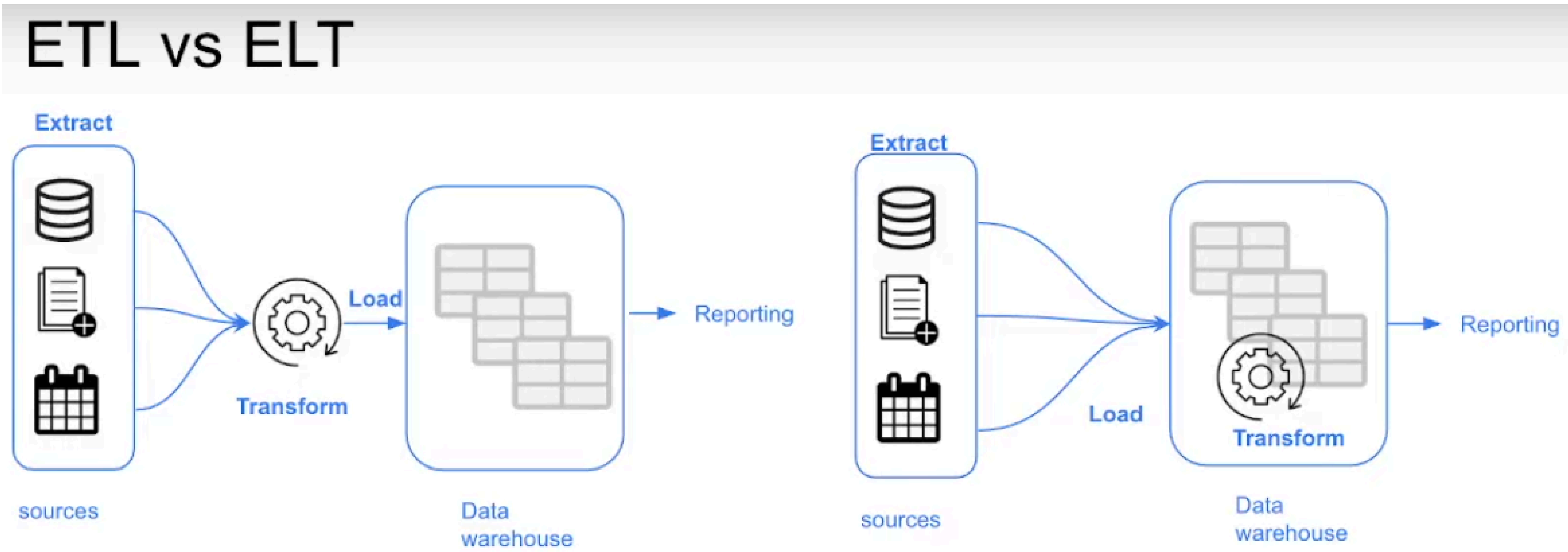
Outline

1. Introduction to dbt

Before discuss about dbt (data build tool), let's first understand the concepts of ETL and ELT. The ETL (Extract, Transform, Load) process is a traditional method of extracting raw data from sources, such as files or databases, transforming this data as necessary, and then loading it into a data warehouse.. On the other hand, the ELT (Extract, Load, Transform) process represents a modern approach, where raw data is first loaded into the data warehouse before being transformed within the warehouse itself. The following diagram can squematicaly represent the ETL and ELT process:



This shift of using ETL instead of ELT is due to the increased processing capabilities of modern data warehouses, which can efficiently handle complex transformations. dbt simplifies the ELT process by enabling data analysts and engineers to define transformations in SQL. This approach allows for the modular, version-controlled, and testable transformation of raw data, making it ready for business use. The following table summarizes benefits of using ETL and ELT:

ETL	ELT
Slightly more stable	Faster and more flexible
Higher storage and compute costs	Lower cost and lower maintenance

There is two ways of using dbt. The first one is using dbt core, which is a command-line tool that enables data analysts and engineers to transform data in their warehouse more effectively. The second one is using dbt cloud, which is a cloud-based service that provides a user interface for dbt core, as well as additional features such as scheduling, monitoring, and collaboration. In this notes, we will focus on the dbt core with the integration of Airflow.

The only dependency we need to install is `dbt-core`:

```
pip install dbt-core
```

Making the `dbt init` command available to create a new dbt project with the necessary directory structure and template files. In cases where we would need to use the dbt core locally, without a containerized environment, we would need to install the database adapter for the database we are using, like postgres and bigquery:

```
pip install dbt-postgres
    pip install dbt-bigquery
```

2. Integration of dbt core with Airflow using Astro CLI

The Astro CLI is designed to help developers easily create, manage, and deploy Airflow projects. The Astro CLI can quickly generate a new Airflow project with the necessary configuration and folders, without the need of configuring all setting in the docker-compose file as would be required to run airflow with docker. The Astro CLI abstracts all this dificults and make it easy to run Airflow locally, by creating four docker containers for the Airflow webserver, scheduler, and triggerer, as well as a Postgres database. Another advantage of using the Astro CLI is that it allows for the integration of dbt core within Airflow, which is the main focus of this notes. This integration make obsolete the need of using the dbt clound, as we can run dbt core within Airflow, schedule dbt runs with Airflow's scheduler, create dbt models with Airflow's DAGs and leverage Airflow's UI to monitor dbt runs.

To install Astro CLI in Linux, we can use the following command:

```
curl -sSL https://install.astronomer.io | sudo bash
```

After installation, we can create a new Airflow project using the following command:

```
astro dev init
```

This will create all the folder structure of airflow and the necessary files to configure and run the Airflow project. The folder structure and files are the following:

```
airflow-project/
    ├── .astro/
    │   ├── config.yaml
    ├── dags/
    │   ├── .airflowignore
    ├── include/
    ├── plugins/
    ├── tests/
    │   ├── dags/
    ├── .dockerignore
    ├── .env                    # Environment variables
    ├── .gitignore
    ├── airflow_settings.yaml   # Setting the connections to databases like postgres
    ├── Dockerfile
    ├── packages.txt            # For OS-level packages to install
    ├── requirements.txt        # Python dependencies for Airflow
```

Now we have our Airflow project ready to run. We can check the documentation how to integrate the dbt with Airflow in here. When checking the documentation, it say that we need to add the following lines of code in Dockerfile:

Dockerfile

```
RUN python -m venv dbt_venv && source dbt_venv/bin/activate && \
        pip install --no-cache-dir dbt-postgres && deactivate
```

instead of using pip install for each library, we can create a `dbt_requeriments.txt` file with many libraries we want for the database providers:

dbt_requeriments.txt

```
dbt-core>=1.7.8
    dbt-postgres>=1.7.8
    dbt-bigquery>=1.7.6
```

make sure to also install the packages locally to run dbt commands outside the container:

```
pip install -r dbt_requirements.txt
```

Now we can slightly change the Dockerfile to copy the `dbt_requirements.txt` file and install the packages in the virtual environment. The content inside the Dockerfile would be:

Dockerfile

```
FROM quay.io/astronomer/astro-runtime:10.3.0

    WORKDIR "/usr/local/airflow"

    COPY dbt-requirements.txt ./
    RUN python -m virtualenv dbt_venv && source dbt_venv/bin/activate && \
        pip install --no-cache-dir -r dbt_requirements.txt && deactivate
```

This way, we create a isolated Python environments, where `dbt_venv` is the name of the new virtual environment to be created. The `source dbt_venv/bin/activate` command activates the virtual environment, and the `pip install --no-cache-dir -r dbt-requirements.txt` command installs the packages listed in the `dbt-requirements.txt` file. The `deactivate` command deactivates the virtual environment. This way we are taking precautions so that the dbt not conflict with the other packages installed in the Airflow environment.

The next step is to create a new directory called `dbt` inside the `dag/` directory. The `dbt-core` can create automatically the folder structure and the files using the following command inside `dbt` directory:

```
dbt init
```

Inside the `dag/dbt/` directory we will have the following structure:

```
airflow-project/
    ├── .astro/
    ├── dags/
    │   ├── dbt/
    │   │   ├── logs
    │   │   ├── dbt-project/
    │   │   │   ├── analyses/
    │   │   │   ├── models/
    │   │   │   ├── macros/
    │   │   │   ├── seeds/
    │   │   │   ├── snapshots/
    │   │   │   ├── tests/
    │   │   │   ├── `dbt_project.yml`
    ├── include/
    ├── plugins/
    ├── tests/
    ├── .dockerignore
    ├── .env
    ├── .gitignore
    ├── airflow_settings.yaml
    ├── Dockerfile
    ├── packages.txt
    ├── requirements.txt
```

The Astro CLI is built on top of Docker Compose, a tool for defining and running multi-container Docker applications. To override the default CLI configurations, add a `docker-compose.override.yml` file to Astro project directory. The values in this file override the default settings when we run `astro dev start`. This information can be found here. The `docker-compose.override.yml` file is used to persist the connection between the Airflow and dbt, so that the dbt models can be created and run within the Airflow environment. The content of the `docker-compose.override.yml` file would be:

docker-compose.override.yml

```yaml
version: "3.1"
services:
scheduler:
    volumes:
    -
/home/user/.google/credentials/google_credentials.json:/usr/local/airflow/google/credentials/google_crede
    - ./dags/dbt:/usr/local/airflow/dags/dbt:rw

webserver:
    volumes:
    -
/home/user/.google/credentials/google_credentials.json:/usr/local/airflow/google/credentials/google_crede
    - ./dags/dbt:/usr/local/airflow/dbt:rw

triggerer:
    volumes:
    -
/home/user/.google/credentials/google_credentials.json:/usr/local/airflow/google/credentials/google_crede
    - ./dags/dbt:/usr/local/airflow/dags/dbt:rw
```

Each service is a container that runs a specific process, such as the webserver, scheduler, and triggerer. The volumes will be the same for all services, and the `volumes` option is used to mount the `google_credentials.json` file and the `dags/dbt` directory.

The `google_credentials.json` file is the file that we download from the Google Cloud Platform when we create a new service account for a project. This is used to authenticate the Airflow environment with the Google Cloud Platform to use the Google Cloud SDK libraries. We also add to the `.env` file the following line to specify the path to the `google_credentials.json` file in the container:

.env

```
GOOGLE_APPLICATION_CREDENTIALS = "/usr/local/airflow/google/credentials/google_credentials.json"
```

---

dbt will automatically read the `.env` file when required to access google cloud. The volume for `dags/dbt` is to automatically synchronize the dbt models with the Airflow environment. The `rw` option is used to give read and write permissions to the Airflow services.

We are almost there, the last step is to add packages to the `packages.txt` file. The `packages.txt` file is used to install OS-level packages in the Airflow environment. The content of the `packages.txt` file would be:

packages.txt

```
gcc
    python3-venv
```

To properly set up our Airflow environment with the necessary astronomer-cosmos package and its specific database integrations, we include the following lines in our `requirements.txt` file:

requirements.txt

```
apache-airflow-providers-google
    astronomer-cosmos[dbt-bigquery]
    astronomer-cosmos[dbt-postgres]
    pyarrow
```

The packages `astronomer-cosmos[dbt-bigquery]` and `astronomer-cosmos[dbt-postgres]` will include all the necessary packages to run dbt within the Airflow environment with all the databases adapters for bigquery and postgres. The Google Cloud SDK libraries is instated with `apache-airflow-providers-google` package to authenticate the Airflow environment with the Google Cloud Platform. The `pyarrow` package is used to read and write parquet files in the Airflow environment.

The last step is now to run the following command inside the root of the Airflow project to start the Airflow environment with the dbt integration:

```
astro dev start
```

The final structure for the directory must be equal to the following:

```
airflow-project/
    ├── .astro/
    ├── dags/
    │   ├── dbt/
    │   │   ├── logs
    │   │   ├── dbt-project/
    │   │   │   ├── analyses/
    │   │   │   ├── models/
    │   │   │   ├── macros/
    │   │   │   ├── seeds/
    │   │   │   ├── snapshots/
    │   │   │   ├── tests/
    │   │   │   ├── dbt_project.yml
    ├── include/
    ├── plugins/
    ├── tests/
    ├── .dockerignore
    ├── `.env`
    ├── .gitignore
    ├── airflow_settings.yaml
    ├── `dbt-requirements.txt`
    ├── `Dockerfile`
    ├── `docker-compose.override.yml`
    ├── `packages.txt`
    ├── `requirements.txt`
```

All edited files are colored. The `astro dev start` command will start the Airflow environment with the dbt integration. The Airflow environment will be available at `http://localhost:8080`.

