CLOUD ANALYTICS AND DATA WAREHOUSE IMPLEMENTATION FOR A HISTORICAL DATASET

Codes and Queries - Group Project (Team - 8)

Shashank Reddy Kandimalla (016799523), Rohith Reddy Vangala (016762109), Sakshi Manish Mukkirwar (016794765), Swati (016702413), Yamini Muthyala (016766165)

1. QUERIES

1. Age Group Wise Analysis of Customers' Interest in Vehicle Insurance:

Query :

```
SELECT
    CASE
      WHEN Age BETWEEN 20 AND 30 THEN '20-30'
      WHEN Age BETWEEN 31 AND 40 THEN '31-40'
      WHEN Age BETWEEN 41 AND 50 THEN '41-50'
      WHEN Age BETWEEN 51 AND 60 THEN '51-60'
      ELSE '60+'
    END AS Age_Group,
    COUNT(*) AS Total_Customers,
    SUM(Response) AS Interested_In_Vehicle_Insurance
FROM "dbms".Customers
GROUP BY Age_Group order by Interested_In_Vehicle_Insurance desc ;
```

Interaction with database: The query selects data from the "Customers" table in the "dbms" database and groups the customers by age range, calculated using a CASE statement. It then counts the total number of customers in each age range and calculates the sum of their responses indicating interest in vehicle insurance. The data is then sorted by the count of interested customers in descending order.

age_group	total_customers	interested_in_vehicle_i…
41-50	101475	16025
31-40	73498	11597
51-60	60288	7716
20-30	215338	7183
60+	57547	4189

Fig-1 Output for Age Group Wise Analysis of Customers' Interest in Vehicle Insurance

2. Number of Customers Interested in Vehicle Insurance by Region Code:

Query:

```
SELECT Region_Code,
    COUNT(*) AS Total_Customers,
```

```
    SUM(Response) AS Interested_In_Vehicle_Insurance
FROM "dbms".Customers
GROUP BY Region_Code order by Interested_In_Vehicle_Insurance desc;
```

Interaction with database: The query is executed on the "dbms" database and selects data from the "Customers" table. It groups the customers by region code and calculates the count and sum of their responses. The query then sorts the data in descending order based on the count of interested customers.

region_code	total_customers	interested_in_vehicle_i...
28	141937	19917
8	44900	3257
41	24400	2224
46	26357	2032
29	14843	1365
3	12349	1181
11	12328	1041
15	17750	958

Fig-2 Output for number of customers interested in vehicle insurance by region code

3. Total number of previously insured customers interested in vehicle insurance:

Query:

```
SELECT COUNT(*) AS Total_Customers,
    SUM(Response) AS Interested_In_Vehicle_Insurance
FROM "dbms".Customers
WHERE Previously_Insured = 1;
```

Interaction with database: The query will be sent to the "dbms" database and executed. The database will scan the "Customers" table to identify the customers who have previously purchased insurance and match the criteria. It will then count the total number of such customers and calculate the sum of their responses indicating interest in vehicle insurance.

total_customers	interested_in_vehicle_i...
233070	158

Fig-3 Output for Total number of previously insured customers interested in vehicle insurance.

4. Count of Customers and Their Interest in Vehicle Insurance by Vehicle Age:

Query:

```
SELECT Vehicle_Age,
    COUNT(*) AS Total_Customers,
    SUM(Response) AS Interested_In_Vehicle_Insurance
FROM "dbms".Customers
INNER JOIN "dbms".Vehicles ON Customers.id = Vehicles.id
GROUP BY Vehicle_Age;
```

Interaction with database: The query fetches the required data by joining the "Customers" and "Vehicles" tables in the "dbms" database using the "INNER JOIN" clause. It groups the data based on the "Vehicle_Age" column and calculates the count of customers and their interest in vehicle insurance for each age group using the "COUNT" and "SUM" aggregate functions, respectively. The query returns the result set showing the count of customers and their interest in vehicle insurance for each vehicle age group.

vehicle_age	total_customers	interested_in_vehicle_i...
2	21326	4702
0	219805	7202
1	267015	34806

Fig-4 Output for count of customers and their interest in vehicle insurance by vehicle age..

5. Grouping customers by premium group and calculating the count of total customers and sum of responses interested in vehicle insurance:

Query:

```
SELECT CASE
        WHEN p.Annual_Premium < 50000 THEN 'Less than 50k'
        WHEN p.Annual_Premium BETWEEN 50000 AND 100000 THEN '50k-100k'
        ELSE 'More than 100k'
      END AS Premium_Group,
      COUNT(*) AS Total_Customers,
      SUM(c.Response) AS Interested_In_Vehicle_Insurance
FROM "dbms".Policies
p inner join "dbms".customers c on p.id=c.id
GROUP BY Premium_Group;
```

Interaction with database: The above SQL query uses a JOIN statement to combine data from the "Policies" and "Customers" tables in the "dbms" database. It calculates the premium group based on the value of the Annual_Premium column in the Policies table, and groups the customers by the premium group. It then counts the total number of customers in each premium group and calculates the sum of their responses indicating interest in vehicle insurance. The result is sorted by the Premium_Group column.

total_customers	interested_in_vehicle_i...	premium_group
465146	41811	Less than 50k
1049	123	More than 100k
41951	4776	50k-100k

Fig-5 Output for Grouping customers by premium group and calculating the count of total customers and sum of responses interested in vehicle insurance.

6. Identifying customers who have a high policy sales channel value and target them with vehicle insurance offers:

Query:

```
SELECT p.policy_sales_channel,COUNT(*) AS Total_Customers
```

```
,SUM(Response) AS Interested_In_Vehicle_Insurance
FROM "dbms".Policies p inner join "dbms".customers c on p.id=c.id
GROUP BY p.policy_sales_channel order by Interested_In_Vehicle_Insurance desc ;
```

Interaction with database: The query interacts with the Policies and Customers tables in the dbms database using an inner join to combine the data based on matching id values. The resulting data is grouped by policy_sales_channel, and the query counts the total number of customers for each group and calculates the sum of their responses indicating interest in vehicle insurance. The resulting data is sorted in descending order based on the count of interested customers.

total_customers	interested_in_vehicle_i...	policy_sales_channel
106594	15891	26
98299	13996	124
179523	3858	152
14313	2297	156
8958	1794	157
13239	1720	122
7988	1474	154
3850	880	163

Fig-6 Output for Identifying customers who have a high policy sales channel value and target them with vehicle insurance offers.

7. Total Customers and Interested Customers in Vehicle Insurance by Vintage:

Query:

```
SELECT Vintage,
    COUNT(*) AS Total_Customers,
    SUM(Response) AS Interested_In_Vehicle_Insurance
FROM "dbms".Policies p inner join "dbms".customers c on p.id=c.id
GROUP BY p.Vintage order by Interested_In_Vehicle_Insurance  desc;
```

Interaction with database: The query runs a join between the "Policies" and "Customers" tables in the "dbms" database using the customer ID as the join key. It groups the data by vintage, counts the total number of customers in each group, and calculates the sum of their responses indicating interest in vehicle insurance. Finally, the data is sorted in descending order based on the count of interested customers.

total_customers	interested_in_vehicle_i...	vintage
1828	192	84
1817	191	11
1795	190	189
1773	190	34
1821	190	282
1823	189	165
1697	186	220
1789	186	298

Fig-7 Output for Total Customers and Interested Customers in Vehicle Insurance by Vintage.

