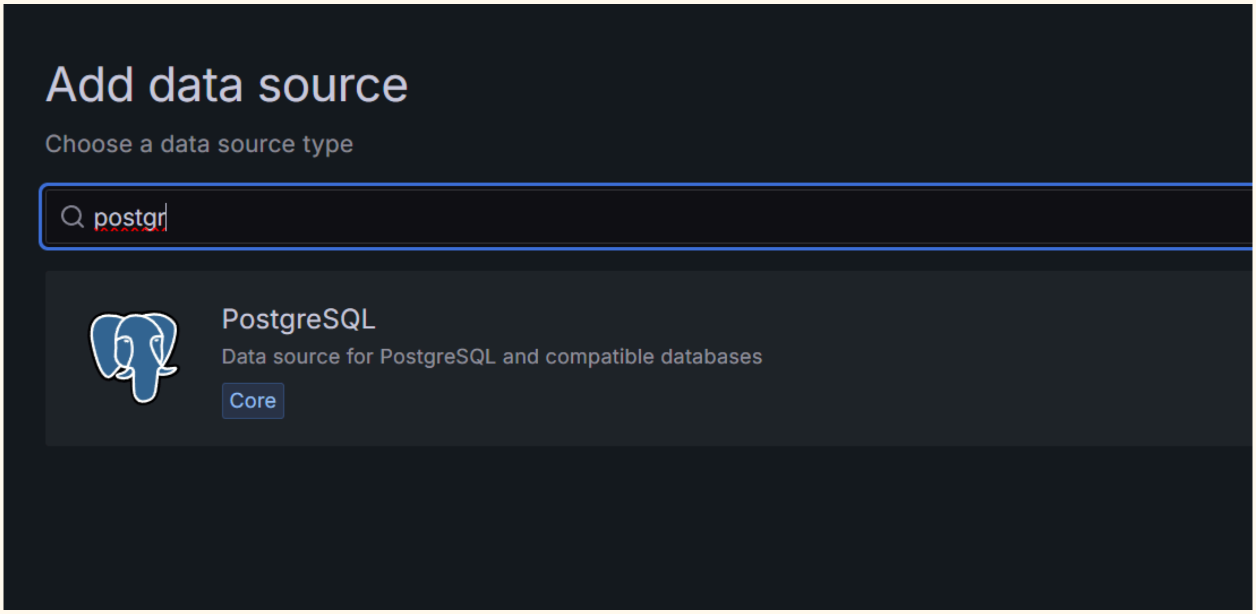


Fig: adding postgres source

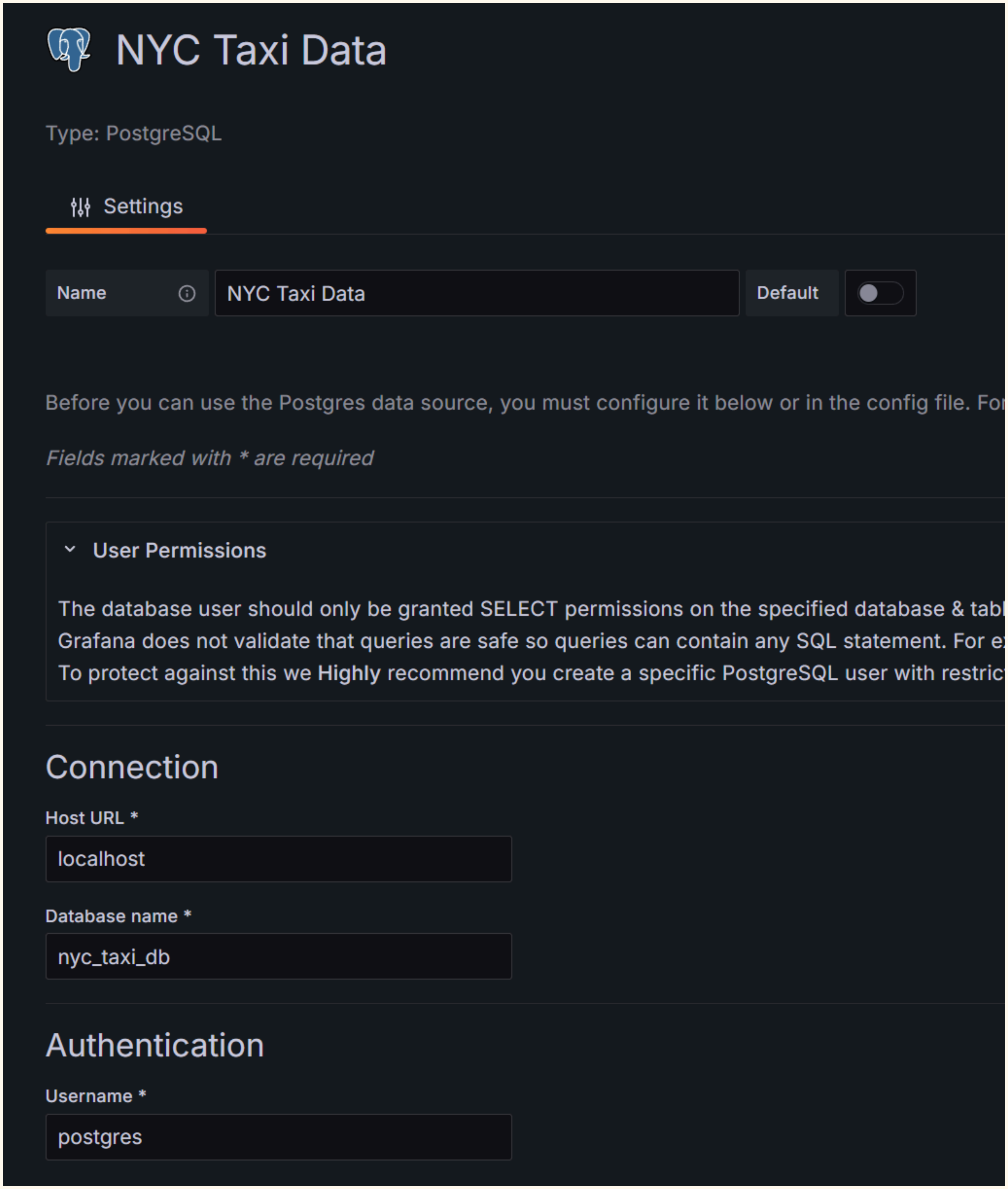
