Dataset Assessment and Interest Report

Enhancing NYC Taxi Services: Analyzing March 2024 Yellow Taxi Trip Data to Optimize Urban Mobility and Passenger Experience

Sample Size: [4 Million Data]

Submitted by: [PAN SHENGXIN]

Date: [6/13/2024]

Word Counts: [5061]

Dataset Assessment and Interest in NYC Taxi & Limousine Commission 2024 March Yellow Taxi Trip Data

Dataset Assessment

Quality

Reliability:

The NYC TLC is a reputable source, providing **a wealth of data collected by the city government.** This data is reliable because it is systematically collected and maintained by the Taxi and Limousine Commission (TLC). The dataset includes trip records from the city's yellow taxis, with attributes such as pickup and drop-off dates/times, locations, trip distances, itemized fares, and payment types. The oversight provided by a governmental body further underscores the dependability of this data.

Judgement Basis:

Reliability is judged based on several factors. **First of all, the reputation of the NYC TLC platform as a trustworthy source is a key indicator.** This platform is recognized for its high standards of data quality and accessibility, managed by the New York City government, which is committed to transparency and the dissemination of valuable public information. **Secondly, the systematic collection and maintenance process by the TLC ensures the data's integrity.** The data collection involves the use of advanced digital systems built into every taxi, which automatically record trip details such as pickup and drop-off locations, times, distances, and fare components. Regular audits and validations conducted by the TLC maintain data integrity by identifying and rectifying any anomalies or inaccuracies. **Lastly, the transparency and public availability of the data promote accountability, enhancing its reliability.** By making data publicly available, the platform allows for anyone to conduct independent verification and analysis, fostering a culture of accountability and confidence in the data's accuracy.

Detail

Amount of Detail:

The dataset includes comprehensive details for each taxi trip. It encompasses ***pickup and drop-off dates and times, geographic coordinates for pickup and drop-off locations, trip duration, distance, itemized fare components, and payment methods***. The level of detail presented in the data supports a wide range of analyses, from operational optimization to strategic transportation planning.

Usefulness:

The granularity of this data **enables in-depth exploration of taxi usage patterns, fare structures, and spatial-temporal trends.** *However there are some limitation to the data as such detailed information is invaluable for applications like transportation planning, economic analysis, and improving operational efficiency.*

Documentation

Clarity of Data Meaning:

The dataset is well-documented, with a detailed data dictionary available on the NYC TLC website. Each field is clearly described, including data types, possible values, and explanations of each field's meaning. The documentation covers 19 columns, *including crucial fields such as VendorID, trip_distance, tpep_pickup_datetime, tpep_dropoff_datetime, total_amount, tolls_amount, tip_amount, store_and_fwd_flag, RatecodeID, PULocationID, payment_type, passenger_count, mta_tax, improvement_surcharge, fare_amount, congestion_surcharge, and airport_fee.* **The clear descriptions provided ensure that users can accurately interpret the data and utilize it effectively.**

Location and Ease of Finding Documentation:

The documentation is available directly on the dataset's webpage, making it easily accessible.

Interrelation

Connecting to Other Datasets:

The taxi trip data can be valuable when integrated with other datasets such as *weather data, public transportation schedules, and socio-economic data.* For example, weather data can help to assess the impact of weather conditions on taxi demand, public transportation data on the other hand can analyze the inter relation between taxi usage and public transit availability, and socio-economic data can examine correlations between taxi usage and different economic factors.

Ease of Connection:

Integration with other datasets can be facilitated by common fields such as geographic coordinates and timestamps. **However, data cleaning and normalization might be required to ensure seamless integration, which can be challenging due to the vast amount of data that needs to be handled.** Furthermore, finding useful relationships, trends and evaluation would require a professional team of data analytics. **These factors increases the difficulty to connect to other data bases, but can be considered manageable given enough period of time.**

Use

The dataset can be used in numerous applications. **In transportation planning,** this data can help to understand taxi usage patterns to inform city transportation policies and infrastructure investments. **For economic analysis**, fare data can be examined to assess the economic impact of taxi services. **Operational insights** can also be gained by analyzing trip and fare data, finding their relationship which will lead to improvements in fleet management.

Key questions that I have concluded include **how taxi trip volumes vary by time of day and location**, **what are the most common payment methods** and **how they correlate with trip distance or fare**, and **how weather conditions affect taxi demand**. These questions helps to better understand a database and its approach as they involve complex queries and data relationships that are best handled through database systems rather than simple spreadsheet sorting or statistical methods.

On the other hand, there are some potentially useful information not included in the dataset, such as specific reasons for trip cancellations. However, the richness of the existing data still allows for comprehensive analyses.

Discoverability

The NYC TLC platform is user-friendly and well-organized, making it relatively straightforward to find datasets within the transportation domain. The dataset is also easily obtainable through the platform or other data repository websites due to its popularity and vast research value.

There are several alternative datasets available on the NYC TLC platform, covering various aspects of transportation. This variety allows for comparative analyses and further enriches research opportunities.

Here are the Sources Visited when finding the data:

- NYC TLC website
- Search engines and data repository websites

Licensing and Terms of Use

The data is provided under the NYC TLC terms of use, which generally permit free use for analysis, research, and application development, provided proper attribution is given. The primary restriction is that the requirement to attribute the source of the data, which does not significantly limit usage but must acknowledge and clearly state where the data came from. **Users must adhere to lawful purposes when accessing and using the data. Prohibited activities include committing criminal offenses, causing civil liability, impersonating others, uploading harmful content, altering or disrupting the site, and misrepresenting affiliations. The City reserves the right to suspend access if these terms are violated, with notice provided to the user.**

Key limitations to my analysis

One of the significant limitations of my analysis is the size of the dataset. The NYC Open Data 2024 Yellow Taxi Trip Data contains over 39 million rows, making it extremely large and complex to process within the limited time frame available. Due to this constraint, I had no choice but to narrow the scope of my analysis to focus only on the data for March.

Impact of Focusing Only on March Data

Seasonal and Monthly Variations:
Focusing solely on March may result in the analysis does not accounting for seasonal variations or monthly trends. Taxi usage can fluctuate significantly throughout the year due to many factors such as weather changes, holidays, and events. By limiting the data to March, the analysis will miss out on understanding these broader patterns, which could provide a more comprehensive view of taxi demand and usage.

Limited Sample Size:
While March data alone still represents a substantial number of trips, it is a smaller sample compared to the entire dataset. This limited sample might not capture the full diversity of taxi trips, leading to less robust insights. Certain trends that are apparent in the complete dataset might not be visible in a single month's data.

Interest and Questions

My Interest in the Dataset

Analyzing transportation patterns in NYC to improve on taxi services provided. The NYC TLC 2024 Yellow Taxi Trip Data presents an invaluable resource for understanding the transportation dynamics of one of the busiest cities in the world. The richness of the dataset offer a unique opportunity to dive into various aspects of urban mobility, which is very crucial for improving taxi services in NYC. By examining this data, useful insights of passenger behavior, operational efficiency, and external factors that are potentially affecting taxi usage can be concluded. These conclusions can then be used to optimize taxi services, improve customer satisfaction, and inform city transportation policies.

Why This Dataset is Interesting

This data set particularly caught my interest due to its detailed coverage of taxi trips, including temporal and spatial information, fare details, and payment methods. It enables a multifaceted analysis of transportation patterns, which can reveal trends and correlations that are not immediately observable. The ability to analyze such data can lead to practical improvements in taxi service management and urban transportation planning.

Questions to Ask the Dataset

What are the peak hours for taxi demand? and How does the average fare amount vary?
By analyzing the timestamps of taxi pickups and drop-offs, it will help to identify the times of day when taxi demand is highest. This information is crucial for optimizing taxi availability and managing fleet operations to meet customer needs efficiently.

Which areas have the highest demand for taxis?
Geospatial analysis of pickup and drop-off locations can highlight areas with the highest concentration of taxi activity. Understanding these hotspots can help in strategically locating more taxis to certain areas, which can help to reduce wait times and improve service coverage.

How does weather affect taxi usage?
By correlating taxi trip data with weather conditions, exploring how different weather scenarios (e.g., rain, snow) influence taxi demand. This analysis can aid in predicting demand spikes and preparing for adverse weather conditions, ensuring reliable service during such events.