8. Analysis of Customers' Interest in Vehicle Insurance based on Vehicle Damage:

Query:

```
SELECT Vehicle_Damage,
    COUNT(*) AS Total_Customers,
    SUM(Response) AS Interested_In_Vehicle_Insurance
FROM "dbms".Customers
INNER JOIN "dbms".Vehicles ON Customers.id = Vehicles.id
GROUP BY Vehicle_Damage;
```

Interaction with database: The query performs an inner join on the "Customers" and "Vehicles" tables using the "id" column. It then groups the data by the "Vehicle_Damage" column from the "Vehicles" table, and calculates the count of customers and the sum of their responses for each group. The data is then sorted by the count of interested customers in descending order.

total_customers	interested_in_vehicle_i...	vehicle_damage
256248	45728	1
251898	982	0

Fig-8 Output for Analysis of Customers' Interest in Vehicle Insurance based on Vehicle Damage.

9. Identifying customers who have a high annual premium but have not yet purchased vehicle insurance:

Query:

```
SELECT Customers.id,
    Policies.Annual_Premium,
    Vehicles.Vehicle_Age,
    Vehicles.Vehicle_Damage
FROM "dbms".Customers
INNER JOIN "dbms".Policies ON Customers.id = Policies.id
INNER JOIN "dbms".Vehicles ON Customers.id = Vehicles.id
WHERE Customers.Response = 0
  AND Policies.Annual_Premium > 100000 ;
```

Interaction with database: The query interacts with the "Customers", "Policies", and "Vehicles" tables in the "dbms" database, performing inner joins to match records based on their ID. It then applies filters to the records to retrieve only those where the customer did not respond and the annual premium for their policy is greater than 100,000. The resulting data is then returned, including the customer ID, annual premium, vehicle age, and vehicle damage status.

id	annual_premium	vehicle_age	vehicle_damage
481	104002	2	1
568	112974	1	1
1141	139130	1	0
2229	125643	1	0
3912	117799	1	1
4798	133098	1	0
4889	103026	1	1
5422	131469	0	0

Fig-9 Output for Identifying customers who have a high annual premium but have not yet purchased vehicle insurance.

10. Retrieving the average annual premium for policies associated with vehicles that have a specified vehicle age:
Query:

```
SELECT AVG(Annual_Premium) AS Avg_Annual_Premium
FROM "dbms".Policies
WHERE id IN (
  SELECT id
  FROM "dbms".Vehicles
  WHERE Vehicle_Age IS NOT NULL
);
```

Interaction with database: In this query, we select the average of the Annual_Premium attribute from the Policies table in the "dbms" database. We then use a subquery to select the IDs of all vehicles from the Vehicles table that have a non-null value for the Vehicle_Age attribute. We use the IN keyword to filter the policies based on those IDs, and calculate the average annual premium for those policies.

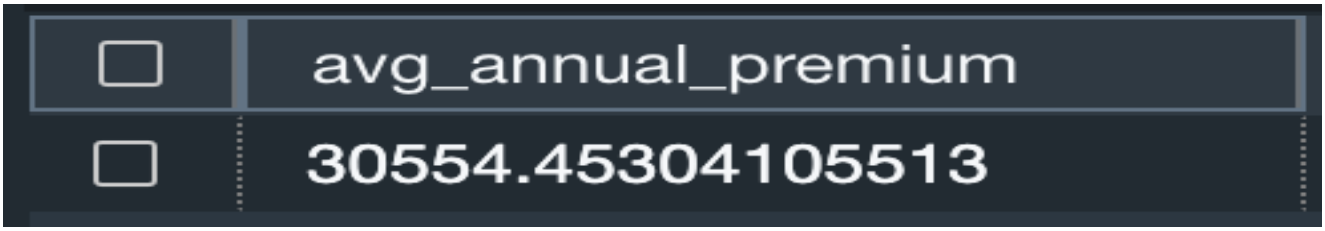


Fig-10 Output for Retrieving the average annual premium for policies associated with vehicles that have a specified vehicle age.

11. Grouping Customers by Age and Gender:

Query:

```
SELECT Gender,
      CASE
        WHEN Age BETWEEN 18 AND 25 THEN '18-25'
        WHEN Age BETWEEN 26 AND 35 THEN '26-35'
        WHEN Age BETWEEN 36 AND 45 THEN '36-45'
        WHEN Age BETWEEN 46 AND 55 THEN '46-55'
        WHEN Age BETWEEN 56 AND 65 THEN '56-65'
        ELSE '65+'
      END AS Age_Group,
      COUNT(DISTINCT id) AS Num_Customers
FROM "dbms".Customers
WHERE id IN (
  SELECT id
  FROM "dbms".Vehicles
  WHERE Vehicle_Age IS NOT NULL
)
GROUP BY Gender, Age_Group;
```

Interaction with database: In this interaction, we are selecting the gender, age group (based on the age of customers), and the number of distinct customers in each group. We are joining the Customers and Vehicles tables on id and filtering out

customers who don't have a vehicle age. We group the results by gender and age group and return the output. The output shows the number of customers in each age group for each gender.

gender	age_group	num_customers
1	36-45	55592
1	46-55	53118
0	18-25	85724
0	56-65	15863
0	46-55	28855
1	65+	21764
1	56-65	29674
0	26-35	49286

Fig-11 Output for grouping customers by age and gender.

12. Getting the average annual premium of customers who responded positively to the vehicle insurance policy and whose vehicle age is known:

Query:

```
SELECT AVG(p.Annual_Premium) AS Average_Annual_Premium
FROM "dbms".Customers c
INNER JOIN "dbms".Policies p ON c.id = p.id
INNER JOIN "dbms".Vehicles v ON c.id = v.id
WHERE c.Response = 1 AND v.Vehicle_Age IS NOT NULL;
```

Interaction with database: This SQL query interacts with the database to calculate the average annual premium for customers who responded positively to the vehicle insurance offer and have a non-null vehicle age. The query uses inner joins to combine data from the Customers, Policies, and Vehicles tables and applies filters to retrieve only the relevant data. The result is a single value representing the average annual premium.

average_annual_premi...
31604.092742453435

Fig-12 Output for the average annual premium of customers who responded positively to the vehicle insurance policy and whose vehicle age is known.