2.1 dbt-core Locally

To test our models in dbt is good to have the local environment to run the dbt commands, like `dbt run` and `dbt seed`. For using dbt-core locally, its required to have a profile file in the `~/.dbt/` directory, for more info go to dbt documentation. We can check where the dbt is looking for the profile file using the following command:

```
dbt debug --config-dir
```

which should return something like:

```
To view your profiles.yml file, run:
    xdg-open /home/user/.dbt
```

The profile file is used to specify the connection to the database, and the credentials to authenticate the connection. The profile file is a YAML file that contains the following information, in this case for a bigquery database:

profile.yml

```yaml
bigquery-db:
        target: dev
        outputs:
            dev:
                type: bigquery
                host: service-account
                project: project-id
                dataset: dataset-id
                threads: 1
                keyfile: /home/user/.google/credentials/google_credentials.json
```

Inside the `dag/dbt/` directory we will have the following structure:

```
airflow-project/
    ├── .astro/
    ├── dags/
    │   ├── dbt/
    │   │   ├── logs
    │   │   ├── dbt-project/
    │   │   │   ├── analyses/
    │   │   │   ├── models/
    │   │   │   ├── macros/
    │   │   │   ├── seeds/
    │   │   │   ├── snapshots/
    │   │   │   ├── tests/
    │   │   │   ├── `dbt_project.yml`
    ├── include/
    ├── plugins/
    ├── tests/
    ├── .dockerignore
    ├── .env
    ├── .gitignore
    ├── airflow_settings.yaml
    ├── dbt-requirements.txt
    ├── Dockerfile
    ├── docker-compose.override.yml
    ├── packages.txt
    ├── requirements.txt
```

Inside the file `dbt_project.yml` have the configuration of the dbt project, where we can specify the name of the project, the profile used for the project and others setups. The part we need to change in the `dbt_project.yml` file is the profile to match the profile name in the `~/.dbt/profiles.yml` file.

dbt_project.yml

```yaml
name: 'taxi_rides_ny'
    version: '1.0.0'
    config-version: 2

    profile: 'bigquery-db'
```

Before running models that depend on seeds, we need to run the following command inside the project directory `dag/dbt/dbt-project/` at the terminal to load the seed data into the database:

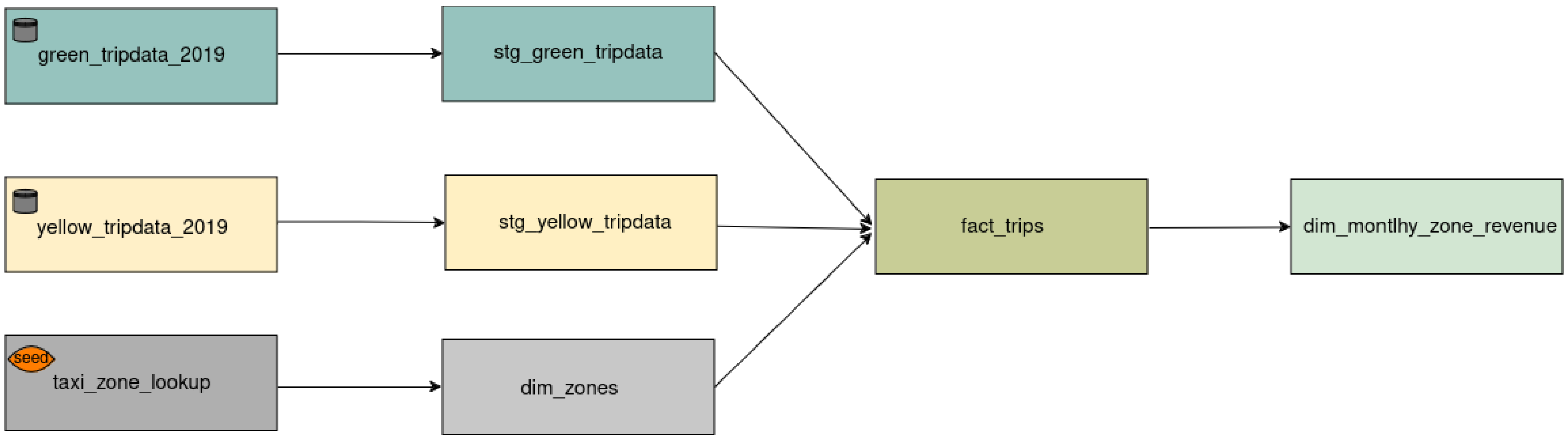
```
dbt seed
```

This command will create tables in our database from the CSV files in our seeds directory. To run a dbt models, we can use the following command inside the `dag/dbt/dbt-project/` directory:

```
dbt run
```

3. Building a ELT pipeline with Airflow and dbt

Here we will build a model using three datasets, the `green_tripdata`, `yellow_tripdata` and the `taxi_zone_lookup` from NYC taxi. The green_tripdata and yellow_tripdata are the datasets that contains the taxi trips, and the taxi_zone_lookup is the dataset that contains the information about the taxi zones.The following diagram shows the structure of the dbt model:



In the first level of this diagram we have the `green_taxi_external_2019` and `yellow_taxi_external_2019` as the datasets ingested into the bigquery and the `taxi_zone_lookup` as the seed data. The Seeds is the directory inside dbt directory where contain a static data that is typically not expected to change frequently.

For the second level of this diagram we have the Staging models that serves as an intermediate layer in the data transformation process. They are responsible for ingesting raw data from the sources `green_taxi_external_2019` and `yellow_taxi_external_2019` performing basic transformations to prepare the data for further processing.

For the third level of this diagram we have the `fact_trips` and `dimension_zones` models. The `fact` model is the model that contains the data that is being measured, and the `dimension` model is the model that contains the data that provides context for the measurements. The `fact` and `dimension` models are the final models that are used to create the reports and dashboards.

3.1 DAG to Extract and Load Data Into GCS

The data used in this project is from the NYC taxi dataset. We are only interested in the Yellow and Green taxi trips dataset from the year 2019 and the taxi zone lookup dataset. For the taxi trip the format of the data is `.parquet` and for the zone lookup is in `.csv` .

To start this project, the idea is to create a pipeline in Airflow to ingest the taxi data into a bucket in Google Cloud Storage and then create a external table in BigQuery. This way we can use the BigQuery as the warehouse to store the data and make the transformation with dbt. For the taxi zone lookup we use as a `seed` data, which is a static data that is typically not expected to change frequently. The Seed is the directory inside dbt directory where contain the static data, shown in the directory structure below:

```
astro-airflow/
    ├── .astro/
    ├── dags/
    │   ├── dbt/
    │   │   ├── logs
    │   │   ├── taxi_rides_ny/
    │   │   │   ├── analyses/
    │   │   │   ├── models/
    │   │   │   ├── macros/
    │   │   │   ├── seeds/
    │   │   │   │   ├── `taxi_zone_lookup.csv`
    ....
```

Before creating the pipeline, let's check the datasets for the yellow and green taxi from 01-2019 and for the taxi zones.

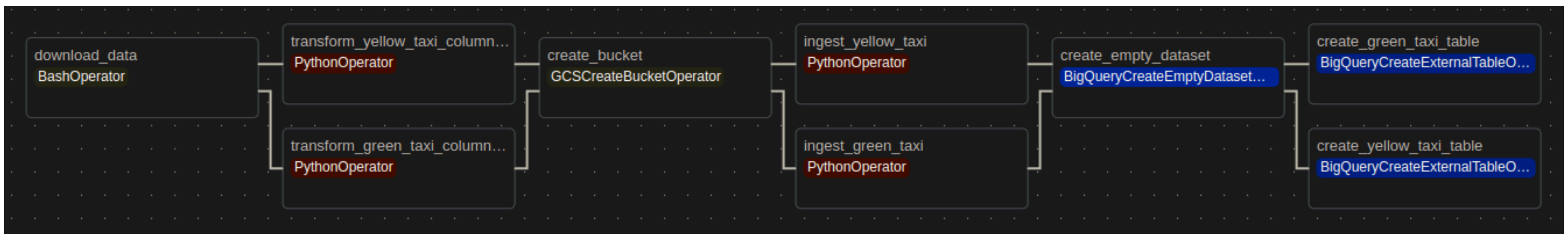
In [ ]:
```python
import pandas as pd

yellow_tripdata_2019_01 = pd.read_parquet('data/yellow_tripdata_2019-01.parquet')
green_tripdata_2019_01 = pd.read_parquet('data/green_tripdata_2019-01.parquet')
taxi_zone_lookup = pd.read_csv('data/taxi_zone_lookup.csv')

display(yellow_tripdata_2019_01.head(2))
display(green_tripdata_2019_01.head(2))
display(taxi_zone_lookup.head(2))
```

	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	RatecodeID	store_and_fwd_flag	PULocat
0	1	2019-01-01 00:46:40	2019-01-01 00:53:20	1.0	1.5	1.0	N	
1	1	2019-01-01 00:59:47	2019-01-01 01:18:59	1.0	2.6	1.0	N	

	VendorID	lpep_pickup_datetime	lpep_dropoff_datetime	store_and_fwd_flag	RatecodeID	PULocationID	DOLocationID	passenger_
0	2	2018-12-21 15:17:29	2018-12-21 15:18:57	N	1.0	264	264	
1	2	2019-01-01 00:10:16	2019-01-01 00:16:32	N	1.0	97	49	

	location_id	borough	zone	service_zone
0	1	EWR	Newark Airport	EWR
1	2	Queens	Jamaica Bay	Boro Zone

In airflow a pipeline is called a Directed Acyclic Graph (DAG). A DAG is a collection of all the tasks we want to run, organized in a way that reflects their relationships and dependencies. The main idea for this DAG is to ingest the Green and Yellow taxi data from the NYC taxi into a bucket in Google Cloud Storage and then create a external table in BigQuery. The following diagram shows the final DAG structure that we desire to construct:



Let's describe each task in the DAG structure:

1. **download_data**: The DAG begins by downloading the latest green and yellow taxi trip data with the `BashOperator`, formatted as parquet files, from specified URLs.

2. **transform_green_taxi_columns_to_snake** and **transform_yellow_taxi_columns_to_snake**: The `PythonOperator` task will transform the column names of the green taxi trip data to snake case.

3. **create_bucket**: create a bucket in Google Cloud Storage to store the taxi data with the `GCSCreateBucketOperator`.

4. **ingest_green_taxi** and **ingest_yellow_taxi**: The `PythonOperator` will ingest the green and yellow taxi trip data into a bucket in Google Cloud Storage.

5. **create_empty_dataset**: The `BigQueryCreateEmptyDatasetOperator` will create a empty dataset in BigQuery to store the taxi tables.

6. **create_green_taxi_table** and **create_yellow_taxi_table**: The `BigQueryCreateExternalTableOperator` will create a external table in BigQuery to store the green and yellow taxi trip data.

For more information about each operator, we can check the Astronomer website. In the search bar we can type the name of the operator and check the documentation and some examples of how to use it.

