Detailed Report on Smart City Traffic & Accident Analytics

---- Sayan Das

Note:

There were two files available—*road_traffic_sensor_data.csv* and *traffic_sensor_data.csv*. Both contained sensor information and traffic conditions. However, to generate this report, I only needed one of these datasets. I chose *road_traffic_sensor_data.csv* because the other file (*traffic_sensor_data.csv*) contained questionable data like pollution sensor, giving speed data..

1. Data Loading & Pre-Processing

I began by loading the provided CSV files for sensor and accident data into Python using the Pandas library. This step involved standardizing column names (converted to lower-case) for consistency.

Python Code Snippet:

```python
import pandas as pd

# Load CSV files
df_sensor = pd.read_csv('road_traffic_sensor_data.csv')
df_accident = pd.read_csv('traffic_accident_data.csv')

# Standardize column names to lower-case
df_sensor.columns = [col.lower() for col in df_sensor.columns]
df_accident.columns = [col.lower() for col in df_accident.columns]
```

This allowed me to work with a consistent dataset before moving on to the quality and transformation steps.

2. Data Quality Checks & Cleaning

To ensure reliable analytics, I performed several data quality checks: - **Uniqueness Check:** Verified that the primary key columns (`sensor_id` in sensor data and `accident_id` in accident data) are unique. - **Missing Data Check:** Assessed the presence of null values. I opted to drop any rows with missing data for simplicity.

Python Code Snippet:

```python
# Check for primary key uniqueness
if df_sensor['sensor_id'].nunique() != len(df_sensor):
    print("Warning: Duplicate sensor_id values found in sensor data!")
else:
    print("All sensor_id values are unique in sensor data.")

if df_accident['accident_id'].nunique() != len(df_accident):
    print("Warning: Duplicate accident_id values found in accident data!")
else:
    print("All accident_id values are unique in accident data.")

# Check for missing values
print("Missing values in sensor data:")
print(df_sensor.isnull().sum())
```

```python
print("\nMissing values in accident data:")
print(df_accident.isnull().sum())

# Drop rows with missing data
df_sensor.dropna(inplace=True)
df_accident.dropna(inplace=True)
```

Additionally, I converted the `date_time` columns into datetime objects to facilitate further time-based transformations.

df_sensor['date_time'] = pd.to_datetime(df_sensor['date_time'])
df_accident['date_time'] = pd.to_datetime(df_accident['date_time'])

3. Data Transformation: Creating Dimension and Fact Tables

Following data cleaning, I transformed the raw data into a structured star schema by creating dimension tables and fact tables.

3.1 Creating Dimension Tables

I built the following dimension tables:

- **dim_time:**
  I combined unique `date_time` values from both datasets, extracted additional attributes (year, month, day, hour, day_of_week), and created a surrogate key `time_id`.

  Python Code Snippet:

  ```python
  all_times = pd.concat([df_sensor[['date_time']], df_accident[['date_time']]])
  all_times = all_times.drop_duplicates().reset_index(drop=True)

  all_times['year'] = all_times['date_time'].dt.year
  all_times['month'] = all_times['date_time'].dt.month
  all_times['day'] = all_times['date_time'].dt.day
  all_times['hour'] = all_times['date_time'].dt.hour
  all_times['day_of_week'] = all_times['date_time'].dt.dayofweek

  all_times.reset_index(inplace=True)
  all_times.rename(columns={'index': 'time_id'}, inplace=True)
  all_times['time_id'] = all_times['time_id'] + 1

  dim_time = all_times[['time_id', 'date_time', 'year', 'month', 'day', 'hour',
  'day_of_week']]
  ```

- **dim_location:**
  I consolidated unique locations from both datasets and assigned a surrogate key `location_id`.

  Python Code Snippet:

  ```python
  locations_sensor = df_sensor[['location']].drop_duplicates()
  locations_accident = df_accident[['location']].drop_duplicates()
  all_locations = pd.concat([locations_sensor,
  locations_accident]).drop_duplicates().reset_index(drop=True)

  all_locations.reset_index(inplace=True)
  all_locations.rename(columns={'index': 'location_id'}, inplace=True)
  ```

```python
all_locations['location_id'] = all_locations['location_id'] + 1

dim_location = all_locations[['location_id', 'location']]
```

- **Additional Dimensions (for Accident Data):**
  I created separate dimensions for `vehicle_type`, `weather_condition`, and `road_condition` with their own surrogate keys (`vehicle_id`, `weather_id`, and `road_id` respectively).