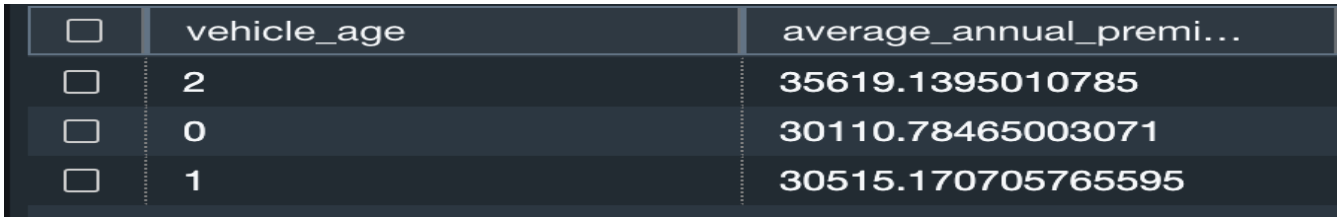
13. Calculating Average Annual Premiums by Vehicle Age:

Query:

```
SELECT Vehicle_Age, AVG(Annual_Premium) AS Average_Annual_Premium
FROM "dbms".Policies p INNER JOIN "dbms".Vehicles v ON p.id = v.id
WHERE Vehicle_Age IS NOT NULL
GROUP BY Vehicle_Age;
```

Interaction with database: The query retrieves data from two tables ("Policies" and "Vehicles") in the "dbms" database, and

performs an inner join to match records based on their ID. It then filters records where the vehicle age is not null and groups the results by vehicle age to calculate the average annual premium for each age group.



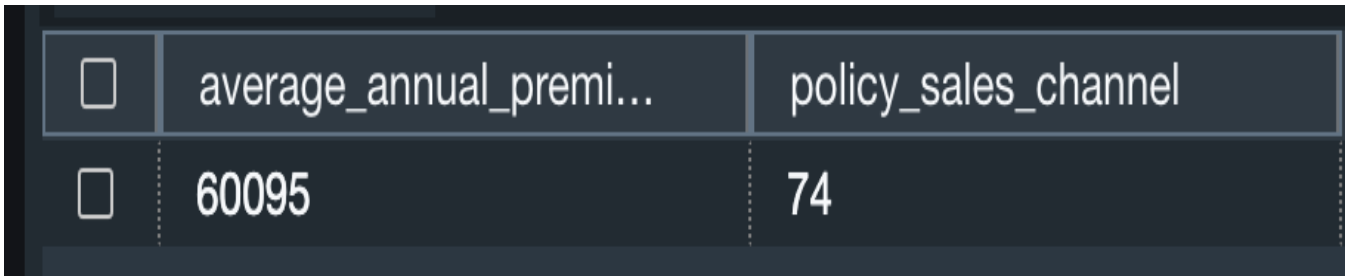
vehicle_age	average_annual_premi...
2	35619.1395010785
0	30110.78465003071
1	30515.170705765595

Fig-13 Output for calculating average annual premiums by vehicle.

14. Finding the Policy Sales Channel with the Highest Average Annual Premium:
Query:

```
SELECT Policy_Sales_Channel, AVG(Annual_Premium) AS Average_Annual_Premium
FROM "dbms".Policies
GROUP BY Policy_Sales_Channel
ORDER BY Average_Annual_Premium DESC
LIMIT 1;
```

Interaction with database: The query retrieves data from the "Policies" table in the "dbms" database and groups the results by policy sales channel to calculate the average annual premium for each channel. It then sorts the results in descending order by the average annual premium and limits the output to the top result (i.e., the policy sales channel with the highest average annual premium).



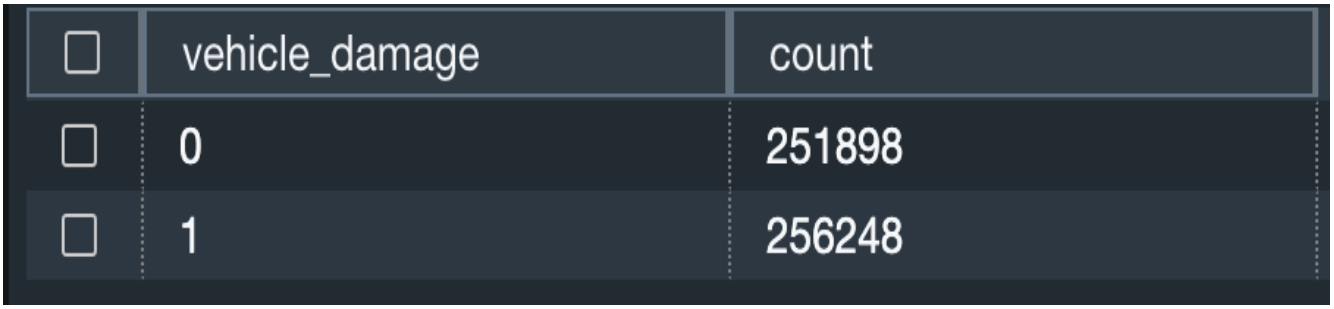
average_annual_premi...	policy_sales_channel
60095	74

Fig-14 Output showing the Policy Sales Channel with the Highest Average Annual Premium

15. Count of Vehicles by Damage Status:
Query:

```
SELECT Vehicle_Damage, COUNT(*) AS count
FROM dbms.Vehicles
GROUP BY Vehicle_Damage;
```

Interaction with database: The query is executed by accessing the "Vehicles" table in the "dbms" database and performing a grouping operation based on the "Vehicle_Damage" column. The "COUNT" function is used to count the number of vehicles with each damage status. The result is a table with two columns: "Vehicle_Damage" and "count".



vehicle_damage	count
0	251898
1	256248

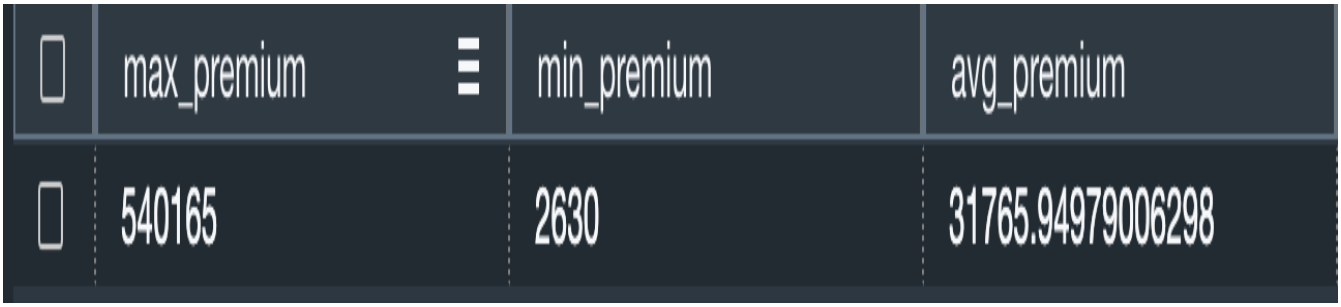
Fig-15 Output for Count of Vehicles by Damage Status.