To create this DAG, first create a `.py` file named `elt_nyc_taxi_bq.py` inside the `dags` directory. The content of the `elt_nyc_taxi_bq.py` file would be:

elt_nyc_taxi_bq.py

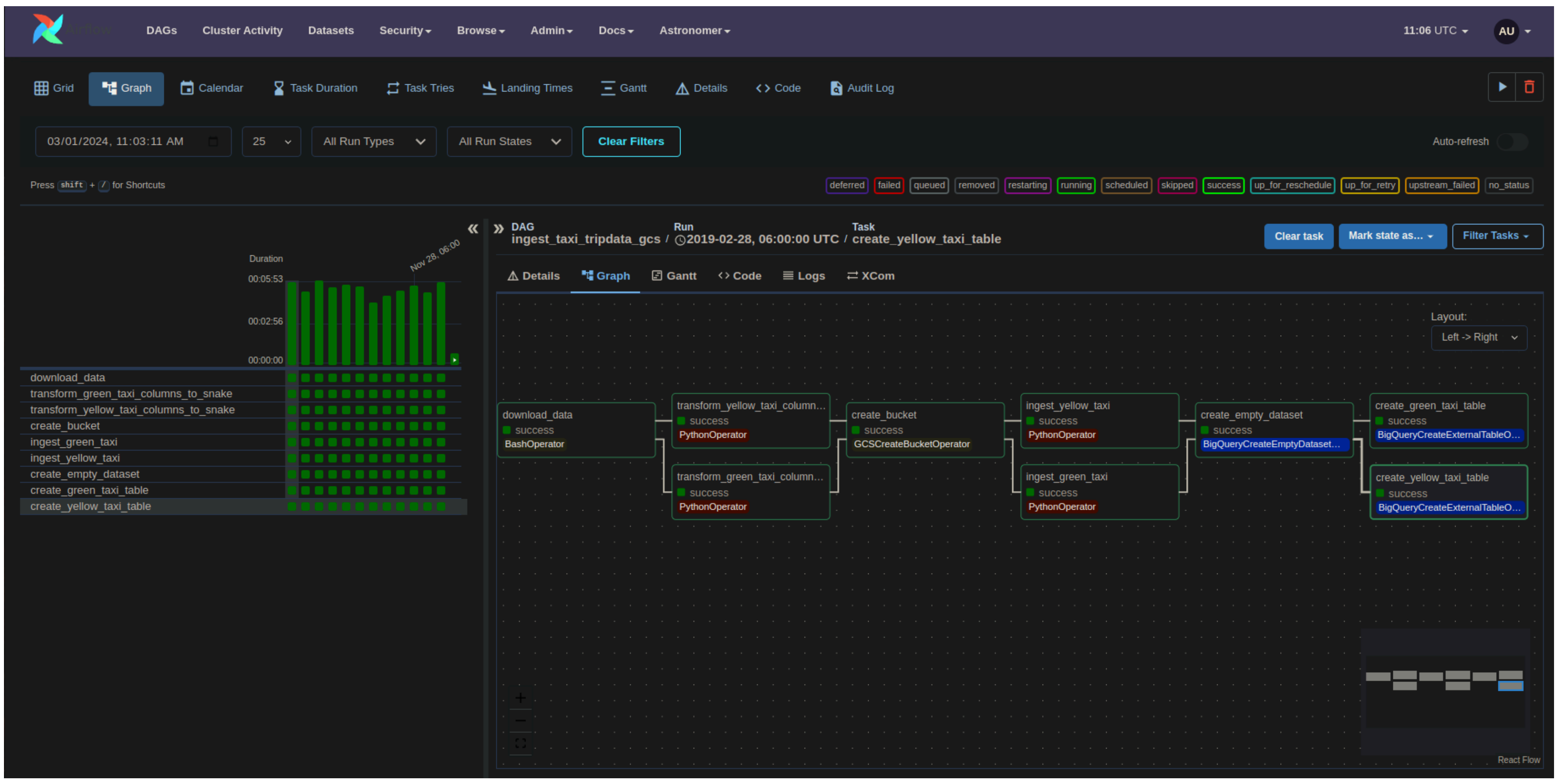
```
# [START import modules]
    from airflow import DAG
    from datetime import datetime
    from google.cloud import storage  # For accessing Google Cloud Storage
    import pandas as pd
    import re
    import pyarrow.parquet as pq
    import pyarrow as pa
    from os import getenv

    # Import specific operators from Airflow
    from airflow.operators.bash import BashOperator
    from airflow.operators.python import PythonOperator
    from airflow.providers.google.cloud.operators.gcs import GCSCreateBucketOperator
    from airflow.providers.google.cloud.transfers.local_to_gcs import LocalFilesystemToGCSOperator
    from airflow.providers.google.cloud.operators.bigquery import BigQueryCreateExternalTableOperator,
BigQueryCreateEmptyDatasetOperator
    # [END import modules]

    # [START Env Variables]-----------------------------------------------------------------
    # Define environment variables for file paths and GCP configurations
    # These variables allow for dynamic data paths and project settings
    BASE_URL_1 = 'https://d37ci6vzurychx.cloudfront.net/trip-data'
    BASE_URL_2 = 'https://d37ci6vzurychx.cloudfront.net/trip-data'
    FILE_NAME_1 = 'green_tripdata_{{ execution_date.strftime(\'%Y-%m\') }}.parquet'
    FILE_NAME_2 = 'yellow_tripdata_{{ execution_date.strftime(\'%Y-%m\') }}.parquet'

    # Complete URLs for taxi data
    URL_1 = f'{BASE_URL_1}/{FILE_NAME_1}'
    URL_2 = f'{BASE_URL_2}/{FILE_NAME_2}'

     # Airflow home directory
    AIRFLOW_HOME = getenv("AIRFLOW_HOME", "/usr/local/airflow")
    # Paths to save taxi data
    FILE_PATH_1 = getenv('FILE_PATH_1', f'{AIRFLOW_HOME}/{FILE_NAME_1}')
    FILE_PATH_2 = getenv('FILE_PATH_2', f'{AIRFLOW_HOME}/{FILE_NAME_2}')
```

```python
DATASET_NAME = getenv("DATASET_NAME", 'nyc_taxi')                   # BigQuery dataset name
TABLE_NAME = 'green_taxi_{{ execution_date.strftime(\'%Y_%m\') }}'  # Table name pattern

# Year to donwload and table if for taxi data
YEAR = 2019
TABLE_ID_1 = f"green_taxi_external_{YEAR}"
TABLE_ID_2 = f"yellow_taxi_external_{YEAR}"

PROJECT_ID = getenv("PROJECT_ID", "de-bootcamp-414215")     # GCP Project ID
REGION = getenv("REGIONAL", "us-east1")
LOCATION = getenv("LOCATION", "us-east1")

BUCKET_NAME = getenv("BUCKET_NAME", 'nyc-taxi-data-414215')
# GCS folder for storing taxi data inside the bucket
GCS_BUCKET_FOLDER = getenv("GCS_BUCKET", 'nyc_taxi_trip_2019')

# Connection ID created in Airflow UI
CONNECTION_ID = getenv("CONNECTION_ID", "gcp_conn")
# [END Env Variables]

# [START default args] ----------------------------------------------------------------
# Define default arguments for the DAG
default_args = {
    "owner": "marcos benicio",
    "email": ['marcosbenicio@id.uff.br'],
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1
}
# [END default args]

# [START Python Functions]------------------------------------------------------------
# Define custom Python functions for data transformation and uploading
def transform_columns_to_snake(file_path):
    """
    Transforms column names from camel case to snake case for consistency.

    Args:
        file_path (str): The file path of the parquet file to transform.
    """
    original_table = pq.read_table(file_path)
    original_column_names = original_table.schema.names
    original_metadata = original_table.schema.metadata

    # Function to convert camel case to snake case
    def camel_to_snake(name):
        return re.sub(r'(?<=[a-z0-9])([A-Z])|(?<=[A-Z])([A-Z])(?=[a-z])', r'_\g<0>', name).lower()

     # Transform column names
    new_column_names = [camel_to_snake(name) for name in original_column_names]
    # Create new fields with transformed names and use to create a new schema and table
    fields = [pa.field(new_name, original_table.schema.field(original_name).type)
              for new_name, original_name in zip(new_column_names, original_column_names)]

    new_schema = pa.schema(fields, metadata=original_metadata)
    new_table = pa.Table.from_arrays(original_table.columns, schema=new_schema)

    # Overwrite the transformed table back to the file
    pq.write_table(new_table, file_path)

def filesystem_to_gcs(bucket, dst, src):
    """
    Uploads a file from the local filesystem to Google Cloud Storage,
    adjusting settings to prevent timeouts on large files.

    Args:
        bucket (str): The name of the GCS bucket.
        dst (str): The destination path and file name within the GCS bucket.
        src (str): The source path and file name on the local filesystem.
    """
    # Set max multipart upload size to 5 MB
    storage.blob._MAX_MULTIPART_SIZE = 5 * 1024 * 1024
    # Set default chunk size to 5 MB to prevent timeouts
    storage.blob._DEFAULT_CHUNKSIZE = 5 * 1024 * 1024

    # Initialize the GCS client and get bucket object
    client = storage.Client()
    bucket = client.bucket(bucket)
```

```
        # Create a blob object with the destination path and upload file
        blob = bucket.blob(dst)
        blob.upload_from_filename(src)

        print(f"File {src} uploaded to {dst} in bucket {bucket}.")
    # [END Python Functions]

    # [START DAG Object]----------------------------------------------------------------
    # Initialize the DAG object
    workflow = DAG(
            dag_id="elt_nyc_taxi_bq",
            default_args = default_args,
            description="""A DAG to export data from NYC taxi web,
            load the taxi trip data into GCS to create a BigQuery external table
            and transform the data with Dbt""",
            tags=['gcs', 'bigquery','data_elt', 'dbt', 'nyc_taxi'],
            schedule_interval="0 6 28 * *",
            start_date = datetime(YEAR, 1, 1),
            end_date = datetime(YEAR, 12, 30)
                )
    # [END DAG Object]

    # [START Workflow]----------------------------------------------------------------
    # Define the workflow using the DAG object
    with workflow:
        # Download taxi data from source URLs
        download_data = BashOperator(
            task_id="download_data",
            bash_command=f"""
                            curl -sSLo {FILE_PATH_1} {URL_1} && \\
                            curl -sSLo {FILE_PATH_2} {URL_2}
                            """
        )

        # Transform column names of the green taxi data to snake case
        transform_green_taxi_columns_to_snake = PythonOperator(
            task_id='transform_green_taxi_columns_to_snake',
            python_callable=transform_columns_to_snake,
            op_kwargs={'file_path': FILE_PATH_1},
        )

        # Transform column names of the yellow taxi data to snake case
        transform_yellow_taxi_columns_to_snake = PythonOperator(
            task_id='transform_yellow_taxi_columns_to_snake',
            python_callable=transform_columns_to_snake,
            op_kwargs={'file_path': FILE_PATH_2},
        )

        # Create a GCS bucket if it doesn't exist
        create_bucket = GCSCreateBucketOperator(
            task_id="create_bucket",
            bucket_name=BUCKET_NAME,
            storage_class="REGIONAL",
            location=LOCATION,
            project_id=PROJECT_ID,
            labels={"env": "dev", "team": "airflow"},
            gcp_conn_id=CONNECTION_ID
        )

        # Upload the transformed green taxi data to GCS
        ingest_green_taxi_gcs = PythonOperator(
            task_id="ingest_green_taxi",
            python_callable=filesystem_to_gcs,
            op_kwargs={"bucket": BUCKET_NAME, "dst": f"{GCS_BUCKET_FOLDER}/{FILE_NAME_1}", "src":
FILE_PATH_1}
        )

        # Upload the transformed yellow taxi data to GCS
        ingest_yellow_taxi_gcs = PythonOperator(
            task_id="ingest_yellow_taxi",
            python_callable=filesystem_to_gcs,
            op_kwargs={"bucket": BUCKET_NAME, "dst": f"{GCS_BUCKET_FOLDER}/{FILE_NAME_2}", "src":
FILE_PATH_2}
        )

        # Create an empty dataset in BigQuery if it doesn't exist
        create_empty_dataset = BigQueryCreateEmptyDatasetOperator(
            task_id="create_empty_dataset",
```

```
            dataset_id=DATASET_NAME,
            project_id=PROJECT_ID,
            location=LOCATION,
            gcp_conn_id=CONNECTION_ID
        )
        # Create an external table in BigQuery for the green taxi data
        bigquery_green_taxi_table = BigQueryCreateExternalTableOperator(
            task_id="create_green_taxi_table",
            table_resource={
                'tableReference': {
                    'projectId': PROJECT_ID,
                    'datasetId': DATASET_NAME,
                    'tableId': TABLE_ID_1,
                },
                'externalDataConfiguration': {
                    'sourceFormat': 'PARQUET',
                    'sourceUris':
[f"gs://{BUCKET_NAME}/{GCS_BUCKET_FOLDER}/green_tripdata_*.parquet"],
                }
            },
            gcp_conn_id=CONNECTION_ID
        )
        # Create an external table in BigQuery for the green taxi data
        bigquery_yellow_taxi_table = BigQueryCreateExternalTableOperator(
            task_id="create_yellow_taxi_table",
            table_resource={
                'tableReference': {
                    'projectId': PROJECT_ID,
                    'datasetId': DATASET_NAME,
                    'tableId': TABLE_ID_2,
                },
                'externalDataConfiguration': {
                    'sourceFormat': 'PARQUET',
                    'sourceUris':
[f"gs://{BUCKET_NAME}/{GCS_BUCKET_FOLDER}/yellow_tripdata_*.parquet"],
                }
            },
            gcp_conn_id=CONNECTION_ID
        )

    download_data >> [transform_green_taxi_columns_to_snake, transform_yellow_taxi_columns_to_snake] \
    >> create_bucket >> [ingest_green_taxi_gcs, ingest_yellow_taxi_gcs ] \
    >> create_empty_dataset >> [bigquery_yellow_taxi_table, bigquery_green_taxi_table]
    # [END Workflow]
```

With the `elt_nyc_taxi_bq.py` inside the `dags` directory, we can now access the Airflow UI and check if the DAG is available and run it. In the Airflow UI we should see the following:



Also, don't forget to configure the connections in the Airflow UI. The connections are the way to connect the Airflow with the Google Cloud Platform. To set a connection go to Admin -> Connections and click on the `Create` button and select the `Google Cloud Platform` option. This will require again a json file with the credentials to authenticate the Airflow with the Google Cloud Platform, the same used before.