Research Questions and Justification for a Database Approach

What are the peak hours for taxi demand? Analyzing timestamps of taxi pickups and drop-offs can help to identify peak demand hours. This question involves complex queries across large datasets to aggregate and analyze data by time intervals, **which is best suited for a database approach. Unlike spreadsheets and csv, databases can efficiently handle the high volume of data and perform advanced time-series analysis.**

Which areas have the highest demand for taxis? Geospatial analysis of pickup and drop-off locations helps determine areas with the highest taxi activity. This requires spatial queries and indexing capabilities provided by databases, which are not feasible with other file formats. **Databases can store and process large amounts of geospatial data, allowing for efficient location-based analysis and visualization.**

How do payment methods correlate with trip distance or fare? Exploring the relationship between payment methods and trip attributes requires joining multiple fields and performing aggregation operations. A database approach can efficiently manage these operations, especially when dealing with thousands or millions of records. **Databases support complex queries that can reveal patterns and correlations, which would be cumbersome in a spreadsheet.**

Data Modeling

E/R Model

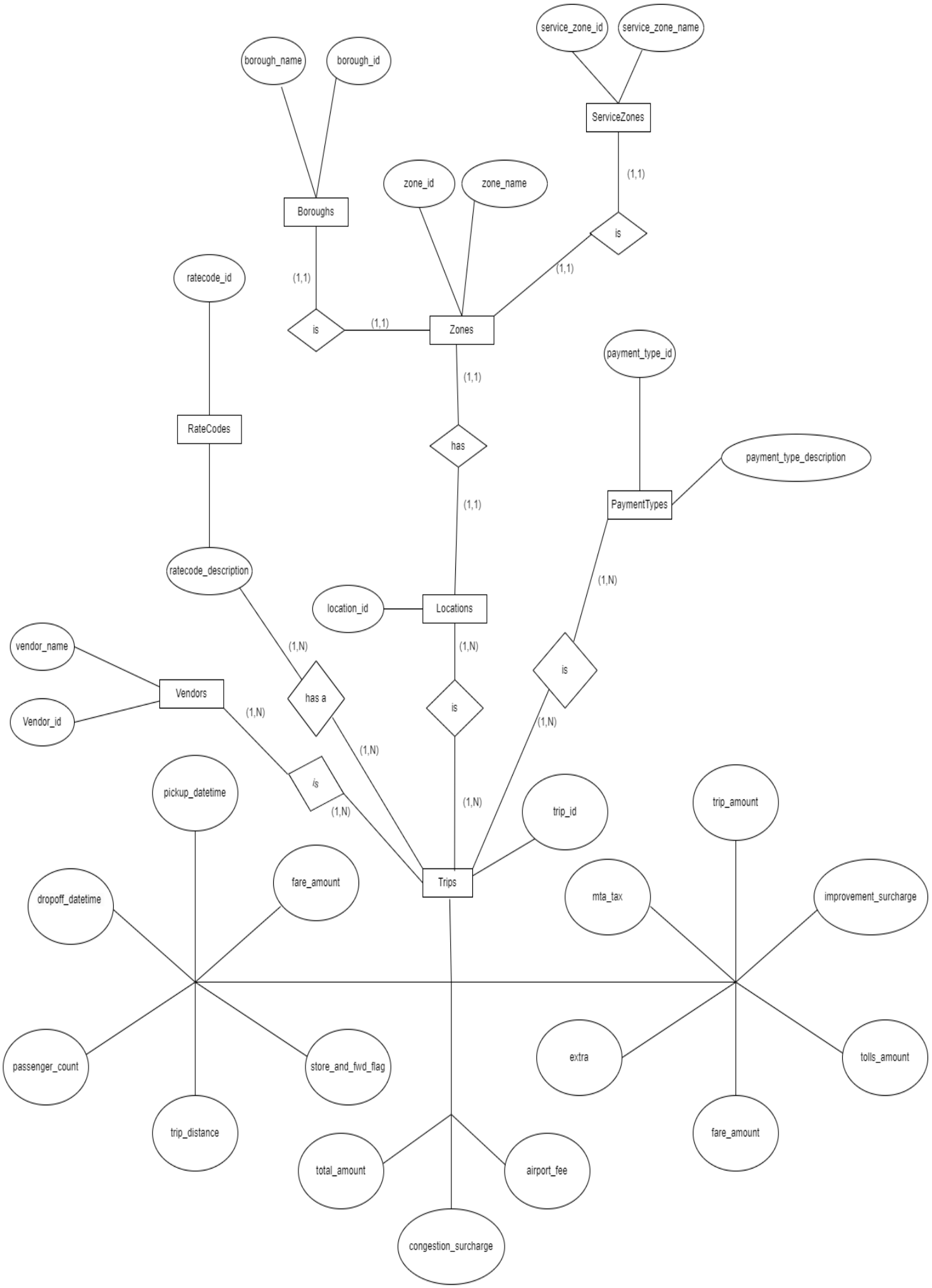
Before drawing diagrams, as the dataset does not include all the information, I had created the completed dataset based on the documentation they had provided on their website. The following table will be created: Locations *(this is then normalized into Boroughs, ServiceZones, Zones and Locations)*, Vendors, PaymentTypes, RateCodes, Weather *(this is then normalized into Weather, Datetimes, Icon and Conditions)*

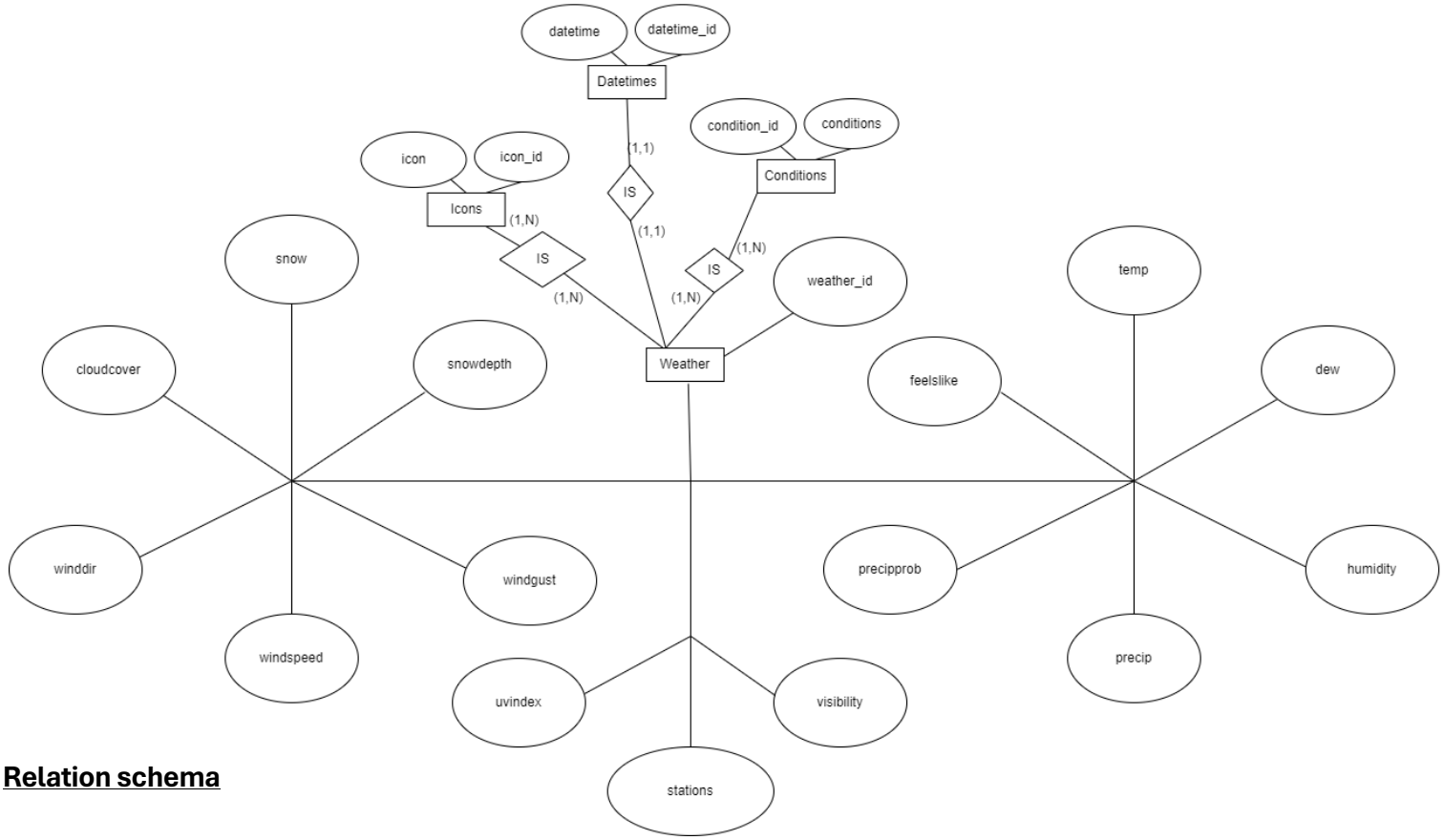
Entities and Fields:

- **Boroughs**: Includes borough_id (Primary Key) and borough_name.
- **ServiceZones**: Contains service_zone_id (Primary Key) and service_zone_name.
- **Zones**: Comprises zone_id (Primary Key), zone_name, borough_id (Foreign Key referencing Boroughs), and service_zone_id (Foreign Key referencing ServiceZones).
- **Locations**: Consists of location_id (Primary Key) and zone_id (Foreign Key referencing Zones).
- **Vendors**: Includes vendor_id (Primary Key) and vendor_name.
- **PaymentTypes**: Contains payment_type_id (Primary Key) and payment_type_description.
- **RateCodes**: Comprises ratecode_id (Primary Key) and ratecode_description.
- **Trips**: Consists of trip_id (Primary Key), vendor_id (Foreign Key referencing Vendors), pickup_datetime, dropoff_datetime, passenger_count, trip_distance, ratecode_id (Foreign Key referencing RateCodes), store_and_fwd_flag, pickup_location_id (Foreign Key referencing Locations), dropoff_location_id (Foreign Key referencing Locations), payment_type (Foreign Key referencing PaymentTypes), fare_amount, extra, mta_tax, tip_amount, tolls_amount, improvement_surcharge, total_amount, congestion_surcharge, and airport_fee.
- **Datetimes**: Includes datetime_id (Primary Key) and datetime.
- **Weather**: Consists of weather_id (Primary Key), datetime_id (Foreign Key referencing Datetimes), temp, feelslike, dew, humidity, precip, precipprob, preciptype, snow, snowdepth, windgust, windspeed, winddir, sealevelpressure, cloudcover, visibility, solarradiation, solarenergy, uvindex, severerisk, conditions_id (Foreign Key referencing Conditions), icon_id (Foreign Key referencing Icons), and stations.
- **Conditions**: Contains conditions_id (Primary Key) and conditions.
- **Icons**: Comprises icon_id (Primary Key) and icon.