16. Retrieving Maximum, Minimum, and Average Annual Premiums for Customers who Responded and have a Vehicle with Damage:

Query:

```
SELECT MAX(p.Annual_Premium) AS max_premium, MIN(p.Annual_Premium) AS min_premium,
AVG(p.Annual_Premium) AS avg_premium
FROM dbms.Customers c
INNER JOIN dbms.Policies p ON c.id = p.id
INNER JOIN dbms.Vehicles v ON c.id = v.id
WHERE c.Response = 1 AND v.Vehicle_Damage = 1;
```

Interaction with database: The database is queried to retrieve data from three tables using an inner join. The query filters records based on two conditions, where the customer responded and their vehicle has damage. The MAX, MIN, and AVG functions are then used to calculate the maximum, minimum, and average annual premiums for the customers who meet these conditions.



max_premium	min_premium	avg_premium
540165	2630	31765.94979006298

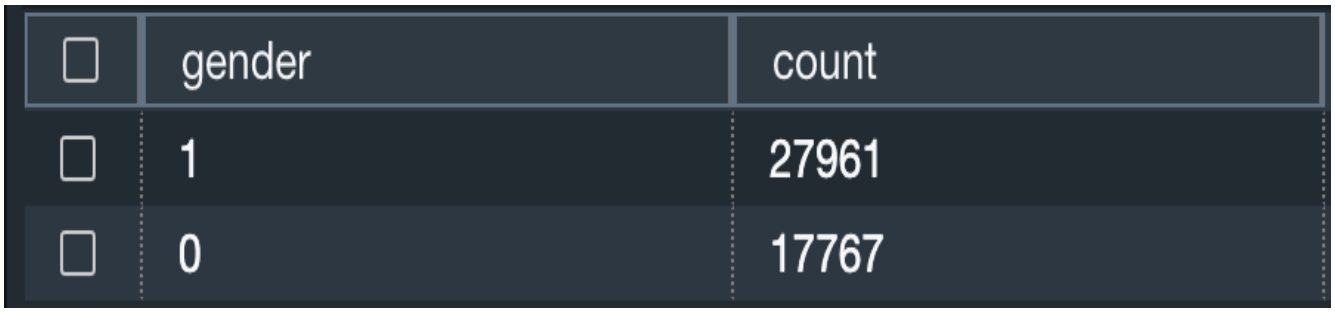
Fig-16 Output for Retrieving Maximum, Minimum, and Average Annual Premiums for Customers who Responded and have a Vehicle with Damage.

17. Counting the number of customers with response=1 and vehicle damage=1 grouped by gender:

Query:

```
SELECT c.Gender, COUNT(*) AS count
FROM dbms.Customers c
INNER JOIN dbms.Vehicles v ON c.id = v.id
WHERE c.Response = 1 AND v.Vehicle_Damage = 1
GROUP BY c.Gender;
```

Interaction with database: The query interacts with the "Customers" and "Vehicles" tables in the "dbms" database and performs an inner join to match records based on their ID. It then filters the records based on specific conditions and groups the results by gender to count the number of customers in each category.



gender	count
1	27961
0	17767

Fig-17 Output for counting the number of customers with response=1 and vehicle damage =1 and grouped by gender

18. Retrieving customer information with high annual premium:

Query:

```
SELECT c.id, c.Age, c.Gender, p.Annual_Premium, v.Vehicle_Age, v.Vehicle_Damage
FROM "dbms".Customers c
JOIN "dbms".Policies p ON c.id = p.id
JOIN "dbms".Vehicles v ON c.id = v.id
WHERE c.Response = 1 AND p.Annual_Premium >= (SELECT AVG(Annual_Premium) FROM "dbms".Policies WHERE id = c.id)
ORDER BY p.Annual_Premium DESC;
```

Interaction with database: The query involves joining three tables ("Customers", "Policies", and "Vehicles") in the "dbms" database based on their ID columns. It uses a subquery to calculate the average annual premium for each customer's policy ID and filters records based on the customer response and policy annual premium. The results are sorted by policy annual premium in descending order and include customer information and policy and vehicle details.

id	age	gender	annual_premium	vehicle_age
54744	26	1	540165	0
172258	40	1	489663	1
136305	50	1	472042	1
281680	45	0	472042	1
59101	41	0	340439	1
102294	43	1	336395	2
37856	47	1	336395	1
170381	44	1	316563	1

Fig-18 Output for Retrieving customer information with high annual premium.

CLOUD ANALYTICS AND DATA WAREHOUSE IMPLEMENTATION FOR A HISTORICAL DATASET

2. DAGS (Directed Acyclic Graph)

DAG1 : preprocess_and_split_dataset.py
Description:
The above DAG (Directed Acyclic Graph) in Airflow is named "preprocess_and_split_dataset" and is designed to preprocess and split a dataset into three tables, namely customer, vehicle, and policy tables. The DAG contains one task named "preprocess_dataset" which executes a Python function to load a CSV file from an S3 bucket, preprocess the data, and split it into three tables. The resulting tables are then converted to CSV format and uploaded to the same S3 bucket in a separate folder. The DAG is scheduled to run manually and has one dependency, i.e., it does not depend on any previous task.

