CUNY Baruch College

Service on the Street and at the Table: Investigating Taxi and Restaurant Issues in NYC

Final Project Report

Group-3 Members:
Dhruv Sharma
Pasang Syangba
Maung Aung
Sujasna Tamang

BUS 9440-27334
DataWarehousing and Analytics
Fall 2024

Professor: Ramah Al Balawi
December 13, 2024

Introduction

Narrative description of the project

New York City, renowned for its fast-paced lifestyle and diverse cultural offerings, heavily relies on two vital service sectors: transportation and dining. However, both sectors face critical issues that impact the city's residents and visitors. Taxi services often attract complaints related to unsafe driving, route disputes, and pickup refusals, while restaurants are frequently evaluated for compliance with health standards, with some receiving poor grades due to critical violations. Addressing these challenges requires an integrated approach to uncover patterns and correlations that may exist between service quality in these sectors.

This project aims to create a comprehensive data warehouse to investigate the interplay between taxi complaints and restaurant inspection results. By analyzing these datasets, we aim to answer the central question: **Can the safety and the service quality of a borough be evaluated through restaurant inspection results and taxi complaints?**

Overview of Source Data

To achieve this goal, we leverage two critical datasets:

1. 311 Taxi Complaints Dataset: This dataset, sourced from NYC Open Data, documents public complaints regarding taxi services, including unsafe driving, route disputes, service refusals, and unauthorized pickups. By analyzing the geographical and temporal distribution of these complaints, we aim to identify systemic service issues across the city.

- Source: NYC Open Data - 311 Service Requests

  https://data.cityofnewyork.us/Social-Services/311-Service-Requests-from-2010-to-Present/erm2-nwe9/about_data

2. DOHMH New York City Restaurant Inspection Results Dataset: This dataset, also sourced from NYC Open Data, provides detailed health inspection records

of New York City restaurants. It includes scores, inspection grades, critical violation flags, and compliance indicators. The data allows us to evaluate neighborhood-level public health standards.

- Source: NYC Open Data - DOHMH Restaurant Inspection Results

  https://data.cityofnewyork.us/Health/DOHMH-New-York-City-Restaurant-Inspection-Results/43nn-pn8j/about_data

The group will analyze the relationship between taxi complaints such as unsafe driving, route issues, service refusals, and New York City restaurant inspection results, including inspection grades, health violations, and compliance flags. By examining both datasets, the group aims to uncover patterns between taxi service issues and restaurant standards within specific boroughs, seeking to determine whether areas with lower restaurant inspection scores also experience higher rates of taxi complaints. This combined analysis could reveal broader trends in borough-wide safety and service quality, providing insights that may guide improvement in transportation and public health regulation across New York City.

Key Performance Indicators (KPIs):

Taxi Complaints Dataset

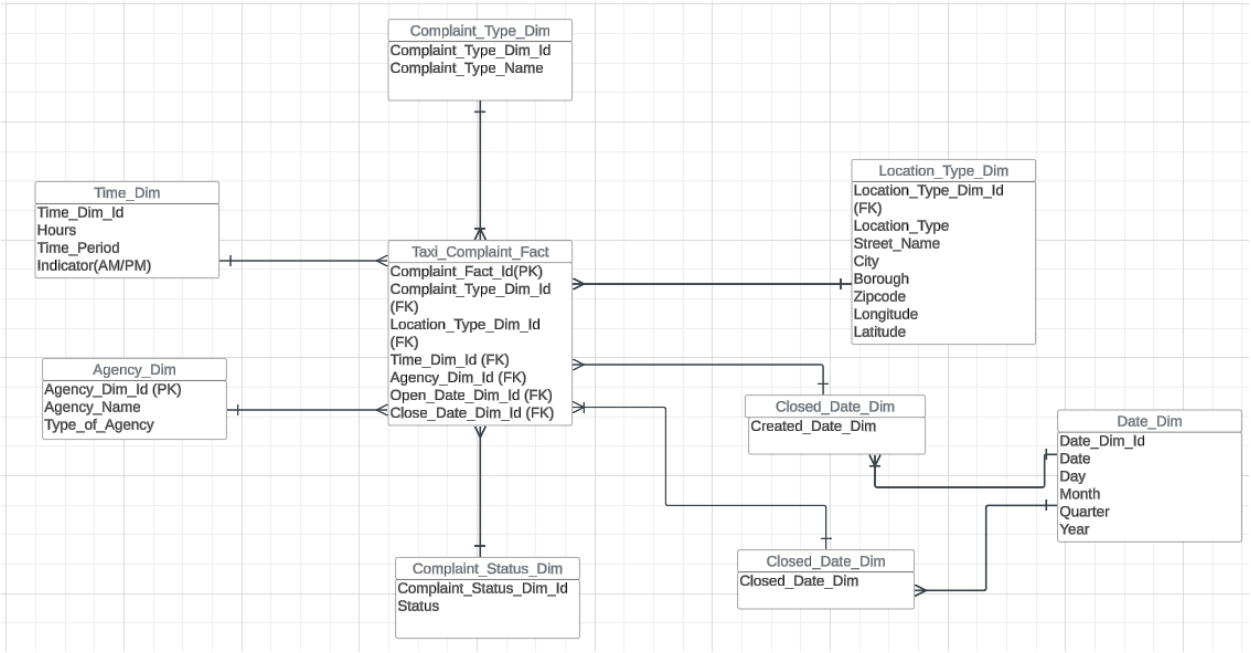
1. Total Number of Complaints by Type
2. Number of Complaints by Location Type
3. Complaint Frequency by Time of Day
4. Average Day of Complaint Closure by Agency
5. Number of Complaints by Status
6. Number of Complaints by Month

Restaurant Inspections Dataset

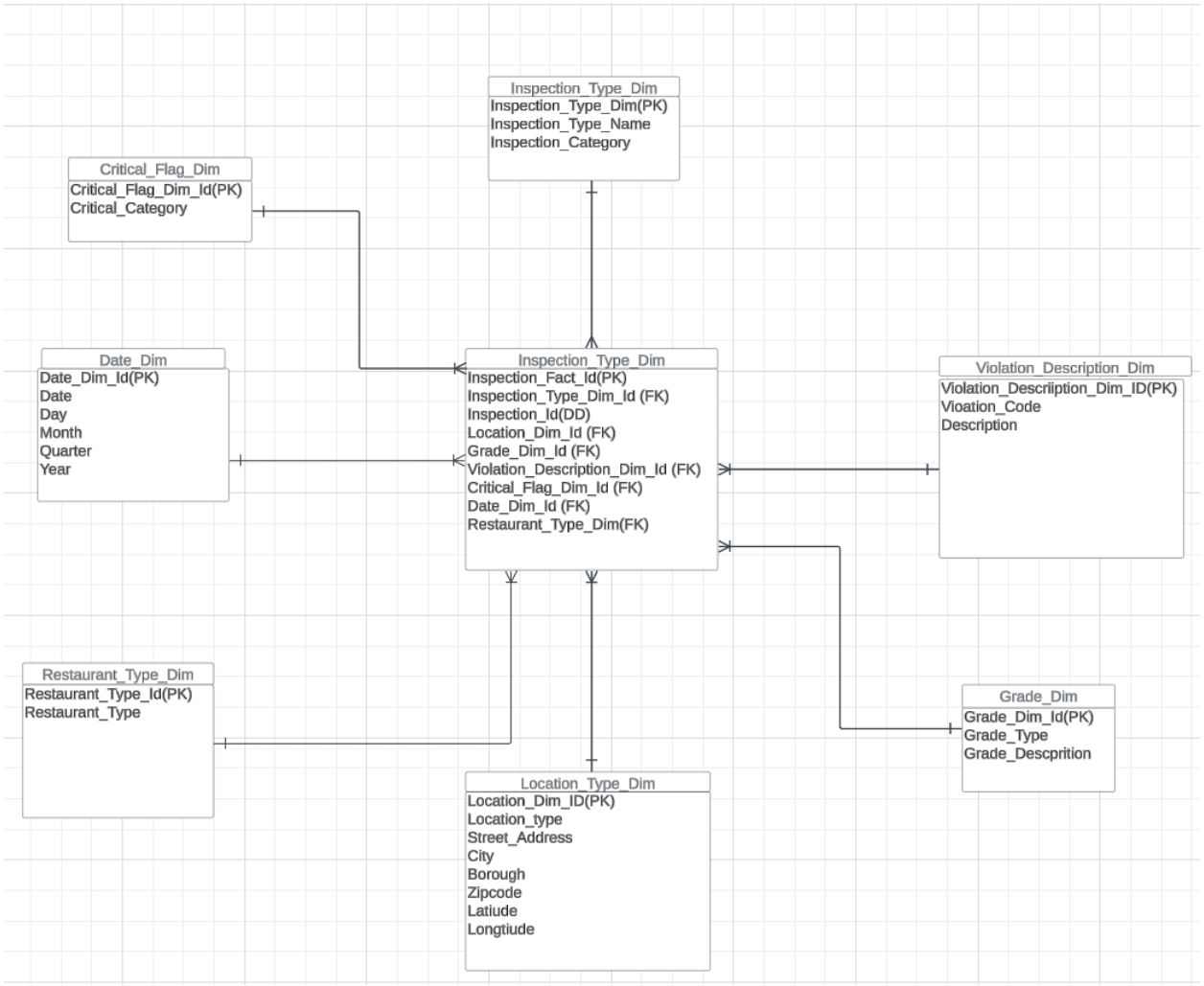
1. Number of inspections by Inspection Type.
2. Number of Inspection scores by Zipcode
3. Number of Inspections by Grade
4. Total Number of Complaints by Critical Flag
5. Number of Inspections by Month

Dimension Model Diagrams

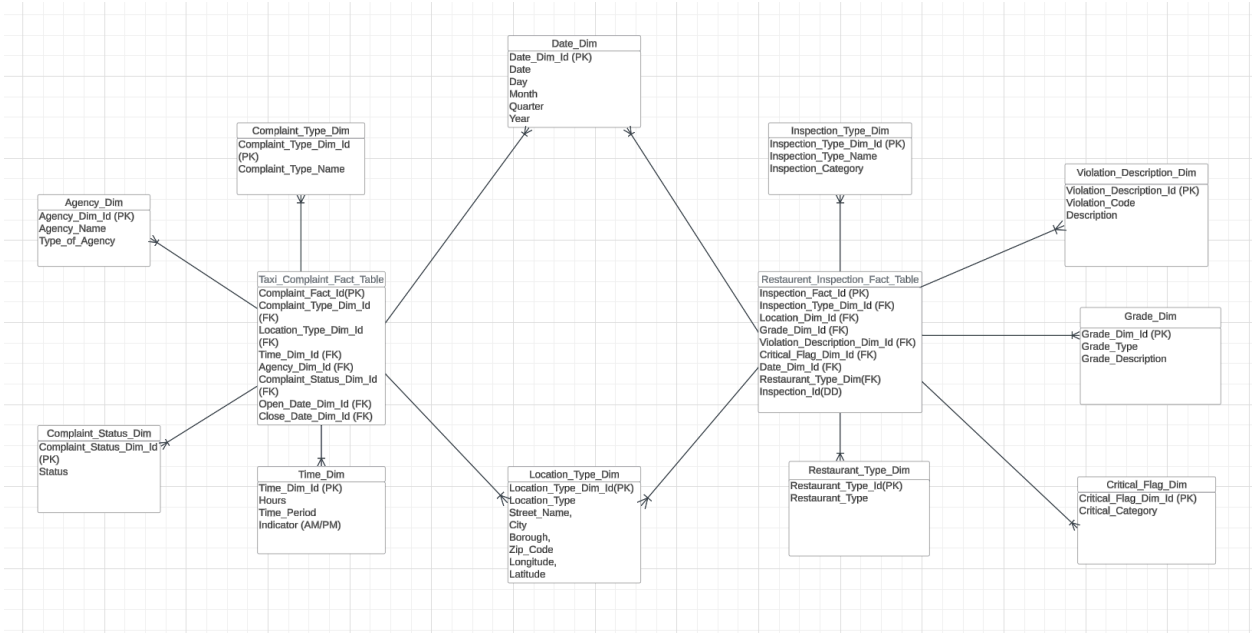
Taxi complaints dimension model



Restaurant Inspection Dimension Model



Integrated Data Warehouse Model



ETL Process

We used Python for the ETL (Extract, Transform, Load) process of our project. The process was carried out as follows:

1. **Extraction and data profiling**

- **Tool Used:** Sodapy API in Jupyter Notebook

Process Overview:

We applied the same methodology to extract and profile both datasets. In this report, we will only provide the code for data extraction and data profiling for the “Taxi Complaints”.

Extraction:

The source data was a publicly available dataset hosted on NYC Open Data and retrieved using sodapy library. The extraction process was streamlined using defined parameters such as filters, date ranges of 01/01/2021 to 10/31/2024, to reduce noise and optimize data transfer. Then, the raw data was saved as CSV files for further processing and backup.

```
[1] !pip install sodapy
```

```
Collecting sodapy
  Downloading sodapy-2.2.0-py2.py3-none-any.whl.metadata (15 kB)
Requirement already satisfied: requests>=2.28.1 in /usr/local/lib/python3.10/dist-packages (from sodapy) (2.32.3)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests>=2.28.1->sodapy
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests>=2.28.1->sodapy) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests>=2.28.1->sodapy) (2.2
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests>=2.28.1->sodapy) (202
Downloading sodapy-2.2.0-py2.py3-none-any.whl (15 kB)
Installing collected packages: sodapy
Successfully installed sodapy-2.2.0
```

```
[3] import pandas as pd
    from sodapy import Socrata
```

```
[5] data_url = 'data.cityofnewyork.us'
    app_token = 'DiYM4Qi0LWQfnpvC11ksbm5ju'

    client = Socrata(data_url, app_token)
    client.timeout = 240
```

```
[12] for x in range(2021, 2025):
         # get data
         start = 0
         chunk_size = 2000
         results = []

         where_clause = f"complaint_type LIKE 'Taxi Complaint%' AND date_extract_y(created_date)={x}"
         data_set = 'erm2-nwe9'
         record_count = client.get(data_set, where=where_clause, select='COUNT(*)')
         print(f'Taxi Complaint data set from {x}')

         while True:
             results.extend(client.get(data_set, where=where_clause, offset=start, limit=chunk_size))
             start += chunk_size
             if (start > int(record_count[0]['COUNT'])):
                 break

         # export data to csv
         df = pd.DataFrame.from_records(results)
         df.to_csv("311_Taxi_Complaint.csv", index=False)
```

```
Taxi Complaint data set from 2021
Taxi Complaint data set from 2022
Taxi Complaint data set from 2023
Taxi Complaint data set from 2024
```

Data profiling:

After extracting, data profiling was conducted to gain insights into the structure of our data structure and quality. This helped us to identify potential issues of missing values, outliers or inconsistencies.