Entity-Relationship Diagram (Weather and Trips)

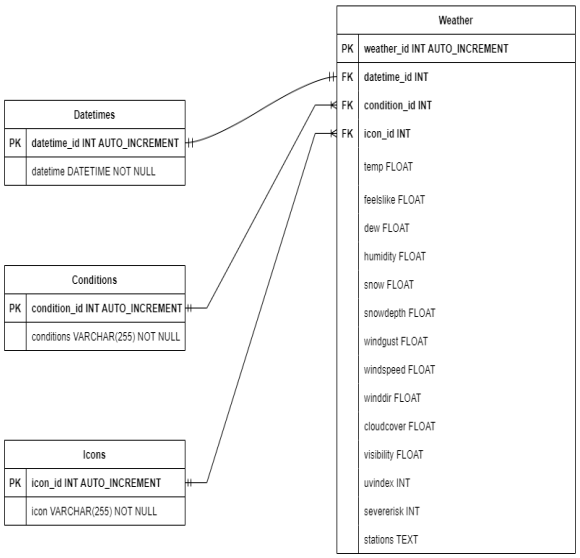
The E/R diagram utilizes ellipses for attributes, rectangles for entities, and rhombuses for relationships. The diagram visually represents the relationships between the entities described above.


service_zone_id
service_zone_name
borough_name
borough_id
ServiceZones
(1,1)
zone_id
zone_name
Boroughs
is
(1,1)
ratecode_id
(1,1)
is
(1,1)
Zones
(1,1)
payment_type_id
RateCodes
has
payment_type_description
(1,1)
PaymentTypes
ratecode_description
(1,N)
location_id
Locations
vendor_name
(1,N)
(1,N)
is
Vendors
has a
is
Vendor_id
(1,N)
(1,N)
(1,N)
is
(1,N)
pickup_datetime
trip_id
trip_amount
(1,N)
dropoff_datetime
fare_amount
Trips
mta_tax
improvement_surcharge
passenger_count
store_and_fwd_flag
extra
tolls_amount
trip_distance
fare_amount
total_amount
airport_fee
congestion_surcharge



Relation schema

Minor Tables Essential for reasearch which are not linked to the Main Tables



Main Tables

Justification of normalizing, Analysis and Normalized forms

Justification of normalizing Locations table

Issues with Initial State:

The initial state presents several issues. Redundancy is a major concern, as each borough name and service zone name is repeated for many locations. This leads to update anomalies, where changing the name of a borough or service zone would necessitate multiple updates. Insert anomalies also arise, requiring duplication of borough and service zone names when adding a new location.

Justification of normalizing Weather table

Issues with Initial State:

The initial state has several issues. Redundancy is evident, as conditions and icons are repeated for each weather entry. This leads to update anomalies, where changing the condition or icon would necessitate multiple updates. Insert anomalies arise, requiring duplication of conditions and icons when adding new weather data.

Normalization Analysis and Normalized forms

Boroughs Table

The Boroughs table, which includes borough_id and borough_name, is already in First Normal Form (1NF) as all columns contain atomic values and there are no repeating groups. It is also in Second Normal Form (2NF) because it has a primary key (borough_id), and all non-key attributes are fully dependent on the primary key. Additionally, it satisfies Third Normal Form (3NF) because there are no transitive dependencies.Since borough_id is the only candidate key, the table is also in Boyce-Codd Normal Form (BCNF). **Thus, the Boroughs table is in 3NF and BCNF.**

ServiceZones Table

The ServiceZones table, containing service_zone_id and service_zone_name, meets the requirements for First Normal Form (1NF) by containing atomic values without repeating groups. It is in Second Normal Form (2NF) as the primary key is service_zone_id, and all non-key attributes depend entirely on this primary key. Furthermore, it is in Third Normal Form (3NF) because there are no transitive dependencies.With service_zone_id as the only candidate key, the table also meets Boyce-Codd Normal Form (BCNF).**Thus, the ServiceZones table is in 3NF and BCNF.**

Zones Table

The Zones table, which consists of zone_id, zone_name, borough_id, and service_zone_id, is in First Normal Form (1NF) because it has atomic values and no repeating groups. It is in Second Normal Form (2NF) because it has a primary key (zone_id), and all non-key attributes depend fully on this primary key. It also meets Third Normal Form (3NF) criteria because there are no transitive dependencies; zone_name, borough_id, and service_zone_id all depend directly on zone_id. Since zone_id is the only candidate key, this table is also in Boyce-Codd Normal Form (BCNF).**Therefore the Zones table is in 3NF and BCNF.**

Locations Table

The Locations table, containing location_id and zone_id, is in First Normal Form (1NF) as it has atomic values and no repeating groups. It is in Second Normal Form (2NF) with a primary key (location_id), and all non-key attributes (zone_id) are fully dependent on the primary key. This table also meets Third Normal Form (3NF) criteria because there are no transitive dependencies. Since location_id is the sole candidate key, it is also in Boyce-Codd Normal Form (BCNF).**Thus, the Locations table is in 3NF and BCNF.**

Vendors Table

The Vendors table, comprising vendor_id and vendor_name, is in First Normal Form (1NF) because it contains atomic values without repeating groups. It meets Second Normal Form (2NF) criteria with a primary key (vendor_id), and all non-key attributes are fully dependent on the primary key. The table is in Third Normal Form (3NF) because there are no transitive dependencies. Since vendor_id is the only candidate key, the table is also in Boyce-Codd Normal Form (BCNF).**The Vendors table is in 3NF and BCNF.**

PaymentTypes Table

The PaymentTypes table, with payment_type_id and payment_type_description, is in First Normal Form (1NF) as it contains atomic values without repeating groups. It is in Second Normal Form (2NF) since the primary key (payment_type_id) fully determines the non-key attributes. The table meets Third Normal Form (3NF) because there are no transitive dependencies; payment_type_description depends directly on payment_type_id. Since payment_type_id is the only candidate key, the table is also in Boyce-Codd Normal Form (BCNF). **Thus the PaymentTypes table is in 3NF and BCNF.**

RateCodes Table
The RateCodes table, consisting of ratecode_id and ratecode_description, is in First Normal Form (1NF) because it has atomic values and no repeating groups. It meets Second Normal Form (2NF) criteria with a primary key (ratecode_id), and all non-key attributes are fully dependent on the primary key. The table is in Third Normal Form (3NF) because there are no transitive dependencies; ratecode_description depends directly on ratecode_id. Since ratecode_id is the only candidate key, the table is also in Boyce-Codd Normal Form (BCNF). **Therefore the RateCodes table is in 3NF and BCNF.**

Trips Table
The Trips table, which includes multiple fields are in First Normal Form (1NF) because it has atomic values and no repeating groups. It meets Second Normal Form (2NF) criteria as it has a primary key (trip_id), and all non-key attributes are fully dependent on the primary key. The table is in Third Normal Form (3NF) because there are no transitive dependencies. Since trip_id is the only candidate key, the table is also in Boyce-Codd Normal Form (BCNF). **Thus Trips table is in 3NF and BCNF.**

Datetimes Table
The Datetimes table, containing datetime_id and datetime, is in First Normal Form (1NF) because it has atomic values without repeating groups. It meets Second Normal Form (2NF) criteria with a primary key (datetime_id), and the non-key attribute (datetime) is fully dependent on the primary key. The table is in Third Normal Form (3NF) because there are no transitive dependencies. Since datetime_id is the only candidate key, the table is also in Boyce-Codd Normal Form (BCNF). **The Datetimes table is in 3NF and BCNF.**

Weather Table
The Weather table is in First Normal Form (1NF) because it has atomic values and no repeating groups. It meets Second Normal Form (2NF) criteria as it has a primary key (weather_id), and all non-key attributes are fully dependent on the primary key. The table is in Third Normal Form (3NF) because there are no transitive dependencies. Since weather_id is the only candidate key, the table is also in Boyce-Codd Normal Form (BCNF). **Therefore the Weather table is in 3NF and BCNF.**

Justification for Normalizing Further

While the database tables have been normalized to BCNF, higher forms of normalization, such as Fourth Normal Form (4NF) and Fifth Normal Form (5NF), were also being considered. However, due to the fact that these higher normal forms address specific types of anomalies that do not apply to the current dataset, I did not normalize these dataset further.

Fourth Normal Form (4NF):

- 4NF deals with multi-valued dependencies, where one attribute in a table can have multiple independent values. In my dataset, there are no multi-value dependencies, as each attribute depends only on the primary key. Normalizing to 4NF would not provide additional benefits and could complicate the schema unnecessarily.