Code:

```
from datetime import datetime, timedelta
from airflow import DAG
from airflow.providers.amazon.aws.hooks.s3 import S3Hook
from airflow.operators.python_operator import PythonOperator
from airflow.hooks.mysql_hook import MySqlHook
import pandas as pd
import io
import boto3


default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2023, 5, 6),
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}
dag = DAG(
    'preprocess_and_split_dataset',
    default_args=default_args,
    description='Preprocess and split dataset into customer, vehicle, and policy tables',
    schedule_interval=None,
)
def preprocess_dataset():
    # Load CSV file from S3 bucket
    s3_hook = S3Hook(aws_conn_id='s3_conn')
    s3_bucket = 'dbms-project-final'
    s3_key = 'input-csv/merged_data.csv'
    file_obj = s3_hook.get_key(key=s3_key, bucket_name=s3_bucket)
    file_content = file_obj.get()['Body'].read().decode('utf-8')
    df = pd.read_csv(io.StringIO(file_content))



    # Split dataset into customer, vehicle, and policy tables
```

```
customers = df[['id', 'Gender', 'Age', 'Driving_License', 'Region_Code', 'Previously_Insured','Response']].drop_duplicates()
vehicles = df[['id', 'Vehicle_Age', 'Vehicle_Damage']].drop_duplicates()
policies = df[['id', 'Annual_Premium', 'Policy_Sales_Channel', 'Vintage']].drop_duplicates()
# For the Customers table
# Convert gender to 1 for male and 0 for female
customers.loc[customers['Gender'] == 'Male', 'Gender'] = 1
customers.loc[customers['Gender'] == 'Female', 'Gender'] = 0
# Fill missing values in Response column with the mode (most frequent value)
customers['Response'].fillna(customers['Response'].mode()[0], inplace=True)
# Convert Response column to integer type
customers['Response'] = customers['Response'].astype(int)
# Convert Region_Code column to integer type
customers['Region_Code'] = customers['Region_Code'].astype(int)

# For the Vehicles table
# Convert Vehicle_Age column to 0 for < 1 year, 1 for 1-2 years, and 2 for > 2 years
vehicles.loc[vehicles['Vehicle_Age'] == '> 2 Years', 'Vehicle_Age'] = 2
vehicles.loc[vehicles['Vehicle_Age'] == '1-2 Year', 'Vehicle_Age'] = 1
vehicles.loc[vehicles['Vehicle_Age'] == '< 1 Year', 'Vehicle_Age'] = 0
# Convert Vehicle_Damage column to 1 for 'Yes' and 0 for 'No'
vehicles.loc[vehicles['Vehicle_Damage'] == 'Yes', 'Vehicle_Damage'] = 1
vehicles.loc[vehicles['Vehicle_Damage'] == 'No', 'Vehicle_Damage'] = 0

# For the Policies table
# Convert Policy_Sales_Channel column to integer type
policies['Policy_Sales_Channel']=policies['Policy_Sales_Channel'].astype(int)
# Convert Annual_Premium column to integer type
policies['Annual_Premium']=policies['Annual_Premium'].astype(int)

# Replace <aws_access_key> and <aws_secret_key> with your AWS access key and secret key
s3 = boto3.resource('s3', aws_access_key_id='AKIA4OKVZYE23QHYVBUU',
aws_secret_access_key='NS6OY3tybIk8L3Wgr+Sd3sdpCQ4v30jC1nk9Afsf')

# Set the name of your S3 bucket
s3_bucket_name = 'dbms-project-final'

# Set the file names for your dataframes
customers_file_name = 'customers.csv'
vehicles_file_name = 'vehicles.csv'
policies_file_name = 'policies.csv'

# Convert dataframes to CSV format
customers_csv = customers.to_csv(index=False)
vehicles_csv = vehicles.to_csv(index=False)
policies_csv = policies.to_csv(index=False)

# Upload CSV data to S3 bucket
s3.Object(s3_bucket_name, 'output-csv/' + customers_file_name).put(Body=customers_csv)
```

```
    s3.Object(s3_bucket_name, 'output-csv/' + vehicles_file_name).put(Body=vehicles_csv)
    s3.Object(s3_bucket_name, 'output-csv/' + policies_file_name).put(Body=policies_csv)


preprocess_data = PythonOperator(
    task_id='preprocess_dataset',
    python_callable=preprocess_dataset,
    dag=dag
)
preprocess_data
```

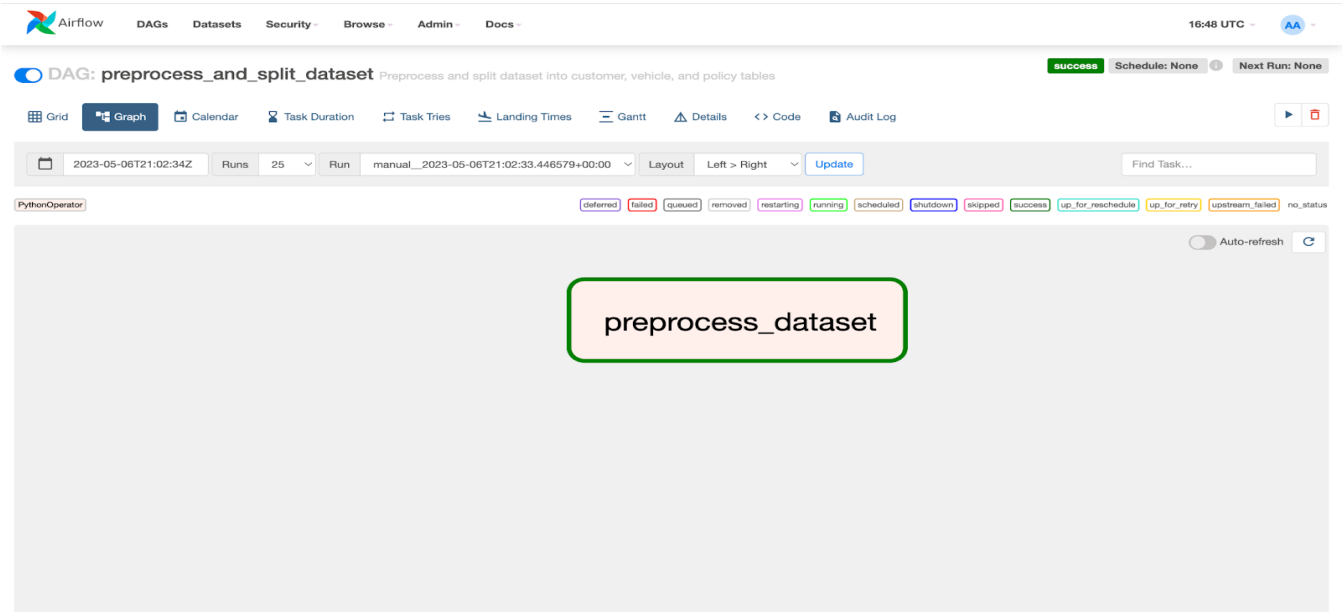


Fig-19 Dag1

DAG2 :load_csv_to_mysql.py

Description:

The above code defines an Airflow DAG named 'load_csv_to_mysql' that loads data from CSV files stored in an S3 bucket and writes them to a MySQL database using SQLAlchemy. The DAG consists of three tasks, each of which calls a Python function to load a specific CSV file into MySQL. The S3 bucket and object keys, as well as the MySQL database credentials, are specified in the Python functions. The task dependencies are defined such that the 'load_customers_to_rds' task is executed first, followed by 'load_vehicles_to_rds', and finally 'load_policies_to_rds'. The DAG is scheduled to run on a manual trigger only, as indicated by the 'schedule_interval=None' argument in the DAG definition.