Taxi Complaint:

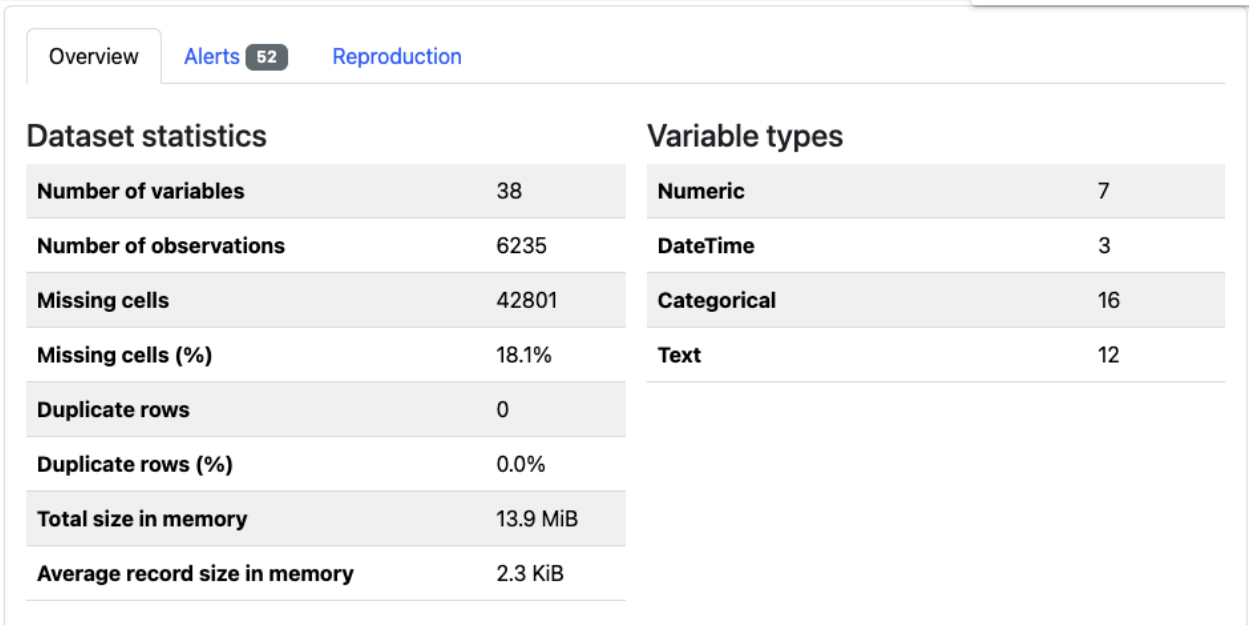
```python
import pandas as pd
from ydata_profiling import ProfileReport

# Load the dataset (replace the file path if necessary)
df = pd.read_csv("311_Taxi_Complaint.csv")

# Generate the data profiling report
profile = ProfileReport(df, title="Taxi Complaint Data Profiling Report", explorative=True)

# Save the profiling report to an HTML file
profile.to_file("taxi_complaint_data_profiling_report.html")

# Optionally, you can display the report in a Jupyter notebook (if using Jupyter)
profile.to_notebook_iframe()
```



Overview | Alerts 52 | Reproduction

Dataset statistics

Number of variables	38
Number of observations	6235
Missing cells	42801
Missing cells (%)	18.1%
Duplicate rows	0
Duplicate rows (%)	0.0%
Total size in memory	13.9 MiB
Average record size in memory	2.3 KiB

Variable types

Numeric	7
DateTime	3
Categorical	16
Text	12

Restaurant Inspection:

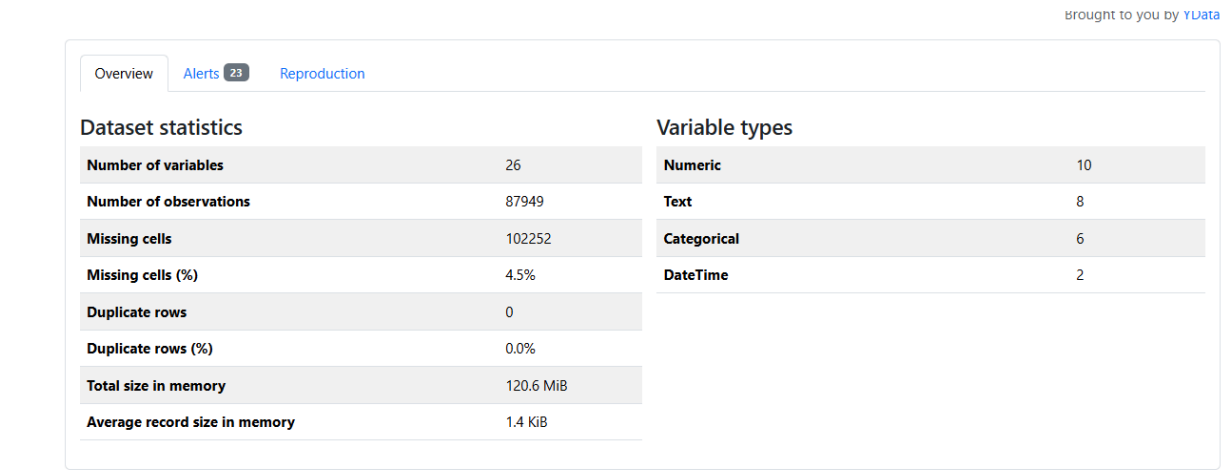
```python
[ ] import pandas as pd
    from ydata_profiling import ProfileReport

    # Load the dataset
    df = pd.read_csv("Restaurant_Inspection.csv")

    # Generate the data profiling report
    profile = ProfileReport(df, title="Restaurant Inspection Data Profiling Report", explorative=True)

    # Save the profiling report to an HTML file
    profile.to_file("Restaurant_inspection_data_profiling_report.html")

    profile.to_notebook_iframe()
```



2. Transformation

After extraction, we performed several data cleansing and transformation steps to ensure the data was accurate and suitable for analysis. Before we defined functions, we imported some necessary libraries. Below are our steps for the transformation process for both data sets.

```
[11] from google.colab import auth
     auth.authenticate_user()
     print('Authenticated')


     # Google Colab load modules for BigQuery
     %load_ext google.cloud.bigquery
     %load_ext google.colab.data_table
```

```
Authenticated
```

```
[12] # ETL Complaint Facts
     # If using the native Google BigQuery API module:
     from google.cloud import bigquery
     from google.cloud.exceptions import NotFound
     # import credentials
     import pandas as pd
     import os
     import pyarrow
     import logging
     from datetime import datetime
```

Taxi Complaints:

```python
from google.cloud import bigquery
from google.oauth2 import service_account
import logging
import os

def create_bigquery_client(logging=None):
    """
    Creates a BigQuery client using the service account key from Google Drive.
    Returns the BigQuery client object or None if creation fails.
    """
    # Use environment variable or hardcoded path for the key file
    key_path = os.getenv('BIGQUERY_KEY_PATH', '/content/drive/MyDrive/CIS9350/sodium-lore-442819-k0-b67a43235790.json')

    try:
        # Construct credentials from the key file
        credentials = service_account.Credentials.from_service_account_file(key_path)

        # Initialize BigQuery client with credentials
        bqclient = bigquery.Client(credentials=credentials, project=credentials.project_id)
        if logging:
        # Log client creation
          logging.info('Created BigQuery client: %s', bqclient)

        return bqclient

    except Exception as err:
        # Log error with exception details
        if logging:
          logging.error('Failed to create BigQuery client.', exc_info=True)
        return None
```

```python
] bqclient = create_bigquery_client(None)
```

```python
] bqclient.create_dataset('311_taxi_complaints',exists_ok=True)
```

```
Dataset(DatasetReference('sodium-lore-442819-k0', '311_taxi_complaints'))
```

The code sets up a connection to BigQuery, a data warehouse service. It first tries to find a service account key from an environment variable or a specific file path. If successful, it uses this key to authenticate and create a BigQuery client. This client can then be used to perform various data operations, such as creating datasets. The code includes error handling to catch potential issues during the connection process.

```
[24] def upload_bigquery_table(logging, bqclient, table_path, write_disposition, df):
        """
        upload_bigquery_table
        Accepts a path to a BigQuery table, the write disposition and a dataframe
        Loads the data into the BigQuery table from the dataframe.
        for credentials.
        The write disposition is either
        write_disposition="WRITE_TRUNCATE"  Erase the target data and load all new data.
        write_disposition="WRITE_APPEND"    Append to the existing table
        """
        try:
            logging.info('Creating BigQuery Job configuration with write_disposition=%s', write_disposition)
            job_config = bigquery.LoadJobConfig(write_disposition=write_disposition)
            logging.info('Submitting the BigQuery job')
            job = bqclient.load_table_from_dataframe(df, table_path, job_config=job_config)
            logging.info('Job  results: %s',job.result())
        except Exception as err:
            logging.error('Failed to load BigQuery Table. %s', err)
```

```
[25] def bigquery_table_exists(bqclient, table_path):
        """
        bigquery_table_exists
        Accepts a path to a BigQuery table
        Checks if the BigQuery table exists.
        Returns True or False
        """
        try:
            bqclient.get_table(table_path)
            return True
        except NotFound:
            return False
```

```
[26] def query_bigquery_table(logging, table_path, bqclient, surrogate_key):
        """
        query_bigquery_table
        Accepts a path to a BigQuery table and the name of the surrogate key
        Queries the BigQuery table but leaves out the update_timestamp and surrogate key columns
        Returns the dataframe
        """
        bq_df = pd.DataFrame
        sql_query = 'SELECT * EXCEPT ( update_timestamp, ' + surrogate_key + ') FROM `' + table_path + '`'
        logging.info('Running query: %s', sql_query)
        try:
            bq_df = bqclient.query(sql_query).to_dataframe()
        except Exception as err:
            logging.info('Error querying the table. %s', err)
        return bq_df
```

```
[27] def add_surrogate_key(df, dimension_name, offset=1):
        """
        add_surrogate_key
        Accepts a data frame and inserts an integer identifier as the first column
        Returns the modified dataframe
        """
        df.reset_index(drop=True, inplace=True)
        df.insert(0, dimension_name + '_dim_id', df.index + offset)
        return df
```

```
def add_update_date(df, current_date):
        """
        add_update_date
        Accepts a data frame and inserts the current date as a new field
        Returns the modified dataframe
        """
        df['update_date'] = pd.to_datetime(current_date)
        return df
```

```
[29] def add_update_timestamp(df):
        """
        add_update_timestamp
        Accepts a data frame and inserts the current datetime as a new field
        Returns the modified dataframe
        """
        df['update_timestamp'] = pd.Timestamp('now', tz='utc').replace(microsecond=0)
        return df
```

```
def build_new_table(logging, bqclient, dimension_table_path, dimension_name, df):
    """
    build_new_table
    Accepts a path to a dimensional table, the dimension name and a data frame
    Add the surrogate key and a record timestamp to the data frame
    Inserts the contents of the dataframe to the dimensional table.
    """
    logging.info('Target dimension table %s does not exit', dimension_table_path)

    if df is not None and not df.empty:
        df = add_surrogate_key(df, dimension_name, 1)
        df = add_update_timestamp(df)
        upload_bigquery_table(logging, bqclient, dimension_table_path, 'WRITE_TRUNCATE', df)
    else:
        logging.warning('No data to insert into the new table.')
```

```
[31] def insert_existing_table(logging, bqclient, dimension_table_path, dimension_name, surrogate_key, df):
        """
        insert_existing_table
        Accepts a path to a dimensional table, the dimension name and a data frame
        Compares the new data to the existing data in the table.
        Inserts the new/modified records to the existing table
        """
        bq_df = pd.DataFrame()
        logging.info('Target dimension table %s exits. Checking for differences.', dimension_table_path)
        bq_df = query_bigquery_table(logging, dimension_table_path, bqclient, surrogate_key)
        new_records_df = df[~df.apply(tuple,1).isin(bq_df.apply(tuple,1))]
        logging.info('Found %d new records.', new_records_df.shape[0])
        if new_records_df.shape[0] > 0:
            new_surrogate_key_value = bq_df.shape[0] + 1
            new_records_df = add_surrogate_key(new_records_df, dimension_name, new_surrogate_key_value)
            new_records_df = add_update_timestamp(new_records_df)
            upload_bigquery_table(logging, bqclient, dimension_table_path, 'WRITE_APPEND', new_records_df)
```

These functions handle data operations in BigQuery, including creating new dimensional tables, inserting new records, and updating existing ones. They check for existing data, add timestamps and unique identifiers, and upload data to the specified tables.

