Final ETL project

William Alejandro Botero Florez

Juan Manuel Lopez Rodriguez

Javier Alejandro Vergara Zorrilla

ETL (Extract, Transform, Load)

Autonomous University of the West

Faculty of Engineering

Santiago de Cali

2024

Documentation and evidence of the Traffic Accident Analysis Project

1. Introduction

This project focuses on the implementation of a traffic accident analysis system using multiple technologies and data analysis tools, with the aim of offering a comprehensive solution for data extraction, transformation, analysis and visualization in real time. The project is divided into six main phases:

- **Data Collection**: Gathering data from a static dataset and an external API to ensure a rich and diverse dataset.
- **Dimensional Modeling**: Designing a dimensional model to organize and store data efficiently for analysis.
- **Kafka Integration**: Implementing Kafka for real-time data streaming and seamless communication between system components.
- **Real-Time Visualization with Looker Studio**: Creating a dynamic dashboard to visualize the latest data updates in real-time.
- **Airflow Orchestration**: Developing an Airflow DAG to automate ETL processes, ensuring a streamlined workflow.
- **Conclusion**: Summarizing the outcomes and the effectiveness of the implemented solution.

2. Data Sources

In this phase, two different data sources were selected to feed the analysis system:

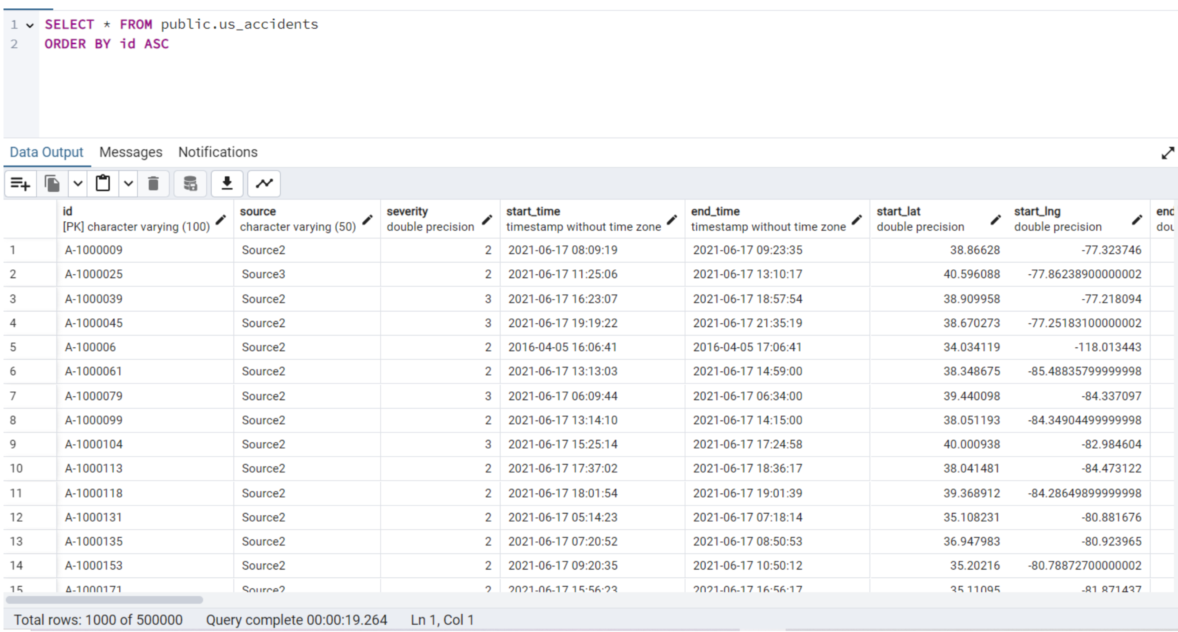
1. **US Car Accident Dataset**: This is historical data on traffic accidents in the USA from 2016 to 2023, stored in CSV format. This data includes detailed information about the accidents, such as the date, location, and number of people injured or killed.

For this data set the following process was carried out

Database

All this information was stored in a postgresql database, with a total of 500 thousand rows and 46 columns

There was information such as accident id, accident severity, accident start time and end time, accident coordinates, accident description, accident location, city, state, region, continent, weather conditions, time zone, time of day etc…



```sql
SELECT * FROM public.us_accidents
ORDER BY id ASC
```

Data Output Messages Notifications

	id [PK] character varying (100)	source character varying (50)	severity double precision	start_time timestamp without time zone	end_time timestamp without time zone	start_lat double precision	start_lng double precision	enc dou
1	A-1000009	Source2	2	2021-06-17 08:09:19	2021-06-17 09:23:35	38.86628	-77.323746	
2	A-1000025	Source3	2	2021-06-17 11:25:06	2021-06-17 13:10:17	40.596088	-77.86238900000002	
3	A-1000039	Source2	3	2021-06-17 16:23:07	2021-06-17 18:57:54	38.909958	-77.218094	
4	A-1000045	Source2	3	2021-06-17 19:19:22	2021-06-17 21:35:19	38.670273	-77.25183100000002	
5	A-100006	Source2	2	2016-04-05 16:06:41	2016-04-05 17:06:41	34.034119	-118.013443	
6	A-1000061	Source2	2	2021-06-17 13:13:03	2021-06-17 14:59:00	38.348675	-85.48835799999998	
7	A-1000079	Source2	3	2021-06-17 06:09:44	2021-06-17 06:34:00	39.440098	-84.337097	
8	A-1000099	Source2	2	2021-06-17 13:14:10	2021-06-17 14:15:00	38.051193	-84.34904499999998	
9	A-1000104	Source2	3	2021-06-17 15:25:14	2021-06-17 17:24:58	40.000938	-82.984604	
10	A-1000113	Source2	2	2021-06-17 17:37:02	2021-06-17 18:36:17	38.041481	-84.473122	
11	A-1000118	Source2	2	2021-06-17 18:01:54	2021-06-17 19:01:39	39.368912	-84.28649899999998	
12	A-1000131	Source2	2	2021-06-17 05:14:23	2021-06-17 07:18:14	35.108231	-80.881676	
13	A-1000135	Source2	2	2021-06-17 07:20:52	2021-06-17 08:50:53	36.947983	-80.923965	
14	A-1000153	Source2	2	2021-06-17 09:20:35	2021-06-17 10:50:12	35.20216	-80.78872700000002	
15	A-1000171	Source2	2	2021-06-17 15:56:23	2021-06-17 16:56:17	35.11095	-81.871437	

Total rows: 1000 of 500000 Query complete 00:00:19.264 Ln 1, Col 1

EDA

Database connection:

In order to use this information and bring it to our domain we had to make a connection to this postgresql database, for this we use the python sqlalchemy libraries and also dotenv, dotenv creates a file for us to have the credentials that we are going to use, for example in this case our .env file would look like this:

PostgreSQL database connection settings

DB_HOST=localhost

DB_PORT=5432

DB_USER=your_postgresql_user

DB_PASS=your_postgresql_password

DB_NAME=ETL_Project1

Once we have our .env file configured we can finish with the connection to our database by establishing a connection

src > db_conexion.py > ...

willyb, last month | 2 authors (willyb and one other)

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from dotenv import load_dotenv        willyb, last month • Añadir DAG de ETL para datos de accidente
import os

# Cargar las variables de entorno desde el archivo .env
load_dotenv()

# Obtener las credenciales desde las variables de entorno
db = os.getenv('DB_NAME')
usuario = os.getenv('DB_USER')
token = os.getenv('DB_PASS')
host1 = os.getenv('DB_HOST')
puerto = os.getenv('DB_PORT')

def establecer_conexion():
    # Crear la cadena de conexión para SQLAlchemy
    connection_string = f'postgresql+psycopg2://{usuario}:{token}@{host1}:{puerto}/{db}'
    engine = create_engine(connection_string)

    # Crear una sesión (opcional, si necesitas trabajar con ORM más adelante)
    Session = sessionmaker(bind=engine)
    session = Session()
    print("Conexión exitosa a la base de datos")
    return engine, session  # Devolver tanto el engine como la session si es necesario

def cerrar_conexion(session):
    session.close()
    print("Conexión cerrada a la base de datos")

# Establecer la conexión
engine, session = establecer_conexion()

# Ahora puedes usar `engine` con pandas sin problemas

```

Extracting the database:

Here we establish the connection again by importing the db_connection and we make a sql query to extract the data and we load the dataset to then save this data in a csv file

```python
import pandas as pd
from db_conexion import establecer_conexion, cerrar_conexion

def extract_data():
    # Establece la conexión usando SQLAlchemy
    engine, session = establecer_conexion()  # Ahora engine es el primero que se recibe

    # Consulta SQL para extraer datos
    query = "SELECT * FROM us_accidents"

    # Carga los datos en un DataFrame de pandas usando el engine de SQLAlchemy
    df = pd.read_sql(query, con=engine)

    # Cierra la sesión y la conexión
    cerrar_conexion(session)
    print("Datos cargados con éxito")

    # Guarda los datos en un archivo CSV para la transformación
    df.to_csv('data/us_accidents_raw.csv', index=False)
    print("Datos guardados en 'data/us_accidents_raw.csv'")

# Puedes llamar a la función `extract_data()` si deseas ejecutar el script directamente
if __name__ == "__main__":
    extract_data()
```

Data cleaning:

In this data cleaning we do the following:

- Connect to the database
- Delete the columns that we will not use later:
- 'id', 'source', 'country', 'description', 'end_lat', 'end_lng', 'civil_twilight', 'nautical_twilight', 'astronomical_twilight'
- Impute some missing values
- Perform other cleaning processes
- Check to see if there are no nulls after

src > data_cleaning.py > clean_data

willyb, last month | 3 authors (willyb and others)

```python
import pandas as pd
from db_conexion import establecer_conexion, cerrar_conexion

def clean_data():
    # Establece la conexión usando SQLAlchemy
    engine, session = establecer_conexion()

    # SQL query para seleccionar todos los datos de la tabla 'us_accidents'
    query = "SELECT * FROM us_accidents"

    # Lee los datos en un DataFrame de pandas usando el engine de SQLAlchemy
    df = pd.read_sql(query, con=engine)

    # Configura pandas para mostrar más filas y columnas
    pd.set_option('display.max_rows', 100)  # Mostrar hasta 100 filas
    pd.set_option('display.max_columns', None)  # Mostrar todas las columnas sin truncar
    pd.set_option('display.width', None)  # Ajustar automáticamente el ancho de visualización

    # Mostrar las primeras 20 filas en formato tabular
    print(df.head(20))

    # Columnas a eliminar
    columns_to_drop = ['id', 'source', 'country', 'description', 'end_lat', 'end_lng',
                       'civil_twilight', 'nautical_twilight', 'astronomical_twilight']

    # Eliminar las columnas especificadas
    df_cleaned = df.drop(columns=columns_to_drop)

    # Imputar valores faltantes en columnas numéricas con la media
    df_cleaned['temperature_f'].fillna(df_cleaned['temperature_f'].mean(), inplace=True)

    # Imputar valores faltantes en columnas categóricas con la moda (valor más frecuente)
    df_cleaned['weather_condition'].fillna(df_cleaned['weather_condition'].mode()[0], inplace=True)

    # Imputar valores faltantes en múltiples columnas numéricas con la media
    num_cols = ['wind_chill_f', 'humidity_percent', 'pressure_in', 'visibility_mi', 'wind_speed_mph', 'precipitation_in']
    df_cleaned[num_cols] = df_cleaned[num_cols].apply(lambda col: col.fillna(col.mean()))
```

After this data cleaning we wanted to see some analysis with the graphics

Dashboard:

2. New York accidents API: To enrich the data, an external API called data.cityofnewyork.us was used, which provides additional information such as the weather conditions at the time of each accident, among other things. The API was integrated into the ETL flow to ensure that each record included the corresponding weather information.

For this API, the following process was carried out

Connection to the API

We defined the function to bring the API to our domain and we defined the parameters that the data limit it brings is 200,000 to finally save that data in a csv in the data folder that we will use later for data cleaning. This API had information such as the date, time of the accident, locations, injured people, dead people, vehicle, street, etc...

src > API_connection.py > ...

willyb, last month | 1 author (willyb)

```
import requests
import pandas as pd

def get_data_from_api():
    # URL of the dataset (API endpoint)
    url = "https://data.cityofnewyork.us/resource/h9gi-nx95.json"

    # Parameters to limit the response to 200,000 records
    params = {
        "$limit": 200000
    }

    # Send a GET request to the API
    response = requests.get(url, params=params)

    # Check if the request was successful
    if response.status_code == 200:
        data = response.json()  # Convert the response to JSON format
        df = pd.DataFrame(data)  # Create a pandas DataFrame from the data
        print(df.head())  # Display the first few records
        # Save the DataFrame to a CSV file
        df.to_csv('data/API_data.csv', index=False, encoding='utf-8')
        print("Data downloaded and saved to data/API_data.csv")
    else:
        # If the request fails, print the error code
        print(f"Error in the request: {response.status_code}")
```

Cleaning the data from the New York API:

In this cleaning, the following transformations were performed:

- The API CSV was loaded
- Crash date and crash time were converted to datetime format
- Duplicate rows were removed
- Unnecessary columns were deleted
- A new column called city was added, in which the only data would be “new york”
- At the end, the clean CSV will be saved in the data folder

src > API_cleaning.py > clean_api_data

