



DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y COMPUTACIÓN
Inteligencia de negocios 202220 – Laboratorio 5
PROFESORA: Haydemar Nuñez

Nombres	Apellidos	Código	Login
María Sofía	Álvarez López	201729031	ms.alvarezl
Brenda Catalina	Barahona Pinilla	201812721	bc.barahona
Alvaro Daniel	Plata Márquez	201820098	ad.plata

Informe de laboratorio #5

En este laboratorio, trabajaremos como consultores de BI de la empresa WWI (World Wide Importers), que es una compañía encargada de realizar importaciones y venderlas a diferentes clientes en diferentes ciudades de Estados Unidos. Ellos desean optimizar sus ganancias, pues consideran que algunos de sus productos no están generando las ganancias que deberían. También, están interesados en saber si hay otros factores que le impiden optimizar sus ganancias. La consultoría de BI consistirá en la creación de la base de datos, la carga de datos y unas consultas iniciales que permitan validar el proceso previo.

El objetivo de este laboratorio es reforzar el conocimiento del proceso de ETL a través del uso de herramientas como Airflow y Apache Hadoop Distributed File System. El primero es una herramienta que administra, estructura y organiza pipelines de datos utilizando gráficos acíclicos dirigidos (DAG). Y el segundo es un sistema de archivos distribuido que proporciona acceso de alto rendimiento a datos guardados en clústeres de Hadoop.

Repositorio asociado a este laboratorio:

https://github.com/alvarodpm/BI_Lab5

Perfilamiento de los datos:

Lo primero que realizamos fue un perfilamiento y preprocesamiento de los datos para que estos estuvieran listos. Para el preprocesamiento nos apoyamos en la herramienta *pandas profiling* para tener un reporte del estado de los datos. Para ver el perfilamiento, remítase a https://github.com/alvarodpm/BI_Lab5/blob/main/preprocesamiento.ipynb. En esto encontramos:

- **Variable dimensión City:**
 - Hay 97 filas con datos con 10 columnas

- No encontramos filas duplicadas
 - Las variables row ID y City_Key tienen un valor distinto en cada fila.
 - No encontramos celdas ausentes.
 - Las variables Country, Continent, Region y Subregion tienen el mismo valor en todas las celdas. (columnas duplicadas)
 - Todos los tipos de datos coinciden con lo que esperan las columnas.
 - Ningún dato excede el tamaño máximo definido en la BD
- **Variable dimensión Customer:**
 - Hay 402 filas con datos con 7 columnas
 - No encontramos filas duplicadas
 - No encontramos celdas ausentes.
 - La Variable Category tiene un único valor.
 - Las variables Bill_To_Customer y Buying_Group aportan información similar.
 - Al comparar los tipos de datos esperados con los del perfilamiento se encuentra que el código postal debe ser un int.
 - Ningún dato excede el tamaño máximo definido en la BD
- **Variable dimensión Date:**
 - Hay 402 filas con datos con 7 columnas
 - Los datos corresponden a todos los días durante 4 años, lo cual incluye un año bisiesto.
 - No encontramos filas duplicadas
 - No encontramos celdas ausentes.
 - Las variables Day_Number y Day_val aportan la misma información, por lo que podemos considerar eliminar una de las 2.
 - Las variables Month_val y Short_Month aportan la misma información, por lo que podemos considerar eliminar una de las 2.
 - Al comparar los tipos de datos esperados con los del perfilamiento se encuentra que Date_key no coincide, pues se espera que esta sea de tipo Date y en realidad se está tomando como Object. Para arreglar este error se usará la función TO_DATE()
- **Variable dimensión Employee:**
 - Hay 212 filas con datos con 7 columnas
 - No encontramos filas duplicadas
 - No encontramos celdas ausentes.
 - Para realizar la carga adecuada de los datos, cambiaremos los valores de True y False a '1' y '0'.
- **Variable dimensión Stock item:**

- Hay 671 filas con datos con 14 columnas
- No encontramos filas duplicadas
- Encontramos 1118 celdas vacías que corresponden al 11.1% de los datos, estos datos se completarán con la función de fillNa de python.
- Observamos que la variable Brand tiene un valor constante en las celdas que tienen un valor asignado, pero además presenta un 90% de ausencias.
- Cambiaremos los valores booleanos por su respectivo valor en String, '1' o '0'.
- Al comparar las columnas que se esperan con las variables que están en el perfilamiento se encuentra que:
 - En los datos del csv no hay ninguna columna denominada WWI_Stock_Item_ID, por lo que debe eliminarse esta de la creación de tablas.
 - Las variables Tax_Rate, Unit_Price, Recommended_Retail_Price y Typical_Weight_Per_Unit se toman como objetos, pero en realidad deben ser números decimales.
- **Tabla fact order:**
 - Tabla de hechos, sin ninguna anomalía. Cuenta con 15 variables, de las cuales 2 son numéricas y las 13 restantes son strings. La tabla de hechos tiene 1000 entradas
- **En general:**
 - Se eliminó la primera fila, con row ID Row0, no aporta información relevante pues todos sus datos están ausentes
 - Ningún dato excede el tamaño máximo definido en la BD

Diagrama de alto nivel describiendo el proceso de ETL:

Según Kimball [1], es necesario tener un esquema general del proyecto antes de realizar un diagrama ETL. Además de ser independiente de la tecnología, debe ilustrar el proceso desde que los datos son extraídos hasta que son subidos a la base de datos final. El proceso ETL elaborado para WWI se encuentra a continuación:

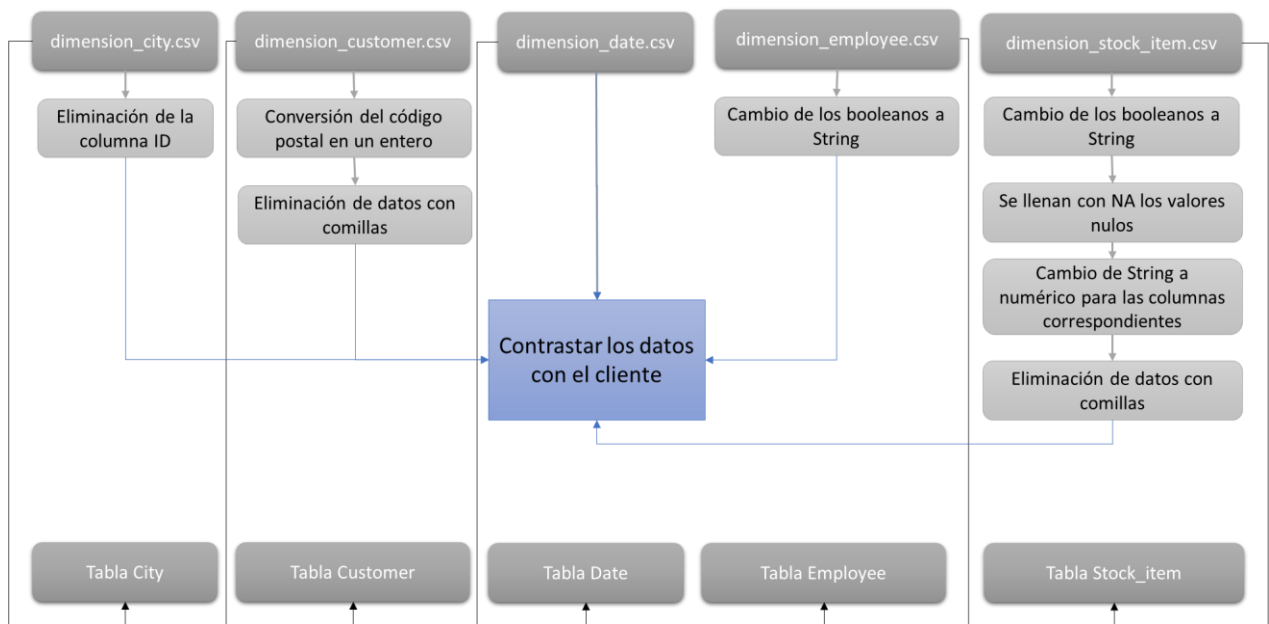


Figura 1: Diagrama de alto nivel describiendo el proceso ETL para WWI.

Partiendo de los archivos csv suministrados por la compañía, y arreglando los errores que traían los datos en ellos (por ejemplo, cambiando las celdas nulas por NA, convirtiendo los datos numéricos que se encontraban en formato string al formato decimal y eliminando las comillas que podían afectar la escritura de las sentencias SQL. Una vez realizadas las modificaciones, se subieron los datos a la BD correspondiente a cada tabla.

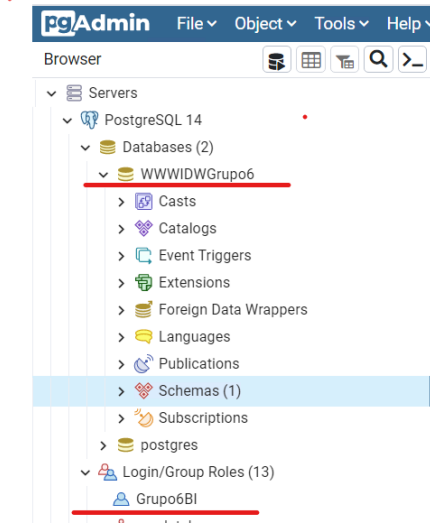
Documentación del proceso y las transformaciones realizadas en Airflow

Para el desarrollo de este laboratorio seguimos los pasos descritos en el tutorial señalado. A continuación, describimos los pasos que llevamos a cabo en la máquina virtual asignada. Asimismo, los archivos de Python asociados a todo esta parte del laboratorio puede encontrarlos en: https://github.com/alvarodpm/BI_Lab5/tree/main/dags .

1. Creación de la base de datos PostgreSQL

- a. Instalamos Postgres junto con pgAdmin4. Para esto seguimos los pasos descritos en la documentación oficial de Postgres ([PostgreSQL: Windows installers](#))
- b. Ejecutamos pgAdmin4 y creamos un usuario y una base de datos, llamados Grupo6BI y WWIDWGrupo6 respectivamente. La contraseña para acceder a la base de datos con este usuario es "usuario". Al usuario Grupo6BI le asignamos los privilegios necesarios para poder hacer login, crear bases de datos y heredar privilegios de los roles padres.

En la siguiente imagen se puede ver el resultado de este paso. Podemos ver la base de datos creada llamada WWIDWGrupo6 y el usuario Grupo6BI



2. Desplegar Airflow

- a. En la carpeta "C:\Users\estudiante", en el archivo ".wslconfig" cambiamos la configuración Memory a 6GB, indicando que será ésta la cantidad de memoria RAM que podrá usar el Docker de Airflow.