Dimension Tables

311 Taxi Complaints Dataset

```
[32] import os
     # Check if the file exists
     print(os.path.exists('/content/drive/My Drive/CIS9350/311_Taxi_Complaint.csv'))
```

```
True
```

```
[33] #gcp_project = 'sodium-lore-442819-k0'
     #bq_dataset = '311_taxi_complaints'
     #dimension_table_path = f'{gcp_project}.taxi_and_restaurant_dataset.{table_name}'
```

```
df1 = pd.read_csv('311_Taxi_Complaint.csv')
print(df1.columns)
```

```
Index(['unique_key', 'created_date', 'agency', 'agency_name', 'complaint_type',
       'descriptor', 'location_type', 'incident_zip', 'incident_address',
       'street_name', 'cross_street_1', 'cross_street_2',
       'intersection_street_1', 'intersection_street_2', 'address_type',
       'city', 'landmark', 'status', 'community_board', 'bbl', 'borough',
       'x_coordinate_state_plane', 'y_coordinate_state_plane',
       'open_data_channel_type', 'park_facility_name', 'park_borough',
       'taxi_pick_up_location', 'latitude', 'longitude', 'location',
       'bridge_highway_name', 'bridge_highway_direction', 'road_ramp',
       'bridge_highway_segment', 'closed_date', 'resolution_description',
       'resolution_action_updated_date', 'taxi_company_borough'],
      dtype='object')
```

[36]
```
import os
import logging
import pandas as pd
import gdown
from datetime import datetime
from google.cloud import bigquery  # Make sure the BigQuery client library is installed
```

[37]
```
# Constants for GCP and file paths
gcp_project = 'sodium-lore-442819-k0'
bq_dataset = '311_taxi_complaints'
csv_file_path = '/content/drive/My Drive/CIS9350/311_Taxi_Complaint.csv'
log_file_dir = '/content/drive/My Drive/CIS9350/logs/'
```

[38]
```
# Dimension dictionary for Taxi Complaints
dim_dict = {
    'complaint_type': ['complaint_type', 'descriptor'],
    'location': ['location_type', 'street_name', 'city', 'borough', 'incident_zip', 'longitude', 'latitude'],
    'agency': ['agency', 'agency_name'],
    'complaints_status': ['status'],
}
```

```
from google.colab import auth
auth.authenticate_user()

# Main ETL process for each dimension
def process_etl(logging, dimension_name, file_path, columns, dimension_table_path):
    """ETL process for a specific dimension."""
    try:
        # Initialize the dataframe
        df = pd.DataFrame()

        # Load and transform data
        df = load_csv_data_file(logging, file_path, df)

        if df.empty:
            logging.warning(f"No data found in {file_path}. Skipping dimension {dimension_name}.")
            return

        df = transform_data(logging, columns, df)
        # Create BigQuery client
        bqclient = create_bigquery_client(logging)
        target_table_exists = bigquery_table_exists(bqclient, dimension_table_path)
        # Load data into BigQuery
        if not target_table_exists:
            build_new_table(logging, bqclient, dimension_table_path, dimension_name, df)  # Pass df as an argument
        else:
          # Set the name of the surrogate key
            surrogate_key = f"{dimension_name}_dim_id"
            insert_existing_table(logging, bqclient, dimension_table_path,dimension_name, surrogate_key, df)

        logging.info(f"Successfully processed dimension {dimension_name}.")
    except Exception as e:
        logging.error(f"Error processing dimension {dimension_name}: {e}")

# Main loop to run ETL for all dimensions
def run_etl():
    for key, value in dim_dict.items():
        dimension_name = key
        columns = value
        table_name = f'{dimension_name}_dimension'
        dimension_table_path = f'{gcp_project}.{bq_dataset}.{table_name}'

        # Process the data for the specific dimension
        process_etl(logging, dimension_name, csv_file_path, columns, dimension_table_path)
```

```
        logging.info(f"ETL process completed for dimension {dimension_name}.")
    logging.shutdown()

# Run the ETL process
if __name__ == "__main__":
    run_etl()
```

```
import os

# Directory where the logs are stored
log_directory = '/content/drive/My Drive/CIS9350/logs/'

# List all log files
log_files = [f for f in os.listdir(log_directory) if f.endswith('.log')]

# Print file details (simulating 'ls -l')
for log_file in log_files:
    file_path = os.path.join(log_directory, log_file)
    file_stats = os.stat(file_path)
    print(f"{log_file} - Size: {file_stats.st_size} bytes - Last Modified: {datetime.fromtimestamp(file_stats.st_mtime)}")
```

```
etl_complaint_type_20241126.log - Size: 0 bytes - Last Modified: 2024-11-28 21:17:06
etl_location_20241126.log - Size: 0 bytes - Last Modified: 2024-11-28 21:17:22
etl_location_20241128.log - Size: 4773 bytes - Last Modified: 2024-11-28 22:32:20
etl_inspection_type_20241128.log - Size: 4899 bytes - Last Modified: 2024-11-28 22:32:22
etl_grade_20241128.log - Size: 4719 bytes - Last Modified: 2024-11-28 22:32:23
etl_violation_code_20241128.log - Size: 4881 bytes - Last Modified: 2024-11-28 22:32:24
etl_critical_flag_20241128.log - Size: 4863 bytes - Last Modified: 2024-11-28 22:32:30
etl_restaurant_type_20241128.log - Size: 2412 bytes - Last Modified: 2024-11-29 00:44:07
etl_inspection_type_20241130.log - Size: 9786 bytes - Last Modified: 2024-11-30 19:51:13
etl_location_20241130.log - Size: 9534 bytes - Last Modified: 2024-11-30 19:51:15
etl_grade_20241130.log - Size: 9426 bytes - Last Modified: 2024-11-30 19:51:16
etl_violation_code_20241130.log - Size: 9750 bytes - Last Modified: 2024-11-30 19:51:18
etl_critical_flag_20241130.log - Size: 9714 bytes - Last Modified: 2024-11-30 19:51:18
etl_restaurant_type_20241130.log - Size: 277511 bytes - Last Modified: 2024-11-30 23:30:30
etl_complaint_fact__20241202.log - Size: 1694 bytes - Last Modified: 2024-12-02 18:40:30
```

These codes helped us create dimensions for 311_Taxi_Complaints dataset other than time and date which are shown below in while creating the location dimension we replaced all the NaN values to “Unknown” in location type, Street name, and city and replaced NaN values to “0” for incident_zip, Latitude and Longitude.

DATE DIMENSION

```
def generate_date_dimension(start, end):
    """
    generate_date_dimension
    Generates a DataFrame with date dimension information.
    """
    df = pd.DataFrame({"full_date": pd.date_range(start=start, end=end)})
    df["weekday_name"] = df.full_date.dt.strftime("%A")
    df["month_name"] = df.full_date.dt.strftime("%B")
    df["day_of_month"] = df.full_date.dt.strftime("%d")
    df["month_of_year"] = df.full_date.dt.strftime("%m")
    df["quarter"] = df.full_date.dt.quarter
    df["year"] = df.full_date.dt.strftime("%Y")
    df["date_dim_Id"] = range(1, len(df) + 1)
    return df
```

Each date is transformed into columns such as the Day of the month can be month and quarter can be 1 and follows.

```
[46] def check_for_null_and_duplicates(df):
        """
        Checks for null values and duplicate rows in the DataFrame.
        """
        null_columns = df.isnull().sum()
        if null_columns.any():
            logging.warning(f"Null values found in columns: {null_columns[null_columns > 0]}")
        else:
            logging.info("No null values found.")

    #checking for duplicates
        duplicate_rows = df.duplicated().sum()
        if duplicate_rows > 0:
            logging.warning(f"{duplicate_rows} duplicate rows found.")
        else:
            logging.info("No duplicate rows found.")
```

```
[47] def build_new_table(logging, bqclient, dimension_table_path, dimension_name, df):
        """
        build_new_table
        Loads a DataFrame into a BigQuery table.
        """
        try:
            job_config = bigquery.LoadJobConfig(write_disposition="WRITE_TRUNCATE")
            job = bqclient.load_table_from_dataframe(df, dimension_table_path, job_config=job_config)
            job.result()  # Wait for the load job to complete
            logging.info(f"Loaded {len(df)} rows into {dimension_table_path}.")
        except Exception as err:
            logging.error(f"Failed to create {dimension_table_path} table.", exc_info=True)
            raise err
```

```python
if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO)

    # Define parameters
    gcp_project = "sodium-lore-442819-k0"
    bq_dataset = "311_taxi_complaints"
    dimension_name = "date"
    table_name = f"{dimension_name}_dimension"
    dimension_table_path = ".".join([gcp_project, bq_dataset, table_name])

    # Date Dimension DataFrame
    df = generate_date_dimension(start="2021-01-01", end="2024-10-31")

    check_for_null_and_duplicates(df)

    bqclient = create_bigquery_client()

    #  Create a new table if it does not exist
    dataset_ref = bqclient.dataset(bq_dataset)
    try:
        bqclient.get_dataset(dataset_ref)
        print(f"Dataset {bq_dataset} already exists.")
    except NotFound:
        print(f"Dataset {bq_dataset} not found, creating it...")
        dataset = bigquery.Dataset(dataset_ref)
        dataset = bqclient.create_dataset(dataset)
        print(f"Dataset {dataset.dataset_id} created.")

    # Check if the table exists and build it if it doesn't
    if not bigquery_table_exists(bqclient, dimension_table_path):
        build_new_table(logging, bqclient, dimension_table_path, dimension_name, df)
        print(f"Table {dimension_table_path} created successfully.")
    else:
        print(f"Table {dimension_table_path} already exists. Will not overwrite it.")

    logging.shutdown()
```

```
Dataset 311_taxi_complaints already exists.
Table sodium-lore-442819-k0.311_taxi_complaints.date_dimension already exists. Will not overwrite it.
```

For date dimensions, we only extracted the ranges from 2021-01-01 to 2024-10-30 as we will only be analyzing from that range.

TIME DIMENSION

```python
import pandas as pd

def create_time_dimension(start_date, end_date):
    """
    Creates a Pandas DataFrame with time dimension columns.

    Args:
        start_date (str): Start date in YYYY-MM-DD format.
        end_date (str): End date in YYYY-MM-DD format.

    Returns:
        pandas.DataFrame: A DataFrame with time dimension columns.
    """

    date_range = pd.date_range(start=start_date, end=end_date)
    df = pd.DataFrame(date_range, columns=['Date'])

    df['Year'] = df['Date'].dt.year
    df['Quarter'] = df['Date'].dt.quarter
    df['Month'] = df['Date'].dt.month
    df['Day'] = df['Date'].dt.day
    df['DayOfWeek'] = df['Date'].dt.dayofweek
    df['DayName'] = df['Date'].dt.day_name()
    df['Hour'] = df['Date'].dt.hour
    df['Minute'] = df['Date'].dt.minute
    df['Second'] = df['Date'].dt.second
    df['WeekOfYear'] = df['Date'].dt.isocalendar().week

    return df

def load_to_bigquery(df, project_id, dataset_id, table_id):
    """
    Loads a Pandas DataFrame to a BigQuery table.

    Args:
        df (pandas.DataFrame): The DataFrame to load.
        project_id (str): The Google Cloud Project ID.
        dataset_id (str): The BigQuery dataset ID.
        table_id (str): The BigQuery table ID.
```

```
    client = bigquery.Client(project=project_id)
    # Create a fully-qualified table ID without redundant project ID in dataset_id
    table_ref = f"{project_id}.{dataset_id.split('.')[-1]}.{table_id}"  # Change is here
    job_config = bigquery.LoadJobConfig(
        write_disposition="WRITE_TRUNCATE"
    )
    job = client.load_table_from_dataframe(df, table_ref, job_config=job_config)
    job.result()


__name__ == "__main__":
    # Replace with your project ID, dataset ID, and table name
    project_id = "sodium-lore-442819-k0"
    dataset_id = "sodium-lore-442819-k0.311_taxi_complaints"  # Includes project ID
    table_id = "time_dimension"

    # Create the time dimension DataFrame
    time_dim_df = create_time_dimension("2021-01-01", "2024-10-31")

    # Load the DataFrame to BigQuery
    load_to_bigquery(time_dim_df, project_id, dataset_id, table_id)
```

It first generates a range of dates and then extracts various time components like year, quarter, month, day of week, and more. This enriched dataset is then loaded into a specified BigQuery table, providing a valuable resource for time-based analysis and reporting.

Creating Fact Table

```
def dimension_lookup(logging, dimension_name='agency', lookup_columns=['agency', 'agency_name'], df=df):
    """
    dimension_lookup
    Lookup the lookup_columns in the dimension_name and return the associated surrogate keys
    Returns dataframe augmented with the surrogate keys
    """
    bq_df = pd.DataFrame
    logging.info("Lookup dimension %s.", dimension_name)
    surrogate_key = dimension_name+"_dim_id"
    dimension_table_path = ".".join([gcp_project,bq_dataset,dimension_name+"_dimension"])
    # Fetch the existing table
    bq_df = query_bigquery_table(logging, dimension_table_path, bqclient, surrogate_key)
    # print(bq_df)
    # Melt the dimension dataframe into an index with the lookup columns
    m = bq_df.melt(id_vars=lookup_columns, value_vars=surrogate_key)
    # print(m)
    # Rename the "value" column to the surrogate key column name
    m=m.rename(columns={"value":surrogate_key})
    # Merge with the fact table record
    df = df.merge(m, on=lookup_columns, how='left')
    # Drop the "variable" column and the lookup columns
    df = df.drop(columns=lookup_columns)
    df = df.drop(columns="variable")
    return df
```