```python
def clean_api_data():
    # Get the absolute path of the file
    file_path = os.path.abspath(os.path.join('data/API_data.csv'))

    # Load the CSV file using pandas
    data = pd.read_csv(file_path)

    # Set pandas to display all columns
    pd.set_option('display.max_columns', None)

    # Convert `crash_date` and `crash_time` to datetime format
    data['crash_date'] = pd.to_datetime(data['crash_date'], errors='coerce')
    data['crash_time'] = pd.to_datetime(data['crash_time'], format='%H:%M', errors='coerce')

    # Fix inconsistent values (e.g., remove whitespace or correct capitalization in the `borough` column)
    data['borough'] = data['borough'].str.strip().str.title()

    # Filter data to keep only rows with valid crash dates and from the year 2021 or later
    data = data[data['crash_date'].notna() & (data['crash_date'].dt.year >= 2021)]

    # Remove duplicates based on the `collision_id` column (assuming it's unique for each accident)
    data = data.drop_duplicates(subset='collision_id')

    # Convert `crash_date` to just the date (drop the time part)
    data['crash_date'] = data['crash_date'].dt.date

    # Drop unnecessary columns
    data = data.drop(['vehicle_type_code_5', 'contributing_factor_vehicle_5',
                      'vehicle_type_code_4', 'contributing_factor_vehicle_4',
                      'vehicle_type_code_3', 'contributing_factor_vehicle_3',
                      'cross_street_name'], axis=1)

    # Drop rows with any missing values
    data = data.dropna()

    # Add a `city` column with the value "New York"
    data['city'] = "New York"
```

Merge:

A merge was performed between the Accidents_Usa_clean and API_data_cleaned csv files, the following transformations were made:

• **Data loading**: Two CSV files were loaded

• **Date conversion**: The date columns in both datasets were converted to datetime format, keeping only the date part.

• **Filtering by city**: Both datasets were filtered to include only data related to New York City.

• **Dataset merge**: An inner join was performed using the date columns (crash_date and start_time), thus combining climate data with accident information.

• **Duplicate column removal**: The duplicate city columns (city_x and city_y) generated after the merge were removed.

• **Date format and new column**: The date column was formatted to include only year and month, and a 'city' column was added with the fixed value "New York".

• **Column reorganization**: Moved the 'city' column to the beginning of the DataFrame.

• **Crash time adjustment**: Converted the crash_time column to display only the time.

• **Integer conversion**: Converted the number_of_persons_injured column to integer type, handling null values.

• **Unnecessary columns removal**: Removed columns not relevant to the analysis, such as coordinates, distances, secondary identifiers, and redundant data.

• **Field merging**: Joined the borough and zip_code columns, and merged city with state.

• **Unique ID assignment**: Assigned a unique identifier to each row and moved it to the beginning of the DataFrame.

• **Final validation**: Checked the number of rows after the merge and checked for null values.

• **Saved to CSV**: The final, cleaned DataFrame was saved to a file named merged_data.csv.

```
src > merge_data.py > merge_data
willyb, 8 hours ago | 2 authors (willyb and one other)
import pandas as pd

def merge_data():
    # Load the CSV files
    API_merge = pd.read_csv('data/API_data_Cleaned.csv')
    db_merge = pd.read_csv('data/us_accidents_cleaned.csv')

    # Convert date columns to datetime format and use only the date
    API_merge['crash_date'] = pd.to_datetime(API_merge['crash_date']).dt.date  # Use only the date
    db_merge['start_time'] = pd.to_datetime(db_merge['start_time']).dt.date  # Use only the date

    # Filter both datasets for rows where the city is 'New York'
    api_data_ny = API_merge[API_merge['city'] == 'New York']
    us_accidents_ny = db_merge[db_merge['city'] == 'New York']
        willyb, 8 hours ago • airflow listo con el modelo dimensional
    # Merge the two datasets based on the date (inner join)
    merged_df = pd.merge(api_data_ny, us_accidents_ny, left_on='crash_date', right_on='start_time', how='inner')

    # Drop duplicate city columns ('city_x' and 'city_y')
    merged_df = merged_df.drop(columns=['city_x', 'city_y'])

    # 1. Asegúrate de que la columna de fecha esté en formato datetime
    merged_df['crash_date'] = pd.to_datetime(merged_df['crash_date'], errors='coerce')

    # 2. Crear una nueva columna que contenga el mes y el año
    # Aquí se formatea como "YYYY-MM"
    merged_df['crash_date'] = merged_df['crash_date'].dt.to_period('M')

    # Add a new column 'city' with the value "New York"
    merged_df['city'] = "New York"

    merged_df = merged_df.sort_values(by='crash_date', ascending=True)

    # Move the 'city' column to the beginning of the DataFrame
    cols = ['city'] + [col for col in merged_df.columns if col != 'city']
    merged_df = merged_df[cols]
```

Merge analysis:

Comparison between day and night accidents

```
Comparison between day and night accidents:
sunrise_sunset
Day        31947
Night      15277
```

Relationship between weather conditions and accident severity

```
Relationship between climate and number of injured people:
weather_condition
Fair             17746
Cloudy            6142
Light Rain        1807
Mostly Cloudy     1760
Partly Cloudy     1310
Heavy Rain         293
Rain               261
Fog                149
Light Snow         128
Haze                64
Snow                15
```

Distribution by type of vehicle involved

```
Distribution by type of vehicle involved:
vehicle_type_code1
Sedan                                  22405
Station Wagon/Sport Utility Vehicle    16225
Taxi                                    1386
Bus                                     1099
Pick-up Truck                           1003
                                       ...
TRACTOR                                    1
Commercial                                 1
MTA bus                                    1
REFG                                       1
Van Camper                                 1
```

Distribtion by time of day

```
2. Distribution by time of day:
     crash_time  count
0         00:00    690
828       14:00    457
888       15:00    440
1008      17:00    413
768       13:00    398
...         ...    ...
226       03:48      1
283       04:47      1
233       03:56      1
234       03:57      1
275       04:38      1
```

Most common contributing factors in serious accidents