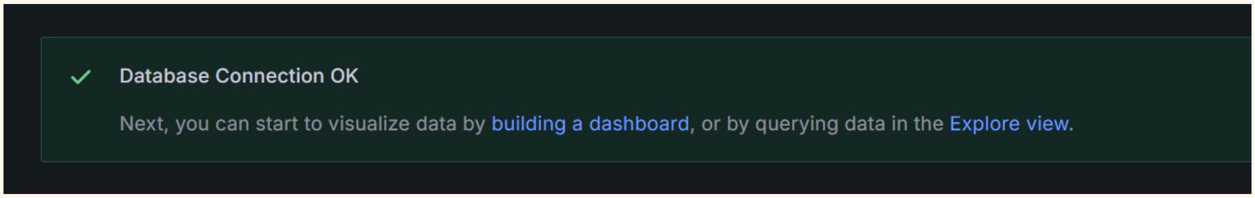
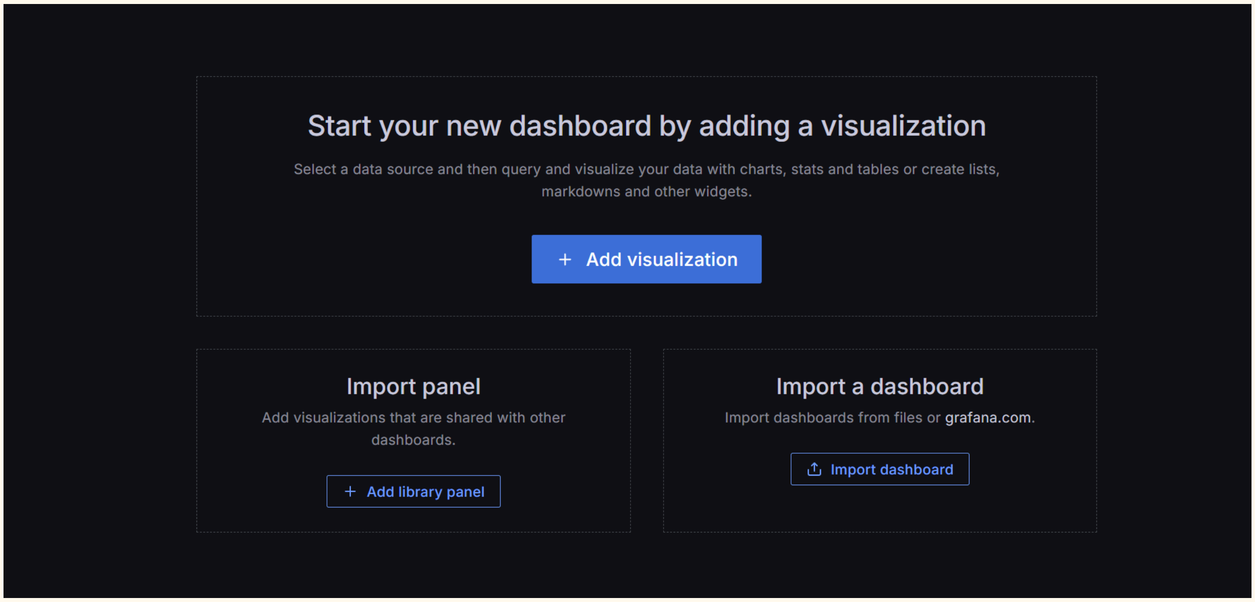