  Python Code Snippet:

```python
# Vehicle Dimension
dim_vehicle =
df_accident[['vehicle_type']].drop_duplicates().reset_index(drop=True)
dim_vehicle.reset_index(inplace=True)
dim_vehicle.rename(columns={'index': 'vehicle_id'}, inplace=True)
dim_vehicle['vehicle_id'] = dim_vehicle['vehicle_id'] + 1
dim_vehicle = dim_vehicle[['vehicle_id', 'vehicle_type']]

# Weather Dimension
dim_weather =
df_accident[['weather_condition']].drop_duplicates().reset_index(drop=True)
dim_weather.reset_index(inplace=True)
dim_weather.rename(columns={'index': 'weather_id'}, inplace=True)
dim_weather['weather_id'] = dim_weather['weather_id'] + 1
dim_weather = dim_weather[['weather_id', 'weather_condition']]

# Road Condition Dimension
dim_road = df_accident[['road_condition']].drop_duplicates().reset_index(drop=True)
dim_road.reset_index(inplace=True)
dim_road.rename(columns={'index': 'road_id'}, inplace=True)
dim_road['road_id'] = dim_road['road_id'] + 1
dim_road = dim_road[['road_id', 'road_condition']]
```

3.2 Creating Fact Tables

Next, I built the fact tables by merging the cleaned data with the respective dimensions: - **fact_traffic:** This fact table includes the key measures from sensor data along with foreign keys linking to `dim_time` and `dim_location`.

Python Code Snippet: ```python # Merge with dim_time and rename to fk_time_id fact_traffic = df_sensor.merge(dim_time[['time_id', 'date_time']], on='date_time', how='left') fact_traffic.rename(columns={'time_id': 'fk_time_id'}, inplace=True)

Merge with dim_location and rename to fk_location_id fact_traffic = fact_traffic.merge(dim_location, on='location', how='left') fact_traffic.rename(columns={'location_id': 'fk_location_id'}, inplace=True)

Select only the necessary columns fact_traffic = fact_traffic[['sensor_id', 'fk_time_id', 'fk_location_id', 'vehicle_count', 'average_speed', 'congestion_level']] ```

- **fact_accident:**
  This fact table holds accident-specific measures and references dimensions including `dim_time`, `dim_location`, `dim_vehicle`, `dim_weather`, and `dim_road`.

  Python Code Snippet:

```
# Merge with dim_time and rename to fk_time_id
fact_accident = df_accident.merge(dim_time[['time_id', 'date_time']],
on='date_time', how='left')
fact_accident.rename(columns={'time_id': 'fk_time_id'}, inplace=True)

# Merge with dim_location and rename to fk_location_id
fact_accident = fact_accident.merge(dim_location, on='location', how='left')
fact_accident.rename(columns={'location_id': 'fk_location_id'}, inplace=True)

# Merge with additional dimensions
fact_accident = fact_accident.merge(dim_vehicle.rename(columns={'vehicle_id':
'fk_vehicle_id'}),
                                    on='vehicle_type', how='left')
fact_accident = fact_accident.merge(dim_weather.rename(columns={'weather_id':
'fk_weather_id'}),
                                    on='weather_condition', how='left')
fact_accident = fact_accident.merge(dim_road.rename(columns={'road_id':
'fk_road_id'}),
                                    on='road_condition', how='left')

# Select only the necessary columns
fact_accident = fact_accident[['accident_id', 'fk_time_id', 'fk_location_id',
'fk_vehicle_id', 'fk_weather_id', 'fk_road_id','accident_severity',
'number_of_vehicles', 'casualties','traffic_density']]
```