Fifth Normal Form (5NF):

- 5NF addresses join dependencies and ensures that every join dependency is implied by the candidate keys. Given the nature of my dataset, the relationships between entities are already well-defined and do not require further decomposition. Normalizing to 5NF is not necessary and would not provide practical benefits.

Data Cleaning and Creating

Extract, Transform, Load(ETL) Process

- **Extract:** I begin by extracting data from CSV files and create DataFrames.
- **Transform:** Then I clean the dataset to remove some unused data columns and missing data. After the data is being cleaned, I create the necessary DataFrames for each tables and merge them where necessary to ensure the data are being transformed to their respective normalized forms. **Below is a screenshot of one of the transform codes:**

```
# Create Datetime Table data
datetimes = df[['datetime']].drop_duplicates().reset_index(drop=True)
datetimes['datetime_id'] = datetimes.index + 1

# Create Conditions Table data
conditions = df[['conditions']].drop_dup
conditions['condition_id'] = conditions.index + 1

# Create Icons Table data
icons = df[['icon']].drop_duplicates().reset_index(drop=True)
icons['icon_id'] = icons.index + 1


# Merge data to create the Weather Table data
df_merged = df.merge(datetimes, on='datetime').merge(conditions, on='conditions').merge(icons, on='icon').merge(preciptypes, on='preciptype')
df_merged.drop(columns=['datetime', 'conditions', 'icon', 'preciptype'], inplace=True)

# Reorder columns for better readability
weather_columns = ['datetime_id', 'condition_id', 'icon_id'] + [col for col in df_merged.columns if col not in ['datetime_id', 'condition_id', 'icon_id']]
df_weather = df_merged[weather_columns]
df_weather = df_weather.replace({np.nan: None})
df_weather = df_weather.drop(columns=['name'])
df_weather = df_weather.drop(columns=['preciptype_id'])
```

(property) index: Index | MultiIndex

- **Load:** Finally the datasets are being loaded into tables respectively

User configuration

The foundation of our database begins with creation of the database and configuration of user access

Step 1: CREATE DATABASE

```
CREATE DATABASE Taxi;
```

Step 2: Configure user

```
CREATE USER 'researcher'@'localhost' IDENTIFIED BY 'Kk123';
GRANT ALL PRIVILEGES ON Taxi.* TO 'researcher'@'localhost';
FLUSH PRIVILEGES;
mysql -u researcher -p
```

Tables: Creation, Structure and Relationships

Order of creation

1. **Boroughs**: This table is created first because it does not depend on any other table. It stores the borough information which is referenced by the Zones table.
2. **ServiceZones**: This table is created second because it also does not depend on any other table. It stores the service zone information which is referenced by the Zones table.
3. **Zones**: This table depends on both the Boroughs and ServiceZones tables because it includes foreign keys referencing these tables. Hence, it is created after Boroughs and ServiceZones.
4. **Locations**: This table depends on the Zones table because it includes a foreign key referencing the zone_id from the Zones table. Therefore, it is created after Zones.
5. **Vendors**: This table is independent and does not have any foreign key dependencies, so it can be created at this point.
6. **PaymentTypes**: This table is independent and does not have any foreign key dependencies, so it can be created at this point.
7. **RateCodes**: This table is independent and does not have any foreign key dependencies, so it can be created at this point.
8. **Trips**: This table has multiple foreign key dependencies on Vendors, Locations, PaymentTypes, and RateCodes. Therefore, it is created last after all the referenced tables are created.

----------Minor tables----------

9. **Datetimes**: This table is created first because it does not depend on any other table. It stores the datetime information which is referenced by the Weather table.
10. **Conditions**: This table is created second because it also does not depend on any other table. It stores the weather conditions which are referenced by the Weather table.
11. **Icons**: This table is created third because it also does not depend on any other table. It stores the icon information which is referenced by the Weather table.
12. **Weather**: This table depends on the Datetimes, Conditions, and Icons tables because it includes foreign keys referencing these tables. Hence, it is created last after all the referenced tables are created.

Data Types Used

1. **INT**: Used for primary keys and foreign keys. It's a standard integer type.
2. **VARCHAR**: Used for storing variable-length strings. The number in parentheses (e.g., VARCHAR(50)) indicates the maximum length of the string.
3. **DECIMAL**: Used for storing precise decimal numbers. The format DECIMAL(m, n) indicates a number with up to m digits, including n digits after the decimal point.
4. **DATETIME**: Used for storing date and time values.
5. **CHAR**: Used for storing fixed-length strings. The number in parentheses indicates the exact length of the string.
6. **FLOAT:** Used for storing floating-point numbers.
7. **TEXT**: Used for storing large text strings.

SQL Scripts for Table Creation

```
create_boroughs_table = """
CREATE TABLE IF NOT EXISTS Boroughs (
    borough_id INT PRIMARY KEY AUTO_INCREMENT,
    borough_name VARCHAR(50) UNIQUE NOT NULL
);
"""

create_servicezones_table = """
CREATE TABLE IF NOT EXISTS ServiceZones (
    service_zone_id INT PRIMARY KEY AUTO_INCREMENT,
    service_zone_name VARCHAR(50) UNIQUE NOT NULL
);
"""
    service_zone_id INT NOT NULL,
    FOREIGN KEY (borough_id) REFERENCES Boroughs(borough_id),
    FOREIGN KEY (service_zone_id) REFERENCES ServiceZones(service_zone_id)
);
"""

create_locations_table = """
CREATE TABLE IF NOT EXISTS Locations (
    location_id INT PRIMARY KEY,
    zone_id INT NOT NULL,
    FOREIGN KEY (zone_id) REFERENCES Zones(zone_id)
);
"""

create_vendors_table = """
CREATE TABLE IF NOT EXISTS Vendors (
    vendor_id INT PRIMARY KEY,
    vendor_name VARCHAR(50) NOT NULL
);
"""

create_paymenttypes_table = """
CREATE TABLE IF NOT EXISTS PaymentTypes (
    payment_type_id INT PRIMARY KEY,
    payment_type_description VARCHAR(50) NOT NULL
);
"""

create_ratecodes_table = """
CREATE TABLE IF NOT EXISTS RateCodes (
    ratecode_id INT PRIMARY KEY,
    ratecode_description VARCHAR(50) NOT NULL
);
"""

create_trips_table = """
CREATE TABLE IF NOT EXISTS Trips (
    trip_id INT PRIMARY KEY AUTO_INCREMENT,
    vendor_id INT NULL,
    pickup_datetime DATETIME NOT NULL,
    dropoff_datetime DATETIME NOT NULL,
    passenger_count INT NULL,
    trip_distance DECIMAL(9,2) NULL,  -- Maximum value 176836.30
    ratecode_id INT NULL,
    store_and_fwd_flag CHAR(1) NULL,
    pickup_location_id INT NULL,
    dropoff_location_id INT NULL,
    payment_type INT NULL,
    fare_amount DECIMAL(6,2) NULL,  -- Maximum value 900.00
    extra DECIMAL(4,2) NULL,  -- Maximum value 14.25
    mta_tax DECIMAL(4,2) NULL,  -- Maximum value 35.84
    tip_amount DECIMAL(6,2) NULL,  -- Maximum value 999.99
    tolls_amount DECIMAL(6,2) NULL,  -- Maximum value 163.00
    improvement_surcharge DECIMAL(3,2) NULL,  -- Maximum value 1.00
    total_amount DECIMAL(7,2) NULL,  -- Maximum value 1021.99
    congestion_surcharge DECIMAL(3,2) NULL,  -- Maximum value 2.50
    airport_fee DECIMAL(3,2) NULL,  -- Maximum value 1.75,
    FOREIGN KEY (vendor_id) REFERENCES Vendors(vendor_id),
    FOREIGN KEY (pickup_location_id) REFERENCES Locations(location_id),
    FOREIGN KEY (dropoff_location_id) REFERENCES Locations(location_id),
    FOREIGN KEY (payment_type) REFERENCES PaymentTypes(payment_type_id),
    FOREIGN KEY (ratecode_id) REFERENCES RateCodes(ratecode_id)
);
"""
```

```
# Create Datetime Table
cursor.execute("""
CREATE TABLE IF NOT EXISTS Datetimes (
    datetime_id INT AUTO_INCREMENT PRIMARY KEY,
    datetime DATETIME NOT NULL
);
""")
conn.commit()

# Create Conditions Table
cursor.execute("""
CREATE TABLE IF NOT EXISTS Conditions (
    condition_id INT AUTO_INCREMENT PRIMARY KEY,
    conditions VARCHAR(255) NOT NULL
);
""")
conn.commit()

# Create Icons Table
cursor.execute("""
CREATE TABLE IF NOT EXISTS Icons (
    icon_id INT AUTO_INCREMENT PRIMARY KEY,
    icon VARCHAR(255) NOT NULL
);
""")
conn.commit()


# Create Weather Table
cursor.execute("""
CREATE TABLE IF NOT EXISTS Weather (
    weather_id INT AUTO_INCREMENT PRIMARY KEY,
    datetime_id INT,
    condition_id INT,
    icon_id INT,
    temp FLOAT,
    feelslike FLOAT,
    dew FLOAT,
    humidity FLOAT,
    precip FLOAT,
    precipprob FLOAT,
    snow FLOAT,
    snowdepth FLOAT,
    windgust FLOAT,
    windspeed FLOAT,
    winddir FLOAT,
    sealevelpressure FLOAT,
    cloudcover FLOAT,
    visibility FLOAT,
    solarradiation FLOAT,
    solarenergy FLOAT,
    uvindex INT,
    severerisk INT,
    stations TEXT,
    FOREIGN KEY (datetime_id) REFERENCES Datetimes(datetime_id),
    FOREIGN KEY (condition_id) REFERENCES Conditions(condition_id),
    FOREIGN KEY (icon_id) REFERENCES Icons(icon_id)
);
""")
conn.commit()
```