Fig: Adding our db creds





Creating a new dashboard

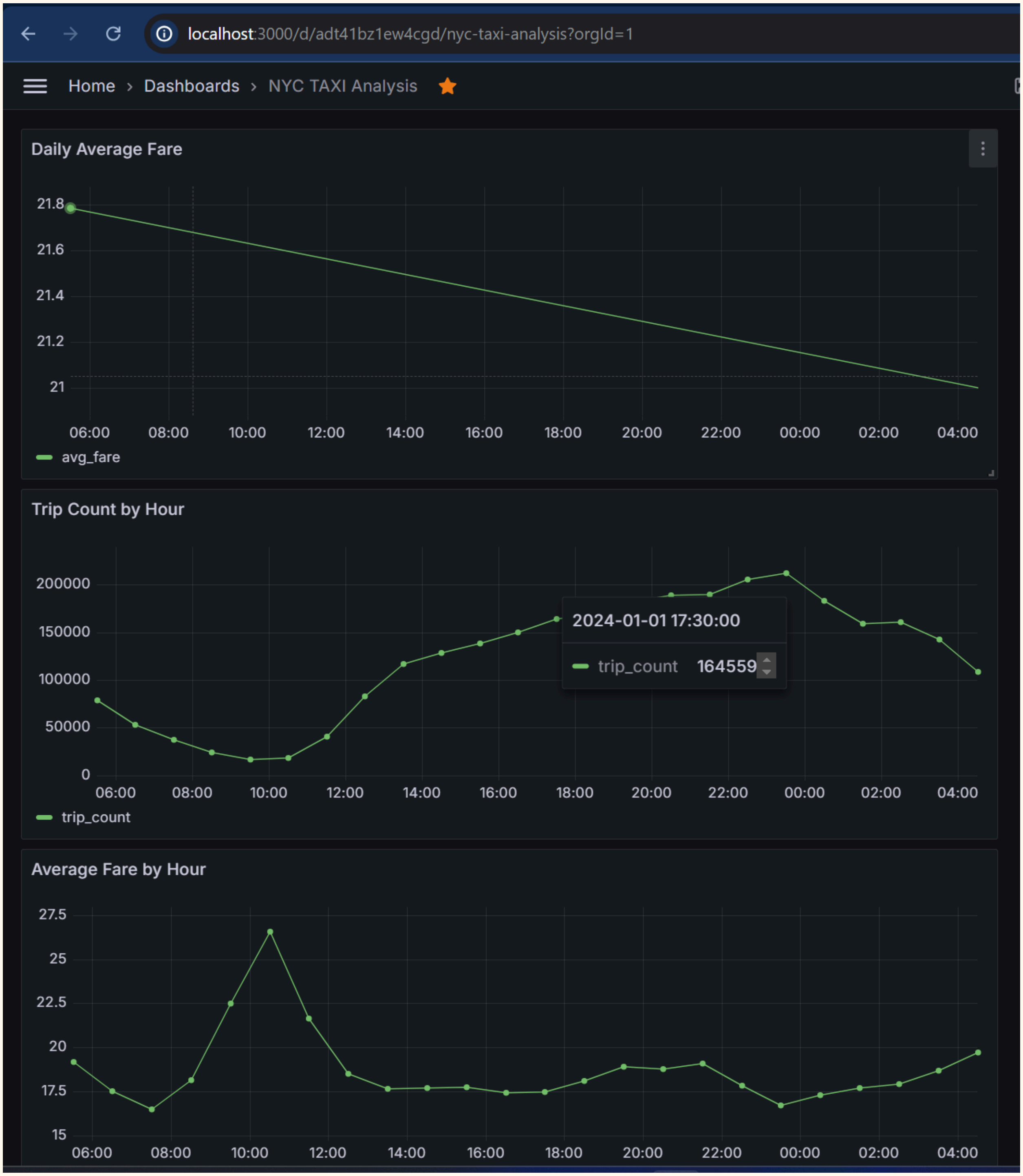


Fig: My final grafana dashboard