This helps merge a fact table with a dimension table to replace natural keys in the fact table with their corresponding surrogate keys from the dimension table.

```
def date_dimension_lookup(logging, dimension_name='date', lookup_column='created_date', df=df):
    """
    date_dimension_lookup
    Lookup the lookup_columns in a date dimension and return the associated surrogate keys
    Returns dataframe augmented with the surrogate keys
    """
    bq_df = pd.DataFrame
    logging.info("Lookup date dimension on column %s.", lookup_column)
    surrogate_key = dimension_name+"_dim_id"
    dimension_table_path = ".".join([gcp_project,bq_dataset,dimension_name+"_dimension"])
    # Fetch the existing table
    bq_df = query_bigquery_table(logging, dimension_table_path, bqclient, surrogate_key)
    bq_df["full_date"] = pd.to_datetime(bq_df.full_date, format="%Y-%m-%d %H:%M:%S")
    # Return just the date portion
    bq_df["full_date"] = bq_df.full_date.dt.date

    # Dates in the 311 data look like this: 2017-08-11T11:57:00.000
    # Extract the date from 'created_date' column
    # Updated the format string to include the "T"
    df[lookup_column] = pd.to_datetime(df[lookup_column], format="%Y-%m-%dT%H:%M:%S.%f")
    # Return just the date portion
    df[lookup_column] = df[lookup_column].dt.date

    # Melt the dimension dataframe into an index with the lookup columns
    m = bq_df.melt(id_vars='full_date', value_vars=surrogate_key)
    # Rename the "value" column to the surrogate key column name
    m=m.rename(columns={"value":lookup_column+"_dim_id"})

    # Merge with the fact table record on the created_date
    df = df.merge(m, left_on=lookup_column, right_on='full_date', how='left')

    # Drop the "variable" column and the lookup columns
    df = df.drop(columns=lookup_column)
    df = df.drop(columns="variable")
    df = df.drop(columns="full_date")
    return df
```

```python
def time_dimension_lookup(logging, dimension_name='time', lookup_column='created_date', df=df):
    """
    time_dimension_lookup
    Lookup the lookup_columns in the time dimension and return the associated surrogate key
    Returns dataframe augmented with the surrogate keys
    """
    bq_df = pd.DataFrame
    logging.info("Lookup time dimension on column %s.", lookup_column)
    surrogate_key = dimension_name+"_dim_id"
    dimension_table_path = ".".join([gcp_project,bq_dataset,dimension_name+"_dimension"])

    # Dates in the 311 data look like this: 2017-08-11T11:57:00.000
    # We can strip off the time portion after the letter "T" to ge the hours and minutes
    # time_dim_id = (hours*60)+minutes+1
    # Example:  Time is 22:07  so  (22*60)+7+1 = 1328
    # Extract the date from 'created_date' column and save it in a temporary column
    df[lookup_column+"_newdate"] = pd.to_datetime(df[lookup_column], format="%Y-%m-%dT%H:%M:%S.%f") # Updated format string
    # Strip off the hours and minutes portions
    df[lookup_column+"_hours"] = df[lookup_column+"_newdate"].dt.strftime("%H").astype(int)
    df[lookup_column+"_minutes"] = df[lookup_column+"_newdate"].dt.strftime("%M").astype(int)
    # Now assign the time_dim_id
    df[surrogate_key] = (df[lookup_column+"_hours"]*60)+df[lookup_column+"_minutes"]+1
    print("Surrogate key is: ", surrogate_key)
    print(df[surrogate_key])
    # Drop the lookup time columns
    df = df.drop(columns=lookup_column+"_newdate")
    df = df.drop(columns=lookup_column+"_hours")
    df = df.drop(columns=lookup_column+"_minutes")
    return df
```

```python
if __name__ == "__main__":
    # df = pd.DataFrame  <-- This was the error. Removed this line.
    # Create the BigQuery Client
    bqclient = create_bigquery_client(logging)
    # Load in the data file
    df = load_csv_data_file(logging, file_source_path, "/content/drive/MyDrive/CIS9350/311_Taxi_Complaint.csv", pd.DataFrame())
    # If city is empty, fill it in with NEW YORK
    df.city = df.city.fillna('NEW YORK')
    # Consider removing columns that we will never use
    df = df.drop(['unique_key', 'cross_street_1', 'cross_street_2', 'intersection_street_1', 'intersection_street_2', 'address_type', 'landmark'
    , 'community_board', 'bbl', 'x_coordinate_state_plane', 'y_coordinate_state_plane', 'park_facility_name'
    , 'park_borough', 'taxi_pick_up_location', 'bridge_highway_name', 'bridge_highway_direction', 'road_ramp'
    , 'bridge_highway_segment', 'resolution_description', 'resolution_action_updated_date', 'taxi_company_borough'], axis=1)

    # Lookup the agency dimension record
    df = dimension_lookup(logging, dimension_name='agency', lookup_columns=['agency', 'agency_name'], df=df)
```

```python
  # Lookup the location dimension record
  df = dimension_lookup(logging, dimension_name='location', lookup_columns=['location_type', 'street_name', 'city', 'borough', 'incident_zip', 'longitude', 'latitude'], df=df)

  # Lookup the complaint_type dimension record
  df = dimension_lookup(logging, dimension_name='complaint_type', lookup_columns=['complaint_type'], df=df)

  # Lookup the complaint_status dimension record
  df = dimension_lookup(logging, dimension_name='complaints_status', lookup_columns=['status'], df=df)



Lookup the time dimension record using the time part of the created_date
  df = time_dimension_lookup(logging, dimension_name='time', lookup_column='created_date', df=df)
  # Rename the resulting column to 'time_dim_id' if it's not already named that
  if 'time_dim_id' not in df.columns and 'created_date_dim_id' in df.columns:
      df = df.rename(columns={'created_date_dim_id': 'time_dim_id'})
  # Lookup the created_date dimension record
  df = date_dimension_lookup(logging, dimension_name='date', lookup_column='created_date', df=df)
  # Rename the resulting column to 'date_dim_id' if it's not already named that
  if 'date_dim_id' not in df.columns and 'created_date_dim_id' in df.columns:
      df = df.rename(columns={'created_date_dim_id': 'date_dim_id'})
  surrogate_keys=['agency_dim_id', 'location_dim_id', 'complaint_type_dim_id', 'complaints_status_dim_id', 'time_dim_id', 'date_dim_id']

  # Remove all of the other non-surrogate key columns
  df = df[surrogate_keys]
```

```
    # Add a complaint count
    df['complaint_count'] = 1
    # Count up the number of complaints per agency, per location, etc.
    df = df.groupby(surrogate_keys)['complaint_count'].agg('count').reset_index()

    # See if the target table exists
    target_table_exists = bigquery_table_exists(fact_table_path, bqclient)
    # If the target table does not exist, load all of the data into a new table
    if not target_table_exists:
        build_new_table(logging, bqclient, fact_table_path, df)
    # If the target table exists, then perform an incremental load
    if target_table_exists:
        insert_existing_table(logging, bqclient, fact_table_path, df)
```

```
Surrogate key is:  time_dim_id
0        1426
1        1426
2        1426
3        1426
4        1352
         ...
28783      25
28784      12
28785      12
28786      12
28787      12
Name: time_dim_id, Length: 28788, dtype: int64
```

```python
# Dimension dictionary for Taxi Complaints
dim_dict = {
    'complaint_type': ['complaint_type', 'descriptor'],
    'location': ['location_type', 'street_name', 'city', 'borough', 'incident_zip', 'longitude', 'latitude'],
    'agency': ['agency', 'agency_name'],
    'complaints_status': ['status'],
}
```

+ Code + Text

```python
df = pd.DataFrame
# Set the name of the dimension
fact_name = 'Taxi'

# Set the GCP Project, dataset and table name
gcp_project = 'sodium-lore-442819-k0'
bq_dataset = '311_taxi_complaints'
table_name = fact_name + '_fact'
# Construct the full BigQuery path to the table
fact_table_path = ".".join([gcp_project,bq_dataset,table_name])

# Set the path to the source data files
# For Linux use something like    /home/username/python_etl
# For Mac use something like     /users/username/python_etl
# file_source_path = 'c:\\Python_ETL'
file_source_path = '/content'
```

```python
# Set up logging
for handler in logging.root.handlers[:]:
    logging.root.removeHandler(handler)
current_date = datetime.today().strftime('%Y%m%d')
log_filename = "_".join(["etl_complaint_fact_",current_date])+".log"
logging.basicConfig(filename=log_filename, encoding='utf-8', format='%(asctime)s %(message)s', level=logging.DEBUG)
logging.info("================================================================================")
logging.info("Starting ETL Run for complaint fact on date "+current_date)
```

It aggregates the data by counting complaints for each combination of dimensions and loads the results into a BigQuery table.

Final Dimension Schema:

Taxi Complaints:

The following screenshots display the final dimensional schema in Google BigQuery, illustrating each dimension after the ETL process has been successfully completed.

1. **Agency Dimension:**

SCHEMA | DETAILS | PREVIEW | TABLE EXPLORER PREVIEW | INSIGHTS | LINEAGE

Row	agency_dim_id	agency	agency_name	update_timestamp
1	1	TLC	Taxi and Limousine Commission	2024-11-28 21:54:27 UTC

2. **Complaint Type Dimension:**

SCHEMA | DETAILS | PREVIEW | TABLE EXPLORER PREVIEW | INSIGHTS | LINEAGE | DAT

Row	complaint_type_	complaint_type	descriptor	update_timestamp
1	1	Taxi Complaint	Driver Complaint - Passenger	2024-11-28 21:54:15 UTC
2	2	Taxi Complaint	Driver Complaint - Non Passen...	2024-11-28 21:54:15 UTC
3	3	Taxi Complaint	Vehicle Complaint	2024-11-28 21:54:15 UTC
4	4	Taxi Complaint	Jewelry	2024-11-28 21:54:15 UTC

3. **Complaint Status Dimension:**

SCHEMA | DETAILS | PREVIEW | TABLE EXPLORER PREVIEW | INSIGHT

Row	complaints_stat	status	update_timestamp
1	1	In Progress	2024-11-28 21:54:31 UTC
2	2	Closed	2024-11-28 21:54:31 UTC

status

4. **Location Dimension:**

SCHEMA | DETAILS | PREVIEW | TABLE EXPLORER PREVIEW | INSIGHTS | LINEAGE | DATA PROFILE | DATA QUALITY

Row	location_dim_id	location_type	street_name	city	borough	incident_zip	longitude	latitude	update_timestamp
1	1105	Street	JFK	JAMAICA	QUEENS	11430.0	-73.788281...	40.6483204...	2024-12-01 22:14:55 UTC
2	1481	Street	JFK	JAMAICA	QUEENS	11430.0	-73.795070...	40.6577034...	2024-12-01 22:14:55 UTC
3	1169	Street	JFK	JAMAICA	QUEENS	11430.0	-73.782071...	40.6573557...	2024-12-01 22:14:55 UTC
4	2668	Street	JFK	JAMAICA	QUEENS	11430.0	-73.779834...	40.6601456...	2024-12-01 22:14:55 UTC
5	1114	Street	LGA	EAST ELMHURST	QUEENS	11369.0	-73.877294...	40.7744208...	2024-12-01 22:14:55 UTC
6	6194	unknown	BOWERY	NEW YORK	MANHATTAN	10003.0	-73.991593...	40.7269331...	2024-12-02 21:00:39 UTC
7	1224	unknown	BOWERY	NEW YORK	MANHATTAN	10003.0	-73.991593...	40.7269331...	2024-12-01 22:14:55 UTC
8	5101	unknown	BOWERY	NEW YORK	MANHATTAN	10003.0	-73.991593...	40.7269331...	2024-12-02 20:58:57 UTC
9	2492	Street	BOWERY	NEW YORK	MANHATTAN	10003.0	-73.991593...	40.7269331...	2024-12-01 22:14:55 UTC
10	4000	unknown	BOWERY	NEW YORK	MANHATTAN	10003.0	-73.991593...	40.7269331...	2024-12-01 22:15:42 UTC
11	2910	unknown	BOWERY	NEW YORK	MANHATTAN	10003.0	-73.991593...	40.7269331...	2024-12-01 17:30:21 UTC
12	3643	Street	BOWERY	NEW YORK	MANHATTAN	10003.0	-73.992012...	40.7257556...	2024-12-01 22:14:55 UTC
13	2264	Bridge	BOWERY	NEW YORK	MANHATTAN	10002.0	-73.994736...	40.7186057...	2024-12-01 22:14:55 UTC
14	3743	Street	BOWERY	NEW YORK	MANHATTAN	10012.0	-73.992391...	40.7247538...	2024-12-01 22:14:55 UTC
15	1806	Street	BOWERY	NEW YORK	MANHATTAN	10013.0	-73.996378...	40.7157622...	2024-12-01 22:14:55 UTC
16	3445	Street	BOWERY	NEW YORK	MANHATTAN	10012.0	-73.993820...	40.7209003...	2024-12-01 22:14:55 UTC

5. Date Dimension:

Row	full_date	weekday_name	month_name	day_of_month	month_of_year	quarter	year
1	2021-01-01T00:00:00	Friday	January	01	01	1	2021
2	2021-01-08T00:00:00	Friday	January	08	01	1	2021
3	2021-01-15T00:00:00	Friday	January	15	01	1	2021
4	2021-01-22T00:00:00	Friday	January	22	01	1	2021
5	2021-01-29T00:00:00	Friday	January	29	01	1	2021
6	2022-01-07T00:00:00	Friday	January	07	01	1	2022
7	2022-01-14T00:00:00	Friday	January	14	01	1	2022
8	2022-01-21T00:00:00	Friday	January	21	01	1	2022
9	2022-01-28T00:00:00	Friday	January	28	01	1	2022
10	2023-01-06T00:00:00	Friday	January	06	01	1	2023
11	2023-01-13T00:00:00	Friday	January	13	01	1	2023
12	2023-01-20T00:00:00	Friday	January	20	01	1	2023
13	2023-01-27T00:00:00	Friday	January	27	01	1	2023
14	2024-01-05T00:00:00	Friday	January	05	01	1	2024
15	2024-01-12T00:00:00	Friday	January	12	01	1	2024
16	2024-01-19T00:00:00	Friday	January	19	01	1	2024
17	2024-01-26T00:00:00	Friday	January	26	01	1	2024

6. Time Dimension:

Row	Date	Year	Quarter	Month	Day	DayOfWeek	DayName	Hour	Minute	Second	WeekOfYear
1	2021-01-04T00:00:00	2021	1	1	4	0	Monday	0	0	0	1
2	2021-01-11T00:00:00	2021	1	1	11	0	Monday	0	0	0	2
3	2021-01-18T00:00:00	2021	1	1	18	0	Monday	0	0	0	3
4	2021-01-25T00:00:00	2021	1	1	25	0	Monday	0	0	0	4
5	2021-02-01T00:00:00	2021	1	2	1	0	Monday	0	0	0	5
6	2021-02-08T00:00:00	2021	1	2	8	0	Monday	0	0	0	6
7	2021-02-15T00:00:00	2021	1	2	15	0	Monday	0	0	0	7
8	2021-02-22T00:00:00	2021	1	2	22	0	Monday	0	0	0	8
9	2021-03-01T00:00:00	2021	1	3	1	0	Monday	0	0	0	9
10	2021-03-08T00:00:00	2021	1	3	8	0	Monday	0	0	0	10
11	2021-03-15T00:00:00	2021	1	3	15	0	Monday	0	0	0	11
12	2021-03-22T00:00:00	2021	1	3	22	0	Monday	0	0	0	12
13	2021-03-29T00:00:00	2021	1	3	29	0	Monday	0	0	0	13
14	2021-04-05T00:00:00	2021	2	4	5	0	Monday	0	0	0	14
15	2021-04-12T00:00:00	2021	2	4	12	0	Monday	0	0	0	15
16	2021-04-19T00:00:00	2021	2	4	19	0	Monday	0	0	0	16
17	2021-04-26T00:00:00	2021	2	4	26	0	Monday	0	0	0	17
18	2021-05-03T00:00:00	2021	2	5	3	0	Monday	0	0	0	18
19	2021-05-10T00:00:00	2021	2	5	10	0	Monday	0	0	0	19
20	2021-05-17T00:00:00	2021	2	5	17	0	Monday	0	0	0	20
21	2021-05-24T00:00:00	2021	2	5	24	0	Monday	0	0	0	21
22	2021-05-31T00:00:00	2021	2	5	31	0	Monday	0	0	0	22
23	2021-06-07T00:00:00	2021	2	6	7	0	Monday	0	0	0	23
24	2021-06-14T00:00:00	2021	2	6	14	0	Monday	0	0	0	24
25	2021-06-21T00:00:00	2021	2	6	21	0	Monday	0	0	0	25
26	2021-06-28T00:00:00	2021	2	6	28	0	Monday	0	0	0	26
27	2021-07-05T00:00:00	2021	3	7	5	0	Monday	0	0	0	27
28	2021-07-12T00:00:00	2021	3	7	12	0	Monday	0	0	0	28
29	2021-07-19T00:00:00	2021	3	7	19	0	Monday	0	0	0	29
30	2021-07-26T00:00:00	2021	3	7	26	0	Monday	0	0	0	30

Taxi Complaints Facts Table:

Row	agency_dim_id	location_dim_id	complaint_type	complaints_stat	time_dim_id	date_dim_id	complaint_count
1	1	1105	1	1	794	1281	1
2	1	1105	2	1	794	1281	1
3	1	1105	3	1	794	1281	1
4	1	1105	4	1	794	1281	1
5	1	1099	1	1	911	1284	1
6	1	1099	1	1	1175	1284	1
7	1	1099	2	1	911	1284	1
8	1	1099	2	1	1175	1284	1
9	1	1099	3	1	911	1284	1
10	1	1099	3	1	1175	1284	1
11	1	1099	4	1	911	1284	1
12	1	1099	4	1	1175	1284	1
13	1	1105	1	1	133	1284	1
14	1	1105	2	1	133	1284	1
15	1	1105	3	1	133	1284	1

Restaurant Inspection:

We began initiating the transformation phase, we developed reusable Python functions to standardize and streamline the ETL process. These functions help us ensure consistency in data handling and allow us to efficiently apply the same transformations across multiple datasets.