Data Normalization and Cleaning

```
# Load the dataset
file_path = 'NYC_Mar_Weather.csv'
df = pd.read_csv(file_path)

# Normalize Datetime
# Create a unique Datetime table with a unique identifier for each datetime
datetimes = df[['datetime']].drop_duplicates().reset_index(drop=True)
datetimes['datetime_id'] = datetimes.index + 1

# Normalize Conditions
# Create a unique Conditions table with a unique identifier for each condition
conditions = df[['conditions']].drop_duplicates().reset_index(drop=True)
conditions['condition_id'] = conditions.index + 1

# Normalize Icons
# Create a unique Icons table with a unique identifier for each icon
icons = df[['icon']].drop_duplicates().reset_index(drop=True)
icons['icon_id'] = icons.index + 1

# Normalize Preciptypes
# Create a unique Preciptypes table with a unique identifier for each preciptype
preciptypes = df[['preciptype']].drop_duplicates().reset_index(drop=True)
preciptypes['preciptype_id'] = preciptypes.index + 1

# Merge data to create the Weather Table data
# Merge the original data with the normalized tables to replace string values with their respective IDs
df_merged = df.merge(datetimes, on='datetime').merge(conditions, on='conditions').merge(icons, on='icon').merge(preciptypes, on='preciptype')

# Drop the original string columns as they have been replaced by their IDs
df_merged.drop(columns=['datetime', 'conditions', 'icon', 'preciptype'], inplace=True)

# Reorder columns for better readability
# Place the ID columns at the beginning for clarity
weather_columns = ['datetime_id', 'condition_id', 'icon_id'] + [col for col in df_merged.columns if col not in ['datetime_id', 'condition_id', 'icon_id']]
df_weather = df_merged[weather_columns]

# Replace NaN values with None for better handling in databases
df_weather = df_weather.replace({np.nan: None})

# Drop columns that are not needed
df_weather = df_weather.drop(columns=['name'])
df_weather = df_weather.drop(columns=['preciptype_id'])
```

```
try:
    # Load data from CSV files
    taxi_zone_lookup = pd.read_csv('taxi_zone_lookup.csv')

    # Inspect data for null values
    print(taxi_zone_lookup.isnull().sum())

    # Handle null values by filling them with a default value, e.g., 'Unknown'
    taxi_zone_lookup['Borough'].fillna('Unknown', inplace=True)
    taxi_zone_lookup['Zone'].fillna('Unknown', inplace=True)
    taxi_zone_lookup['service_zone'].fillna('Unknown', inplace=True)

    # Normalize Boroughs
    # Insert unique boroughs into the Boroughs table, handling duplicates with ON DUPLICATE KEY UPDATE
    boroughs = taxi_zone_lookup['Borough'].drop_duplicates()
    for borough in boroughs:
        cursor.execute("INSERT INTO Boroughs (borough_name) VALUES (%s) ON DUPLICATE KEY UPDATE borough_name=borough_name", (borough,))

    # Normalize Service Zones
    # Insert unique service zones into the ServiceZones table, handling duplicates with ON DUPLICATE KEY UPDATE
    service_zones = taxi_zone_lookup['service_zone'].drop_duplicates()
    for service_zone in service_zones:
        cursor.execute("INSERT INTO ServiceZones (service_zone_name) VALUES (%s) ON DUPLICATE KEY UPDATE service_zone_name=service_zone_name", (service_zone,))

    # Commit the transactions to save changes
    conn.commit()
    print("Boroughs and ServiceZones tables populated successfully")

    # Fetch ids for Boroughs to create a mapping dictionary
    cursor.execute("SELECT borough_id, borough_name FROM Boroughs")
    borough_dict = {row[1]: row[0] for row in cursor.fetchall()}

    # Fetch ids for ServiceZones to create a mapping dictionary
    cursor.execute("SELECT service_zone_id, service_zone_name FROM ServiceZones")
    service_zone_dict = {row[1]: row[0] for row in cursor.fetchall()}

    # Normalize Zones
    # Insert unique zones into the Zones table, linking them to borough and service zone ids
    zones = taxi_zone_lookup[['Zone', 'Borough', 'service_zone']].drop_duplicates()
    for _, row in zones.iterrows():
        borough_id = borough_dict[row['Borough']]
        service_zone_id = service_zone_dict[row['service_zone']]
        cursor.execute("INSERT INTO Zones (zone_name, borough_id, service_zone_id) VALUES (%s, %s, %s) ON DUPLICATE KEY UPDATE zone_name=zone_name", (row['Zone'], borough_id, service_zone_id))

    # Commit the transactions to save changes
    conn.commit()
    print("Zones table populated successfully")

    # Fetch ids for Zones to create a mapping dictionary
    cursor.execute("SELECT zone_id, zone_name FROM Zones")
    zone_dict = {row[1]: row[0] for row in cursor.fetchall()}

    # Populate Locations table
    # Insert data into Locations table, linking each LocationID to its corresponding zone id
    for _, row in taxi_zone_lookup.iterrows():
        zone_id = zone_dict[row['Zone']]
        cursor.execute("INSERT INTO Locations (location_id, zone_id) VALUES (%s, %s)", (row['LocationID'], zone_id))

    # Commit the transactions to save changes
    conn.commit()
    print("Locations table populated successfully")
except mysql.connector.Error as err:
    print(f"Error: {err}")
```

Data population into tables

```python
def insert_vendors(cursor):
    vendors = {
        1: 'Creative Mobile Technologies, LLC',
        2: 'VeriFone Inc.'
    }
    for vendor_id, vendor_name in vendors.items():
        cursor.execute(
            "INSERT INTO Vendors (vendor_id, vendor_name) VALUES (%s, %s) ON DUPLICATE KEY UPDATE vendor_name=VALUES(vendor_name)",
            (vendor_id, vendor_name)
        )
    conn.commit()

def insert_paymenttypes(cursor):
    payment_types = {
        1: 'Credit Card',
        2: 'Cash',
        3: 'No charge',
        4: 'Dispute',
        5: 'Unknown',
        6: 'Voided trip'
    }
    for payment_type_id, payment_type_description in payment_types.items():
        cursor.execute(
            "INSERT INTO PaymentTypes (payment_type_id, payment_type_description) VALUES (%s, %s) ON DUPLICATE KEY UPDATE payment_type_description=VALUES(payment_type_description)",
            (payment_type_id, payment_type_description)
        )
    conn.commit()

def insert_ratecodes(cursor):
    rate_codes = {
        1: 'Standard rate',
        2: 'JFK',
        3: 'Newark',
        4: 'Nassau or Westchester',
        5: 'Negotiated fare',
        6: 'Group ride'
    }
    for ratecode_id, ratecode_description in rate_codes.items():
        cursor.execute(
            "INSERT INTO RateCodes (ratecode_id, ratecode_description) VALUES (%s, %s) ON DUPLICATE KEY UPDATE ratecode_description=VALUES(ratecode_description)",
            (ratecode_id, ratecode_description)
        )
    conn.commit()

# Insert data into tables
insert_vendors(cursor)
# insert_locations(cursor, taxi_zone_lookup)
insert_paymenttypes(cursor)
insert_ratecodes(cursor)

print("Reference tables populated successfully")
```

```python
try:
    # Disable foreign key checks and non-primary indexes
    cursor.execute("SET foreign_key_checks = 0;")
    cursor.execute("ALTER TABLE Trips DISABLE KEYS;")
    conn.commit()

    # Load data using LOAD DATA INFILE
    load_query = """
    LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/yellow_tripdata_Mar.csv'
    INTO TABLE Trips
    FIELDS TERMINATED BY ','
    ENCLOSED BY '"'
    LINES TERMINATED BY '\n'
    IGNORE 1 ROWS
    (vendor_id, pickup_datetime, dropoff_datetime, passenger_count, trip_distance, ratecode_id, store_and_fwd_flag,
    pickup_location_id, dropoff_location_id, payment_type, fare_amount, extra, mta_tax, tip_amount, tolls_amount,
    improvement_surcharge, total_amount, congestion_surcharge, Airport_fee)
    """

    cursor.execute(load_query)
    conn.commit()
    print("Data loaded successfully")

except mysql.connector.Error as err:
    print(f"Error during data loading: {err}")
    conn.rollback()  # Rollback the transaction on error

finally:
    # Re-enable indexes and foreign key checks
    cursor.execute("ALTER TABLE Trips ENABLE KEYS;")
    cursor.execute("SET foreign_key_checks = 1;")
    conn.commit()
```

```python
# Insert data into Datetimes table
for index, row in datetimes.iterrows():
    cursor.execute("INSERT INTO Datetimes (datetime) VALUES (%s)", (row['datetime'],))
conn.commit()

# Insert data into Conditions table
for index, row in conditions.iterrows():
    cursor.execute("INSERT INTO Conditions (conditions) VALUES (%s)", (row['conditions'],))
conn.commit()

# Insert data into Icons table
for index, row in icons.iterrows():
    cursor.execute("INSERT INTO Icons (icon) VALUES (%s)", (row['icon'],))
conn.commit()


# Insert data into Weather table
for index, row in df_weather.iterrows():
    row_values = (
        row['datetime_id'], row['condition_id'], row['icon_id'], row['temp'], row['feelslike'], row['dew'], row['humidity'], row['precip'],
        row['precipprob'], row['snow'], row['snowdepth'], row['windgust'], row['windspeed'], row['winddir'],
        row['sealevelpressure'], row['cloudcover'], row['visibility'], row['solarradiation'], row['solarenergy'], row['uvindex'],
        row['severerisk'], row['stations']
    )

    # Print the row values before executing the SQL statement
    print(f"Inserting row {index + 1}/{len(df_weather)}: {row_values}")

    # Print the SQL query for debugging
    sql_query = """
    INSERT INTO Weather (datetime_id, condition_id, icon_id, temp, feelslike, dew, humidity, precip, precipprob, snow, snowdepth, windgust, winds
    VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
    """
    print(f"SQL Query: {sql_query}")
    print(f"Parameters: {row_values}")

    try:
        cursor.execute(sql_query, row_values)
    except mysql.connector.Error as e:
        print(f"Error inserting row {index + 1}: {e}")
        break
conn.commit()
```