We are now with the following directory structure:

```
astro-airflow/
    ├── .astro/
    ├── dags/
    │   ├── dbt/
    │   │   ├── logs
    │   │   ├── taxi_rides_ny/
    │   │   ├── `elt_nyc_taxi_bq.py`
    ├── include/
    ├── plugins/
    ├── tests/
    ├── .dockerignore
    ├── .env
    ├── .gitignore
    ├── airflow_settings.yaml
    ├── dbt-requirements.txt
    ├── Dockerfile
    ├── docker-compose.override.yml
    ├── packages.txt
    ├── requirements.txt
```

3.2 dbt Macros and Packages

In dbt, macros are pieces of code written in Jinja that are used for generating SQL queries. They are essentially functions that can be defined to encapsulate logic using `if` and `for` statements within SQL code for reuse across a dbt project. Macros can be used to perform operations like data manipulation, formatting, and conditional logic, making dbt models more dynamic and modular. They help to keep the code DRY (Don't Repeat Yourself) by allowing to write a piece of logic once and reuse it in multiple models or analyses.

Let's create some macros that will be used in our project. To create a macro, we need to create a file with the `.sql` extension inside the `dags/dbt/taxi_rides_ny/macros` directory. The content of the `get_payment_type_description.sql` file would be:

get_payment_type_description.sql

```
{#
        This macro returns the description of the payment_type
    #}

    {% macro get_payment_type_description(payment_type) -%}

        case ( {{ payment_type }} as integer)
            when 1 then 'Credit card'
            when 2 then 'Cash'
            when 3 then 'No charge'
            when 4 then 'Dispute'
            when 5 then 'Unknown'
            when 6 then 'Voided trip'
            else 'EMPTY'
        end

    {%- endmacro %}
```

This macro is designed to return the description of a payment type based on its numerical value. `get_payment_type_description` is the function name, and `payment_type` is the parameter that will be passed to the function. The `payment_type` parameter is used to determine the payment type description.

The `{%-` indicates to Jinja to strip any whitespace that appears immediately after the tag. It ensures that there is no whitespace before the end of the macro, which can be important when the macro is used in generating code or queries, where extra whitespace could cause syntax errors or unintended formatting. The same idea is used for `{%-` at the end, ensuring that any whitespace immediately before the macro tag is removed.

We can also import packages from dbt package hub like libraries in other programming languages. By adding packages to the `packages.yml` file, we can use the functions and macros defined in the packages in our dbt project. Let's create the `packages.yml` file inside the `dags/dbt/taxi_rides_ny` directory and add the `dbt_utils` and the `codegen` packages to it.

packages.yml

```
packages:
      - package: dbt-labs/dbt_utils
        version: 1.1.1
      - package: dbt-labs/codegen
        version: 0.12.1
```

To instal the packages, we need to run the following command inside the `dags/dbt/taxi_rides_ny` directory, where the `dbt_project.yml` is located directory:

```
dbt deps
```

We should see the new directory `dbt_packages` inside the `dags/dbt/taxi_rides_ny` directory. This directory contains the packages that was defined in the `packages.yml` file. The final directory structure would be:

```
astro-airflow/
    ├── .astro/
    ├── dags/
    │   ├── dbt/
    │   │   ├── logs
    │   │   ├── taxi_rides_ny/
    │   │   │   ├── analyses/
    │   │   │   ├── `dbt_packages/`
    │   │   │   ├── models/
    │   │   │   ├── macros/
    │   │   │   │   ├── `get_payment_type_description.sql`
    │   │   │   ├── seeds/
    │   │   │   ├── snapshots/
    │   │   │   ├── tests/
    │   │   │   ├── dbt_project.yml
    │   │   │   ├── `packages.yml`
    │   │   │   ├── `packages.lock.yml`
        ...
```

3.3 Staging Model

Staging models are typically the first step in transforming raw data into a format that's more suitable for analysis. These models are crucial for ensuring consistency, cleanliness, and reliability of the data. Let's create inside `dags/dbt/taxi_rides_ny/model/` the `staging` directory with a `schema.yml`, `stg_staging_green_tripdata.sql` , `stg_staging_yellow_tripdata.sql` . The directory structure would be:

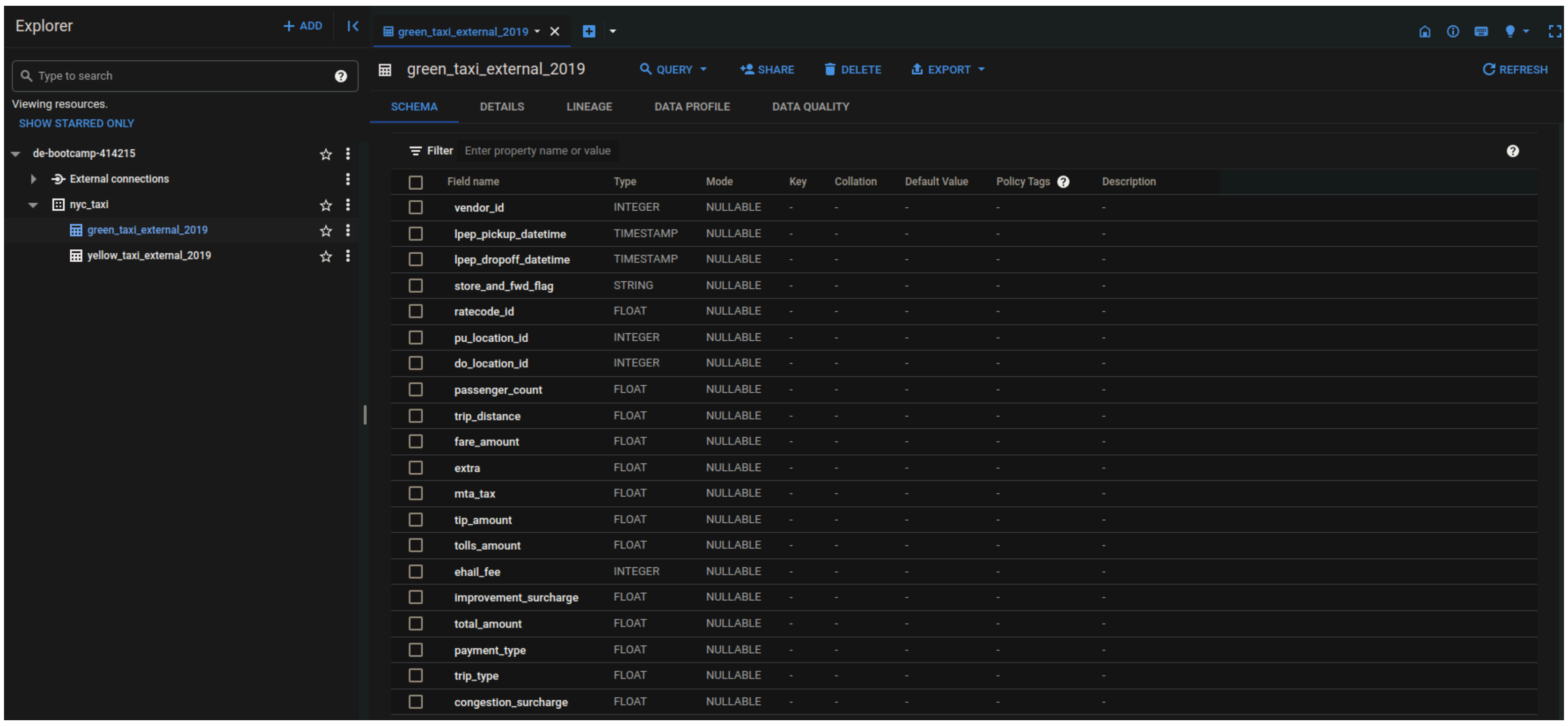
```
astro-airflow/
    ├── .astro/
    ├── dags/
    │   ├── dbt/
    │   │   ├── logs
    │   │   ├── taxi_rides_ny/
    │   │   │   ├── analyses/
    │   │   │   ├── dbt_packages/
    │   │   │   ├── models/
    │   │   │   │    ├── staging/
    │   │   │   │    │    ├── `schema.yml`
    │   │   │   │    │    ├── `stg_green_tripdata.sql`
    │   │   │   │    │    ├── `stg_yellow_tripdata.sql`
    │   │   │   ├── macros/
    │   │   │   ├── seeds/
    │   │   │   ├── snapshots/
    │   │   │   ├── tests/
    │   │   │   ├── dbt_project.yml
    │   │   │   ├── packages.yml
    │   │   │   ├── packages.lock.yml
        ...
```

Starting by the `schema.yml`, we configure as follows for the tables that we previously ingested into the BigQuery:

schema.yml

```yaml
version: 2

  sources:
    - name: staging                           # directory name in dbt/models
      database: de-bootcamp-414215            # dataset name in BigQuery
      schema: nyc_taxi                        # schema name in BigQuery where the table is located
      tables:                                 # list of tables in the schema
        - name: "green_taxi_external_2019"
        - name: "yellow_taxi_external_2019"
```