CREATING FUNCTIONS

```
[13] def create_bigquery_client(logging):
         try:
             bqclient = bigquery.Client.from_service_account_json("/content/drive/My Drive/CIS9350/my-service-account.json")
             logging.info('BigQuery Client created successfully.')
             return bqclient
         except Exception as err:
             logging.error('Failed to create BigQuery Client.', exc_info=True)
             raise err
```

```
[14] def load_csv_data_file(logging, file_source_path, file_name, df):
         try:
             full_path = f"{file_source_path}/{file_name}"
             logging.info(f"Loading file: {full_path}")
             df = pd.read_csv(full_path)
             logging.info(f"Loaded {len(df)} rows from {file_name}.")
             return df
         except FileNotFoundError:
             logging.error(f"File not found: {file_source_path}/{file_name}.")
             raise
         except Exception as e:
             logging.error(f"Error loading file {file_name}: {e}", exc_info=True)
```

```
[15] def bigquery_table_exists(bqclient, table_path):
         try:
             bqclient.get_table(table_path)
             return True
         except NotFound:
             return False
```

```
[16] def build_new_table(logging, bqclient, table_path, df):
          try:
              logging.info(f"Creating new table: {table_path}")
              #job_config = bigquery.LoadJobConfig(write_disposition="WRITE_TRUNCATE")
              job = bqclient.load_table_from_dataframe(df, table_path) #job_config=job_config)
              job.result()
              logging.info(f"Table {table_path} created with {len(df)} rows.")
          except Exception as e:
              logging.error(f"Failed to create table {table_path}: {e}", exc_info=True)
              raise
```

```
[17] def insert_into_existing_table(logging, bqclient, table_path, surrogate_key, df):
          try:
              logging.info(f"Inserting new records into existing table: {table_path}")
              # Query to fetch existing surrogate keys
              query = f"SELECT {surrogate_key} FROM `{table_path}`"
              existing_keys = bqclient.query(query).to_dataframe()[surrogate_key].tolist()
              # Filter out records already in the table
              df_new = df[~df[surrogate_key].isin(existing_keys)]
              if not df_new.empty:
                  job_config = bigquery.LoadJobConfig(write_disposition="WRITE_APPEND")
                  job = bqclient.load_table_from_dataframe(df_new, table_path, job_config=job_config)
                  job.result()
                  logging.info(f"Inserted {len(df_new)} new rows into {table_path}.")
              else:
                  logging.info("No new records to insert.")
          except Exception as e:
              logging.error(f"Error inserting new records into {table_path}: {e}", exc_info=True)
              raise
```

Inspection_Type

```python
def transform_inspection_type_data(logging, df):
    try:
        logging.info("Transforming data for inspection type dimension.")

        df = df[['inspection_type']]

        df = df.drop_duplicates()

        df['inspection_type'] = df['inspection_type'].fillna('Unknown')

        # categorizing inspection types and assigning numerical codes
        def categorize_inspection_type(inspection_type):
            # Define numerical codes for broader categories
            if 'Re-inspection' in inspection_type:
                return 2  # Re-inspection
            elif 'Initial Inspection' in inspection_type:
                return 1  # Initial Inspection
            elif 'Reopening Inspection' in inspection_type:
                return 3  # Reopening Inspection
            elif 'Second Compliance Inspection' in inspection_type:
                return 4  # Second Compliance Inspection
            elif 'Compliance Inspection' in inspection_type:
                return 5  # Compliance Inspection
            else:
                return 0  # Miscellaneous or unknown types

        # Applying categorize function to assign inspection_category as numerical codes
        df['inspection_category_code'] = df['inspection_type'].apply(categorize_inspection_type)

        df['inspection_type_dim_id'] = range(1, len(df) + 1)

        logging.info("Data transformation completed successfully.")
        return df

    except KeyError as e:
        logging.error(f"Missing required columns in the DataFrame: {e}")
        raise
    except Exception as e:
        logging.error(f"Error transforming inspection type data: {e}", exc_info=True)
        raise
```

The inspection type field was mapped to corresponding numeric values to create a new field, inspection_category_code. Each inspection type was assigned a unique numeric code for clarity and traceability. For instance, an “initial Inspection” was mapped to 1, a “Re-inspection” to 2 and so on.

```
[19] if __name__ == "__main__":
        logging.basicConfig(level=logging.INFO)

        # Parameters
        file_source_path = "/content/drive/My Drive/CIS9350"
        file_name = "Restaurant_Inspection.csv"
        gcp_project = "my-project-0577-434418"
        bq_dataset = "restaurant_inspection"
        table_name = "inspection_type_dimension"
        table_path = f"{gcp_project}.{bq_dataset}.{table_name}"
        surrogate_key = "inspection_type_dim_id"


        df = pd.DataFrame()

        # Load in the data file
        df = load_csv_data_file(logging, file_source_path, file_name, df)

        df = transform_inspection_type_data(logging, df)

        # Create the BigQuery client
        bqclient = create_bigquery_client(logging)

        # See if the target table exists
        if not bigquery_table_exists(bqclient, table_path):
            # Create a new table if it does not exist
            build_new_table(logging, bqclient, table_path, df)
        else:
            # Insert new records if the table alreadyexists
            insert_into_existing_table(logging, bqclient, table_path, surrogate_key, df)

        logging.shutdown()
```

Location_Type

```
[20] def transform_location_data(logging, df):
        try:
            logging.info("Transforming data for location dimension.")
            # Columns for Location dimension
            df = df[['street', 'boro', 'zipcode', 'latitude', 'longitude']]
            # Dropping duplicates
            df = df.drop_duplicates()

            # Handling null values
            df['street'] = df['street'].fillna('Unknown Street')
            df['boro'] = df['boro'].fillna('Unknown Borough')
            df['zipcode'] = df['zipcode'].fillna(99999)  # Replaced missing ZipCode with 99999
            df['latitude'] = df['latitude'].fillna(0.0)
            df['longitude'] = df['longitude'].fillna(0.0)

            # Setting city to "New York" for all rows
            df['city'] = "New York"

             # Reordered the columns to place 'city' after 'boro'
            df = df[['street', 'boro', 'city', 'zipcode', 'latitude', 'longitude']]  # Adjusted column order for granuality

            df['location_dim_id'] = range(1, len(df) + 1)

            logging.info("Data transformation complete.")
            return df
        except KeyError as e:
            logging.error(f"Missing required columns in the DataFrame: {e}")
            raise
        except Exception as e:
            logging.error(f"Error transforming location data: {e}", exc_info=True)
            raise
```

Since some columns had duplicates and some columns were missing, we replaced the missing zipcode with 99999 and street as unknown and added columns named “city” and assigned as “New York”.

```
[21] if __name__ == "__main__":
         logging.basicConfig(level=logging.INFO)

         # Parameters
         file_source_path = "/content/drive/My Drive/CIS9350"
         file_name = "Restaurant_Inspection.csv"
         gcp_project = "my-project-0577-434418"
         bq_dataset = "restaurant_inspection"
         table_name = "location_dimension"
         table_path = f"{gcp_project}.{bq_dataset}.{table_name}"
         surrogate_key = "location_dim_id"

         df = pd.DataFrame()

         df = load_csv_data_file(logging, file_source_path, file_name, df)

         df = transform_location_data(logging, df)

         bqclient = create_bigquery_client(logging)

         # See if the target table exists
         if not bigquery_table_exists(bqclient, table_path):
             # Create a new table if it does not exist
             build_new_table(logging, bqclient, table_path, df)
         else:
             # Insert new records if the table exists
             insert_into_existing_table(logging, bqclient, table_path, surrogate_key, df)

         logging.shutdown()
```

Grade_Dimension

```
[22] def assign_grade(score):
        """Assigns a letter grade based on the score."""
        if score <= 13:
            return 'A'
        elif score <= 27:
            return 'B'
        elif score >= 28:
            return 'C'
        else:
            return 'Unknown'
    def transform_grade_data(logging, df):
        try:
            logging.info("Transforming data for grade dimension.")

            # Assigning grades based on score ranges
            df['grade'] = df['score'].apply(assign_grade)

            # Create grade_description based on grade
            def get_grade_description(grade):
                if grade == 'A':
                    return 'Excellent'
                elif grade == 'B':
                    return 'Good'
                elif grade == 'C':
                    return 'Fair'
                elif grade == 'P':
                    return 'Pending'
                elif grade == 'Z':
                    return 'Closed'
                else:
                    return 'Unknown'  # For any other grade that is not A, B, C, P, or Z

            # Assigning grade descriptions based on the grade
            df['grade_description'] = df['grade'].apply(get_grade_description)

            # Created score ranges as numeric categories
            bins = [0, 13, 27, float('inf')]
            labels = ['0-13', '14-27', '28+']
            df['score_range'] = pd.cut(df['score'], bins=bins, labels=labels, right=True, include_lowest=True)

            df['grade'] = df['grade'].fillna('Unknown')
```

```
            df['grade'] = df['grade'].fillna('Unknown')
            df['grade_description'] = df['grade_description'].fillna('No description')
            df['score_range'] = df['score_range'].cat.add_categories('Unknown range').fillna('Unknown range')

            grade_dim = df[['grade', 'grade_description', 'score_range']].drop_duplicates()

            grade_dim['grade_dim_id'] = range(1, len(grade_dim) + 1)

            #created extra column named _index_level_0 and dropping it
            grade_dim = grade_dim.reset_index(drop=True)

            logging.info("Data transformation complete for grade dimension.")
            return grade_dim
        except KeyError as e:
            logging.error(f"Missing required columns in the DataFrame: {e}")
            raise
        except Exception as e:
            logging.error(f"Error transforming grade data: {e}", exc_info=True)
            raise
```

The restaurant score follows the below grades:

- *0-13 points = Grade A (Excellent)*
- *14-27 points = Grade B (Good)*
- *28+ points = Grade C (Fair)*
- *Grade P* (Pending) for reinspection, Grade N (Not rated) or Grade Z (Closed) if severe issues are identified

The grade function steps:

1. Assign Letter Grade: Based on the score, a letter grade is assigned (e.g., a score of 0-13 gets an "A" and follows).
2. Add Grade Description: After assigning the letter grade, a description is added to explain it (e.g., "A" is described as "Excellent" and follows).
3. Define Score Range: Finally, the score range is categorized based on score into three ranges 0-13, 14-27, and 28+ using pd. cut() array elements into different groups.
4. Exclusion of Grades P, Z and N from binning: They are not related to the numerical scoring system as these grades are assigned based on inspection status or conditions and unrelated to point values.

This process helps organize the data clearly, linking scores to grades, descriptions, and ranges without repeating the same letter grades for different records.

```
[22] # Main program execution for Grade Dimension
     if __name__ == "__main__":
         logging.basicConfig(level=logging.INFO)
         file_source_path = "/content/drive/My Drive/CIS9350"
         file_name = "Restaurant_Inspection.csv"
         gcp_project = "my-project-0577-434418"
         bq_dataset = "restaurant_inspection"
         table_name = "grade_dimension"
         table_path = f"{gcp_project}.{bq_dataset}.{table_name}"
         surrogate_key = "grade_dim_id"

         # Create an empty DataFrame
         df = pd.DataFrame()

         # Load the data file
         df = load_csv_data_file(logging, file_source_path, file_name, df)

         # Transform the data for the grade dimension
         df = transform_grade_data(logging, df)

         # Create BigQuery client
         bqclient = create_bigquery_client(logging)

         # Check if the target table exists
         if not bigquery_table_exists(bqclient, table_path):
             # Create a new table if it does not exist
             build_new_table(logging, bqclient, table_path, df)
         else:
             # Insert new records if the table exists
             insert_into_existing_table(logging, bqclient, table_path, surrogate_key, df)

         logging.shutdown()
```

Violation_Dimension

```
[24] def transform_violation_description_data(logging, df):
         try:
             logging.info("Transforming data for violation description dimension.")

             # Selecting columns
             df = df[['violation_code', 'violation_description']]

             df = df.drop_duplicates()

             df['violation_code'] = df['violation_code'].fillna('Unknown Code')
             df['violation_description'] = df['violation_description'].fillna('No Description')

             df['violation_description_dim_id'] = range(1, len(df) + 1)

             logging.info("Data transformation complete.")
             return df
         except KeyError as e:
             logging.error(f"Missing required columns in the DataFrame: {e}")
             raise
         except Exception as e:
             logging.error(f"Error transforming violation description data: {e}", exc_info=True)
             raise
```

```
[25] if __name__ == "__main__":
         logging.basicConfig(level=logging.INFO)


         file_source_path = "/content/drive/My Drive/CIS9350"
         file_name = "Restaurant_Inspection.csv"
         gcp_project = "my-project-0577-434418"
         bq_dataset = "restaurant_inspection"
         table_name = "violation_description_dimension"
         table_path = f"{gcp_project}.{bq_dataset}.{table_name}"
         surrogate_key = "violation_description_dim_id"

         df = pd.DataFrame()

         df = load_csv_data_file(logging, file_source_path, file_name, df)

         df = transform_violation_description_data(logging, df)

         bqclient = create_bigquery_client(logging)

          # See if the target table exists
         if not bigquery_table_exists(bqclient, table_path):
             # Create a new table if it does not exist
             build_new_table(logging, bqclient, table_path, df)
         else:
             # Insert new records if the table exists
             insert_into_existing_table(logging, bqclient, table_path, surrogate_key, df)

         logging.shutdown()
```