```
Most common contributing factors in serious accidents:
contributing_factor_vehicle_1
Driver Inattention/Distraction                            5855
Unspecified                                               3572
Failure to Yield Right-of-Way                             2820
Traffic Control Disregarded                               1733
Following Too Closely                                     1218
Unsafe Speed                                               820
Passing or Lane Usage Improper                             757
Turning Improperly                                         623
Other Vehicular                                            490
Pedestrian/Bicyclist/Other Pedestrian Error/Confusion      431
Driver Inexperience                                        366
Unsafe Lane Changing                                       334
Alcohol Involvement                                        317
View Obstructed/Limited                                    301
```

Accidents by (borough)

```
Accidents by (borough):
borough
Brooklyn - 11207.0     1427
Brooklyn - 11236.0      922
Brooklyn - 11234.0      831
Queens - 11434.0        771
Brooklyn - 11208.0      758
                       ...
Manhattan - 10280.0       5
Manhattan - 10115.0       4
Manhattan - 10069.0       4
Queens - 11109.0          2
Manhattan - 10169.0       1
```

Correlation between type of vehicle and number of injured people

```
Correlation between type of vehicle and number of injured people:
vehicle_type_code1
Sedan                                  14633
Station Wagon/Sport Utility Vehicle    10300
Taxi                                     971
Bike                                     617
Pick-up Truck                            536
                                       ...
MINIVAN                                    0
MINI BUS                                   0
Lunch Wagon                                0
LOCOMOTIVE                                 0
van                                        0
```

Accidents by Wind Speed

```
Accidents by Wind Speed:
wind_speed_mph
0.000000     9099
3.000000     8589
5.000000     7806
6.000000     7099
7.000000     3737
7.681347     2513
8.000000     2403
9.000000     2011
10.000000    1356
13.000000     709
12.000000     685
15.000000     416
18.000000     308
16.000000     306
14.000000     187
```

Comparison between Injured Persons, Pedestrians, Cyclist and Motorist

```
Comparison between injured pedestrians, cyclists and motorcyclists:
{'Injured People': 29675, 'Injured Pedestrians': 219, 'Injured Cyclist': 4104, 'Injured Motorist': 23444}
```

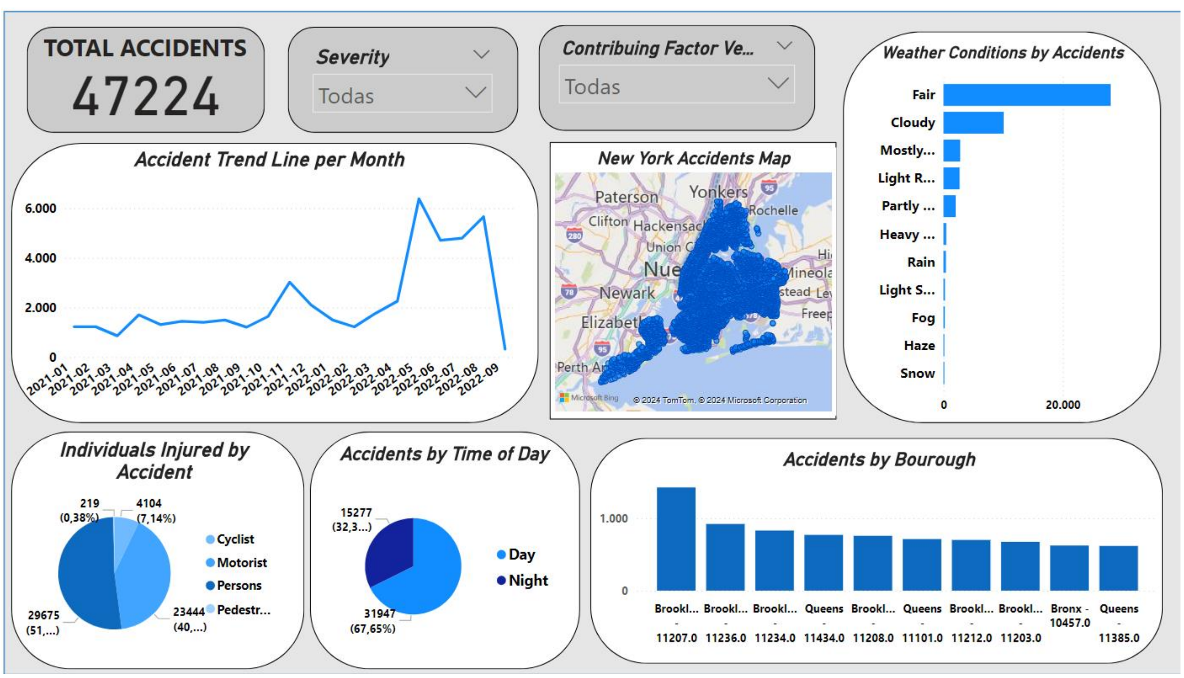
Correlation between temperature and number of accidents

```
Correlation between type of vehicle and number of injured people:
vehicle_type_code1
Sedan                                  14633
Station Wagon/Sport Utility Vehicle    10300
Taxi                                     971
Bike                                     617
Pick-up Truck                            536
                                       ...
MINIVAN                                    0
MINI BUS                                   0
Lunch Wagon                                0
LOCOMOTIVE                                 0
van                                        0
```

Most common factors in accidents without injuries