Next, in the `stg_staging_green_tripdata.sql` create the code using sql and jinja to transforms raw data from the `green_taxi_external_2019` table in BigQuery, as our source, into a view with properly formatted and casted columns, including a description of the payment type, and applies filtering to select only the first row for each combination of `vendor_id` and `lpep_pickup_datetime`. In BigQuery we should be able to see the `nyc_taxi` schema with the tables `green_taxi_external_2019` and `yellow_taxi_external_2019` as external tables as follows:



stg_staging_green_tripdata.sql

```
{{  config( materialized='view')  }}

  WITH tripdata AS
  (
      SELECT *,
      row_number() OVER(PARTITION BY vendor_id, lpep_pickup_datetime) AS rn
      FROM {{ source('staging','green_taxi_external_2019') }}
      WHERE vendor_id IS NOT NULL
  )
  SELECT
      -- identifiers
      {{ dbt_utils.generate_surrogate_key(['vendor_id', 'lpep_pickup_datetime']) }} AS trip_id,
      {{ dbt.safe_cast("vendor_id", api.Column.translate_type("integer")) }} AS vendor_id,
      {{ dbt.safe_cast("ratecode_id", api.Column.translate_type("integer")) }} AS ratecode_id,
      {{ dbt.safe_cast("pu_location_id", api.Column.translate_type("integer")) }} AS
pickup_location_id,
      {{ dbt.safe_cast("do_location_id", api.Column.translate_type("integer")) }} AS
dropoff_location_id,

      -- timestamps
      cast(lpep_pickup_datetime AS timestamp) AS pickup_datetime,
      cast(lpep_dropoff_datetime AS timestamp) AS dropoff_datetime,

      -- trip info
      store_and_fwd_flag,
      {{ dbt.safe_cast("passenger_count", api.Column.translate_type("integer")) }} AS passenger_count,
      cast(trip_distance AS numeric) AS trip_distance,
      {{ dbt.safe_cast("trip_type", api.Column.translate_type("integer")) }} AS trip_type,

      -- payment info
      cast(fare_amount AS numeric) AS fare_amount,
      cast(extra AS numeric) AS extra,
      cast(mta_tax AS numeric) AS mta_tax,
      cast(tip_amount AS numeric) AS tip_amount,
      cast(tolls_amount AS numeric) AS tolls_amount,

      cast(improvement_surcharge AS numeric) AS improvement_surcharge,
      cast(total_amount AS numeric) AS total_amount,
      coalesce({{ dbt.safe_cast("payment_type", api.Column.translate_type("integer")) }},0) AS
payment_type,
      {{ get_payment_type_description("payment_type") }} AS payment_type_description
  FROM tripdata
  WHERE rn = 1


  -- dbt build --select <model_name> --vars '{'is_test_run': 'false'}'
  {% if var('is_test_run', default=true) %}

      LIMIT 100

  {% endif %}
```

Let's better understand this code dissecting it into parts:

Part 1 - Configuration

The first piece of code is a configuration block used within a dbt model to specify the materialization type of the model.

```
{{  config( materialized='view')  }}
```

The `view` materialization is used to create a view in the database, which is a virtual table that does not store data, but instead retrieves data from the underlying tables when queried. Views provide a convenient way to represent and query data subsets or transformations without duplicating the underlying data.

Part 2 - Common Table Expression

The second piece of code is a common table expression (CTE) in SQL that is used to create a temporary result set that can be referenced within the main query. The `tripdata` CTE is used to create a temporary result set that contains the raw data from the `green_taxi_external_2019` source in BigQuery, as well as a row number for each combination of `vendor_id` and `lpep_pickup_datetime`.

```
WITH tripdata AS
  (
    SELECT *,
      ROW_NUMBER() OVER(PARTITION BY vendor_id, lpep_pickup_datetime) AS rn
    FROM {{ source('staging','green_taxi_external_2019') }}
    WHERE vendor_id IS NOT NULL
  )
```

The `ROW_NUMBER()` function assigns a unique sequential integer to each row within a partition. In this query, it's used to generate a row number (`rn`) for each row within each partition defined by the combination of `vendor_id` and `lpep_pickup_datetime`. The `PARTITION BY` clause partitions the result set into groups based on the specified columns (`vendor_id` and `lpep_pickup_datetime`). For each distinct combination of `vendor_id` and `lpep_pickup_datetime`, the row numbers will start from 1 and increment for each subsequent row.

Part 3 - Main Query

The third piece of code is the main query that transforms the raw data from the `green_taxi_external_2019` staging source into a view with properly formatted and casted columns, including a description of the payment type, and applies filtering to select only the first row for each combination of `vendor_id` and `lpep_pickup_datetime`.

```
SELECT
      -- identifiers
      {{ dbt_utils.generate_surrogate_key(['vendor_id', 'lpep_pickup_datetime']) }} AS trip_id,
      {{ dbt.safe_cast("vendor_id", api.Column.translate_type("integer")) }} AS vendor_id,
      {{ dbt.safe_cast("ratecode_id", api.Column.translate_type("integer")) }} AS ratecode_id,
      {{ dbt.safe_cast("pu_location_id", api.Column.translate_type("integer")) }} AS
pickup_location_id,
      {{ dbt.safe_cast("do_location_id", api.Column.translate_type("integer")) }} AS
dropoff_location_id,

      -- timestamps
      CAST(lpep_pickup_datetime AS TIMESTAMP) AS pickup_datetime,
      CAST(lpep_dropoff_datetime AS TIMESTAMP) AS dropoff_datetime,

      -- trip info
      store_and_fwd_flag,
      {{ dbt.safe_cast("passenger_count", api.Column.translate_type("integer")) }} AS passenger_count,
      CAST(trip_distance AS NUMERIC) AS trip_distance,
      {{ dbt.safe_cast("trip_type", api.Column.translate_type("integer")) }} AS trip_type,

      -- payment info
      CAST(fare_amount AS NUMERIC) AS fare_amount,
      CAST(extra AS NUMERIC) AS extra,
      CAST(mta_tax AS NUMERIC) AS mta_tax,
      CAST(tip_amount AS NUMERIC) AS tip_amount,
      CAST(tolls_amount AS NUMERIC) AS tolls_amount,
      CAST(improvement_surcharge AS NUMERIC) AS improvement_surcharge,
      CAST(total_amount AS NUMERIC) AS total_amount,
      COALESCE({{ dbt.safe_cast("payment_type", api.Column.translate_type("integer")) }},0) AS
payment_type,
      {{ get_payment_type_description("payment_type") }} AS payment_type_description
  FROM tripdata
  WHERE rn = 1
```

The `{{ dbt_utils.generate_surrogate_key(['vendor_id', 'lpep_pickup_datetime']) }}` function from the package `dbt_utils` is used to generate a surrogate key for the `trip_id` column based on the combination of `vendor_id` and `lpep_pickup_datetime`. A surrogate key is a unique identifier for each row in a table that is not derived from the data itself, but rather generated by the system. It is often used as a primary key in a data warehouse to uniquely identify each row in a table. The `dbt.safe_cast()` and `api.translate_type()` are functions used to cast the specific column to a data type, where the `dbt` and `api` are packages pre built in dbt. At the end, the `get_payment_type_description` is the macro that we defined in the `get_payment_type_description.sql`.

The `WHERE rn = 1` clause is used to filter the result set to select only the first row for each combination of `vendor_id` and `lpep_pickup_datetime`. Selecting only the first row for each combination can serve as a simple form of data sampling to analyze a representative subset of the dataset without processing the entire dataset.

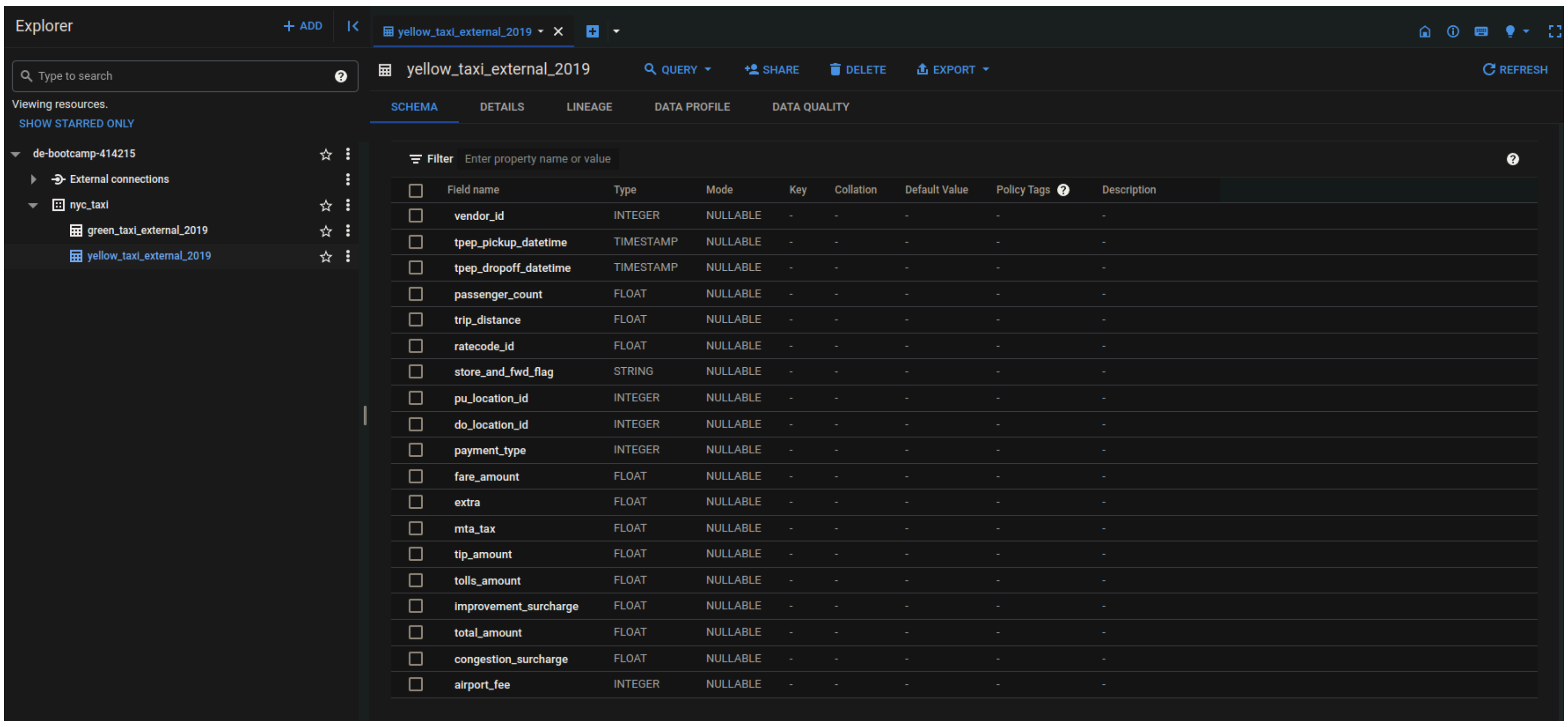
Part 4 - Conditional Statement

The last piece of code is a conditional statement that limits the number of rows returned by the query when the `is_test_run` variable is set to `true`.

```
{% if var('is_test_run', default=true) %}

    LIMIT 100

  {% endif %}
```

This conditional statement is used to limit the number of rows returned by the query when running dbt in test mode, which can be useful for testing and debugging purposes. The `default=true` argument specifies that the `is_test_run` variable defaults to `true` if it is not explicitly set when running dbt. **This can me deleted in the final version of the code.**

Following the same logic, we can do the same for the `stg_yellow_tripdata.sql` file. The schema in Bigquery should be as follows:



The content of the `stg_yellow_tripdata.sql` file would be:

stg_staging_yellow_tripdata.sql

```
{{ config(materialized='view') }}

  WITH tripdata AS
  (
    SELECT *,
      row_number() OVER(PARTITION BY vendor_id, tpep_pickup_datetime) AS rn
    FROM {{ source('staging','yellow_taxi_external_2019') }}
    WHERE vendor_id IS NOT NULL
  )
  SELECT
    -- identifiers
      {{ dbt_utils.generate_surrogate_key(['vendor_id', 'tpep_pickup_datetime']) }} AS trip_id,
      {{ dbt.safe_cast("vendor_id", api.Column.translate_type("integer")) }} AS vendor_id,
      {{ dbt.safe_cast("ratecode_id", api.Column.translate_type("integer")) }} AS ratecode_id,
      {{ dbt.safe_cast("pu_location_id", api.Column.translate_type("integer")) }} AS
pickup_location_id,
      {{ dbt.safe_cast("do_location_id", api.Column.translate_type("integer")) }} AS
dropoff_location_id,

      -- timestamps
      CAST(tpep_pickup_datetime AS timestamp) AS pickup_datetime,
      CAST(tpep_dropoff_datetime AS timestamp) AS dropoff_datetime,

      -- trip info
      store_and_fwd_flag,
      {{ dbt.safe_cast("passenger_count", api.Column.translate_type("integer")) }} AS passenger_count,
      CAST(trip_distance AS numeric) AS trip_distance,
      -- yellow cabs are always street-hail
      1 AS trip_type,

      -- payment info
```

```sql
      CAST(fare_amount AS numeric) AS fare_amount,
      CAST(extra AS numeric) AS extra,
      CAST(mta_tax AS numeric) AS mta_tax,
      CAST(tip_amount AS numeric) AS tip_amount,
      CAST(tolls_amount AS numeric) AS tolls_amount,
      CAST(improvement_surcharge AS numeric) AS improvement_surcharge,
      CAST(total_amount AS numeric) AS total_amount,
      COALESCE({{ dbt.safe_cast("payment_type", api.Column.translate_type("integer")) }},0) AS
payment_type,
      {{ get_payment_type_description('payment_type') }} AS payment_type_description
  FROM tripdata
  WHERE rn = 1

  -- dbt build --select <model.sql> --vars '{'is_test_run: false}'
  {% if var('is_test_run', default=true) %}

    LIMIT 100

  {% endif %}
```

3.4 Core Models

The Core folder within the Models directory of a dbt project is intended for storing models that perform more complex transformations on data that has already been pre-processed in the Staging models. These Core models typically include:

- **Fact Tables**: These are the tables that contain the measures and metrics that will be analyze. Fact tables usually result from joining various staging models together and aggregating data to a level suitable for analysis.
- **Dimension Tables**: Dimension tables are used to store descriptive attributes or dimensions through which fact data can be analyzed. They provide context to the numerical metrics stored in fact tables. Each dimension table contains a set of attributes (columns) that describe aspects of the business process represented in a fact table

For example, a fact table might record a sale with a numeric sale amount and foreign keys linking to dimension tables; the dimension tables would then describe the product sold, the customer who bought it, and the date of the sale.