Reflection on the database

1. **Data Integrity and Consistency**:
   - By normalizing the tables and enforcing foreign key constraints, the database ensures that relationships between tables are maintained correctly. This prevents orphan records and maintains referential integrity across the database. For instance, the separation of boroughs, zones, service zones, and locations into distinct tables ensures that each entity's data remains consistent and dependable.
2. **Reduction of Redundancy**:
   - The normalization process has effectively reduced data redundancy. For example, the initial combined table for boroughs, zones, service zones, and locations had significant redundancy. By splitting these into separate tables and establishing foreign key relationships, the redundancy is minimized, and the database is more efficient in terms of storage and maintenance. This also applies to the weather-related tables where conditions and icons are separated into their own tables.

Queries to address the research questions

What are the peak hours for taxi demand?

```
const query = `
    SELECT HOUR(pickup_datetime) AS hour, COUNT(*) AS count
    FROM Trips
    GROUP BY hour
    ORDER BY count DESC `;
```

How does the average fare amount vary?

```
const query = `
    SELECT HOUR(pickup_datetime) AS hour, AVG(fare_amount) AS avg_fare
    FROM Trips
    GROUP BY hour
    ORDER BY hour`;
```

Which areas have the highest demand for taxis?

```
const query = `
    SELECT Z.zone_name, COUNT(*) AS count
    FROM Trips T
    JOIN Locations L ON T.pickup_location_id = L.location_id
    JOIN Zones Z ON L.zone_id = Z.zone_id
    GROUP BY Z.zone_name
    ORDER BY count DESC
    LIMIT 10`;
```

How does weather affect taxi usage?

```
const query = `
    SELECT C.conditions AS weather_condition, AVG(T.fare_amount) AS avg_fare
    FROM Trips T
    JOIN Datetimes D ON T.pickup_datetime = D.datetime
    JOIN Weather W ON D.datetime_id = W.datetime_id
    JOIN Conditions C ON W.condition_id = C.condition_id
    GROUP BY weather_condition
    ORDER BY avg_fare DESC `;
```

What is the fare by zone during peak hours?

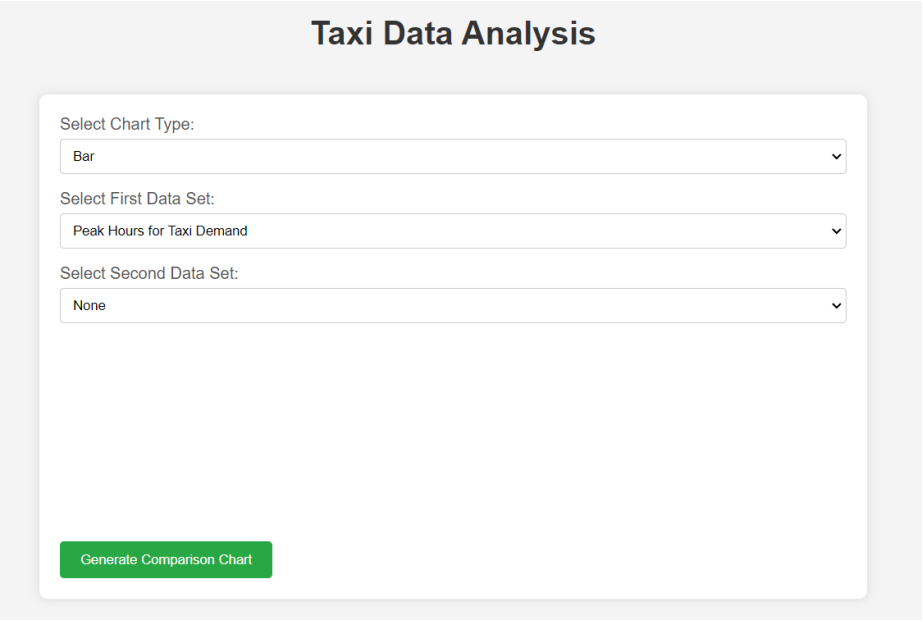
```
const query = `
    WITH PeakHoursTrips AS (
      SELECT
        t.trip_id,
        t.pickup_datetime,
        t.fare_amount,
        l.zone_id AS pickup_zone_id,
        z.zone_name AS pickup_zone_name
      FROM
        Trips t
      JOIN
        Locations l ON t.pickup_location_id = l.location_id
      JOIN
        Zones z ON l.zone_id = z.zone_id
      WHERE
        (HOUR(t.pickup_datetime) BETWEEN 7 AND 9 OR HOUR(t.pickup_datetime) BETWEEN 17 AND 19)
    ),
    AverageFarePerZone AS (
      SELECT
        pzt.pickup_zone_name,
        AVG(pzt.fare_amount) AS average_fare_amount
      FROM
        PeakHoursTrips pzt
      GROUP BY
        pzt.pickup_zone_name
    )
    SELECT
      afz.pickup_zone_name,
      afz.average_fare_amount,
      (SELECT AVG(fare_amount) FROM Trips) AS overall_average_fare,
      afz.average_fare_amount - (SELECT AVG(fare_amount) FROM Trips) AS fare_difference
    FROM
      AverageFarePerZone afz
    ORDER BY
      fare_difference DESC    `;
```

How do payment methods correlate with trip distance or fare?

```
const query = `
    SELECT
      P.payment_type_description AS payment_type,
      AVG(T.trip_distance) AS avg_distance,
      AVG(T.fare_amount) AS avg_fare
    FROM
      Trips T
    JOIN
      PaymentTypes P ON T.payment_type = P.payment_type_id
    GROUP BY
      P.payment_type_description
    ORDER BY
      avg_fare DESC `;
```

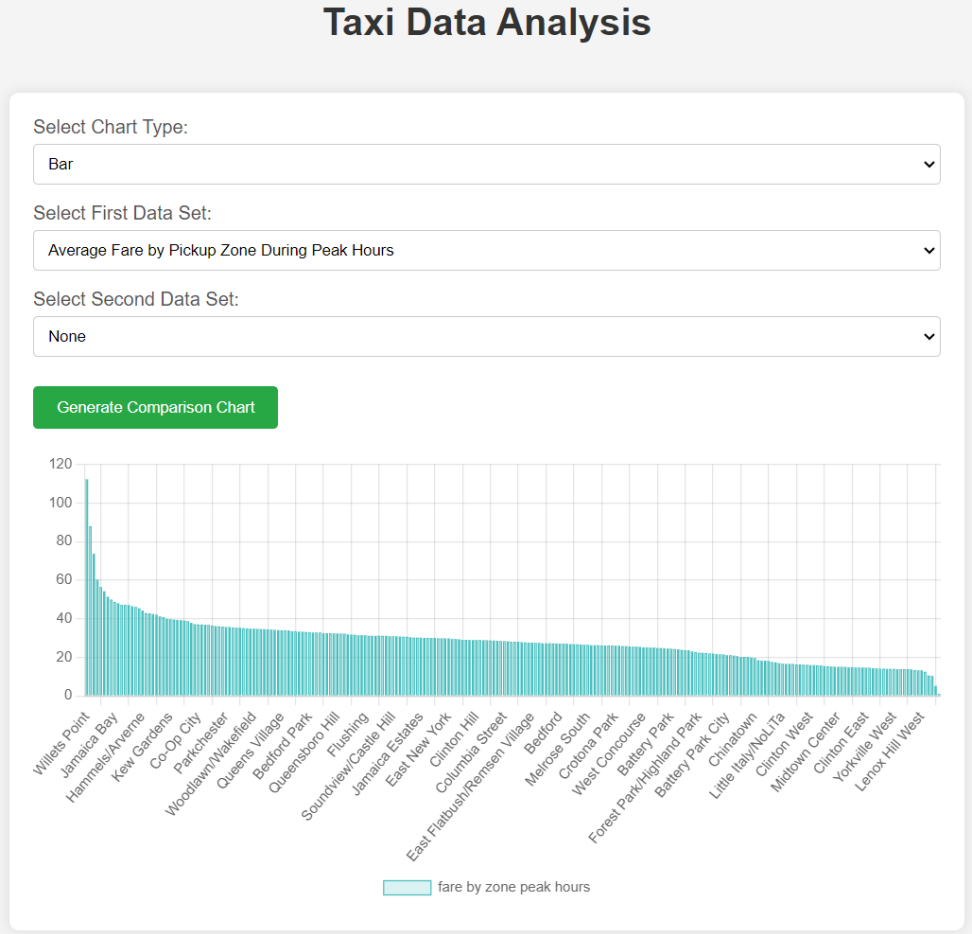
Web App (Visualization of the Research Questions)

The aim of this web app is to help visualize the results of the research questions through a user-friendly interface. The web application is designed to present data insights in a clear and interactive manner, allowing users to explore taxi trip data effectively.