.wslconfig: Bloc de notas

Archivo Edición Formato Ver Ayuda

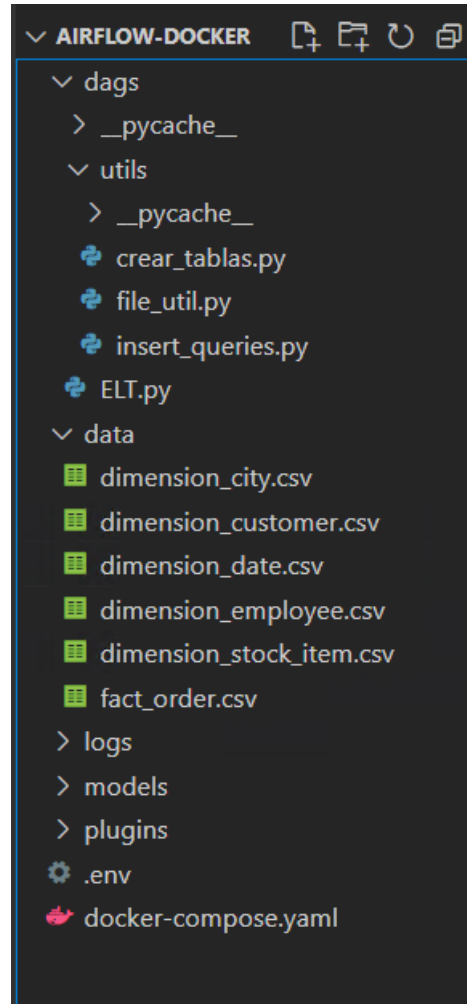
[wsl2]

```
memory=6GB # Limits VM memory in WSL 2 up to 3GB
processors=4 # Makes the WSL 2 VM use two virtual processors
```

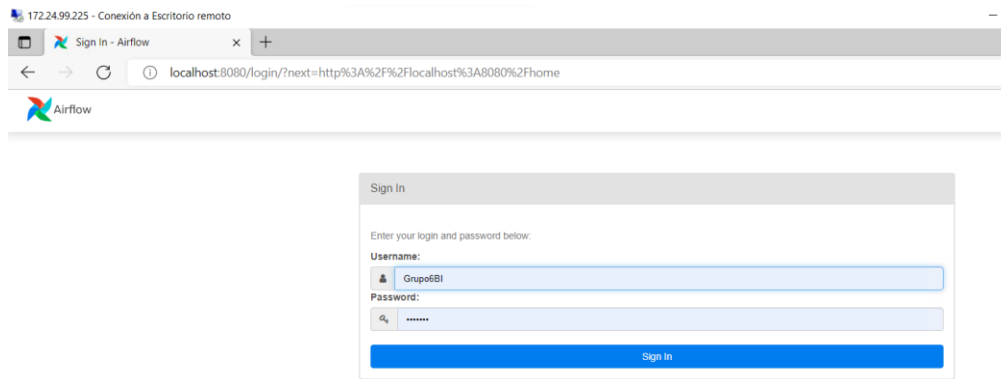
- b. En la carpeta Airflow-Docker que se encuentra en el escritorio, identificamos los siguientes archivos y carpetas:
- docker-compose.yaml: en este archivo se define el scheduler que supervisa las tareas y DAG, el servidor web para acceder a la interfaz de Airflow, el worker que ejecuta las tareas dadas por el scheduler y el servicio de inicialización
 - ./dags: aquí van sus colecciones de tareas en airflow.
 - ./logs: Contiene registros de la ejecución de tareas.
 - ./files: Aquí van los archivos CSV
 - ./models: Aquí van los modelos .joblib, en este laboratorio no será necesario usar esta carpeta.
 - ./utils: Aquí van a almacenar las funciones que son llamadas por cada tarea de Airflow.

vii. .env: aquí configuramos el User Id y el Grupo Id de Airflow

Observamos la estructura final de la carpeta Airflow-Docker

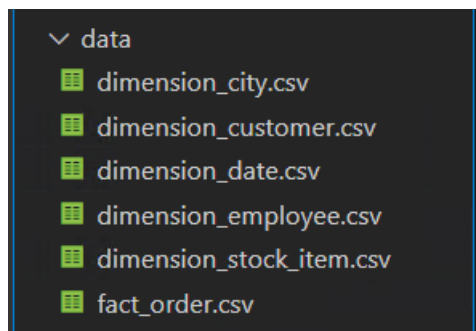


- c. Desde la carpeta Airflow-Docker ejecutamos los siguientes comandos:
- i. `docker-compose up airflow-init`: usado para crear la imagen de airflow y todos sus servicios
 - ii. `docker-compose up`: usado para lanzar todos los servicios
 - iii. Ingresamos desde el navegador a la interfaz de Airflow y verificamos que podemos hacer login