4. Loading Transformed Data to MySQL

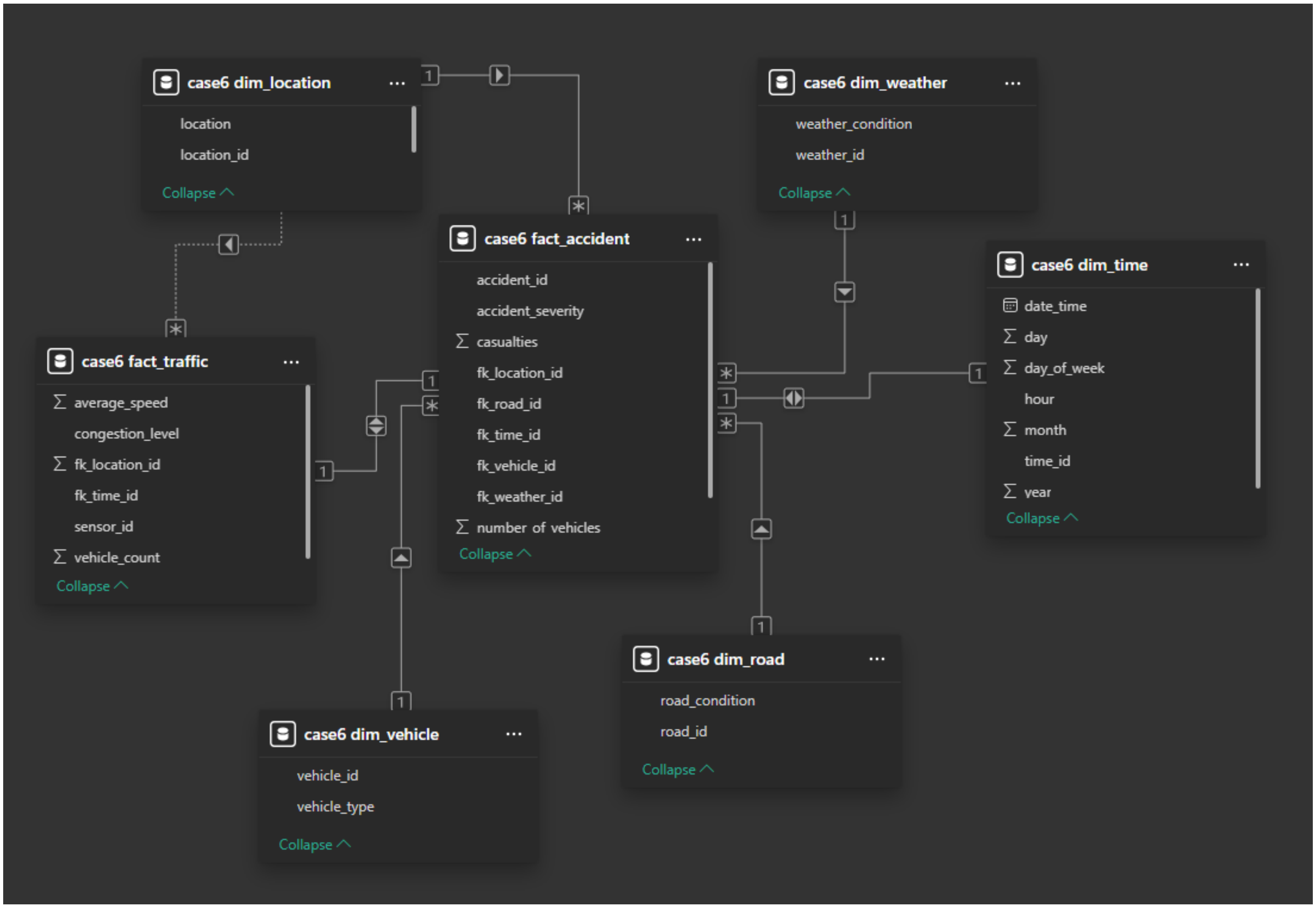
After the ETL process, I loaded the dimension and fact tables into a MySQL database using SQLAlchemy. I configured the connection parameters and used the `to_sql()` method with `if_exists='replace'` to update the tables.

Python Code Snippet:

```
from sqlalchemy import create_engine

# MySQL connection parameters
username = 'root'
password = '12345'
host = 'localhost'
port = '3306'
database = 'case6'

# Create the SQLAlchemy engine
engine = create_engine(f'mysql+pymysql://{username}:{password}@{host}:{port}/{database}')

# Load tables into MySQL
dim_time.to_sql('dim_time', con=engine, index=False, if_exists='replace')
dim_location.to_sql('dim_location', con=engine, index=False, if_exists='replace')
dim_vehicle.to_sql('dim_vehicle', con=engine, index=False, if_exists='replace')
dim_weather.to_sql('dim_weather', con=engine, index=False, if_exists='replace')
dim_road.to_sql('dim_road', con=engine, index=False, if_exists='replace')
fact_traffic.to_sql('fact_traffic', con=engine, index=False, if_exists='replace')
fact_accident.to_sql('fact_accident', con=engine, index=False, if_exists='replace')

print("\nAll tables have been loaded successfully into MySQL!")
```

5. SQL Queries for Insights

Schema:



- **Traffic Congestion (Real time data not available)**

```sql
USE case6;

-- 1. Show traffic congestion by location and time
SELECT
    dl.location,
    dt.date_time,
    ft.congestion_level,
    ft.vehicle_count,
    ft.average_speed
FROM fact_traffic AS ft
JOIN dim_time AS dt
    ON ft.fk_time_id = dt.time_id
JOIN dim_location AS dl
    ON ft.fk_location_id = dl.location_id
ORDER BY dt.date_time;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

location	date_time	congestion_level	vehicle_count	average_speed
Downtown	2024-01-01 00:00:00	High	90	59
Downtown	2024-01-01 01:00:00	Moderate	324	73
Highway	2024-01-01 02:00:00	Moderate	61	56
Highway	2024-01-01 03:00:00	High	90	56
Residential Zone	2024-01-01 04:00:00	High	72	77
Downtown	2024-01-01 05:00:00	Moderate	243	61
Residential Zone	2024-01-01 06:00:00	Moderate	353	74
Industrial Area	2024-01-01 07:00:00	High	475	46
Highway	2024-01-01 08:00:00	Moderate	61	53
Suburbs	2024-01-01 09:00:00	Low	375	44
Downtown	2024-01-01 10:00:00	Moderate	441	21
Industrial Area	2024-01-01 11:00:00	Moderate	203	76
Residential Zone	2024-01-01 12:00:00	Moderate	398	70
Industrial Area	2024-01-01 13:00:00	High	78	50
Downtown	2024-01-01 14:00:00	Low	420	48
Suburbs	2024-01-01 15:00:00	Low	337	63
Highway	2024-01-01 16:00:00	Moderate	430	41

- **Accident-Prone Areas**

```sql
-- 2. Show top 10 accident-prone areas
SELECT
    dl.location,
    COUNT(*) AS total_accidents
FROM fact_accident AS fa
JOIN dim_location AS dl
    ON fa.fk_location_id = dl.location_id
GROUP BY dl.location
ORDER BY total_accidents DESC
LIMIT 10;
```

Result Grid | Filter Rows:

location	total_accidents
Highway	1018
Suburbs	1017
Residential Zone	998
Industrial Area	993
Downtown	974

```sql
-- Top locations with the highest count of 'Fatal' accidents
SELECT
    dl.location,
    COUNT(*) AS fatal_accidents
FROM fact_accident AS fa
JOIN dim_location AS dl
    ON fa.fk_location_id = dl.location_id
WHERE fa.accident_severity = 'Fatal'
GROUP BY dl.location
ORDER BY fatal_accidents DESC
LIMIT 10;
```

Result Grid | Filter Rows:

location	fatal_accidents
Suburbs	275
Highway	260
Industrial Area	253
Residential Zone	243
Downtown	222

- **Peak Hour Analysis**

```sql
-- 3. Peak hour analysis for accidents
SELECT
    dt.hour,
    COUNT(*) AS total_accidents
FROM fact_accident AS fa
JOIN dim_time AS dt
    ON fa.fk_time_id = dt.time_id
GROUP BY dt.hour
ORDER BY total_accidents DESC;
```

Result Grid | Filter Row

hour	total_accidents
0	209
1	209
2	209
3	209
4	209
5	209
6	209
7	209
8	208
9	208
10	208
11	208
12	208
13	208

```sql
-- 4. Peak Hour for Traffic Volume
SELECT
    dt.hour,
    SUM(ft.vehicle_count) AS total_vehicles
FROM fact_traffic AS ft
JOIN dim_time AS dt
    ON ft.fk_time_id = dt.time_id
GROUP BY dt.hour
ORDER BY total_vehicles DESC;
```

Result Grid | Filter Rov

hour	total_vehicles
18	4295
15	4274
10	4110
2	4063
4	3974
12	3888
1	3721
7	3661
5	3641
6	3603
3	3502
[illegible]	[illegible]