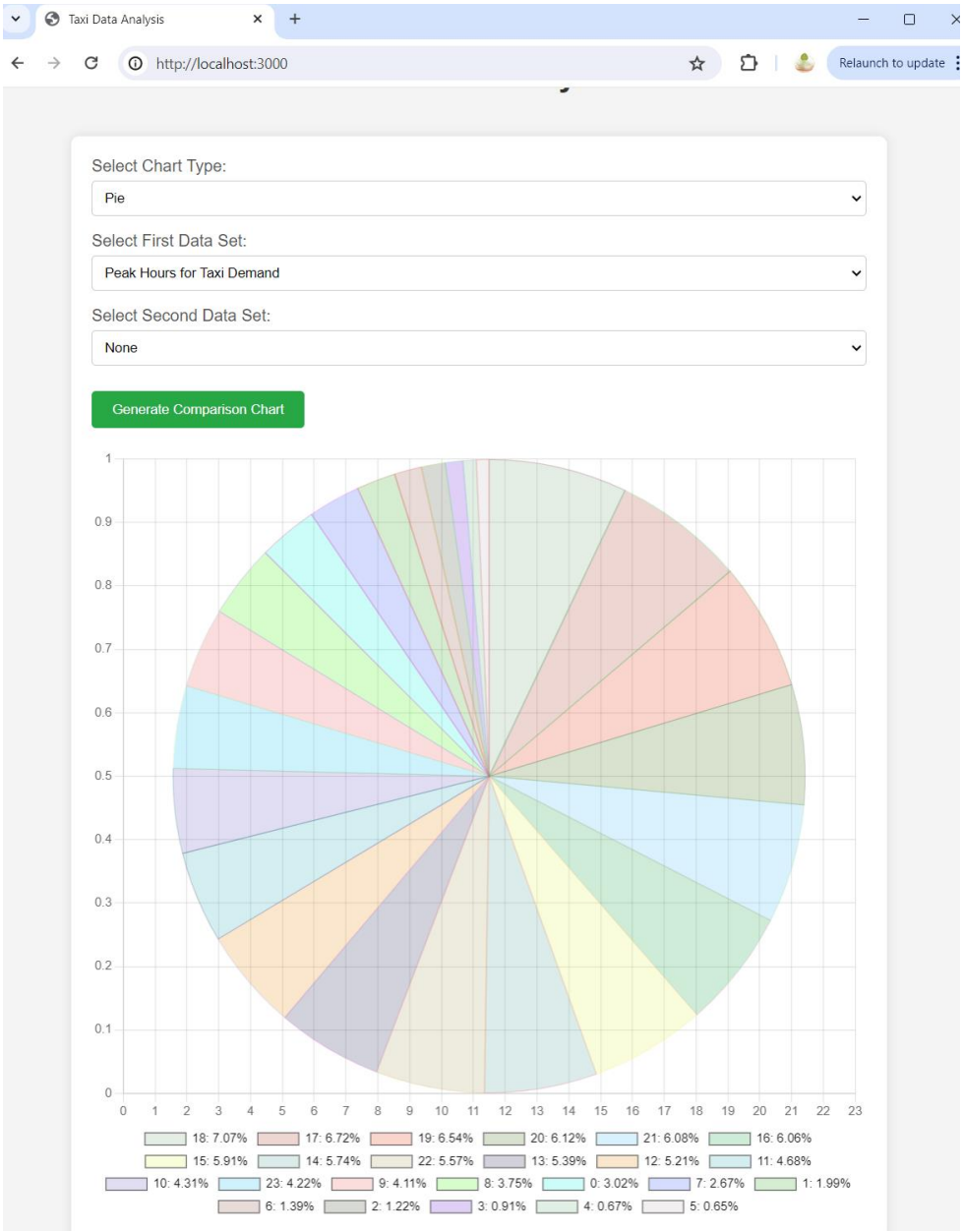
Research Questions Visualization

What is the fare by zone during peak hours?



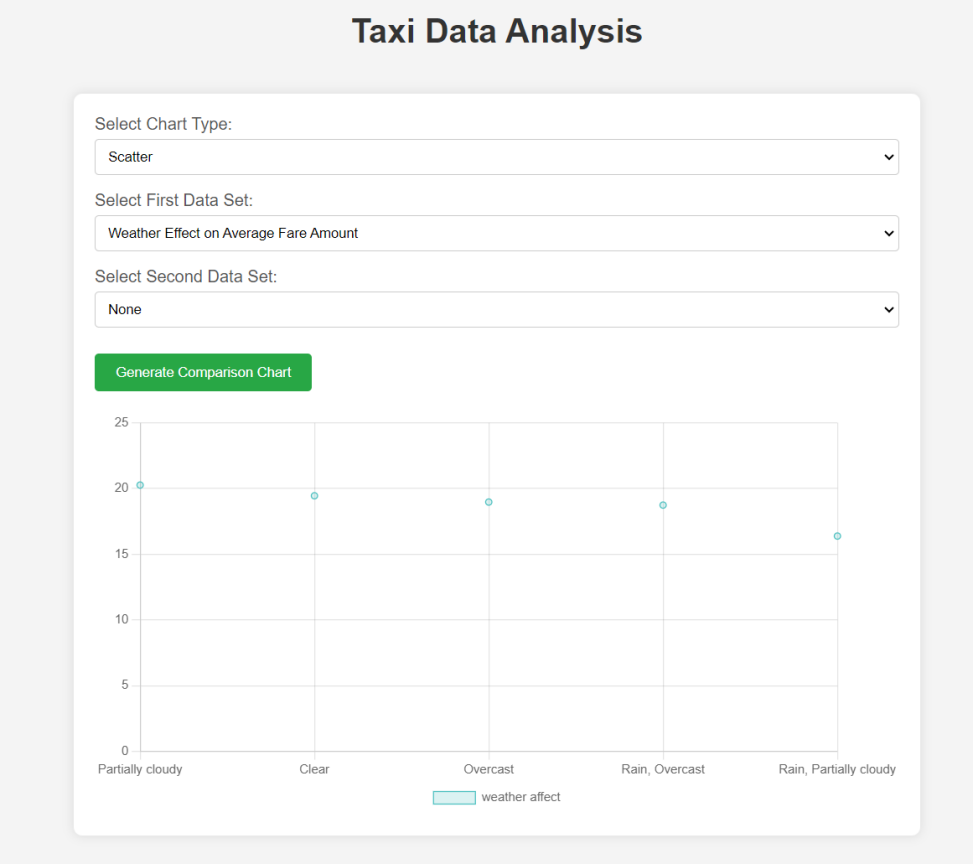
The query used here calculates the average fare amount by zone during peak hours (7-9 AM and 5-7 PM). It uses a Common Table Expression (CTE) to first select trips that occur during peak hours. Another CTE calculates the average fare per zone for these peak hours. The final query selects the average fare per zone, compares it to the overall average fare, and orders the results by the fare difference in descending order to highlight zones with the highest and lowest fares during peak hours.

What are the peak hours for taxi demand?



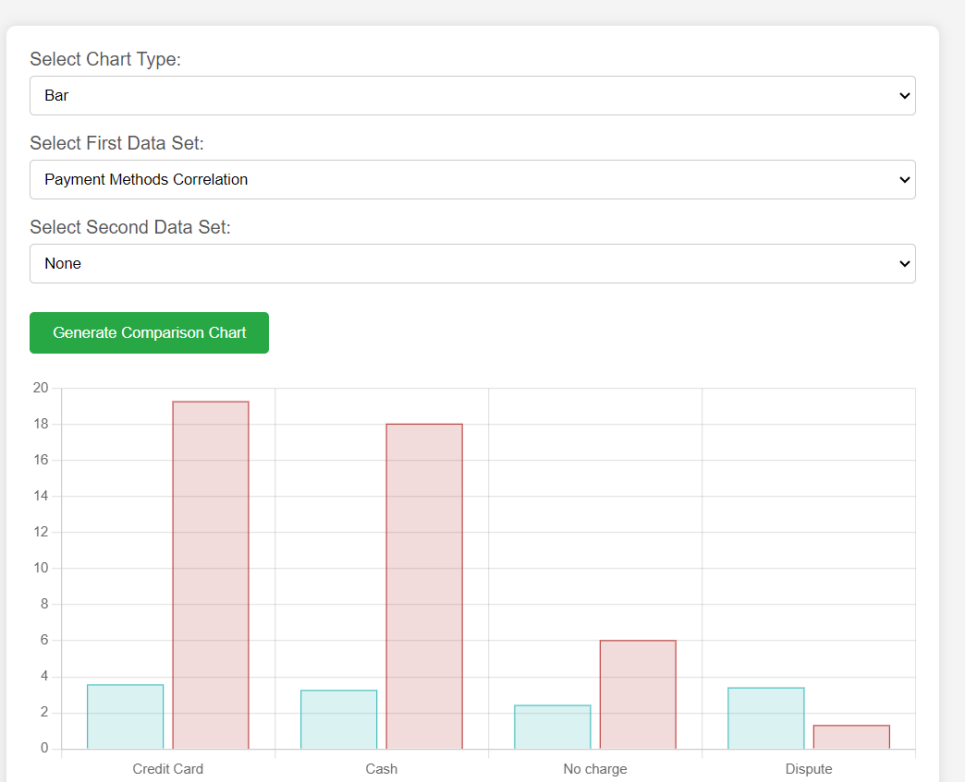
The query used here helps to identifies the peak hours for taxi demand by counting the number of trips that start in each hour of the day. It groups the trips by the hour extracted from the pickup_datetime field and orders the results in descending order of trip count. The hour with the highest count represents the peak hour for taxi demand.