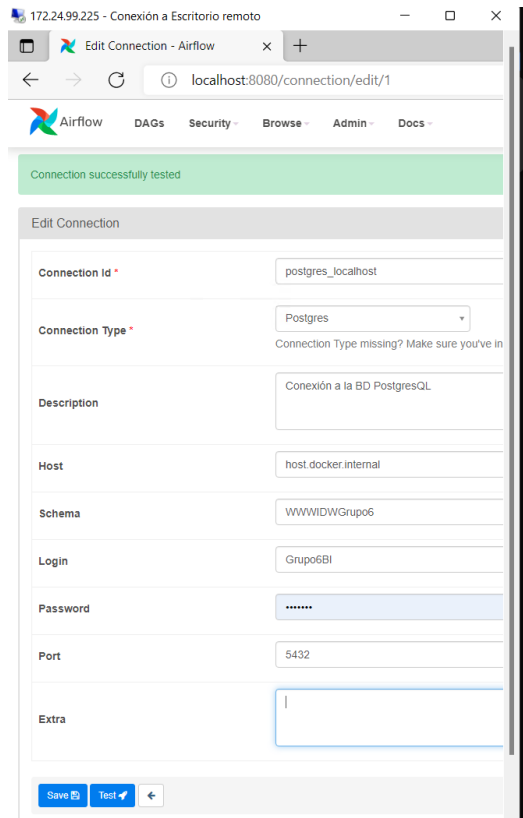
3. Detallar y preprocesar los datos

- a. Realizamos la identificación y preprocesamiento de los datos con los que realizaremos el proceso ETL. La explicación detallada de este proceso se encuentra en la sección _____.
- b. Una vez hallamos realizado este proceso, guardamos los datos preprocesados en la carpeta Airflow-Docker/data, para poder ser usados por Airflow en el proceso ETL



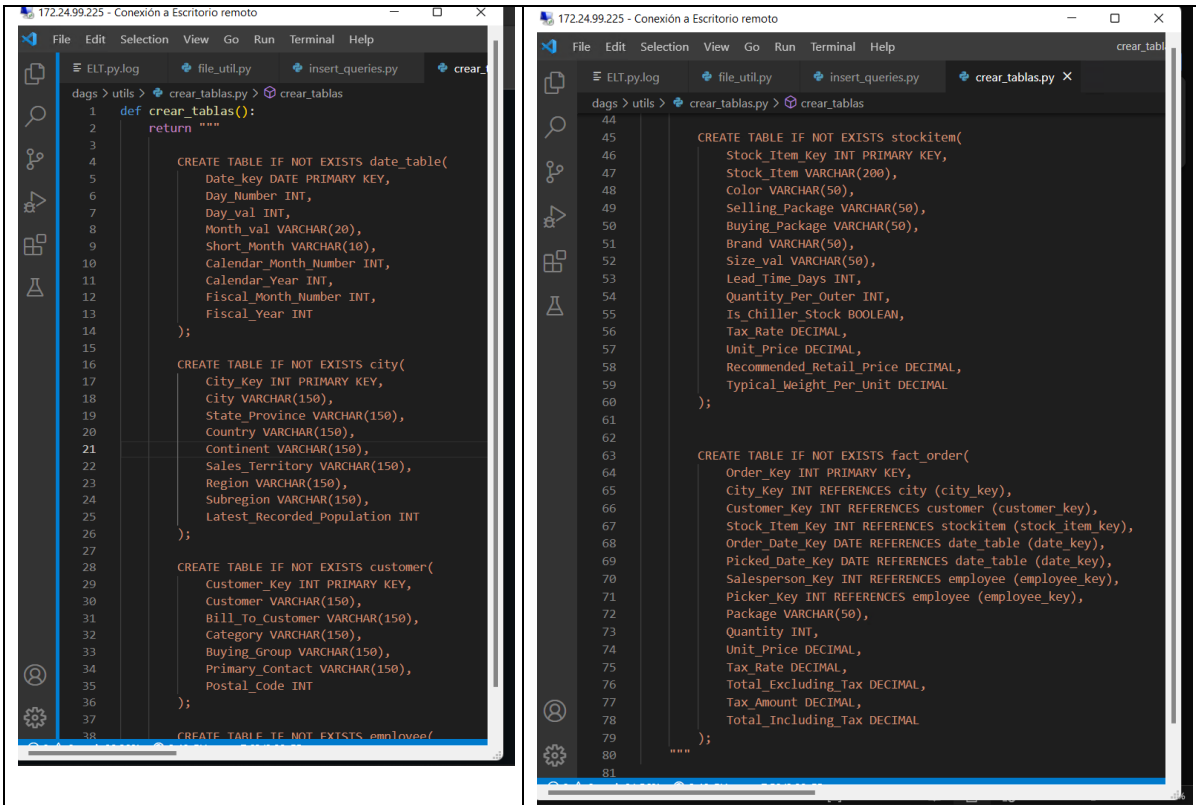
4. Creación de una conexión postgres en Airflow

- i. En la interfaz de airflow, en la pestaña Admin -> Connections, creamos la conexión que tendrá Airflow con la base de datos Postgres de la VM. Verificamos que la conexión es exitosa.



5. Creación de los archivos de utilidad para el DAG

- a. En la carpeta Airflow-Docker/utils creamos los siguientes archivos:
 - i. crear_tablas-py: definimos el script de creación de tablas del modelo multidimensional



- ii. file_util.py: definimos las funciones de escritura y lectura de csv:



- iii. insert_queries.py: definimos las funciones que transformarán el contenido de archivos csv a queries de inserción en SQL. Se realizó una función por cada una de las tablas, cuidando los parámetros y sus nombres específicos, las comillas y los tipos de datos esperados por la base de datos. Esto se puede ver en las siguientes figuras:

```
def insert_query_city(**kwargs):
    insert = f"INSERT INTO city (City_Key,City,State_Province,Country,Continent,Sales_Territory,Region,Subregion,Latest_Recorded_Population) VALUES "
    insertQuery = ""
    dataframe = cargar_datos(kwargs['csv_path'])
    for index, row in dataframe.iterrows():
        insertQuery += insert + f"({row.City_Key},{row.City},{row.State_Province},{row.Country},{row.Continent},{row.Sales_Territory},{row.Region},{row.Subregion},{row.Latest_Recorded_Population});\n"
    return insertQuery

def insert_query_customer(**kwargs):
    insert = f"INSERT INTO customer (Customer_Key,Customer,Bill_To_Customer,Category,Buying_Group,Primary_Contact,Postal_Code) VALUES "
    insertQuery = ""
    dataframe = cargar_datos(kwargs['csv_path'])
    for index, row in dataframe.iterrows():
        insertQuery += insert + f"({row.Customer_Key},{row.Customer},{row.Bill_To_Customer},{row.Category},{row.Buying_Group},{row.Primary_Contact},{row.Postal_Code});\n"
    return insertQuery

def insert_query_date(**kwargs):
    insert = f"INSERT INTO date_table (date_key,date_number,day_val,month_val,short_month,calendar_month_number,calendar_year,fiscal_month_number,fiscal_year) VALUES "
    insertQuery = ""
    dataframe = cargar_datos(kwargs['csv_path'])
    for index, row in dataframe.iterrows():
        insertQuery += insert + f"({row.Date_Key},{row.Date_Number},{row.Day_Val},{row.Month_Val},{row.Short_Month},{row.Calendar_Month},{row.Calendar_Year},{row.Fiscal_Month},{row.Fiscal_Year});\n"
    return insertQuery

def insert_query_employee(**kwargs):
    insert = f"INSERT INTO employee (Employee_Key,Employee,Preferred_Name,Is_Salesperson) VALUES "
    insertQuery = ""
    dataframe = cargar_datos(kwargs['csv_path'])
    for index, row in dataframe.iterrows():
        insertQuery += insert + f"({row.Employee_Key},{row.Employee},{row.Preferred_Name},{row.Is_Salesperson});\n"
    return insertQuery

def insert_query_stock(**kwargs):
    insert = f"INSERT INTO stockitem (Stock_Item_Key,Stock_Item,Color,Selling_Package,Buying_Package,Brand,Size_val,Lead_Time_Days,Quantity_Per_Outer,Is_Chiller_Stock,Tax) VALUES "
    insertQuery = ""
    dataframe = cargar_datos(kwargs['csv_path'])
    for index, row in dataframe.iterrows():
        insertQuery += insert + f"({row.Stock_Item_Key},{row.Stock_Item},{row.Color},{row.Selling_Package},{row.Buying_Package},{row.Brand},{row.Size_val},{row.Lead_Time_Days},{row.Quantity_Per_Outer},{row.Is_Chiller_Stock},{row.Tax});\n"
    return insertQuery

def insert_query_fact_order(**kwargs):
    insert = f"INSERT INTO fact_order (Order_Key,City_Key,Customer_Key,Stock_Item_Key,Order_Date_Key,Picked_Date_Key,Salesperson_Key,Picker_Key,Package,Quantity,Unit_Price) VALUES "
    insertQuery = ""
    dataframe = cargar_datos(kwargs['csv_path'])
    for index, row in dataframe.iterrows():
        insertQuery += insert + f"({row.order_key},{row.city_key},{row.customer_key},{row.stock_item_key},{row.order_date_key},{row.picked_date_key},{row.salesperson_key},{row.picker_key},{row.package},{row.quantity},{row.unit_price});\n"
    return insertQuery
```

6. Implementar el DAG de ETL

Apache Airflow usa data pipelines o flujos de trabajo para realizar funciones para consumo. Para ello utiliza DAGs (Directed Acyclic Graphs). Estos son una colección de todas las tareas que se desean ejecutar, organizadas de una manera que refleja sus relaciones y dependencias.

- a. Creamos un DAG que consta de 3 pasos. Todos ellos utilizarán operador de Airflow llamado "PostgresOperator" el cual es el encargado de manejar conexiones y procesos relacionados con bases de datos PostgreSQL:
 - i. Crear tablas en la base de datos.
 - ii. Pobl原因ar tablas de dimensiones. Como la inserción de cada una de estas dimensiones es un proceso estrechamente relacionado, agruparemos todas estas tareas en un TaskGroup de Airflow.
 - iii. Pobl原因ar tabla de hechos. Una vez terminada la inserción de las dimensiones la tabla de hechos debe guardarse en Postgres.