Code:

```
from datetime import datetime
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.providers.amazon.aws.hooks.s3 import S3Hook
```

```
import boto3
import pandas as pd
from io import StringIO
from sqlalchemy import create_engine
import io

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2023, 5, 6),
    'retries': 1
}

dag = DAG('load_csv_to_mysql', default_args=default_args, schedule_interval=None)
# AWS credentials
aws_access_key_id = 'YOUR_AWS_ACCESS_KEY_ID'
aws_secret_access_key = 'YOUR_AWS_SECRET_ACCESS_KEY'
# Define the function to read the CSV file from S3 and write to MySQL
def load_customers_to_rds():
    s3_hook = S3Hook(aws_conn_id='s3_conn')
    s3_bucket = 'dbms-project-final'
    s3_key = 'output-csv/customers.csv'
    file_obj = s3_hook.get_key(key=s3_key, bucket_name=s3_bucket)
    file_content = file_obj.get()['Body'].read().decode('utf-8')
    df = pd.read_csv(io.StringIO(file_content))
    # Write DataFrame to MySQL database using SQLAlchemy
    engine =
create_engine('mysql+pymysql://admin:team8nohate@final-project.csqhs1mydagp.us-east-2.rds.amazonaws.com/final_proje
ct')
    df.to_sql('Customers', con=engine, index=False, if_exists='append')
# Define the function to read the CSV file from S3 and write to MySQL
def load_vehicles_to_rds():
    # S3 bucket configuration
    s3_hook = S3Hook(aws_conn_id='s3_conn')
    s3_bucket = 'dbms-project-final'
    s3_key = 'output-csv/vehicles.csv'
    file_obj = s3_hook.get_key(key=s3_key, bucket_name=s3_bucket)
    file_content = file_obj.get()['Body'].read().decode('utf-8')
    df = pd.read_csv(io.StringIO(file_content))
    # Write DataFrame to MySQL database using SQLAlchemy
    engine =
create_engine('mysql+pymysql://admin:team8nohate@final-project.csqhs1mydagp.us-east-2.rds.amazonaws.com/final_proje
ct')
    df.to_sql('Vehicles', con=engine, index=False, if_exists='append')
def load_policies_to_rds():
    s3_hook = S3Hook(aws_conn_id='s3_conn')
    s3_bucket = 'dbms-project-final'
    s3_key = 'output-csv/policies.csv'
```

```
    file_obj = s3_hook.get_key(key=s3_key, bucket_name=s3_bucket)
    file_content = file_obj.get()['Body'].read().decode('utf-8')
    df = pd.read_csv(io.StringIO(file_content))
    # Write DataFrame to MySQL database using SQLAlchemy
    engine =
create_engine('mysql+pymysql://admin:team8nohate@final-project.csqhs1mydagp.us-east-2.rds.amazonaws.com/final_proje
ct')
    df.to_sql('Policies', con=engine, index=False, if_exists='append')
# Define the DAG tasks
t1 = PythonOperator(
    task_id='load_customers_to_rds',
    python_callable=load_customers_to_rds,
    dag=dag
)
t2= PythonOperator(
    task_id='load_vehicles_to_rds',
    python_callable=load_vehicles_to_rds,
    dag=dag
)
t3= PythonOperator(
    task_id='load_policies_to_rds',
    python_callable=load_policies_to_rds,
    dag=dag
)
# Define the task dependencies
t1>>t2>>t3
```

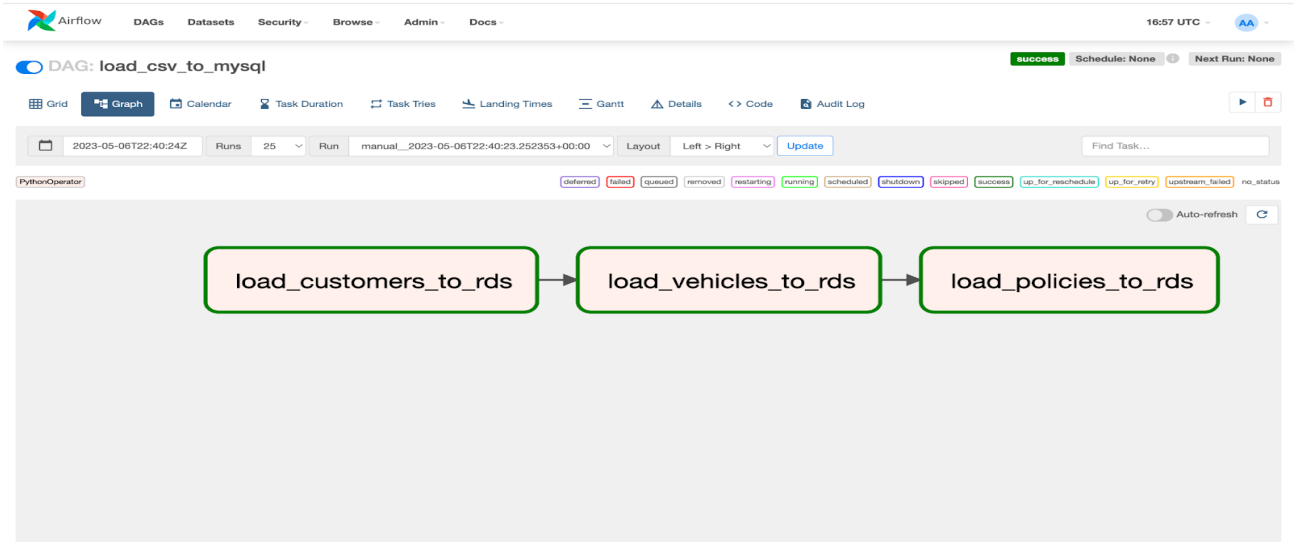


Fig-20 Dag2

CLOUD ANALYTICS AND DATA WAREHOUSE IMPLEMENTATION FOR A HISTORICAL DATASET

DAG3 :export_to_s3.py

Description:

This is an Airflow DAG that exports data from a MySQL RDS instance to CSV files and then uploads them to an S3 bucket. The exported tables include Customers, Vehicles, and Policies. Each table has a corresponding Python function that exports the data and uploads the CSV file to S3 using an S3Hook. The DAG is scheduled to run once a day.