How does weather affect taxi usage?



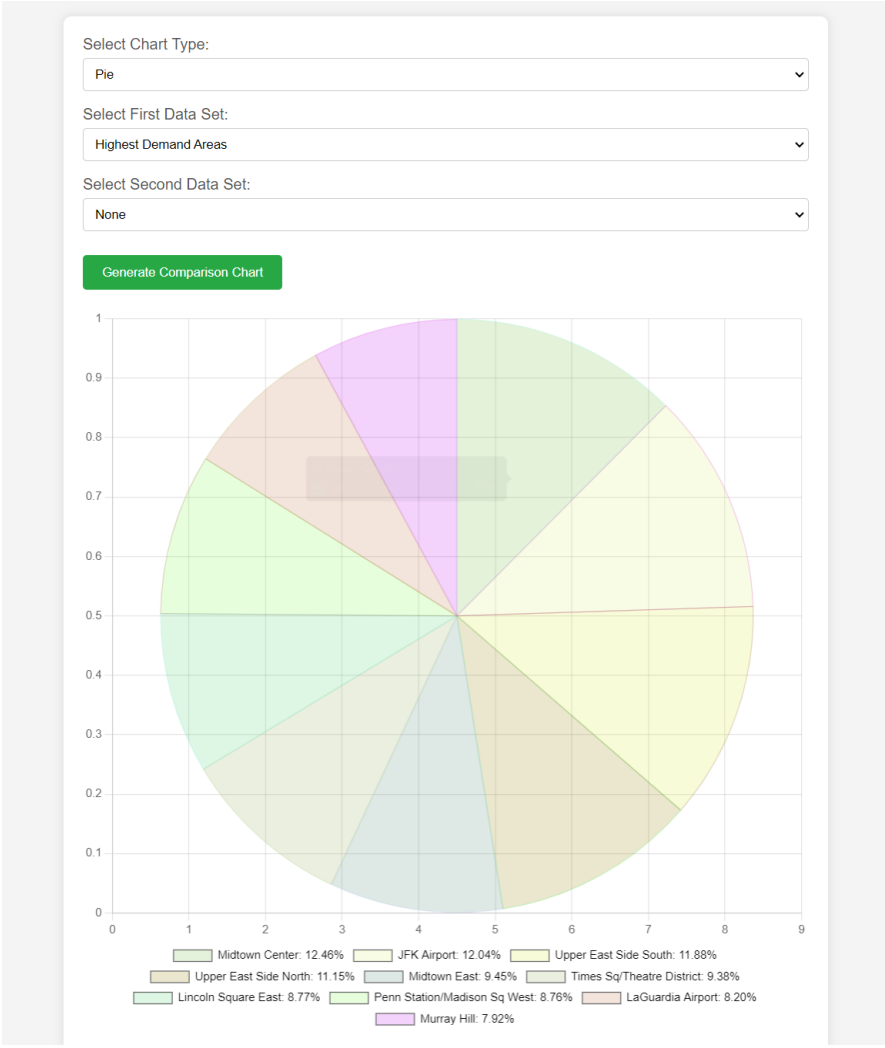
The query here explores the impact of weather on taxi usage by calculating the average fare amount under different weather conditions. It joins the Trips table with the Datetimes, Weather, and Conditions tables to get the weather conditions for each trip and then groups the results by weather condition. The results are ordered by the average fare amount in descending order to identify which weather conditions correlate with higher fares.

How do payment methods correlate with trip distance or fare?



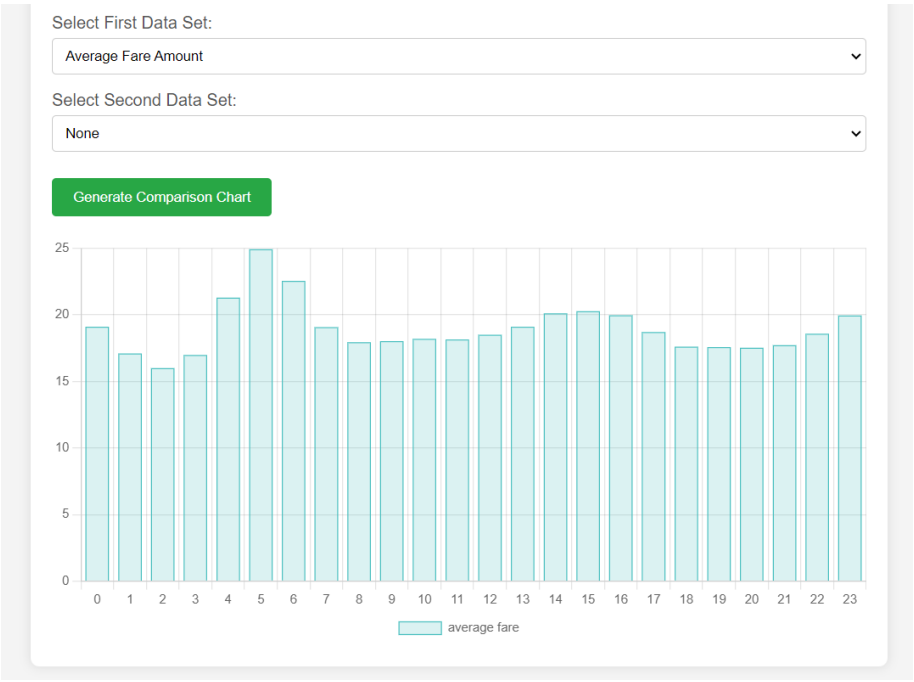
This examines the correlation between payment methods and trip distance or fare. It joins the Trips table with the PaymentTypes table to get the description of each payment type. The query then calculates the average trip distance and average fare amount for each payment type, grouping the results by payment type description and ordering them by average fare in descending order.

Which areas have the highest demand for taxis?



The query here identifies the areas with the highest taxi demand by counting the number of trips that start in each zone. It joins the Trips table with the Locations and Zones tables to get the zone names and then groups the trips by zone name, ordering the results in descending order of trip count. The top 10 zones are selected to highlight the areas with the highest demand.

How does the average fare amount vary?



This query examines the variation in average fare amounts throughout the day. It calculates the average fare amount for trips starting in each hour, groups the results by the hour, and orders them chronologically. This helps understand how the average fare changes during different times of the day.

Conclusion

Database Connection Security

To ensure the security of our database connections, all sensitive credentials are being stored as environment variables using the dotenv package. This approach keeps sensitive information like database usernames and passwords secure and ensures that they are not included in the source code or version control system. By keeping the .env file out of the public code repository, to prevent unauthorized access and enhance the confidentiality of our database connections. This method not only protects against potential data breaches but also complies with best practices for secure application development.

Effective Database Implementation

The relational database was carefully constructed and normalized to 3NF and BCNF, ensuring data integrity and supporting complex SQL queries essential for the analysis. This structure proved effective for querying and managing taxi trip data, underscoring the importance of rigorous database design in this project.

Web Application Functionality

The Node.js web application assisted dynamic data exploration, allowing users to interact with the data through a series of structured queries. This not only made the data more accessible but also demonstrated how web technologies can be leveraged to enhance data visibility.

Recommendations for Future Enhancements

While the current project met the current coursework requirements, future enhancements could include integrating additional datasets and improving user interface designs. These improvements would extend the project's utility and provide a more inclusive and detailed examination of the NYC TLC datasets.

Overall this coursework project successfully addressed the coursework objectives by analyzing, modeling, and implementing MySQL database to explore taxi trip data. The creation of a web application in Node.js further enabled interactive user engagement with the database, highlighting practical applications of the data.

References

1. **Draw.io**
   - Draw.io. (n.d.). Retrieved from https://www.draw.io/
2. **NYC TLC 2024 March Yellow Taxi Data**
   - NYC Open Data. (2024). Yellow Taxi Trip Records. Retrieved from https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page
3. **Chart.js Documentation**
   - Chart.js. (n.d.). Chart.js Documentation. Retrieved from https://www.chartjs.org/docs/latest/
4. **Node.js**
   - Node.js. (n.d.). Node.js Documentation. Retrieved from https://nodejs.org/en/docs/
5. **dotenv package**
   - dotenv. (n.d.). dotenv package on npm. Retrieved from https://www.npmjs.com/package/dotenv