```
Most common factors in accidents without injuries:
contributing_factor_vehicle_1
Driver Inattention/Distraction                          6780
Unspecified                                             4674
Failure to Yield Right-of-Way                           1879
Passing or Lane Usage Improper                          1701
Following Too Closely                                   1582
Passing Too Closely                                     1239
Traffic Control Disregarded                             1091
Turning Improperly                                      1040
Backing Unsafely                                         911
Unsafe Speed                                             789
Other Vehicular                                          690
Alcohol Involvement                                      591
Driver Inexperience                                      545
Unsafe Lane Changing                                     530
Reaction to Uninvolved Vehicle                           323
```

Data merge dashboard:



3. Dimensional model:

The goal of this model is to organize data in a way that makes it easy to query and analyze. It is based on the star schema methodology, where a central fact table is surrounded by several dimension tables. In this case, the fact table is fact_accidents, and it is related to the dimension tables dim_time, dim_weather, dim_vehicle, and dim_location.

First we create the tables that we already mentioned with the sql that is inside the repository so that we can fill those tables with this merge.py code

dim_time table

- Purpose: Stores details related to the date and time of the accident.
- Content:
- date: Year and month of the accident (YYYY-MM format).
- day: Name of the day (e.g. "Monday").
- month: Name of the month (e.g. "January").
- year: Year of the accident.
- time: Specific time of the accident.

This table allows for temporal analysis, such as identifying patterns on specific days of the week, months, or years.

dim_climate table

- Purpose: Contains weather information relevant to the time of the accident.
- Contents:
- temperature_f: Temperature in degrees Fahrenheit.
- wind_chill_f: Wind chill.
- humidity_percent: Relative humidity in percentage.
- pressure_in: Atmospheric pressure in inches.
- visibility_mi: Visibility in miles.
- wind_direction: Wind direction.
- wind_speed_mph: Wind speed in miles per hour.
- precipitation_in: Precipitation in inches.

This table helps to understand the influence of weather on the occurrence of accidents.

dim_vehicle table

- Purpose: Describes the type of vehicle and contributing factors to the accident.
- Contents:
- vehicle_type_code1: Type of vehicle involved.
- contributing_factor_vehicle_1: Contributing factor to the vehicle-related accident (e.g. "Distracted Driving").

Allows for analysis based on vehicle types and common causes of accidents.

dim_location table

- Purpose: Contains geographic and location information for accidents.
- Content:
- city: City where the accident occurred (in this case, always "New York").
- borough: District or area of the city (e.g. "Manhattan").
- location: Coordinates or specific details of the accident location.

Facilitates the analysis of accidents by specific geographic locations.

fact_accidents table

- Purpose: This is the central table of the dimensional model that records the specific details of each accident.
- Content:
- Accident details such as date (crash_date), time (crash_time), number of people injured or killed, severity (severity), and weather conditions (weather_condition).
- Foreign keys that connect to the dimension tables:
  - dim_tiempo_id: Relates to the dim_tiempo table.
  - dim_climate_id: Relates to the dim_climate table.
  - dim_vehicle_id: Relates to the dim_vehicle table.
  - dim_location_id: Relates to the dim_location table.

Model Population Process

The code you provided performs the following steps to populate the dimensional model in PostgreSQL:

1. Connecting to the database using SQLAlchemy, with connection parameters stored in a .env file.

2. Loading the merged dataset (merged_data_cleaned.csv), which contains accident information along with climate and geographic data.

3. Converting and creating unique IDs for each dimension table (dim_weather, dim_climate, dim_vehicle, dim_location), ensuring that each row in these tables has a unique identifier.

4. Inserting data into the dimension tables using to_sql to send the data to the PostgreSQL database.

5. Assigning dimension IDs in the fact_accidents fact table by performing index-based merges to ensure foreign key integrity.

6. Filtering data to ensure consistency in the fact_accidents table, removing rows with null or inconsistent IDs.

7. Final insertion of records into the fact_accidents table.

src > load_dimensional_model.py > dimensional_model

```python
import os

def dimensional_model():


    # Cargar las variables de entorno del archivo .env
    load_dotenv()

    # Obtener las variables de entorno
    DB_HOST = os.getenv('DB_HOST')
    DB_PORT = os.getenv('DB_PORT')
    DB_USER = os.getenv('DB_USER')
    DB_PASS = os.getenv('DB_PASS')
    DB_NAME = os.getenv('DB_NAME')

    # Configurar la conexión a la base de datos
    DATABASE_URL = f'postgresql://{DB_USER}:{DB_PASS}@{DB_HOST}:{DB_PORT}/{DB_NAME}'
    engine = create_engine(DATABASE_URL)

    # Cargar el archivo CSV de accidentes
    merged_df = pd.read_csv('data/merged_data_cleaned.csv', encoding='utf-8')


    # Convertir crash_date y crash_time a datetime
    merged_df['crash_datetime'] = pd.to_datetime(merged_df['crash_date'] + ' ' + merged_df['crash_time'], errors='coerce')

    # Verificar si hay errores de conversión de fecha
    if merged_df['crash_datetime'].isnull().any():
        print("Advertencia: Algunas fechas no se pudieron convertir.")

    # Extraer información de tiempo
    merged_df['dim_tiempo_id'] = range(1, len(merged_df) + 1)
    merged_df['fecha'] = merged_df['crash_datetime'].dt.strftime('%Y-%m')  # Convertir a formato de cadena 'YYYY-MM'
    merged_df['hora'] = merged_df['crash_datetime'].dt.strftime('%H:%M')  # Formato HH:MM
    merged_df['dia'] = merged_df['crash_datetime'].dt.day_name()
    merged_df['mes'] = merged_df['crash_datetime'].dt.month_name()
    merged_df['año'] = merged_df['crash_datetime'].dt.year
```

Rationale for the Dimensional Model

This model is designed to improve the efficiency of querying and analyzing accident data:

- **Fast query**: By segmenting the data into specific dimensions, faster queries and analysis can be performed for common questions, such as "In which months do more accidents occur?" or "What type of weather is most associated with serious accidents?".
- **Flexibility**: Allows new dimensions to be added in the future without affecting the core structure of the model.
- **Clarity in analysis**: By separating data into dimension tables, redundancy is minimized and clarity in the presentation of information is improved.
- **Storage optimization**: Dimension tables store information that is repeated in multiple records in the fact table, saving space and making maintenance easier.

4. Kafka Implementation

4.1. Configuration with Docker

To manage Apache Kafka and Zookeeper, Docker Compose was used. The configuration included:

4.2. Productor de Kafka

- The Kafka Producer For the Kafka implementation, we use the metric of total injured people per month from data collected in our traffic accident database.

```
import json
import time  # Importa el módulo time para usar sleep
from kafka import KafkaProducer

def kafka_producer():
    # Cargar las variables de entorno del archivo .env
    load_dotenv()

    # Obtener las variables de entorno
    DB_HOST = os.getenv('DB_HOST')
    DB_PORT = os.getenv('DB_PORT')
    DB_USER = os.getenv('DB_USER')
    DB_PASS = os.getenv('DB_PASS')
    DB_NAME = os.getenv('DB_NAME')

    # Configurar la conexión a la base de datos
    DATABASE_URL = f'postgresql://{DB_USER}:{DB_PASS}@{DB_HOST}:{DB_PORT}/{DB_NAME}'
    engine = create_engine(DATABASE_URL)

    # Conexión a PostgreSQL para obtener el número total de personas heridas
    try:
        # Consulta SQL para obtener el total de personas heridas por día
        query = """
        SELECT
            crash_date AS date,
            SUM(number_of_persons_injured +
                number_of_pedestrians_injured +
                number_of_cyclist_injured +
                number_of_motorist_injured) AS total_injured
        FROM
            fact_accidents
        GROUP BY
            crash_date      You, 2 days ago • implementacion de kafka
        ORDER BY
            crash_date;
        """

        # Ejecutar la consulta usando pandas para obtener el resultado en un DataFrame
```

The Kafka producer takes daily traffic accident data from our PostgreSQL database and streams the total injured people metric to the consumer. Each message contains the date of the accident and the total number of injured people on that day.