Code:

```
import os
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.operators.bash_operator import BashOperator
from airflow.providers.amazon.aws.hooks.s3 import S3Hook
import csv
import pymysql.cursors
# AWS credentials
aws_access_key_id = "AKIA4OKVZYE22LCPNEUC"
aws_secret_access_key = "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"
# RDS configuration
rds_host = "final-project.csqhs1mydagp.us-east-2.rds.amazonaws.com"
db_name = "final_project"
username = "admin"
password = "team8nohate"
# Define default arguments
default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": datetime(2023, 5, 6),
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
}

# Define DAG
dag = DAG("export_to_s3", default_args=default_args, schedule_interval=timedelta(days=1))
# Function to export data to CSV file
def export_vehicle_data_to_s3():
    # S3 configuration
    s3_bucket = "dbms-project-final"
    s3_key = "input-for-redshift/vehicles.csv"
    # Connect to RDS database
    conn = pymysql.connect(host=rds_host, user=username, password=password, db=db_name,
cursorclass=pymysql.cursors.DictCursor)
    # Execute SQL query to export data to CSV file
    with conn.cursor() as cursor:
        query = "SELECT * FROM Vehicles;"
        cursor.execute(query)
        rows = cursor.fetchall()

    # Write data to CSV file
```

```
    with open("/tmp/mytable.csv", "w", newline='') as file:
        writer = csv.writer(file, delimiter=",", quotechar='"', quoting=csv.QUOTE_MINIMAL)
        writer.writerow([i[0] for i in cursor.description])  # Write header row
        for row in rows:
            writer.writerow(row.values())

    # Upload CSV file to S3
    s3_hook = S3Hook(aws_conn_id="s3_conn")
    s3_hook.load_file(
        filename="/tmp/mytable.csv",
        bucket_name=s3_bucket,
        key=s3_key,
        replace=True,
        encrypt=False,
    )

    # Delete the CSV file from the server
    os.remove("/tmp/mytable.csv")

# Function to export data to CSV file
def export_customer_data_to_s3():

    # S3 configuration
    s3_bucket = "dbms-project-final"
    s3_key = "input-for-redshift/customers.csv"
    # Connect to RDS database
    conn = pymysql.connect(host=rds_host, user=username, password=password, db=db_name,
cursorclass=pymysql.cursors.DictCursor)

    # Execute SQL query to export data to CSV file
    with conn.cursor() as cursor:
        query = "SELECT * FROM Customers;"
        cursor.execute(query)
        rows = cursor.fetchall()

    # Write data to CSV file
    with open("/tmp/mytable.csv", "w", newline='') as file:
        writer = csv.writer(file, delimiter=",", quotechar='"', quoting=csv.QUOTE_MINIMAL)
        writer.writerow([i[0] for i in cursor.description])  # Write header row
        for row in rows:
            writer.writerow(row.values())

    # Upload CSV file to S3
    s3_hook = S3Hook(aws_conn_id="s3_conn")
    s3_hook.load_file(
        filename="/tmp/mytable.csv",
        bucket_name=s3_bucket,
        key=s3_key,
```

```
        replace=True,
        encrypt=False,
    )
    # Delete the CSV file from the server
    os.remove("/tmp/mytable.csv")
def export_policy_data_to_s3():
        # S3 configuration
    s3_bucket = "dbms-project-final"
    s3_key = "input-for-redshift/policies.csv"
    # Connect to RDS database
    conn = pymysql.connect(host=rds_host, user=username, password=password, db=db_name,
cursorclass=pymysql.cursors.DictCursor)
    # Execute SQL query to export data to CSV file
    with conn.cursor() as cursor:
        query = "SELECT * FROM Policies;"
        cursor.execute(query)
        rows = cursor.fetchall()
    # Write data to CSV file
    with open("/tmp/mytable.csv", "w", newline='') as file:
        writer = csv.writer(file, delimiter=",", quotechar='"', quoting=csv.QUOTE_MINIMAL)
        writer.writerow([i[0] for i in cursor.description])  # Write header row
        for row in rows:
            writer.writerow(row.values())

    # Upload CSV file to S3
    s3_hook = S3Hook(aws_conn_id="s3_conn")
    s3_hook.load_file(
        filename="/tmp/mytable.csv",
        bucket_name=s3_bucket,
        key=s3_key,
        replace=True,
        encrypt=False,
    )

    # Delete the CSV file from the server
    os.remove("/tmp/mytable.csv")
# Define tasks
task_export_customer_to_csv = PythonOperator(
    task_id="export_customer_data_to_s3",
    python_callable=export_customer_data_to_s3,
    dag=dag,
)
task_export_vehicle_to_csv = PythonOperator(
    task_id="export_vehicle_data_to_s3",
    python_callable=export_vehicle_data_to_s3,
    dag=dag,
)
task_export_policy_to_csv = PythonOperator(
```

```
    task_id="export_policy_data_to_s3",
    python_callable=export_policy_data_to_s3,
    dag=dag,
)
# Define task dependencies
task_export_customer_to_csv >> task_export_vehicle_to_csv>>task_export_policy_to_csv
```

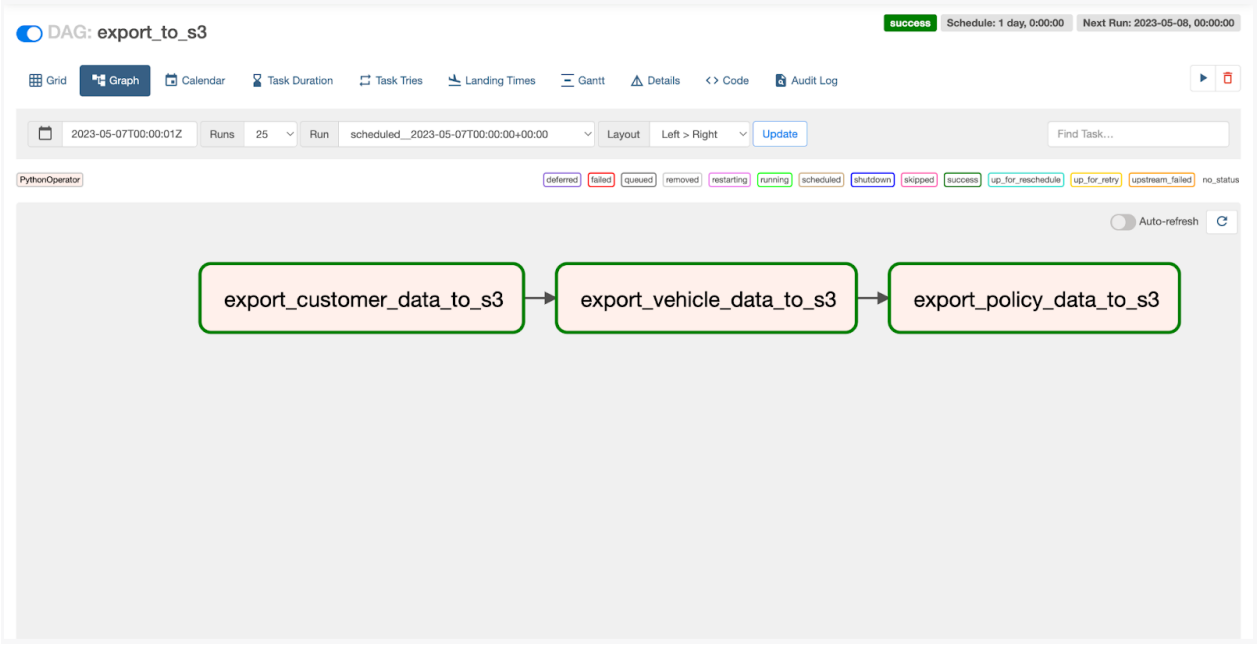


Fig-21 Dag3

DAG4 :copy_data_from_s3_to_redshift.py

Description:

This DAG copies data from S3 to Redshift in three tables: customers, vehicles, and policies. It uses PythonOperator to call three separate functions, each of which connects to Redshift and S3, truncates the target table, and copies the data from S3 to Redshift using the COPY command. The DAG has default arguments such as retries and retry delay, and it does not have a scheduled interval, meaning it will only run when manually triggered.