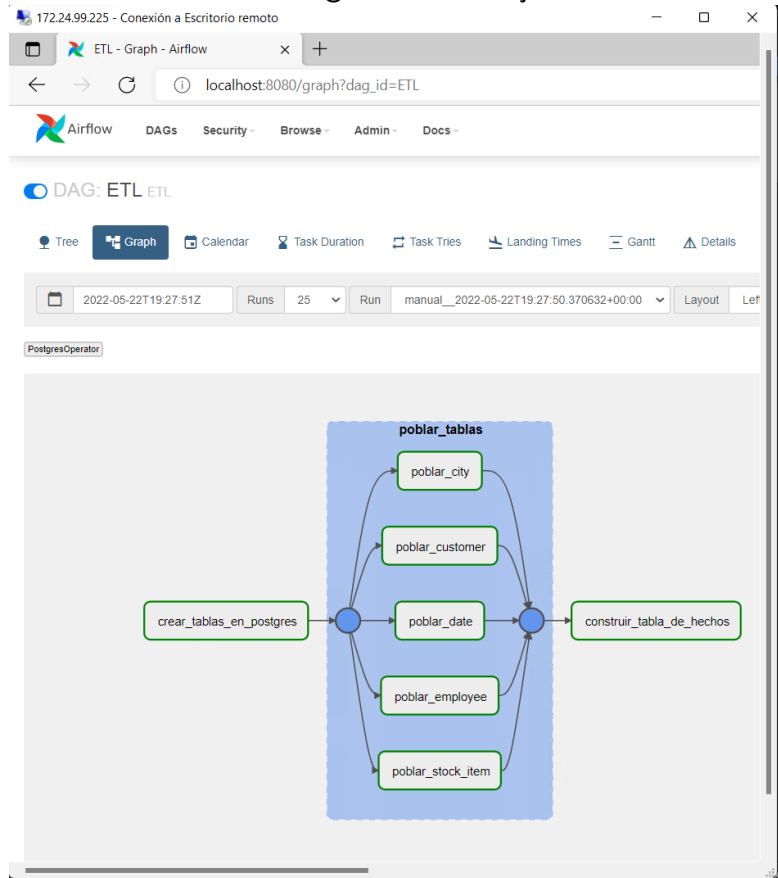
- b. Para realizar el proceso ETL, definimos el archivo "ELT.py" que contiene el código correspondiente a la implementación del DAG. Este archivo se creó en la carpeta dags

```
172.24.99.225 - Conexión a Escritorio remoto
File Edit Selection View Go Run Terminal Help
ELTpy - Airflow-Docker - Visual Studio Code
insert_queries.py ELTpy 3 X
dags > ELTpy > ...
1 # Utilidades de airflow
2 from airflow.models import DAG
3 from airflow.providers.postgres.operators.postgres import PostgresOperator
4 from airflow.utils.task_group import TaskGroup
5
6 # Utilidades de python
7 from datetime import datetime
8
9 # Funciones ETL
10 from utils.crear_tablas import crear_tablas
11 from utils.insert_queries import *
12
13
14 default_args = {
15     'owner': 'Estudiante',
16     'email_on_failure': False,
17     'email': ['estudiante@uniandes.edu.co'],
18     'start_date': datetime(2022, 5, 5) # inicio de ejecución
19 }
20
21 with DAG(
22     'ETL',
23     description='ETL', # ejecución diaria del DAG
24     schedule_interval='@daily',
25     default_args=default_args,
26     catchup=False) as dag:
27
28     # task: 1 crear las tablas en la base de datos postgres
29     crear_tablas_db = PostgresOperator(
30         task_id='crear_tablas_en_postgres',
31         postgres_conn_id='postgres_localhost', # Nótese que es el mismo ID definido en la conexión Postgres de la interfaz de Airflow
32         sql=crear_tablas()
33     )
34
35     # task: 2 poblar las tablas de dimensiones en la base de datos
36     with TaskGroup('poblar_tablas') as poblar_tablas_dimensiones:
37
38         # task: 2.1 poblar tabla city
39         poblar_city = PostgresOperator(
40             task_id='poblar_city',
41             postgres_conn_id='postgres_localhost',
42             sql=insert_query_city(csv_path = "dimension_city")
43         )
44
45         # task: 2.2 poblar tabla customer
46         poblar_customer = PostgresOperator(
47             task_id='poblar_customer',
48             postgres_conn_id='postgres_localhost',
49             sql=insert_query_customer(csv_path = "dimension_customer")
50         )
51
52         # task: 2.3 poblar tabla date
53         poblar_date = PostgresOperator(
54             task_id='poblar_date',
55             postgres_conn_id='postgres_localhost',
56             sql=insert_query_date(csv_path = "dimension_date")
57         )
58
59         # task: 2.4 poblar tabla employee
60         poblar_employee = PostgresOperator(
61             task_id='poblar_employee',
62             postgres_conn_id='postgres_localhost',
63             sql=insert_query_employee(csv_path = "dimension_employee")
64         )
65
66         # task: 2.5 poblar tabla stock item
67         poblar_stock_item = PostgresOperator(
68             task_id='poblar_stock_item',
69             postgres_conn_id='postgres_localhost',
70             sql=insert_query_stock(csv_path = "dimension_stock_item")
71         )
72
73     # task: 3 poblar la tabla de hechos
74     poblar_fact_order = PostgresOperator(
75         task_id='construir_tabla_de_hechos',
76         postgres_conn_id='postgres_localhost',
77         sql=insert_query_fact_order(csv_path = "fact_order")
78     )
79
80     # flujo de ejecución de las tareas
81     crear_tablas_db >> poblar_tablas_dimensiones >> poblar_fact_order
82
```