```
$ python src/kafka_producer.py
Enviado a Kafka: {'date': '2021-01', 'total_injured': 1146}
Enviado a Kafka: {'date': '2021-02', 'total_injured': 1154}
Enviado a Kafka: {'date': '2021-03', 'total_injured': 954}
Enviado a Kafka: {'date': '2021-04', 'total_injured': 2122}
Enviado a Kafka: {'date': '2021-05', 'total_injured': 1658}
Enviado a Kafka: {'date': '2021-06', 'total_injured': 1794}
Enviado a Kafka: {'date': '2021-07', 'total_injured': 1656}
Enviado a Kafka: {'date': '2021-08', 'total_injured': 1968}
Enviado a Kafka: {'date': '2021-09', 'total_injured': 1633}
Enviado a Kafka: {'date': '2021-10', 'total_injured': 1866}
Enviado a Kafka: {'date': '2021-11', 'total_injured': 3611}
Enviado a Kafka: {'date': '2021-12', 'total_injured': 2495}
Enviado a Kafka: {'date': '2022-01', 'total_injured': 1645}
Enviado a Kafka: {'date': '2022-02', 'total_injured': 1270}
Enviado a Kafka: {'date': '2022-03', 'total_injured': 1945}
Enviado a Kafka: {'date': '2022-04', 'total_injured': 2907}
```

4.3. Kafka Consumer

- The Kafka Consumer The Kafka Consumer receives real-time accident data and sends it to a spreadsheet in Google Sheets. This data, organized by date, contains the metric of the total number of injured people.
- Each consumer message has a time.sleep

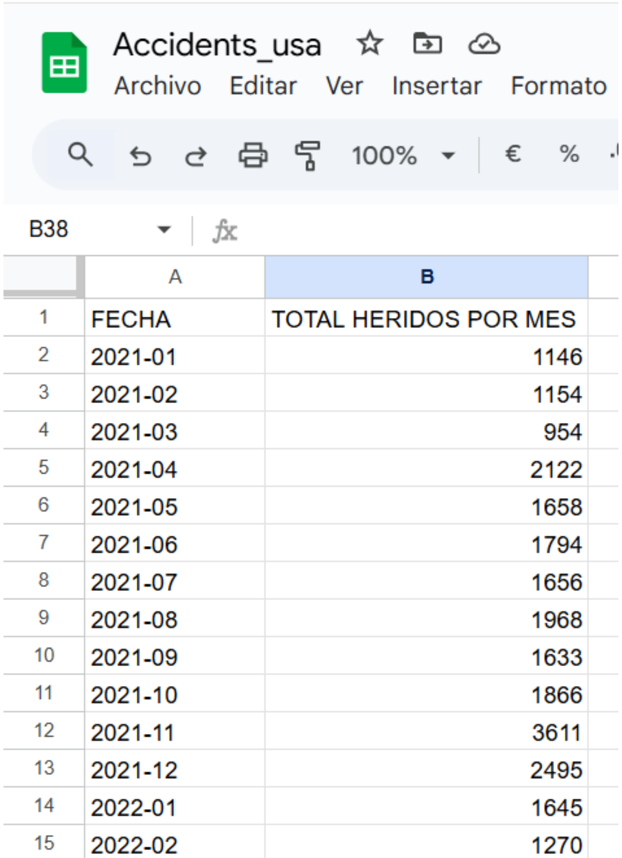
src > kafka_consumer.py > kafka_consumer_to_google_sheets

```python
from google.oauth2.service_account import Credentials
from googleapiclient.errors import HttpError

def kafka_consumer_to_google_sheets():
    # Configuración del consumidor de Kafka
    consumer = KafkaConsumer(
        'injuries_by_day',  # Nombre del topic de Kafka
        bootstrap_servers='localhost:9092',  # Servidor de Kafka
        value_deserializer=lambda x: json.loads(x.decode('utf-8'))  # Deserialización de mensajes
    )

    # Ruta al archivo de credenciales JSON de la cuenta de servicio de Google
    creds = Credentials.from_service_account_file(
        'accidents_usa.json',  # Ruta del archivo de credenciales JSON
        scopes=["https://www.googleapis.com/auth/spreadsheets"]
    )

    # ID de la hoja de cálculo de Google Sheets y el rango donde escribir los datos
    SPREADSHEET_ID = '1Dflw-mewAhm6Nqa60dxc9BgLTklrMrbnyWf485XWOFE'  # ID de la hoja de cálculo
    RANGE_NAME = 'Hoja1!A:A'  # Columna A para referencia

    # Crear el servicio de Google Sheets
    service = build('sheets', 'v4', credentials=creds)        You, 2 days ago • implementacion de kafka
    sheet = service.spreadsheets()

    # Función para obtener la última fila de la hoja de cálculo
    def get_last_row():
        try:
            result = sheet.values().get(
                spreadsheetId=SPREADSHEET_ID,
                range=RANGE_NAME
            ).execute()
            values = result.get('values', [])
            return len(values) + 1  # La próxima fila disponible
        except HttpError as err:
            print(f'Error al obtener la última fila: {err}')
            return 2  # Retorna la fila 2 en caso de error
```

381 0 Select Postgres Server You, 2 days ago Ln 27, Col 55 Spaces: 4 UTF-8 CRLF Python 3.12.4 ('.venv': venv)