Code:

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime
from psycopg2 import OperationalError
import psycopg2
from datetime import datetime, timedelta


default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
```

```
    'start_date': datetime(2023, 5, 6),
    'retries': 3,
    'retry_delay': timedelta(minutes=5)
}

dag = DAG(
    'copy_data_from_s3_to_redshift',
    default_args=default_args,
    schedule_interval=None
)

def copy_data_from_s3_to_redshift_customers():

    # Set the connection parameters
    host = 'redshift-aws-bart.cxqbfjz9n9jj.us-east-2.redshift.amazonaws.com'
    dbname = 'final-project'
    port = '5439'
    user = 'awsuser'
    password = 'Shashank_9292'

    # Connect to the database
    con = psycopg2.connect(
        host=host,
        dbname=dbname,
        port=port,
        user=user,
        password=password
    )

    # Set the schema search path
    schema_name = 'dbms'
    cur = con.cursor()
    cur.execute("SET search_path TO {0}".format(schema_name))
    con.commit()

     # Truncate target table
    truncate_command = "TRUNCATE TABLE customers"
    cur.execute(truncate_command)
    con.commit()

    # Copy data from S3 to Redshift
    copy_command = """COPY customers FROM 's3://dbms-project-final/input-for-redshift/customers.csv'
                    IAM_ROLE 'arn:aws:iam::855415308597:role/shashank_role'
                    FORMAT AS CSV DELIMITER ',' QUOTE '"' IGNOREHEADER 1 REGION AS 'us-east-2'"""
    cur.execute(copy_command)
    con.commit()

    # Close the cursor and the connection
```

```
    cur.close()
    con.close()

def copy_data_from_s3_to_redshift_vehicles():

    # Set the connection parameters
    host = 'redshift-aws-bart.cxqbfjz9n9jj.us-east-2.redshift.amazonaws.com'
    dbname = 'final-project'
    port = '5439'
    user = 'awsuser'
    password = 'Shashank_9292'

    # Connect to the database
    con = psycopg2.connect(
        host=host,
        dbname=dbname,
        port=port,
        user=user,
        password=password
    )

    # Set the schema search path
    schema_name = 'dbms'
    cur = con.cursor()
    cur.execute("SET search_path TO {0}".format(schema_name))
    con.commit()

     # Truncate target table
    truncate_command = "TRUNCATE TABLE vehicles"
    cur.execute(truncate_command)
    con.commit()

    # Copy data from S3 to Redshift
    copy_command = """COPY vehicles FROM 's3://dbms-project-final/input-for-redshift/vehicles.csv'
                IAM_ROLE 'arn:aws:iam::855415308597:role/shashank_role'
                FORMAT AS CSV DELIMITER ',' QUOTE '"' IGNOREHEADER 1 REGION AS 'us-east-2'"""
    cur.execute(copy_command)
    con.commit()

    # Close the cursor and the connection
    cur.close()
    con.close()

def copy_data_from_s3_to_redshift_policies():

    # Set the connection parameters
    host = 'redshift-aws-bart.cxqbfjz9n9jj.us-east-2.redshift.amazonaws.com'
    dbname = 'final-project'
```

```
    port = '5439'
    user = 'awsuser'
    password = 'Shashank_9292'

    # Connect to the database
    con = psycopg2.connect(
        host=host,
        dbname=dbname,
        port=port,
        user=user,
        password=password
    )

    # Set the schema search path
    schema_name = 'dbms'
    cur = con.cursor()
    cur.execute("SET search_path TO {0}".format(schema_name))
    con.commit()

     # Truncate target table
    truncate_command = "TRUNCATE TABLE policies"
    cur.execute(truncate_command)
    con.commit()

    # Copy data from S3 to Redshift
    copy_command = """COPY policies FROM 's3://dbms-project-final/input-for-redshift/policies.csv'
                    IAM_ROLE 'arn:aws:iam::855415308597:role/shashank_role'
                    FORMAT AS CSV DELIMITER ',' QUOTE '"' IGNOREHEADER 1 REGION AS 'us-east-2'"""
    cur.execute(copy_command)
    con.commit()

    # Close the cursor and the connection
    cur.close()
    con.close()

task_copy_data_customers= PythonOperator(
    task_id='copy_data_from_s3_to_redshift_customers',
    python_callable=copy_data_from_s3_to_redshift_customers,
    dag=dag
)

task_copy_data_vehicles= PythonOperator(
    task_id='copy_data_from_s3_to_redshift_vehicles',
    python_callable=copy_data_from_s3_to_redshift_vehicles,
    dag=dag
)

task_copy_data_policies= PythonOperator(
```

```
    task_id='copy_data_from_s3_to_redshift_policies',
    python_callable=copy_data_from_s3_to_redshift_policies,
    dag=dag
)

task_copy_data_customers>>task_copy_data_vehicles>>task_copy_data_policies
```

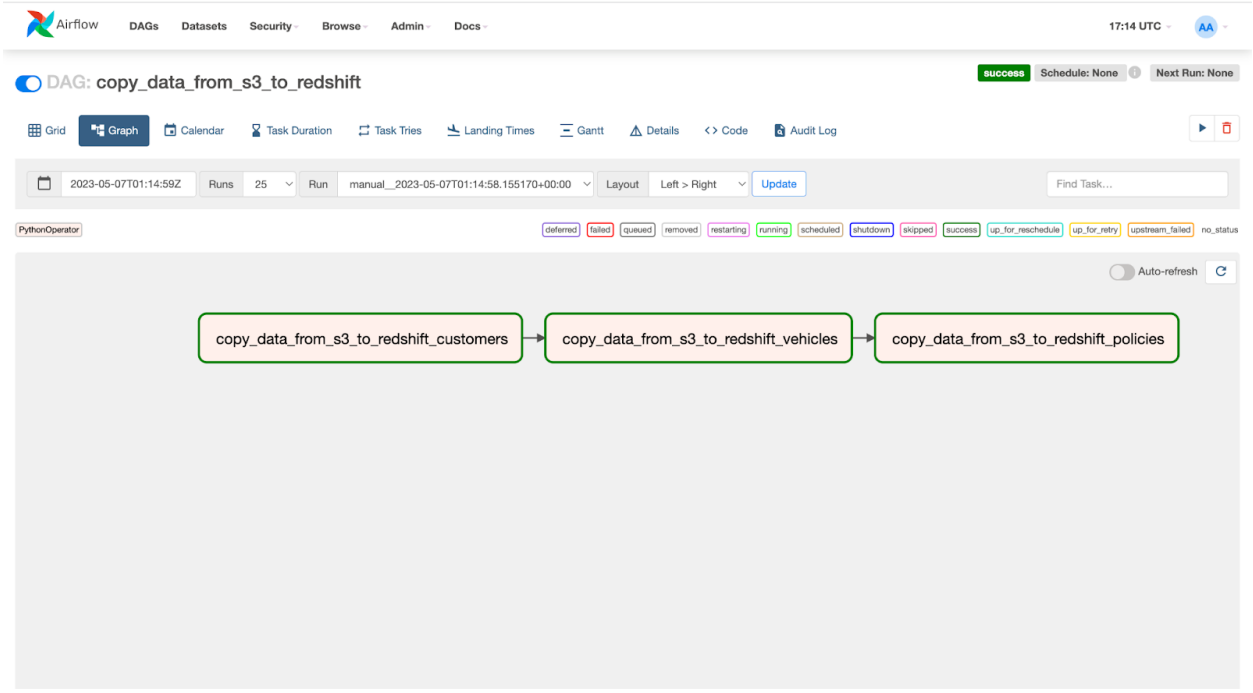


Fig-22 Dag4

THE END