```sql
-- 5. Average speed by hour
SELECT
    dt.hour,
    AVG(ft.average_speed) AS avg_speed
FROM fact_traffic AS ft
JOIN dim_time AS dt
    ON ft.fk_time_id = dt.time_id
GROUP BY dt.hour
ORDER BY avg_speed;
```

Result Grid | Filter Row

hour	avg_speed
8	40.0769
3	41.3077
10	42.0000
16	43.9167
22	45.5000
20	46.0833
17	46.8333
15	47.0833
6	48.3077
2	48.3846
23	48.8333
0	49.5385
13	49.6667
11	50.4615
1	50.6923
9	50.7692

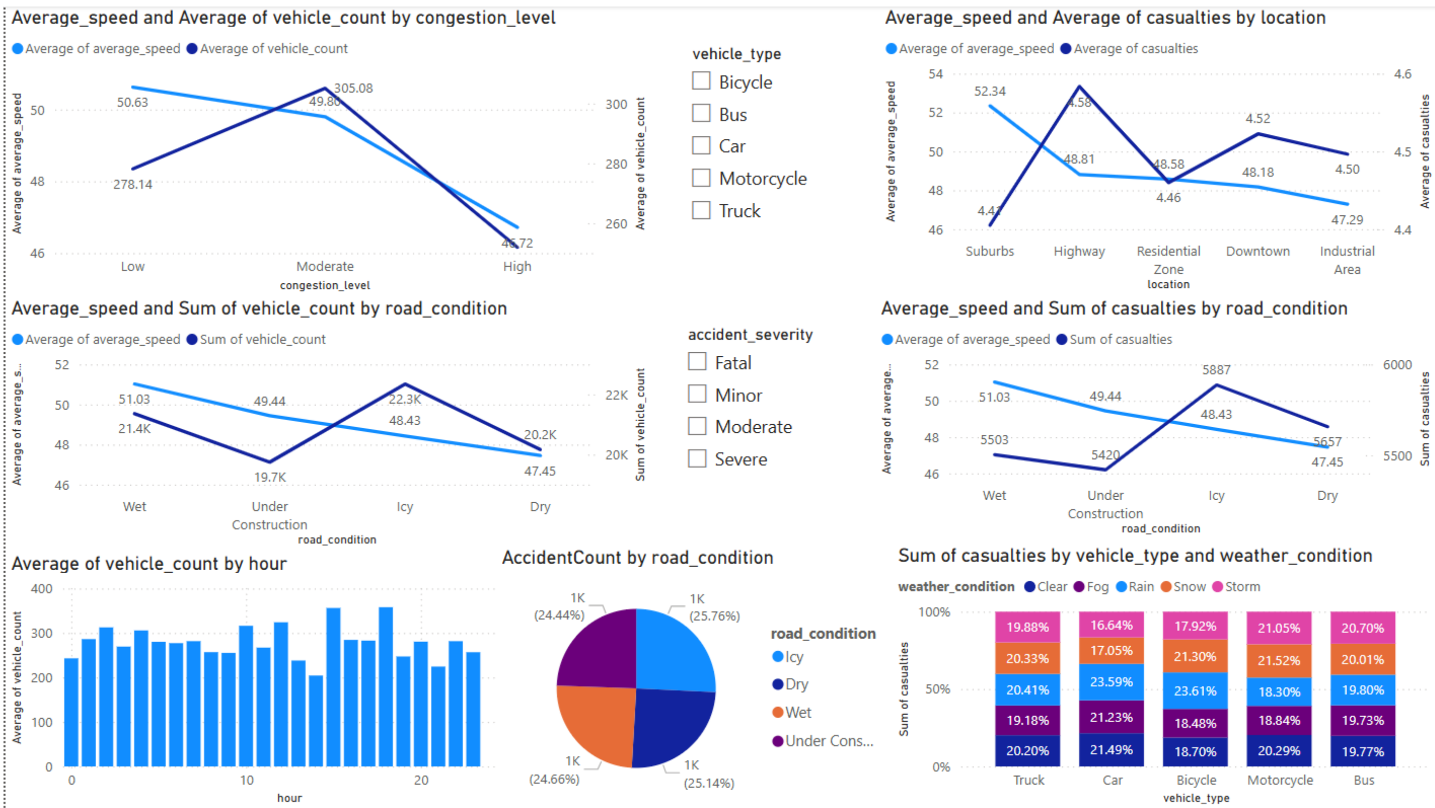
-- 4. Peak hour analysis for traffic volume

```sql
SELECT
    dt.hour,
    SUM(ft.vehicle_count) AS total_vehicles
FROM fact_traffic AS ft
JOIN dim_time AS dt
    ON ft.fk_time_id = dt.time_id
GROUP BY dt.hour
ORDER BY total_vehicles DESC;
```

Result Grid | Filter Rows:

hour	total_vehicles
18	4295
15	4274
10	4110
2	4063
4	3974
12	3888
1	3721
7	3661
5	3641
6	3603
3	3502
11	3472

6. PowerBI Report



Conclusion

- **Focus on High-Congestion Zones/Times:** Implement dynamic traffic signals, ramp metering, or congestion pricing in the hours and locations where congestion is severe and speed is low.
- **Improve Road Safety in Hazardous Conditions:**
  - **Icy or Wet Roads:** Deploy salting trucks, improved drainage, or better signage.
  - **Under Construction Areas:** Enhance signage, set lower speed limits, and ensure construction zones are well-lit.

- **Target Vehicle-Specific Interventions:**
    - **Buses**: Possibly large casualties due to high passenger count—enforce rigorous driver training and vehicle maintenance.
    - **Motorcycles**: High vulnerability in bad weather—promote protective gear, stricter licensing.