Critical_Flag

```
[26] def transform_critical_flag_data(logging, df):
         try:
             logging.info("Transforming data for Critical Flag dimension.")

             df = df[['critical_flag']].drop_duplicates().reset_index(drop=True)

             df['critical_flag'] = df['critical_flag'].fillna('Unknown')

             df['critical_flag_dim_id'] = range(1, len(df) + 1)

             logging.info("Data transformation complete for Critical Flag.")
             return df
         except KeyError as e:
             logging.error(f"Missing required columns in the DataFrame: {e}")
             raise
         except Exception as e:
             logging.error(f"Error transforming critical flag data: {e}", exc_info=True)
             raise
```

```
[27] if __name__ == "__main__":
         logging.basicConfig(level=logging.INFO)

         # Parameters
         file_source_path = "/content/drive/My Drive/CIS9350"
         file_name = "Restaurant_Inspection.csv"
         gcp_project = "my-project-0577-434418"
         bq_dataset = "restaurant_inspection"
         table_name = "critical_flag_dimension"
         table_path = f"{gcp_project}.{bq_dataset}.{table_name}"
         surrogate_key = "critical_flag_dim_id"

         df = pd.DataFrame()

         df = load_csv_data_file(logging, file_source_path, file_name, df)

         df = transform_critical_flag_data(logging, df)

         bqclient = create_bigquery_client(logging)

          # See if the target table exists
         if not bigquery_table_exists(bqclient, table_path):
             # Create a new table if it does not exist
             build_new_table(logging, bqclient, table_path, df)
         else:
             # Insert new records if the table exists
             insert_into_existing_table(logging, bqclient, table_path, surrogate_key, df)

         logging.shutdown()
```

Restaurant_Type_Dimension

```
[28] def transform_restaurant_type_data(logging, df):
         try:
             logging.info("Transforming data for Restaurant Type dimension.")

             df = df[['cuisine_description']]

             df = df.drop_duplicates()

             df['cuisine_description'] = df['cuisine_description'].fillna('Unknown Cuisine')

             df['restaurant_type_dim_id'] = range(1, len(df) + 1)

             logging.info("Data transformation complete for Restaurant Type.")
             return df
         except KeyError as e:
             logging.error(f"Missing required columns in the DataFrame: {e}")
             raise
         except Exception as e:
             logging.error(f"Error transforming restaurant type data: {e}", exc_info=True)
             raise
```

```
[29] if __name__ == "__main__":
        logging.basicConfig(level=logging.INFO)

        # Parameters
        file_source_path = "/content/drive/My Drive/CIS9350"
        file_name = "Restaurant_Inspection.csv"
        gcp_project = "my-project-0577-434418"
        bq_dataset = "restaurant_inspection"
        table_name = "restaurant_type_dimension"
        table_path = f"{gcp_project}.{bq_dataset}.{table_name}"
        surrogate_key = "restaurant_type_dim_id"

        df = pd.DataFrame()

        df = load_csv_data_file(logging, file_source_path, file_name, df)

        df = transform_restaurant_type_data(logging, df)

        bqclient = create_bigquery_client(logging)

         # See if the target table exists
        if not bigquery_table_exists(bqclient, table_path):
            # Create a new table if it does not exist
            build_new_table(logging, bqclient, table_path, df)
        else:
            # Insert new records if the table exists
            insert_into_existing_table(logging, bqclient, table_path, surrogate_key, df)

        logging.shutdown()
```

Date_Dimension

Creating Dimensions

```
[30] def generate_date_dimension(start, end):
        """
        generate_date_dimension
        Generates a DataFrame with date dimension information.
        """
        df = pd.DataFrame({"full_date": pd.date_range(start=start, end=end)})
        df["weekday_name"] = df.full_date.dt.strftime("%A")
        df["month_name"] = df.full_date.dt.strftime("%B")
        df["day_of_month"] = df.full_date.dt.strftime("%d")
        df["month_of_year"] = df.full_date.dt.strftime("%m")
        df["quarter"] = df.full_date.dt.quarter
        df["year"] = df.full_date.dt.strftime("%Y")
        df["date_dim_Id"] = range(1, len(df) + 1)
        return df
```

```
[31] def generate_date_dimension(start, end):
        """
        generate_date_dimension
        Generates a DataFrame with date dimension information.
        """
        df = pd.DataFrame({"full_date": pd.date_range(start=start, end=end)})
        df["weekday_name"] = df.full_date.dt.strftime("%A")
        df["month_name"] = df.full_date.dt.strftime("%B")
        df["day_of_month"] = df.full_date.dt.strftime("%d")
        df["month_of_year"] = df.full_date.dt.strftime("%m")
        df["quarter"] = df.full_date.dt.quarter
        df["year"] = df.full_date.dt.strftime("%Y")
        df["date_dim_id"] = range(1, len(df) + 1)
        return df
```

```
[32] def check_for_null_and_duplicates(df):
        """
        Checks for null values and duplicate rows in the DataFrame.
        """
        null_columns = df.isnull().sum()
        if null_columns.any():
            logging.warning(f"Null values found in columns: {null_columns[null_columns > 0]}")
        else:
            logging.info("No null values found.")

    #checking for duplicates
        duplicate_rows = df.duplicated().sum()
        if duplicate_rows > 0:
            logging.warning(f"{duplicate_rows} duplicate rows found.")
        else:
            logging.info("No duplicate rows found.")
```

```
[33] def build_new_table(logging, bqclient, dimension_table_path, dimension_name, df):
        """
        build_new_table
        Loads a DataFrame into a BigQuery table.
        """
        try:
            job_config = bigquery.LoadJobConfig(write_disposition="WRITE_TRUNCATE")
            job = bqclient.load_table_from_dataframe(df, dimension_table_path, job_config=job_config)
            job.result()  # Wait for the load job to complete
            logging.info(f"Loaded {len(df)} rows into {dimension_table_path}.")
        except Exception as err:
            logging.error(f"Failed to create {dimension_table_path} table.", exc_info=True)
            raise err
```

```
[34] if __name__ == "__main__":
         logging.basicConfig(level=logging.INFO)

         # Define parameters
         gcp_project = "my-project-0577-434418"
         bq_dataset = "restaurant_inspection"
         dimension_name = "date"
         table_name = f"{dimension_name}_dimension"
         dimension_table_path = ".".join([gcp_project, bq_dataset, table_name])

         # Date Dimension DataFrame
         df = generate_date_dimension(start="2021-01-01", end="2024-10-31")

         check_for_null_and_duplicates(df)

         bqclient = create_bigquery_client(logging)

         #  Create a new table if it does not exist
         dataset_ref = bqclient.dataset(bq_dataset)
         try:
             bqclient.get_dataset(dataset_ref)
             print(f"Dataset {bq_dataset} already exists.")
         except NotFound:
             print(f"Dataset {bq_dataset} not found, creating it...")
             dataset = bigquery.Dataset(dataset_ref)
             dataset = bqclient.create_dataset(dataset)
             print(f"Dataset {dataset.dataset_id} created.")

         # Check if the table exists and build it if it doesn't
         if not bigquery_table_exists(bqclient, dimension_table_path):
             build_new_table(logging, bqclient, dimension_table_path, dimension_name, df)
             print(f"Table {dimension_table_path} created successfully.")
         else:
             print(f"Table {dimension_table_path} already exists. Will not overwrite it.")

         logging.shutdown()
```

```
Dataset restaurant_inspection already exists.
Table my-project-0577-434418.restaurant_inspection.date_dimension created successfully.
```

Creating Fact Table

```
[38] def load_csv_data_file(logging, file_path, df):
         """
         Loads data from a CSV file into a Pandas DataFrame.

         Args:
             logging: The logging module for logging messages.
             file_path (str): The path to the CSV file.
             df (pd.DataFrame, optional): An existing DataFrame to append to. Defaults to None.

         Returns:
             pd.DataFrame: The DataFrame containing the loaded data.
         """
         try:
             # Read the CSV file into a DataFrame
             logging.info(f"Reading data from CSV file: {file_path}")  # Log the file path
             df = pd.read_csv(file_path)
             logging.info(f"Successfully read data from CSV file: {file_path}")  # Log success
             return df
         except FileNotFoundError:
             logging.error(f"Error: CSV file not found at path: {file_path}")
             raise  # Re-raise the exception to halt execution
         except pd.errors.EmptyDataError:
             logging.error(f"Error: CSV file is empty: {file_path}")
             raise  # Re-raise the exception to halt execution
         except Exception as e:
             logging.error(f"Error reading CSV file: {file_path}. Error: {e}")
             raise  # Re-raise the exception to halt execution
```

```python
def insert_existing_table(logging, bqclient, fact_table_path, df):
    try:

        df['inspection_id'] = df.index  # Generating inspection_id dynamically

        # Specify schema explicitly
        job_config = bigquery.LoadJobConfig(
            schema=[
                bigquery.SchemaField("critical_flag_dim_id", "INTEGER"),
                bigquery.SchemaField("location_dim_id", "INTEGER"),
                bigquery.SchemaField("grade_dim_id", "INTEGER"),
                bigquery.SchemaField("inspection_type_dim_id", "INTEGER"),
                bigquery.SchemaField("inspection_date_dim_id", "INTEGER"),
                bigquery.SchemaField("restaurant_type_dim_id", "INTEGER"),
                bigquery.SchemaField("violation_description_dim_id", "INTEGER"),
                bigquery.SchemaField("inspection_id", "INTEGER"),
                bigquery.SchemaField("inspection_count", "INTEGER")
            ]
        )
        bqclient.load_table_from_dataframe(df, fact_table_path, job_config=job_config)
        logging.info(f"Data successfully inserted into {fact_table_path}.")
    except Exception as e:
        logging.error(f"Error inserting data into {fact_table_path}: {str(e)}")
```

```python
[42] def query_bigquery_table(logging, table_path, bqclient, surrogate_key):
        """
        Queries a BigQuery table and returns the result as a Pandas DataFrame.

        Args:
            logging: The logging module for logging messages.
            table_path (str): The full path to the BigQuery table.
            bqclient: The BigQuery client object.
            surrogate_key (str): The name of the surrogate key column.

        Returns:
            pd.DataFrame: The DataFrame containing the query results.
        """
        try:
            logging.info(f"Querying BigQuery table: {table_path}")
            query = f"SELECT *, {surrogate_key} FROM `{table_path}`"  # Include surrogate key in query
            bq_df = bqclient.query(query).to_dataframe()  # Execute query and convert to DataFrame
            logging.info(f"Successfully queried BigQuery table: {table_path}")
            return bq_df
        except Exception as e:
            logging.error(f"Error querying BigQuery table: {table_path}. Error: {e}")
            raise  # Re-raise the exception to halt execution
```

```
[40] def dimension_lookup(logging, dimension_name, lookup_columns, df):
        '''
        dimension_lookup
        Lookup the lookup_columns in the dimension_name and return the associated surrogate keys
        Returns dataframe augmented with the surrogate keys
        '''
        bq_df = pd.DataFrame
        logging.info('Lookup dimension %s.', dimension_name)
        surrogate_key = dimension_name + '_dim_id'
        dimension_table_path = '.'.join([gcp_project, bq_dataset, dimension_name + '_dimension'])
        bq_df = query_bigquery_table(logging, dimension_table_path, bqclient, surrogate_key)
        m = bq_df.melt(id_vars=lookup_columns, value_vars=surrogate_key)
        m = m.rename(columns={'value':surrogate_key})
        df = df.merge(m, on=lookup_columns, how='left')
        df = df.drop(columns=lookup_columns)
        df = df.drop(columns='variable')
        return df
```

```
[48] def date_dimension_lookup(logging, dimension_name, lookup_column, df):
         '''
         dimension_lookup
         Lookup the lookup_columns in the dimension_name and return the associated surrogate keys
         Returns dataframe augmented with the surrogate keys
         '''
         bq_df = pd.DataFrame
         logging.info('Lookup date dimension on column %s.', lookup_column)
         surrogate_key = dimension_name + '_dim_id'
         dimension_table_path = '.'.join([gcp_project, bq_dataset, dimension_name + '_dimension'])
         bq_df = query_bigquery_table(logging, dimension_table_path, bqclient, surrogate_key)
         bq_df['full_date'] = pd.to_datetime(bq_df['full_date'])
         bq_df['full_date'] = bq_df.full_date.dt.date

         df[lookup_column] = pd.to_datetime(df[lookup_column])
         df[lookup_column] = df[lookup_column].dt.date

         m = bq_df.melt(id_vars='full_date', value_vars=surrogate_key)
         # Ensure the renamed column name matches the surrogate key
         m = m.rename(columns={'value': surrogate_key})
         df = df.merge(m, left_on=lookup_column, right_on='full_date', how='left')

         df = df.drop(columns=lookup_column)
         df = df.drop(columns='variable')
         df = df.drop(columns='full_date')
         return df
```

It generates inspection_id by setting an index on the DataFrame to assign a unique identifier. The schema is defined for a table that is being loaded, which matches the structure of the data frame.

```
[53] for year in range(2021, 2025):
        logging.info('========================================================================')
        if __name__ == '__main__':
            df = pd.DataFrame

            bqclient = create_bigquery_client(logging)

            df = load_csv_data_file(logging, 'Restaurant_Inspection.csv', df)

            df = dimension_lookup(logging, dimension_name='critical_flag', lookup_columns=['critical_flag'], df=df)
            df = dimension_lookup(logging, dimension_name='location', lookup_columns=['street', 'boro', 'zipcode'], df=df)
            df = dimension_lookup(logging, dimension_name='grade', lookup_columns=['grade'], df=df)
            df = dimension_lookup(logging, dimension_name='inspection_type', lookup_columns=['inspection_type'], df=df)
            df = dimension_lookup(logging, dimension_name='restaurant_type', lookup_columns=['cuisine_description'], df=df)
            df = dimension_lookup(logging, dimension_name='violation_description', lookup_columns=['violation_code', 'violation_description'], df=df)


            df = date_dimension_lookup(logging, dimension_name='date', lookup_column='inspection_date', df=df)

            surrogate_keys=['critical_flag_dim_id','location_dim_id','grade_dim_id','inspection_type_dim_id','date_dim_id','restaurant_type_dim_id','violation_description_dim_id']
            df = df[surrogate_keys]

            #for daily transcational snapshot grain)
            df['inspection_count'] = 1
            df = df.groupby(surrogate_keys)['inspection_count'].agg('count').reset_index()

            # See if the target table exists
            target_table_exists = bigquery_table_exists(bqclient, fact_table_path)
            # If the target table does not exist, load all of the data into a new table
            if not target_table_exists:
                build_new_table(logging, bqclient, fact_table_path, df)
            # If the target table exists, then perform an incremental load
            if target_table_exists:
                insert_existing_table(logging, bqclient, fact_table_path, df)
            logging.shutdown()
```

`df['inspection_id'] = df.index:` creates a new column to assign values

```
        df = df.group by (surrogate_keys + ['inspection_id'])['inspection_count'].agg('count').reset_index()

df = df[surrogate_keys + ['inspection_id']]
```

It assigns a constant value of 1 to insepction_id and then groups the DataFrame by surrogate keys and insepction_id to count the occurrences of each group. The record represents a single transaction of inspections. This captures a granular snapshot of surrogate keys with dimensions.