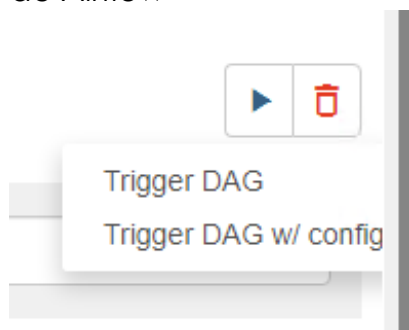
```
172.24.99.225 - Conexión a Escritorio remoto
File Edit Selection View Go Run Terminal Help
ELTpy
insert_queries.py ELTpy 3 X
dags > ELTpy > ...
38 poblar_city = PostgresOperator(
39     task_id="poblar_city",
40     postgres_conn_id="postgres_localhost",
41     sql=insert_query_city(csv_path = "dimension_city")
42 )
43
44 # task: 2.2 poblar tabla customer
45 poblar_customer = PostgresOperator(
46     task_id="poblar_customer",
47     postgres_conn_id="postgres_localhost",
48     sql=insert_query_customer(csv_path = "dimension_customer")
49 )
50
51 # task: 2.3 poblar tabla date
52 poblar_date = PostgresOperator(
53     task_id="poblar_date",
54     postgres_conn_id="postgres_localhost",
55     sql=insert_query_date(csv_path = "dimension_date")
56 )
57
58 # task: 2.4 poblar tabla employee
59 poblar_employee = PostgresOperator(
60     task_id="poblar_employee",
61     postgres_conn_id="postgres_localhost",
62     sql=insert_query_employee(csv_path = "dimension_employee")
63 )
64
65 # task: 2.5 poblar tabla stock item
66 poblar_stock_item = PostgresOperator(
67     task_id="poblar_stock_item",
68     postgres_conn_id="postgres_localhost",
69     sql=insert_query_stock(csv_path = "dimension_stock_item")
70 )
71
72
73 # task: 3 poblar la tabla de hechos
74 poblar_fact_order = PostgresOperator(
75     task_id="construir_tabla_de_hechos",
76     postgres_conn_id="postgres_localhost",
77     sql=insert_query_fact_order(csv_path = "fact_order")
78 )
79
80 # flujo de ejecución de las tareas
81 crear_tablas_db >> poblar_tablas_dimensiones >> poblar_fact_order
82
```

Note que estas funciones contienen todos los operadores de Postgres definidos (crear tablas y poblarlas todas), cada una su task definida. La definición de cada uno de estos conceptos se hará más adelante.

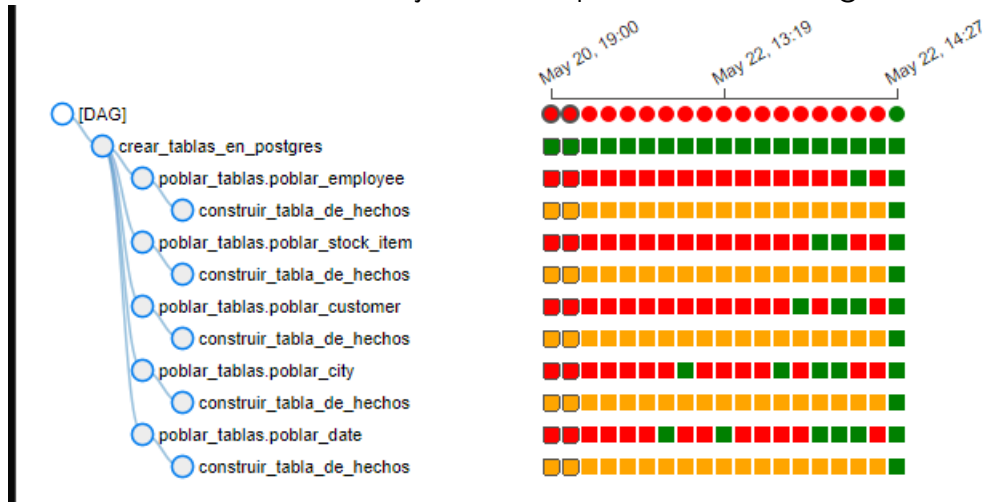
- c. Una vez finalizado este proceso de implementación, en la interfaz de airflow se ver el grafo de ejecución de la siguiente manera:



- d. Oprimimos el botón de trigger DAG que se encuentra en la interfaz de Airflow



- e. Observamos el árbol de ejecución que se ve de la siguiente manera:



7. Principales retos observados en el proceso

- a. Durante el paso de ejecutar el DAG en la interfaz web de Airflow, como se puede observar en el árbol de ejecución, tuvimos una gran cantidad de intentos fallidos en los que se presentaban errores en la mayoría de los tasks del DAG. Lo que hicimos entonces fue revisar cada uno de los Logs de los tasks que fallaron e identificar la fuente de los errores. Algunos de ellos fueron: Ya había un registro con la Primary Key que se intentaba ingresar en la base de datos, producto de la ejecución anterior del DAG, o había pasos faltantes en el preprocesamiento de los datos. Por ejemplo, tuvimos que volver al notebook de preprocesamiento para eliminar las comillas de las palabras que las tenían pues generaban errores. Igualmente agregamos el formateo correcto de los datos de fechas en las sentencias INSERT SQL pues se estaba presentando el error de que el tipo de dato esperado por la base de datos no era correcto.
- b. Uno de los errores que más tiempo nos tomó resolver fue el de establecer la conexión entre Airflow y la base de datos PostgreSQL. Lo que al final encontramos fue un problema de autenticación con las credenciales que habíamos establecido al inicio de la configuración. Reemplazamos las credenciales por unas diferentes y pudimos establecer la conexión con la base de datos.

5. Resultados

Una vez completado el proceso ETL y corregidos los errores que surgieron en la ejecución, como fue exhaustivamente descrito en la sección anterior de este documento, se obtuvo una ejecución correcta del ETL como puede verse en las imágenes del árbol (Figura 4.1) y grafo DAG (Figura 4.2). En la sección de anexos de este documento puede encontrar las imágenes donde se observa que, en efecto, el árbol y el grafo construidos corresponden a los del grupo 6.