```
$ python src/kafka_consumer.py
Mensaje recibido: {'date': '2021-01', 'total_injured': 1146}
Datos actualizados en Google Sheets: {'spreadsheetId': '1Dflw-mewAhm6Nqa60dxc9BgLTklrMrbnyWf485XWOFE', 'updatedRange': 'Hoja1!A2:B2',
ws': 1, 'updatedColumns': 2, 'updatedCells': 2}
Mensaje recibido: {'date': '2021-02', 'total_injured': 1154}
Datos actualizados en Google Sheets: {'spreadsheetId': '1Dflw-mewAhm6Nqa60dxc9BgLTklrMrbnyWf485XWOFE', 'updatedRange': 'Hoja1!A3:B3',
ws': 1, 'updatedColumns': 2, 'updatedCells': 2}
Mensaje recibido: {'date': '2021-03', 'total_injured': 954}
Datos actualizados en Google Sheets: {'spreadsheetId': '1Dflw-mewAhm6Nqa60dxc9BgLTklrMrbnyWf485XWOFE', 'updatedRange': 'Hoja1!A4:B4',
ws': 1, 'updatedColumns': 2, 'updatedCells': 2}
Mensaje recibido: {'date': '2021-04', 'total_injured': 2122}
Datos actualizados en Google Sheets: {'spreadsheetId': '1Dflw-mewAhm6Nqa60dxc9BgLTklrMrbnyWf485XWOFE', 'updatedRange': 'Hoja1!A5:B5',
ws': 1, 'updatedColumns': 2, 'updatedCells': 2}
Mensaje recibido: {'date': '2021-05', 'total_injured': 1658}
Datos actualizados en Google Sheets: {'spreadsheetId': '1Dflw-mewAhm6Nqa60dxc9BgLTklrMrbnyWf485XWOFE', 'updatedRange': 'Hoja1!A6:B6',
ws': 1, 'updatedColumns': 2, 'updatedCells': 2}
```

Here the Google Sheets table is filled in



	A	B
1	FECHA	TOTAL HERIDOS POR MES
2	2021-01	1146
3	2021-02	1154
4	2021-03	954
5	2021-04	2122
6	2021-05	1658
7	2021-06	1794
8	2021-07	1656
9	2021-08	1968
10	2021-09	1633
11	2021-10	1866
12	2021-11	3611
13	2021-12	2495
14	2022-01	1645
15	2022-02	1270

To implement data integration from Kafka to Google Sheets, a process was followed that involved setting up credentials in Google Cloud, authorizing access to the Google Sheets spreadsheet, and creating the script that consumes the data from Kafka to insert it into Google Sheets.

Setting up the Google Sheets API:

- In the Google Cloud console, enable the Google Sheets API.
- Create a service account in Google Cloud and download the JSON credentials file.
- Share the Google Sheets spreadsheet with the email generated by the service account (e.g. accidents@accidents-usa.iam.gserviceaccount.com).

Preparing the Kafka Consumer:

- Set up a Kafka consumer in Python to connect to the specific topic (in this case, injuries_by_day).
- Deserialize the messages received from Kafka using json.

Authentication in Google Sheets:

- Use the credentials from the JSON file to authenticate and connect to the Google Sheets API with googleapiclient.
- Specify the spreadsheet ID and cell range to write data to.

Google Sheets Update:

- Read the last available row in Google Sheets to avoid overwrites.
- Insert data received from Kafka into the cells of the specified sheet.

Data Consumption and Update:

- Consume Kafka messages in a continuous loop.
- Send each received data to Google Sheets, updating the sheet in real time.
- Include a 3-second wait between messages to avoid overload.

5. Real-Time Visualization with Looker Studio

The dashboard in Looker Studio is configured to display updated information in real time. Below is a brief description of the process and the tools used:

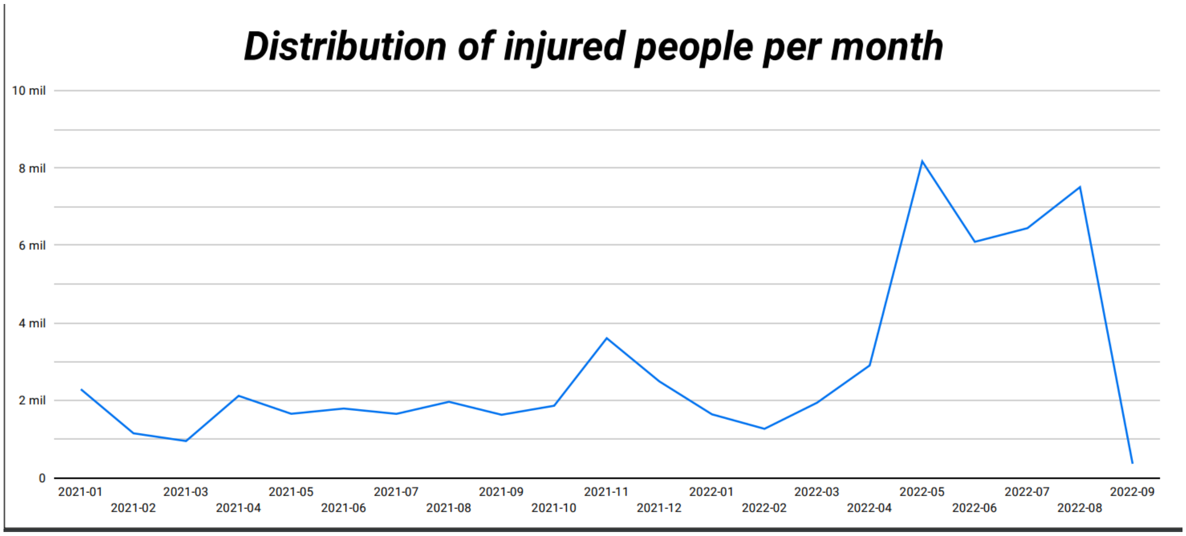
1. Data Connection:

- The Google Sheets file is used as a data source for the dashboard.
- Whenever data is updated in Google Sheets (via the Kafka consumer), Looker Studio is automatically refreshed to reflect the latest information.

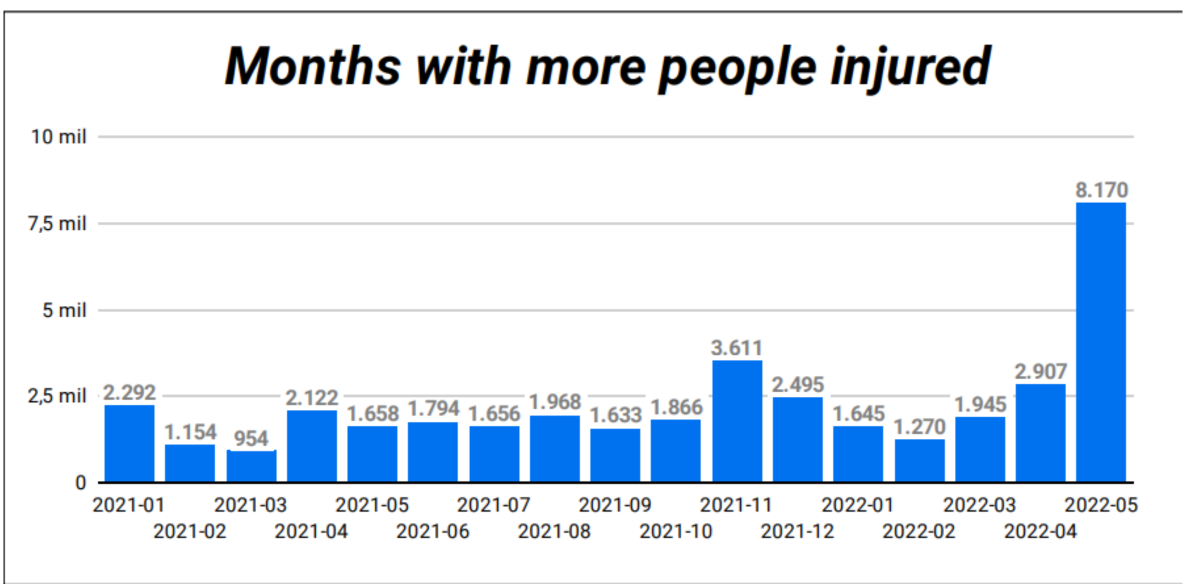
2. Dashboard Setup:

Several key visualizations are created to display accident and injury metrics:

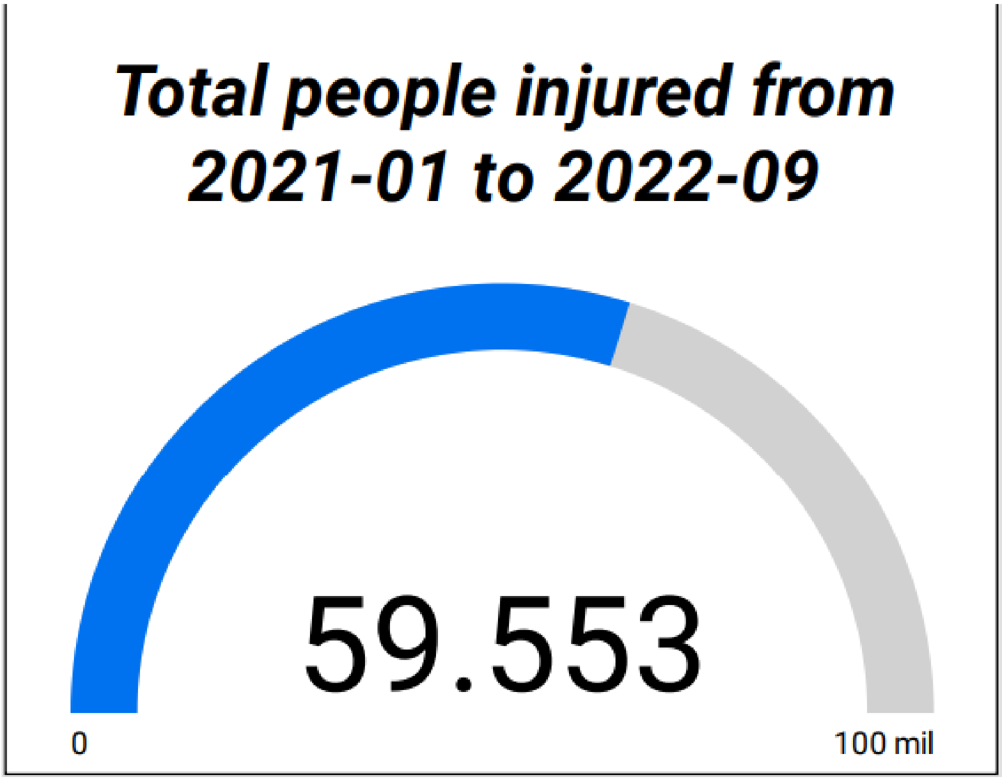
Line Chart: Visualizes injury and accident trends over time, allowing you to identify spikes and patterns.



Bar Chart: Comparison of categorical data, such as the number of accidents by day of the week or the distribution of injuries.



Indicator: Displays critical metrics, such as the total number of injuries or fatalities, providing a clear view of current values.



6. DAG Implementation in Airflow

For the orchestration of the ETL process, Apache Airflow was used on a Linux Ubuntu virtual machine. The tasks of the created DAG are detailed below:

DAG Tasks

1. Data Extraction from the API

2. Data Cleaning from the API

3. Data Extraction from the CSV

4. Data Cleaning from the CSV

5. Data Unification

6. Loading into the Dimensional Model

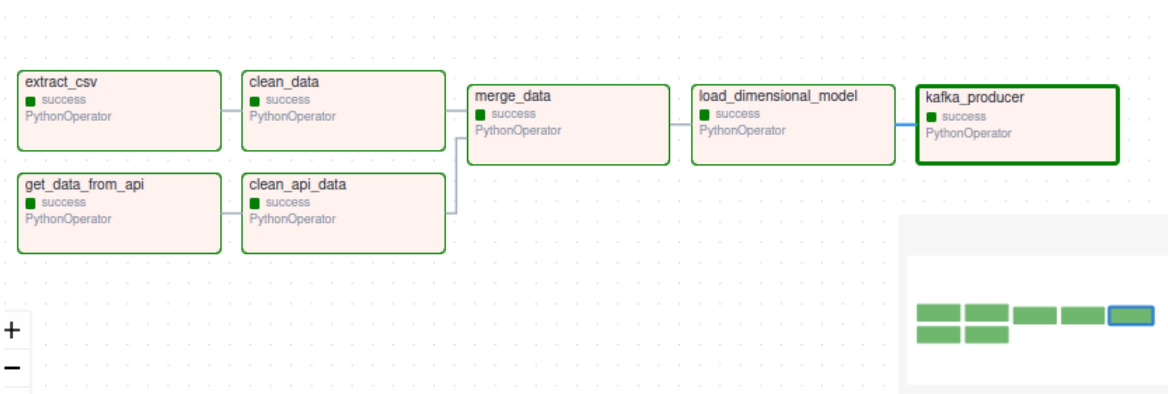
7. Data Streaming with Kafka

dags > etl_dag.py > ...

willyb, 9 hours ago | 2 authors (willyb and one other)