Final Dimension Schema:

Restaurant Inspection:

The following screenshots display the final dimensional schema in Google BigQuery, illustrating each dimension after the ETL process has been successfully completed.

1. Inspection Type Dimension

Row	inspection_type	inspection_category_code	inspection_type_dim_id
1	Inter-Agency Task Force / Initial Inspection	1	2
2	Pre-permit (Operational) / Initial Inspection	1	4
3	Trans Fat / Initial Inspection	1	7
4	Administrative Miscellaneous / Initial Inspection	1	9
5	Smoke-Free Air Act / Initial Inspection	1	12

2. Location Dimension

Row	street	boro	city	zipcode	latitude	longitude	location_dim_id
1	BATTERY PLACE	Manhattan	New York	10280.0	40.7057987...	-74.018001...	12089
2	LIBERTY STREET	Manhattan	New York	10280.0	40.7115301...	-74.015672...	1585
3	SOUTH END AVENUE	Manhattan	New York	10280.0	40.7105282...	-74.016209...	6975
4	MURRAY ST	Manhattan	New York	10282.0	40.7151231...	-74.014797...	10508
5	WEST STREET	Manhattan	New York	10282.0	40.7138387...	-74.013812...	12330
6	VESEY STREET	Manhattan	New York	10282.0	40.7145439...	-74.015684...	7593
7	MURRAY STREET	Manhattan	New York	10282.0	40.7155210...	-74.015388...	14015
8	RIVER TERRACE	Manhattan	New York	10282.0	40.7155071...	-74.016662...	5180

3. Grade Dimension

Row	grade	grade_description	score_range	grade_dim_id
1	C	Fair	28+	1
2	A	Excellent	0-13	2
3	B	Good	14-27	3
4	Unknown	Unknown	Unknown range	4

4. Violation Code Dimensions

Row	violation_code	violation_description	violation_description_dim_id
1	02A	Time/Temperature Control for Safety (TCS) food not cooked to required minimum internal temperature. • Poultry, poultry parts, ground and comminuted poultry, all stuffing containing poultry, meats, fish or ratites to or above 165 °F for 15 seconds with no interruption of the cooking process • Gro...	6
2	02A	Time/Temperature Control for Safety (TCS) food not cooked to required minimum internal temperature. • Poultry, poultry parts, ground and comminuted poultry, all stuffing containing poultry, meats, fish or ratites to or above 165 °F for 15 seconds with no interruption of the cooking process • Ground...	8
3	02B	Hot TCS food item not held at or above 140 °F.	
4	02C	Hot TCS food item that has been cooked and cooled is being held for service without first being reheated to 165° F or above within 2 hours.	2
5	02C	Hot TCS food item that has been cooked and cooled is being held for service without first being reheated to 165° F or above within 2 hours.	8
6	02D	Commercially processed pre-cooked TCS in hermetically sealed containers and precooked TCS in intact packages from non-retail food processing establishments not heated to 140 °F within 2 hours of removal from container or package.	8
7	02F	Meat, fish, molluscan shellfish, unpasteurized raw shell eggs, poultry or other TCS offered or served raw or undercooked and written notice not provided to consumer.	8

Load more

5. Critical Flag Dimension

Row	critical_flag	critical_flag_dim_id
1	Critical	1
2	Not Critical	2
3	Not Applicable	3

6. Restaurant Type Dimension

Row	cuisine_description	restaurant_type
1	Chinese	1
2	Vegan	2
3	Chicken	3
4	Italian	4
5	Caribbean	5
6	Bakery Products/Desserts	6
7	Mediterranean	7
8	Sandwiches	8
9	American	9
10	Fruits/Vegetables	10
11	Other	11
12	Jewish/Kosher	12

7. Date Dimension

Row	full_date	weekday_name	month_name	day_of_month	month_of_year	quarter	year	date_dim_id
1	2021-01-01T00:00:00	Friday	January	01	01	1	2021	1
2	2021-01-08T00:00:00	Friday	January	08	01	1	2021	8
3	2021-01-15T00:00:00	Friday	January	15	01	1	2021	15
4	2021-01-22T00:00:00	Friday	January	22	01	1	2021	22
5	2021-01-29T00:00:00	Friday	January	29	01	1	2021	29
6	2022-01-07T00:00:00	Friday	January	07	01	1	2022	372
7	2022-01-14T00:00:00	Friday	January	14	01	1	2022	379
8	2022-01-21T00:00:00	Friday	January	21	01	1	2022	386
9	2022-01-28T00:00:00	Friday	January	28	01	1	2022	393
10	2023-01-06T00:00:00	Friday	January	06	01	1	2023	736
11	2023-01-13T00:00:00	Friday	January	13	01	1	2023	743
12	2023-01-20T00:00:00	Friday	January	20	01	1	2023	750

Restaurant Inspection Fact Table:

Row	critical_flag_dim	location_dim_id	grade_dim_id	inspection_type_dim_id	date_dim_id	restaurant_type_dim_id	violation_description_dim_id	inspection_count
1	1	112	2	4	1280	1	11	1
2	1	126	2	1	1280	1	5	1
3	1	126	2	1	1280	1	18	1
4	1	167	2	4	1280	1	11	1
5	1	390	2	1	1280	1	5	1
6	1	390	2	1	1280	1	18	1
7	1	452	2	1	1280	1	5	1
8	1	452	2	1	1280	1	18	1
9	1	465	2	1	1280	1	5	1
10	1	465	2	1	1280	1	18	1
11	1	550	2	1	1280	1	5	1
12	1	550	2	1	1280	1	18	1
13	1	579	2	1	1280	1	5	1
14	1	579	2	1	1280	1	18	1
15	1	831	2	1	1280	1	5	1
16	1	831	2	1	1280	1	18	1
17	1	914	2	1	1280	1	5	1
18	1	914	2	1	1280	1	18	1
19	1	991	2	1	1280	1	5	1
20	1	991	2	1	1280	1	18	1

Results per page: 50 1 – 50 of 2309440

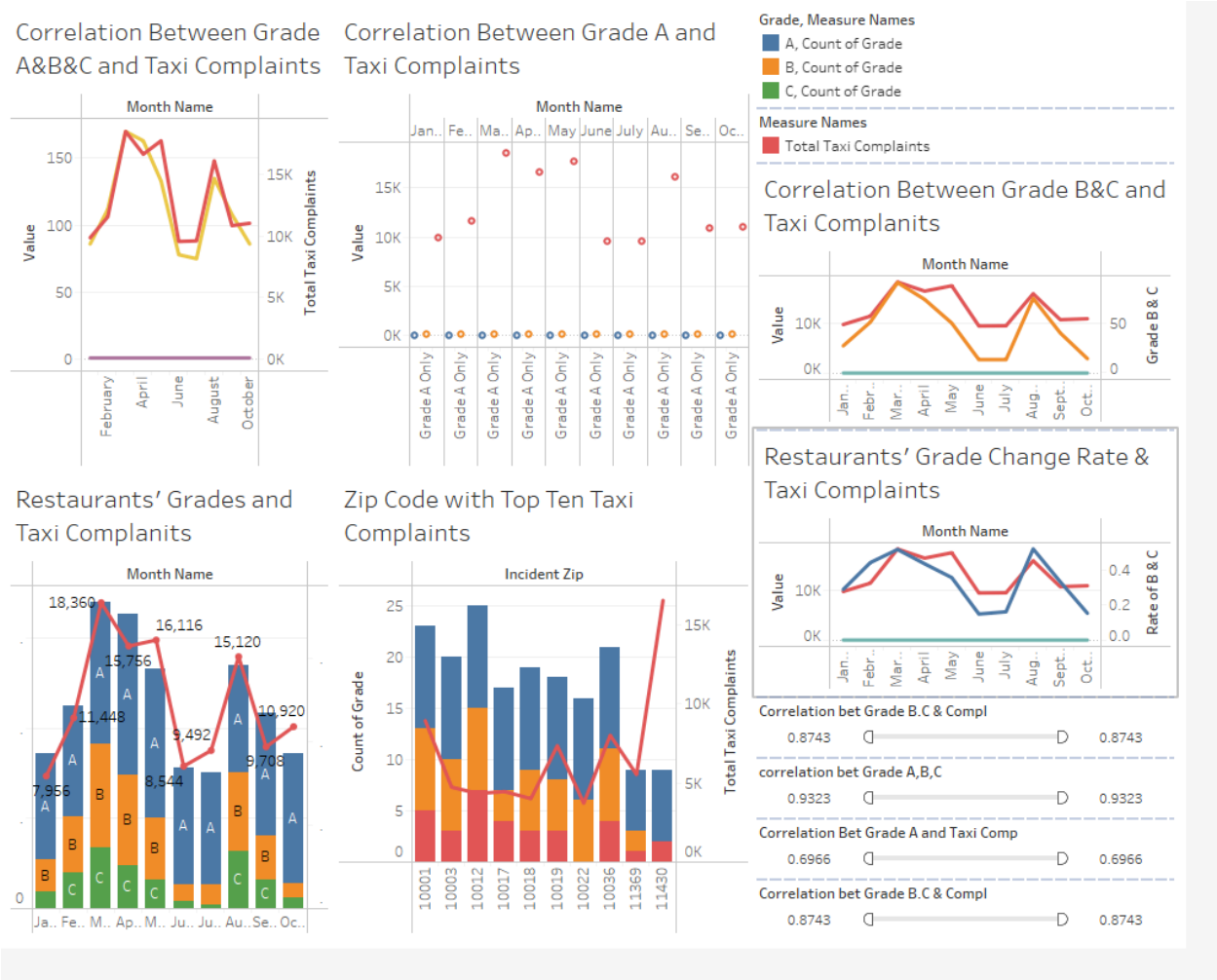
3. Loading

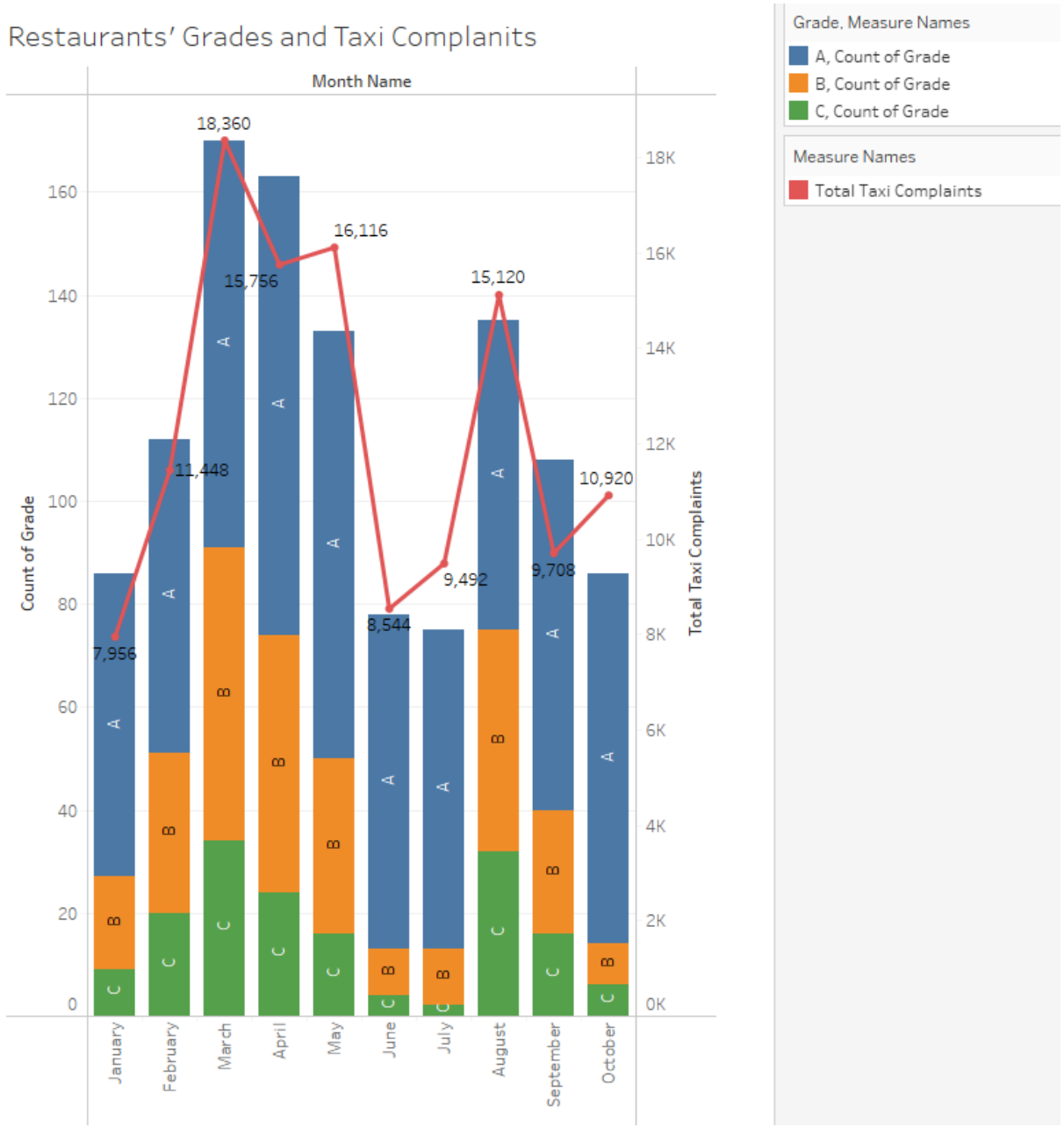
- **Tools Used:** Python and BigQuery on Google Cloud Platform
- The transformed data was loaded into our cloud-based data warehouse. We defined the table structures and loaded the transformed data into our dimensional schema and BigQuery served as the final storage and querying tool for our data warehouse.
- Tables were organized into fact and dimension tables based on the star schema design.
    - **Fact Table:** Stored the aggregated metrics and KPIs.
    - **Dimension Tables:** Contained descriptive information such as restaurant types, inspection grades, and geographic locations.

Data visualization

After identifying dimensions and fact tables, we integrated both datasets to generate visualizations. Tableau was selected as the primary visualization tool due to its robust interactive features, user-friendly interface, and ability to connect with Google BigQuery for real-time data analysis.

The screenshot below showcases a data visualization dashboard. Below is a detailed explanation of each graph included in the dashboard.