Let's create inside `dags/dbt/taxi_rides_ny/model/` the `core` directory the `fact_trips.sql` and `dim_zones.sql` files. The directory structure would be:

```
astro-airflow/
    ├── .astro/
    ├── dags/
    │   ├── dbt/
    │   │   ├── logs
    │   │   ├── taxi_rides_ny/
    │   │   │   ├── analyses/
    │   │   │   ├── dbt_packages/
    │   │   │   ├── models/
    │   │   │   │    ├── staging/
    │   │   │   │    ├── core/
    │   │   │   │    │    ├── `fact_trips.sql`
    │   │   │   │    │    ├── `dim_zones.sql`
    │   │   │   ├── macros/
    │   │   │   ├── seeds/
    │   │   │   │    ├── `taxi_zone_lookup.csv`
    │   │   │   ├── snapshots/
    │   │   │   ├── tests/
    │   │   │   ├── dbt_project.yml
    │   │   │   ├── packages.yml
    │   │   │   ├── packages.lock.yml
```

For the `taxi_zone_lookup.csv` table, we can create the `dim_zones.sql` file. The content of the `dim_zones.sql` file would be:

dim_zones.sql

```sql
{{ config(materialized='table') }}
SELECT
    location_id,
    borough,
    zone,
    REPLACE(service_zone,'Boro','Green') AS service_zone
FROM {{ ref('taxi_zone_lookup') }}
```

The `{{ ref('taxi_zone_lookup') }}` function is used to reference the `taxi_zone_lookup.csv` table inside the `seeds` directory. Remember that the `taxi_zone_lookup` seed data contains information about the taxi zones in New York City, including the location ID, borough, zone, and service zone. The `SELECT` statement is used to select the location ID, borough, zone, and service zone from the `taxi_zone_lookup` seed data, and the `REPLACE()` function is used to replace the word 'Boro' with 'Green' in the service zone column. The Green taxis, officially known as Boro Taxis, were introduced to provide street hail service to areas outside of the central districts served by yellow taxis, this is why we are replacing the word 'Boro' with 'Green' in the service zone column.

For the `fact_trips.sql` file, the content would be:

fact_trips.sql

```
{{ config(materialized='table') }}

    WITH green_tripdata AS (
        SELECT *,
            'Green' AS service_type
        FROM {{ ref('stg_green_tripdata') }}
    ),
    yellow_tripdata AS (
        SELECT *,
            'Yellow' AS service_type
        FROM {{ ref('stg_yellow_tripdata') }}
    ),
    trips_unioned AS (
        SELECT * FROM green_tripdata
        UNION ALL
        SELECT * FROM yellow_tripdata
    ),
    dim_zones AS (
        SELECT * FROM {{ ref('dim_zones') }}
        WHERE borough != 'Unknown'
    )
    SELECT
        trips_unioned.trip_id,
        trips_unioned.vendor_id,
        trips_unioned.service_type,
        trips_unioned.ratecode_id,
        trips_unioned.pickup_location_id,
        pickup_zone.borough AS pickup_borough,
        pickup_zone.zone AS pickup_zone,
        trips_unioned.dropoff_location_id,
        dropoff_zone.borough AS dropoff_borough,
        dropoff_zone.zone AS dropoff_zone,
        trips_unioned.pickup_datetime,
        trips_unioned.dropoff_datetime,
        trips_unioned.store_and_fwd_flag,
        trips_unioned.passenger_count,
        trips_unioned.trip_distance,
        trips_unioned.trip_type,
        trips_unioned.fare_amount,
        trips_unioned.extra,
        trips_unioned.mta_tax,
        trips_unioned.tip_amount,
        trips_unioned.tolls_amount,
        trips_unioned.improvement_surcharge,
        trips_unioned.total_amount,
        trips_unioned.payment_type,
        trips_unioned.payment_type_description
    FROM trips_unioned
    INNER JOIN dim_zones AS pickup_zone
    ON trips_unioned.pickup_location_id = pickup_zone.location_id
    INNER JOIN dim_zones AS dropoff_zone
    ON trips_unioned.dropoff_location_id = dropoff_zone.location_id
```

Let's better understand this code dissecting it into parts:

Part 1 - Configuration

The jinja macro now is used to materialize this model as a physical table in the database.

```
{{ config(materialized='table') }}
```

Part 2 - Common Table Expression

This second part of the code create a CTE table for the `green_tripdata` and `yellow_tripdata` tables, and then union them together to create the `trip_unioned` table. At the last part of the code we create a CTE table for the `dim_zones` table for borough rows with values different from unknown.

```
WITH green_tripdata AS (
        SELECT *,
            'Green' AS service_type
        FROM {{ ref('stg_green_tripdata') }}
    ),
    yellow_tripdata AS (
        SELECT *,
            'Yellow' AS service_type
        FROM {{ ref('stg_yellow_tripdata') }}
    ),
    trips_unioned AS (
        SELECT * FROM green_tripdata
        UNION ALL
        SELECT * FROM yellow_tripdata
    ),
    dim_zones AS (
        SELECT * FROM {{ ref('dim_zones') }}
        WHERE borough != 'Unknown'
    )
```

Part 3 - Main Query

This part it selects various fields from the trips_unioned CTE, which contains combined data from both green and yellow taxi trips. The `dim_zones` table is joined with INNER JOIN to ensures that only records with matching location IDs in both `trips_unioned` and `dim_zones` are selected. This means the query will only return trip records that have a known pickup and dropoff location within the zones defined in the `dim_zones` table.

```
SELECT
        trips_unioned.trip_id,
        trips_unioned.vendor_id,
        trips_unioned.service_type,
        trips_unioned.ratecode_id,
        trips_unioned.pickup_location_id,
        pickup_zone.borough AS pickup_borough,
        pickup_zone.zone AS pickup_zone,
        trips_unioned.dropoff_location_id,
        dropoff_zone.borough AS dropoff_borough,
        dropoff_zone.zone AS dropoff_zone,
        trips_unioned.pickup_datetime,
        trips_unioned.dropoff_datetime,
        trips_unioned.store_and_fwd_flag,
        trips_unioned.passenger_count,
        trips_unioned.trip_distance,
        trips_unioned.trip_type,
        trips_unioned.fare_amount,
        trips_unioned.extra,
        trips_unioned.mta_tax,
        trips_unioned.tip_amount,
        trips_unioned.tolls_amount,
        trips_unioned.improvement_surcharge,
        trips_unioned.total_amount,
        trips_unioned.payment_type,
        trips_unioned.payment_type_description
    FROM trips_unioned
    INNER JOIN dim_zones AS pickup_zone
    ON trips_unioned.pickup_location_id = pickup_zone.location_id
    INNER JOIN dim_zones AS dropoff_zone
    ON trips_unioned.dropoff_location_id = dropoff_zone.location_id
```

Finally, for the last part of our dbt model, we can create the `dim_monthly_zone_revenue.sql` file. The content of the `dim_monthly_zone_revenue.sql` file would be:

dim_monthly_zone_revenue.sql

```
{{ config(materialized='table') }}

    WITH trips_data AS (
        SELECT * FROM {{ ref('fact_trips') }}
    )
        SELECT
            -- Reveneue grouping
            pickup_zone AS revenue_zone,
            {{ dbt.date_trunc("month", "pickup_datetime") }} AS revenue_month,
            service_type,
            -- Revenue calculation
            SUM(fare_amount) AS revenue_monthly_fare,
```

```
        SUM(extra) AS revenue_monthly_extra,
        SUM(mta_tax) AS revenue_monthly_mta_tax,
        SUM(tip_amount) AS revenue_monthly_tip_amount,
        SUM(tolls_amount) AS revenue_monthly_tolls_amount,
        SUM(improvement_surcharge) AS revenue_monthly_improvement_surcharge,
        SUM(total_amount) AS revenue_monthly_total_amount,

        -- Additional calculations
        COUNT(trip_id) AS total_monthly_trips,
        AVG(passenger_count) AS avg_monthly_passenger_count,
        AVG(trip_distance) AS avg_monthly_trip_distance
    FROM trips_data
    GROUP BY 1,2,3
```

The `{{ dbt.date_trunc("month", "pickup_datetime") }}` truncate the `pickup_datetime` column to the first day of each month, effectively grouping data by month, and to refer to this truncated date as revenue_month in the output of the query. The use of this macro is a cross database macro to abstract the underlying SQL flavour and provide a consistent interface for date truncation across different databases.

The `SUM()` function is used to calculate the total revenue for each revenue category, and the `AVG()` function is used to calculate the average passenger count and trip distance for each month. The `COUNT()` function is used to calculate the total number of trips for each month. The `GROUP BY 1, 2, 3` is shorthand syntax in SQL that references the columns selected in the SELECT clause to group the revenue data by `revenue_zone`, `revenue_month`, and `service_type`.

The final structure of files should be the following:

```
astro-airflow/
    ├── .astro/
    ├── dags/
    │   ├── dbt/
    │   │   ├── logs
    │   │   ├── taxi_rides_ny/
    │   │   │   ├── analyses/
    │   │   │   ├── dbt_packages/
    │   │   │   ├── models/
    │   │   │   │    ├── staging/
    │   │   │   │    ├── core/
    │   │   │   │    │    ├── `fact_trips.sql`
    │   │   │   │    │    ├── `dim_zones.sql`
    │   │   │   │    │    ├── `dim_monthly_zone_revenue.sql`
    │   │   │   ├── macros/
    │   │   │   ├── seeds/
    │   │   │   │    ├── `taxi_zone_lookup.csv`
    │   │   │   ├── snapshots/
    │   │   │   ├── tests/
    │   │   │   ├── dbt_project.yml
    │   │   │   ├── packages.yml
    │   │   │   ├── packages.lock.yml
```

3.5 Running dbt models in Airflow

To run the dbt models in Airflow, we need to increment our `elt_nyc_taxi_bq.py` file with the necessary dags to run the dbt models. This allows for the orchestration of dbt runs as part of an automated workflow, enabling data transformation processes to be scheduled, monitored, and managed within the Airflow environment. To run all the dbt models that was created, we use the operators from cosmos library, like `DbtTaskGroup` to create a DAG, and the operators `ProfileConfig` and `ProfileConfig` to configure the dbt profile and the dbt project. We add at the `elt_nyc_taxi_bq.py` the following code:

```python
from cosmos import DbtTaskGroup, ProjectConfig, ProfileConfig
    from cosmos.profiles import GoogleCloudServiceAccountFileProfileMapping

    CONNECTION_ID = "gcp_conn"
    SCHEMA_NAME = "nyc_taxi"
    # Path to the project inside dbt folder
    DBT_ROOT_PATH = '/usr/local/airflow/dags/dbt/taxi_rides_ny'
    # Name to create a temporary profile for when running dbt with DAG
    PROFILE_NAME = "bigquery-bq"


    profile_config = ProfileConfig(
        profile_name = PROFILE_NAME,
        target_name="dev",
        profile_mapping = GoogleCloudServiceAccountFileProfileMapping(
            conn_id = CONNECTION_ID,
            profile_args = {
                #"keyfile": KEYFILE_ROOT,
                "project": "de-bootcamp-414215",
```

```
                "dataset": SCHEMA_NAME
            }

        )
    )

    project_config = ProjectConfig(DBT_ROOT_PATH)

    dbt_workflow = DbtTaskGroup(
        group_id = 'dbt_workflow',
        project_config = project_config,
        profile_config = profile_config,

    )
```