```python
import sys
import os

# Agregar la ruta al paquete src
sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '../src')))

from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime, timedelta

# Importar funciones desde los módulos en src
from API_connection import get_data_from_api
from API_cleaning import clean_api_data
from extract import extract_data
from data_cleaning import clean_data
from merge_data import merge_data
from load_dimensional_model import dimensional_model
from kafka_producer import kafka_producer


# Definir el DAG
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2024, 11, 13),
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=1)
}

with DAG(
    'accidents_etl_pipeline',
    default_args=default_args,    willyb, last month • Añadir DAG de ETL para datos de accidentes, lim…
    description='ETL Pipeline for Accidents Data',
    schedule_interval='@daily',
```

In the Ariflow interface it would look like this after having completed all the tasks:



7. Conclusion

This project provides a comprehensive solution for managing and analyzing traffic accident data, integrating multiple technologies into a single workflow. The combination of Airflow, Kafka, PostgreSQL and Looker Studio allowed not only to store and analyze data, but also to visualize metrics in real time, offering valuable insights for decision making.

Technologies used:

- **Airflow**
- **Kafka**
- **Docker**
- **PostgreSQL**
- **Google Sheets y Looker Studio**
- **Power BI**
- **Github**
- **Jupiter notebooks**
- **Kaggle**
- **Pandas**