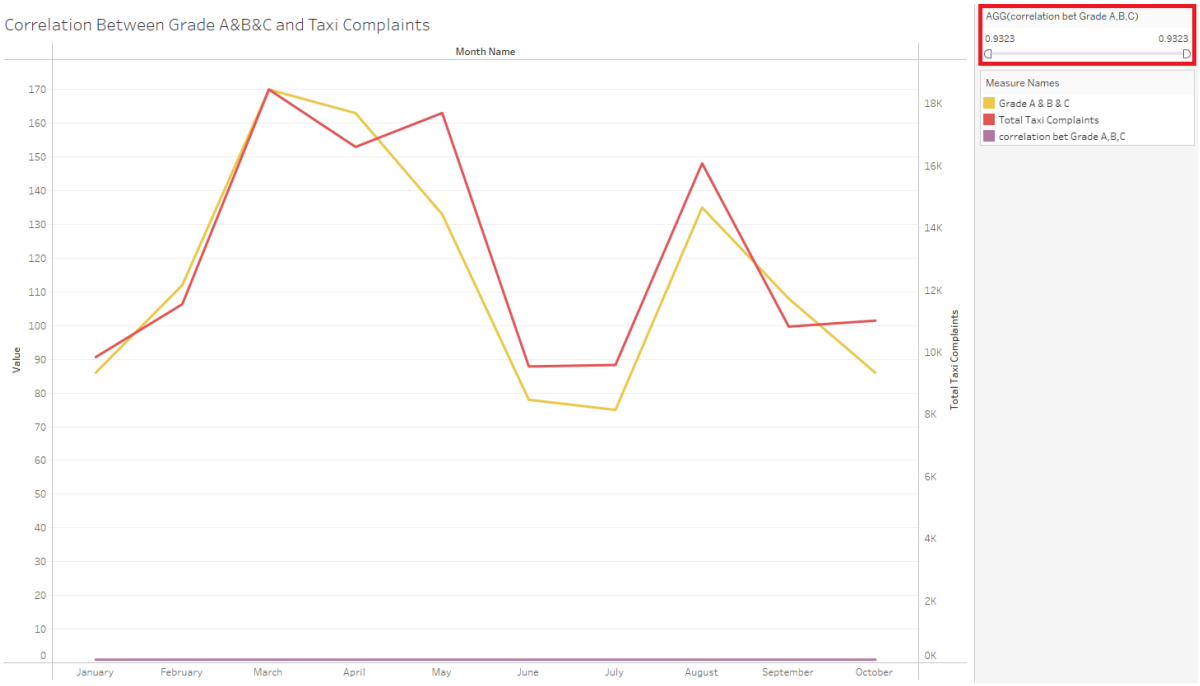




For this analysis, we selected the year 2024 to ensure that observations and correlations are based on recent and relevant data only. Using this approach, we can capture current patterns and relationships that reflect current realities and are more actionable.

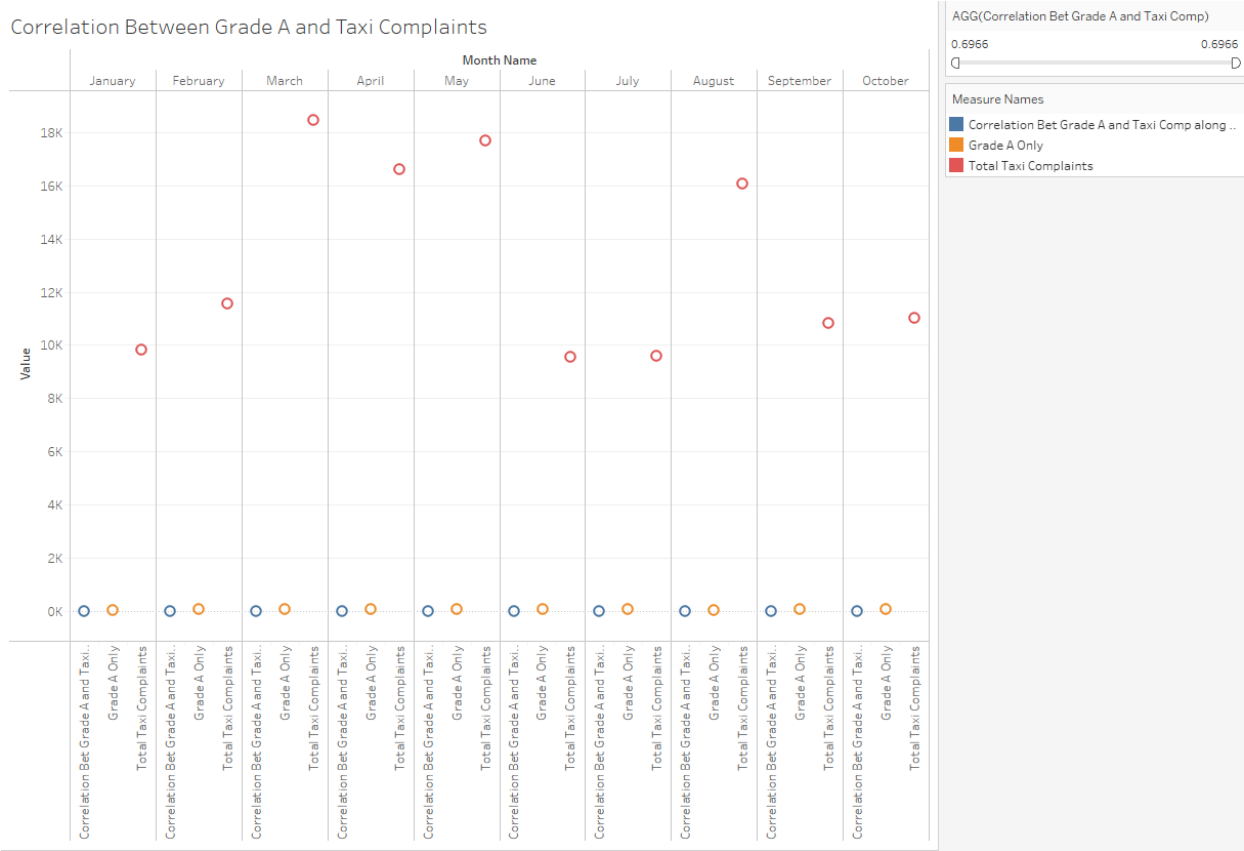
The above graph displays the count of restaurant inspections categorized by their respective grades (A, B, and C) alongside the total number of taxi complaints, organized

by month. Observing the trends, it appears that months with an increase in Grade B and C counts often correspond to a rise in taxi complaints. However, this is merely an observation and should not be interpreted as a definitive correlation without further analysis or statistical validation.

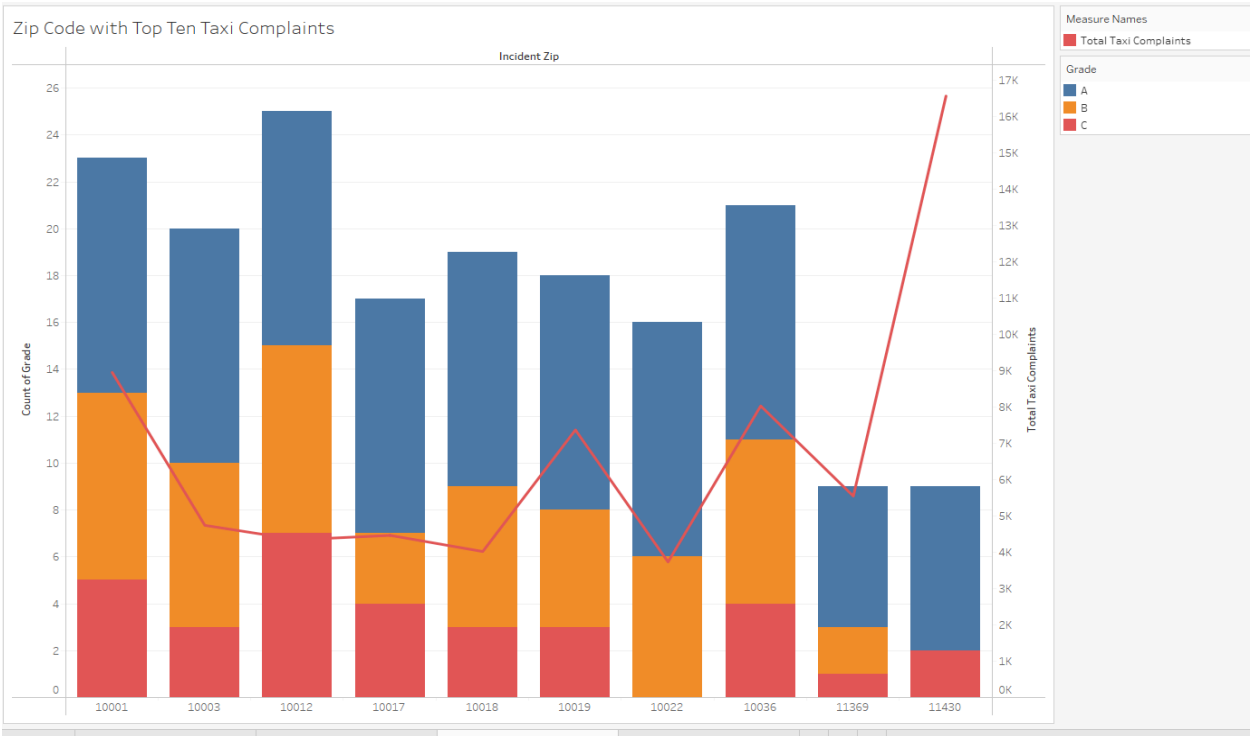


To confirm the relationship, we analyzed the correlation between restaurant grades and taxi complaints. The results show a correlation value of 0.9, indicating a strong positive relationship between neighborhoods with varying restaurant grades and the volume of taxi complaints. This supports the hypothesis that restaurant quality impacts taxi complaints.

Narrowing the focus to neighborhoods with a higher concentration of Grade B and C restaurants, the correlation value is 0.87, still a strong positive relationship. This highlights that areas with more Grade B and C restaurants are likely to experience higher taxi complaints.



When focusing exclusively on Grade A restaurants, the correlation with taxi complaints drops to 0.69, a weaker but still positive relationship. This aligns with the hypothesis that neighborhoods with better-graded restaurants tend to experience fewer taxi complaints.



We wanted to examine the neighborhoods with the highest taxi complaints. In this graph, the zip codes are displayed along the bottom, with the red line representing the number of taxi complaints. If you focus on the bars, which indicate the count of Grade B and C restaurants, you'll notice a pattern: as the count of Grade B and C restaurants increases, the taxi complaints line also rises. Similarly, when the bars decrease, the line for taxi complaints drops as well. This suggests a potential relationship between the prevalence of Grade B and C restaurants and the volume of taxi complaints.

This above graph examines the ratio of lower grades (B and C) to all restaurant grades over time. This ratio represents the proportion of lower-graded restaurants (B and C) compared to the total number of restaurants, including Grade A, providing a relative measure of the concentration of lower-grade restaurants in each period. The blue line represents this ratio, while the red line shows taxi complaints. Both lines follow similar trends, with a correlation value of 0.87, reinforcing the hypothesis that neighborhoods with a higher proportion of lower-grade restaurants tend to have more taxi complaints.

Descriptions of Tools Used

Our project leveraged various tools and technologies to design, implement, and analyze the data warehouse. Below is an overview of the tools used across different milestones of the project:

Databases

Google BigQuery: Used as the primary data warehouse to efficiently store, manage, and query large datasets. It enabled seamless integration of taxi complaints and restaurant inspection data.
Lucidchart: Designed dimensional models, star schemas, and integrated schema diagrams.

ETL (Extract, Transform, Load)

Python: Utilized for data profiling, transformation, and cleaning tasks. Python libraries like Pandas were used to handle missing values, format data, and apply business logic.

Google Colab: Used for running ETL scripts and generating surrogate keys for all the dimensional tables.

Visualization Tools

Tableau: Developed KPI dashboards and visualizations to analyze trends and relationships between taxi complaints and restaurant inspections.

Conclusion

a) Software and Database Tools Used

- Lucidchart: The tool was used to visualize and design the dimensional models and schema diagrams.
- Google BigQuery: It was the primary database for storing and querying large datasets efficiently.
- Python: Mainly used for ETL processes, data profiling, and transformations with libraries like Pandas.

- Tableau: Created interactive dashboards for visualizing main KPIs and trends.
- Google Colab: Used to execute Python scripts and manage ETL processes collaboratively.

b) Team's Experience with the Project

Challenges:

- The major challenges we faced were when doing the ETL process, particularly understanding DBT's setup and capabilities. Switching to Python simplified the task but required significant learning to optimize transformations.
- Integrating two distinct datasets (taxi complaints and restaurant inspections) posed difficulties in aligning dimensions and handling missing data.

Easiest Step:

Building dimensional models in Lucidchart was relatively seamless once the schema design was finalized.

Key Learnings:

- We gained insights into advanced data warehouse concepts, including star schemas and surrogate key creation.
- Mastered each step of ETL techniques and handled large datasets efficiently.
- Realized the importance of clear communication amongst groups and collaboration tools for managing group tasks.

What We Would Do Differently:

- Allocate more time for ETL tool exploration as it was the most challenging part to avoid mid-project changes.
- Establish stricter deadlines for all milestones to ensure better time management.
- Conduct a pilot test with smaller datasets to identify integration issues earlier.

c) Proposed Benefits Realization

This project provides actionable insights by correlating taxi complaints and restaurant inspections, enabling targeted interventions to improve public health and service quality. It supports efficient resource allocation for regulatory agencies, educates the public, and lays a foundation for broader application and future research.

d) Final Comments and Conclusions

This project has been a valuable learning experience for us. It has equipped the whole team with collaborative strategies and technical skills. The results also demonstrate the potential for integrated datasets to address real-world issues comprehensively. While challenges were encountered, the project's outcomes validate the effort and showcase the significance of data warehousing in decision-making processes.

References

1. **NYC Open Data**
   Dataset: Taxi Complaints
   Source:
   https://data.cityofnewyork.us/Social-Services/311-Service-Requests-from-2010-to-Present/erm2-nwe9/about_data

2. **NYC Open Data**
   Dataset: Restaurant Inspections
   Source:
   https://data.cityofnewyork.us/Health/DOHMH-New-York-City-Restaurant-Inspection-Results/43nn-pn8j/about_data

3. **Professor's Code**

   Code provided by Professor Ramah Al Balaw for extracting and transforming inspection data, which was used to preprocess the Restaurant Inspections dataset. The code was not available on a public website.

4. **GitHub Repository: NYC 311 Complaints Data Warehouse**

   The repository provided by GitHub User: jli82 was used as a guide to create the fact table for restaurant inspections.
   https://github.com/jli82/nyc-311-complaints-data-warehouse/blob/main/Final%20Project%20Report.pdf

   Date Accessed: 11/15/2024