The full code for the pipeline to extract the data from the NYC taxi, create a bucket and load/ingest the data into the Google Cloud Storage, create a dataset in BigQuery, create external tables in BigQuery, and finally run the dbt models to make the necessary transformations would be:

elt_nyc_taxi_bq.py

```
# [START import modules]---------------------------------------------------------------------------
-----------
from airflow import DAG
from datetime import datetime
from google.cloud import storage
import pandas as pd
import re
import pyarrow.parquet as pq
import pyarrow as pa
from os import getenv

from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
from airflow.providers.google.cloud.operators.gcs import GCSCreateBucketOperator
from airflow.providers.google.cloud.operators.bigquery import BigQueryCreateExternalTableOperator,\
    BigQueryCreateEmptyDatasetOperator
from cosmos import DbtTaskGroup, ProjectConfig, ProfileConfig
from cosmos.profiles import GoogleCloudServiceAccountFileProfileMapping
# [END import modules]

# [START Env Variables]----------------------------------------------------------------------------
----------
##From https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page##
URL_PREFIX_1 = 'https://d37ci6vzurychx.cloudfront.net/trip-data'
URL_PREFIX_2 = 'https://d37ci6vzurychx.cloudfront.net/trip-data'
FILE_NAME_1 = 'green_tripdata_{{ execution_date.strftime(\'%Y-%m\') }}.parquet'
FILE_NAME_2 = 'yellow_tripdata_{{ execution_date.strftime(\'%Y-%m\') }}.parquet'

URL_1 = f'{URL_PREFIX_1}/{FILE_NAME_1}'
URL_2 = f'{URL_PREFIX_2}/{FILE_NAME_2}'

AIRFLOW_HOME = getenv("AIRFLOW_HOME", "/usr/local/airflow")
FILE_PATH_1 = getenv('FILE_PATH_1', f'{AIRFLOW_HOME}/{FILE_NAME_1}')
FILE_PATH_2 = getenv('FILE_PATH_2', f'{AIRFLOW_HOME}/{FILE_NAME_2}')

DATASET_NAME= getenv("DATASET_NAME", 'nyc_taxi')
TABLE_NAME = 'green_taxi_{{ execution_date.strftime(\'%Y_%m\') }}'
YEAR = 2019
TABLE_ID_1 = f"green_taxi_external_{YEAR}"
TABLE_ID_2 = f"yellow_taxi_external_{YEAR}"

PROJECT_ID = getenv("PROJECT_ID", "de-bootcamp-414215")
REGION = getenv("REGIONAL", "us-east1")
LOCATION = getenv("LOCATION", "us-east1")

BUCKET_NAME = getenv("BUCKET_NAME", 'nyc-taxi-data-414215')
GCS_BUCKET_FOLDER= getenv("GCS_BUCKET", f'nyc_taxi_trip_{YEAR}')
CONNECTION_ID = getenv("CONNECTION_ID", "gcp_conn")
SCHEMA_NAME = "nyc_taxi"
DBT_ROOT_PATH = '/usr/local/airflow/dags/dbt/taxi_rides_ny'
PROFILE_NAME = "bigquery-db"
# [END Env Variables]

# [START default args]-----------------------------------------------------------------------------
---------
default_args = {
    "owner": "marcos benicio",
```

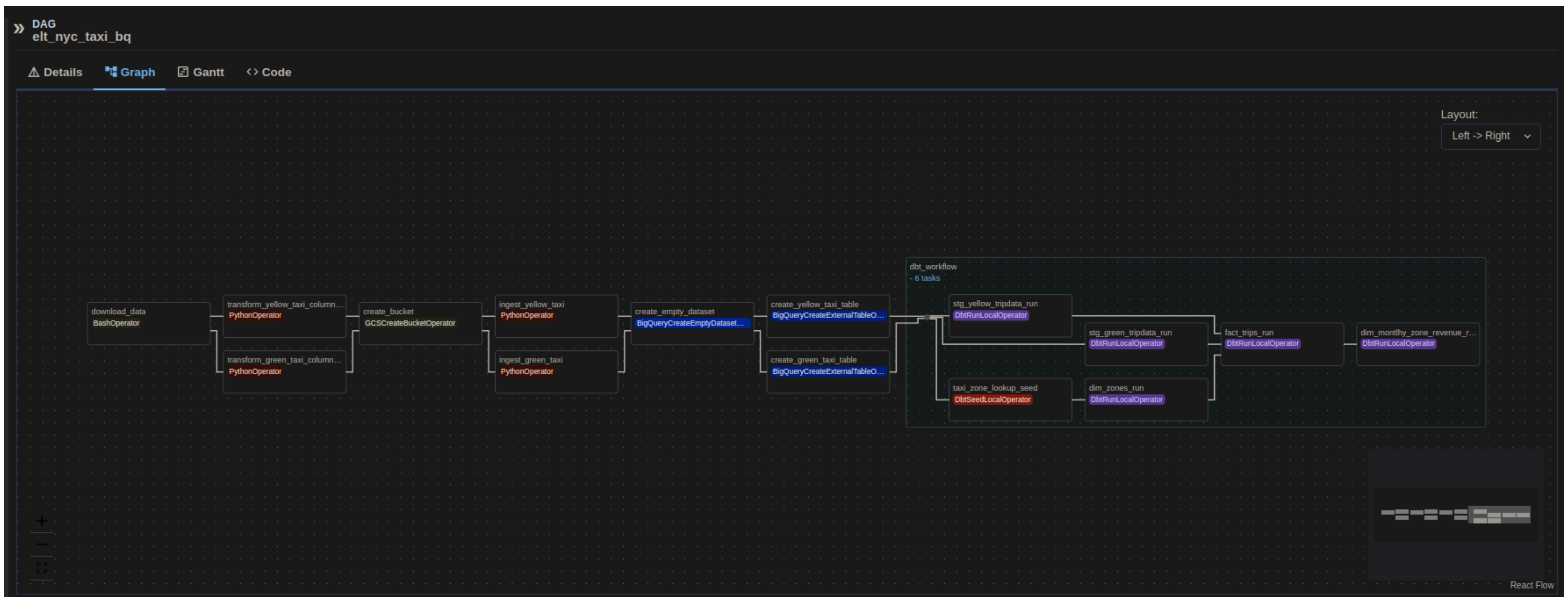
```python
    "email": ['marcosbenicio@id.uff.br'],
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1
}
# [END default args]

# [START Python Functions]------------------------------------------------------------------------------
-------------
def transform_columns_to_snake(file_path):

    # Load the parquet file metadata (schema) without reading the data
    original_table = pq.read_table(file_path)
    original_column_names = original_table.schema.names
    original_metadata = original_table.schema.metadata

    # convert all camel columns names to snake
    def camel_to_snake(name):
        return re.sub(r'(?<=[a-z0-9])([A-Z])|(?<=[A-Z])([A-Z])(?=[a-z])', r'_\g<0>', name).lower()

    new_column_names = [camel_to_snake(name) for name in original_column_names]
    fields = [pa.field(new_name, original_table.schema.field(original_name).type)
            for new_name, original_name in zip(new_column_names, original_column_names)]

    new_schema = pa.schema(fields, metadata=original_metadata)

    new_table = pa.Table.from_arrays(original_table.columns, schema=new_schema)

    pq.write_table(new_table, file_path)

# Takes 20 mins, at an upload speed of 800kbps. Faster if your internet has a better upload speed
def filesystem_to_gcs(bucket, dst, src):
    """
    Uploads a file from the local filesystem to Google Cloud Storage.

    :param bucket: Name of the GCS bucket.
    :param dst: Destination path & file-name within the GCS bucket.
    :param src: Source path path & file-name on the local filesystem.
    """
    # Adjust the maximum multipart upload size and chunk
    # to prevent timeout for files > 6 MB on 800 kbps upload speed.
    storage.blob._MAX_MULTIPART_SIZE = 5 * 1024 * 1024  # 5 MB
    storage.blob._DEFAULT_CHUNKSIZE = 5 * 1024 * 1024  # 5 MB

    # Initialize the GCS client and get the bucket
    client = storage.Client()
    bucket = client.bucket(bucket)

    # Create a blob object and upload the file
    blob = bucket.blob(dst)
    blob.upload_from_filename(src)

    print(f"File {src} uploaded to {dst} in bucket {bucket}.")
# [END Python Functions]

# [START DAG Object]------------------------------------------------------------------------------------
-------
workflow = DAG(
                dag_id="elt_nyc_taxi_bq",
                default_args = default_args,
                description="""A DAG to export data from NYC taxi web,
                load the taxi trip data into GCS to create a BigQuery external table
                and transform the data with Dbt""",
                tags=['gcs', 'bigquery','data_elt', 'dbt', 'nyc_taxi'],
                schedule_interval="0 6 28 * *",
                start_date = datetime(YEAR, 1, 1),
                end_date = datetime(YEAR, 12, 30),
                )
# [END DAG Object]

# [START Workflow]--------------------------------------------------------------------------------------
-----
### Configure the profile and project to use with DbtTaskGroup
profile_config = ProfileConfig(
    profile_name = PROFILE_NAME,
    target_name="dev",
    profile_mapping = GoogleCloudServiceAccountFileProfileMapping(
        conn_id = CONNECTION_ID,
        profile_args = {
```

```
            "project": "de-bootcamp-414215",
            "dataset": SCHEMA_NAME
        }

    )
)
project_config = ProjectConfig(DBT_ROOT_PATH)
###

# Start the workflow
with workflow:

    download_data = BashOperator(
        task_id="download_data",
        bash_command =  f"""
                        curl -sSLo {FILE_PATH_1} {URL_1} && \\
                        curl -sSLo {FILE_PATH_2} {URL_2}
                        """
    )
    transform_green_taxi_columns_to_snake = PythonOperator(
        task_id='transform_green_taxi_columns_to_snake',
        python_callable=transform_columns_to_snake,
        op_kwargs={'file_path': FILE_PATH_1},
    )
    transform_yellow_taxi_columns_to_snake = PythonOperator(
        task_id='transform_yellow_taxi_columns_to_snake',
        python_callable=transform_columns_to_snake,
        op_kwargs={'file_path': FILE_PATH_2},
    )
    create_bucket = GCSCreateBucketOperator(
        task_id="create_bucket",
        bucket_name=BUCKET_NAME,
        storage_class="REGIONAL",
        location=LOCATION,
        project_id=PROJECT_ID,
        labels={"env": "dev", "team": "airflow"},
        gcp_conn_id= CONNECTION_ID
    )
    ingest_green_taxi_gcs = PythonOperator(
        task_id="ingest_green_taxi",
        python_callable=filesystem_to_gcs,
        op_kwargs={ "bucket": BUCKET_NAME,
                    "dst": f"{GCS_BUCKET_FOLDER}/{FILE_NAME_1}",
                    "src": FILE_PATH_1
                    }
    )
    ingest_yellow_taxi_gcs = PythonOperator(
        task_id="ingest_yellow_taxi",
        python_callable=filesystem_to_gcs,
        op_kwargs={ "bucket": BUCKET_NAME,
                    "dst": f"{GCS_BUCKET_FOLDER}/{FILE_NAME_2}",
                    "src": FILE_PATH_2
                    }
    )
    create_empty_dataset = BigQueryCreateEmptyDatasetOperator(
        task_id="create_empty_dataset",
        dataset_id=DATASET_NAME,
        project_id=PROJECT_ID,
        location=LOCATION,
        gcp_conn_id=CONNECTION_ID
    )
    bigquery_green_taxi_table = BigQueryCreateExternalTableOperator(
        task_id="create_green_taxi_table",
        table_resource={
            'tableReference': {
                'projectId': PROJECT_ID,
                'datasetId': DATASET_NAME,
                'tableId': TABLE_ID_1,
            },
            'externalDataConfiguration': {
                'sourceFormat': 'PARQUET',
                'sourceUris': [f"gs://{BUCKET_NAME}/{GCS_BUCKET_FOLDER}/green_tripdata_*.parquet"],
                'schema_field': {'name': 'ehail_fee', 'type': 'FLOAT64', 'mode': 'NULLABLE'}
                }
        },
        gcp_conn_id=CONNECTION_ID
    )
    bigquery_yellow_taxi_table = BigQueryCreateExternalTableOperator(
        task_id="create_yellow_taxi_table",
```

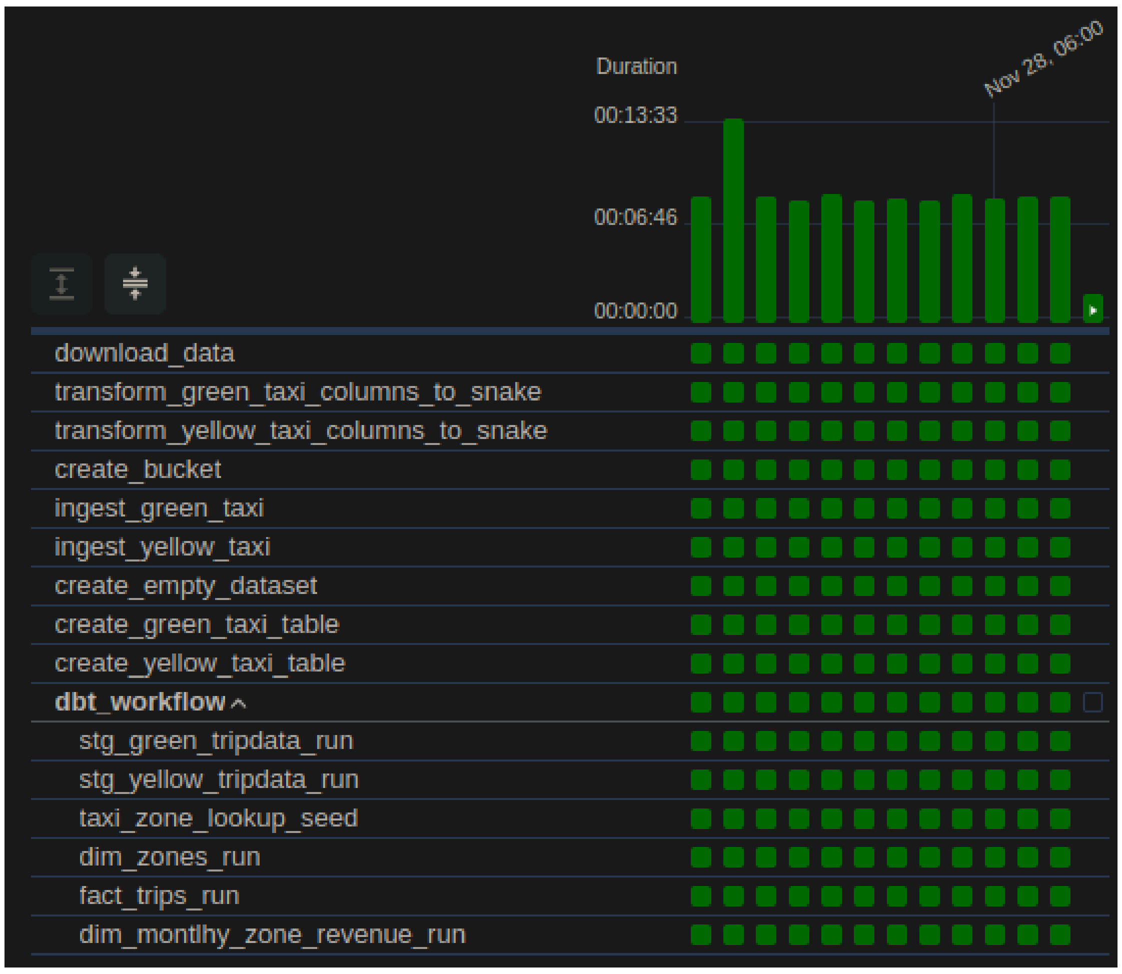
```python
            table_resource={
                'tableReference': {
                    'projectId': PROJECT_ID,
                    'datasetId': DATASET_NAME,
                    'tableId': TABLE_ID_2,
                },
                'externalDataConfiguration': {
                    'sourceFormat': 'PARQUET',
                    'sourceUris': [f"gs://{BUCKET_NAME}/{GCS_BUCKET_FOLDER}/yellow_tripdata_*.parquet"],
                }
            },
            gcp_conn_id=CONNECTION_ID
        )
        dbt_workflow = DbtTaskGroup(
            group_id = 'dbt_workflow',
            project_config = project_config,
            profile_config = profile_config,

        )


download_data >> [transform_green_taxi_columns_to_snake, transform_yellow_taxi_columns_to_snake] \
>> create_bucket >> [ingest_green_taxi_gcs, ingest_yellow_taxi_gcs ] \
>> create_empty_dataset >> [bigquery_yellow_taxi_table, bigquery_green_taxi_table] \
>> dbt_workflow
# [END Workflow]
```

Remember to set the connection in the airflow UI, and because airflow manage to create a temporary `profile.yml` to map the connection in airflow onto the dbt profile, we need to set the `gcp_conn` connection in the airflow UI using the Keyfile Path instead of the directly passing the key to Keyfile JSON. The `gcp_conn` connection is used to authenticate with Google Cloud Platform (GCP) services, and it is used to create the BigQuery dataset and external tables, as well as to upload the taxi trip data to Google Cloud Storage.

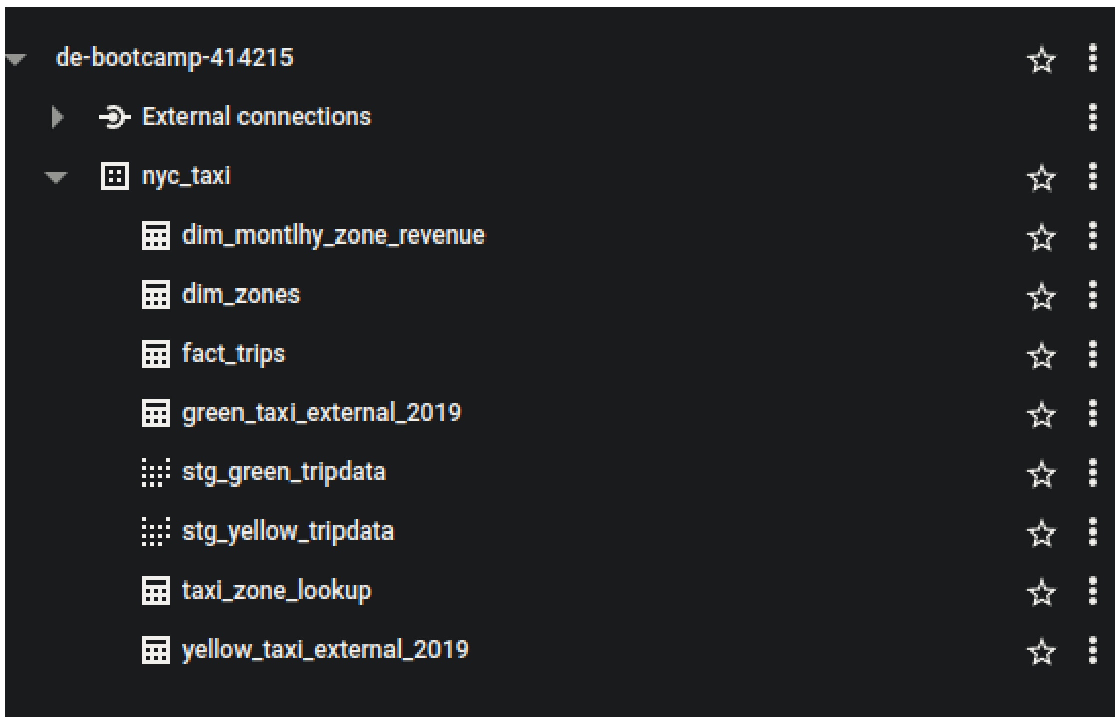
To run the dbt models in Airflow, now we can go to the UI at `http://localhost:8080` and trigger the `elt_nyc_taxi_bq` DAG. We should see the following DAG in the Airflow UI:



Running this dag should complete the entire pipeline, showing a green success message in the Airflow UI.



We can also check if all tables was created in BigQuery:



3.6. Testing and Documenting

Test are assumptions that we make about our data, and they are used to ensure that the data is accurate, consistent, and reliable. We can add tests and descriptions for each column inside the `schema.yml` file. To easily generate the content of the `schema.yml` file for the `staging` and `core` directory we can use the package `dbt-labs/codegen` that was installed.

We can call a macro operation in the bash to generate the `schema.yml` file for the models in the `staging` and `core` directories. Go to the terminal, and inside the directory `dags/dbt/taxi_rides_ny` run the following command:

```
OUTPUT_STG=$(dbt run-operation generate_model_yaml --args '{"model_names": ["stg_green_tripdata",
"stg_yellow_tripdata"]}')
    echo "$OUTPUT_STG" >> models/staging/schema.yml
```

and for the Models inside the core directory:

```
OUTPUT_CORE=$(dbt run-operation generate_model_yaml --args '{"model_names": ["fact_trips",
"dim_zones", "dim_montlhy_zone_revenue"]}')
    echo "$OUTPUT_CORE" >> models/core/schema.yml
```

This will automatically append the output from the macro at the end of each `schema.yml` file. After running the command, check the content of the `schema.yml` file for the `staging` and `core` directory, and format if necessary. We should have the following content for the `schema.yml` files:

staging/schema.yml

```
version: 2

sources:
  - name: staging                          # directory name in dbt/models
```

```yaml
    database: de-bootcamp-414215           # dataset name in BigQuery
    schema: nyc_taxi                       # schema name in BigQuery where the table is located
    tables:                                # list of tables in the schema
      - name: "green_taxi_external_2019"
      - name: "yellow_taxi_external_2019"

models:
  - name: stg_green_tripdata
    description: ""
    columns:
      - name: trip_id
        data_type: string
        description: ""

      - name: vendor_id
        data_type: int64
        description: ""
    .
    .
    .

    - name: stg_yellow_tripdata
        description: ""
        columns:
        - name: trip_id
            data_type: string
            description: ""

        - name: vendor_id
            data_type: int64
            description: ""
    .
    .
    .
```

---

core/schema.yml

```yaml
version: 2

models:
  - name: fact_trips
    description: ""
    columns:
      - name: trip_id
        data_type: string
        description: ""
  .
  .
  .
  - name: dim_zones
    description: ""
    columns:
      - name: location_id
        data_type: numeric
        description: ""
  .
  .
  .
  - name: dim_montlhy_zone_revenue
      description: ""
      columns:
        - name: revenue_zone
          data_type: string
          description: ""

        - name: revenue_month
          data_type: timestamp
          description: ""
  .
  .
  .
```

We can now fill the `description` and `data_type` fields for each column in the `schema.yml` file and also add tests using the `tests` field. The `tests` field is used to define the tests that should be run on the data in the model. For example, we could add a `unique` test and a `not_null` test to ensure that the `trip_id` column in the `fact_trips` model is unique and not null. The `severity` field is used to specify the severity level of the test, which can be `error`, `warn`, or `info`.

```yaml
models:
    - name: fact_trips
      description: ""
      columns:
        - name: trip_id
          data_type: string
          description: ""
          tests:
            - unique
              severity: warn
            - not_null
              severity: warn
```

4. Visualising the Data with Google Data Studio

The following table was created using Google Data Studio. With Google Data Studio, we can create interactive dashboards and reports by directly connecting to our BigQuery dataset that was created using dbt. Here we use the fact_trips table to create a report.