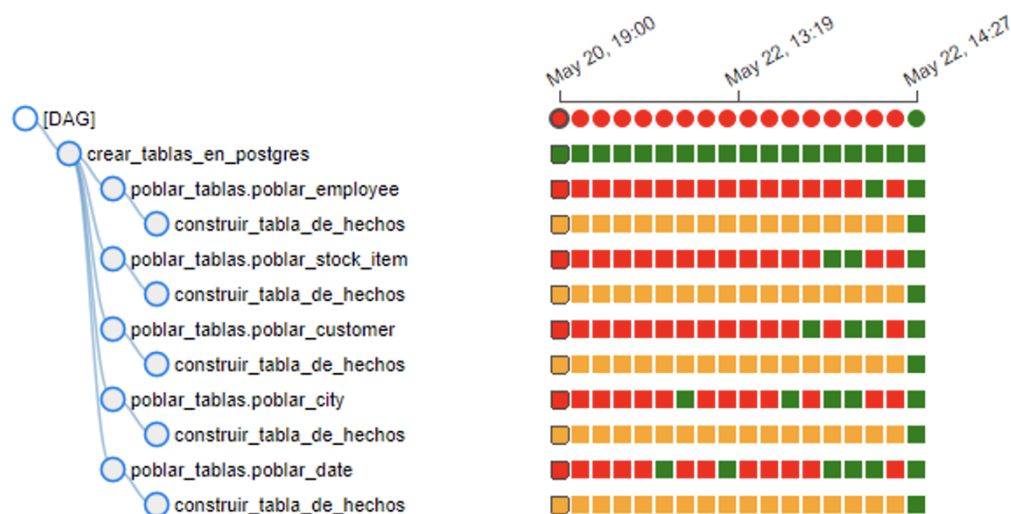


Figura 4.1: Árbol de ejecuciones de Airflow.

La figura 4.1 ilustra el árbol de ejecuciones de Airflow. Después de varios intentos y solucionando los errores que describimos en la sección anterior de este documento, logramos crear todas las tablas, poblarlas y construir la tabla de hechos, como se ve en la última ejecución del árbol.

Por su parte, la figura 4.2 muestra el grafo (ya en verde) que fue generado como resultado de una ejecución exitosa en Airflow. Podemos ver que se genera un DAG (Grafo dirigido acíclico), donde primero se crean las tablas, luego son pobladas y, finalmente, se construye la tabla de hechos con las tablas ya pobladas.

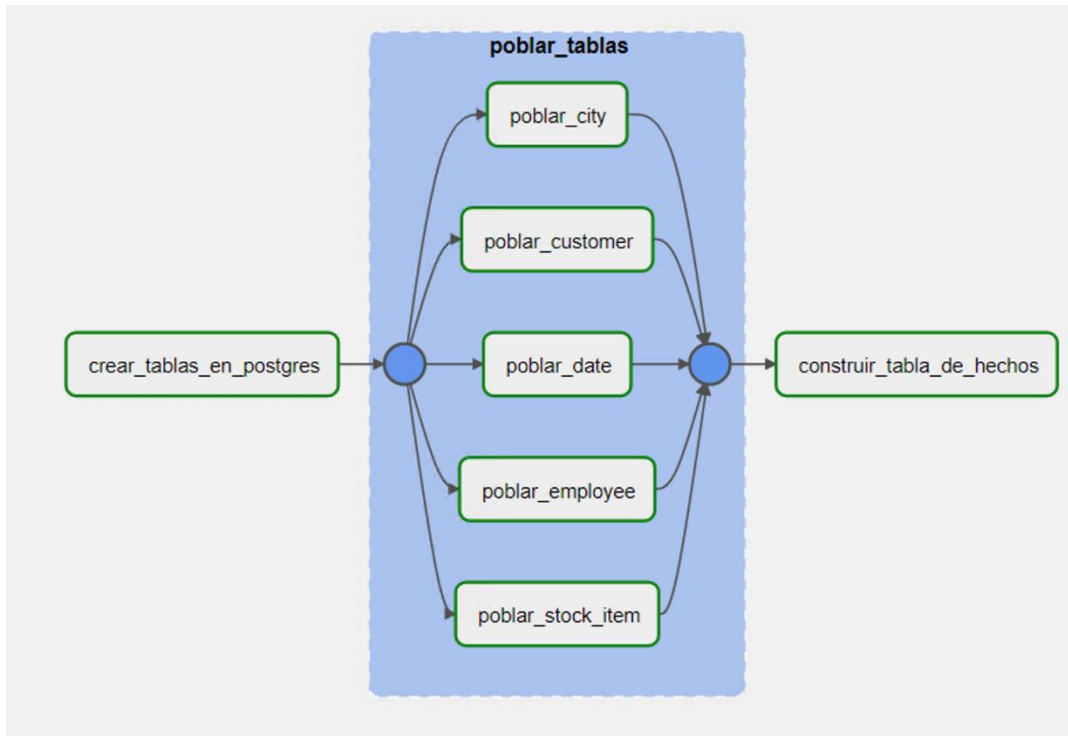


Figura 4.2: Grafo de ejecución DAG en Airflow.

Con el fin de revisar que los datos se cargaron correcta y completamente, se realizaron algunas sentencias SQL sobre la base de datos de PostgreSQL. Para ello, se revisó que la cantidad de filas de cada tabla coincidiera con la cantidad de filas de cada uno de los archivos csv limpios (esto, con el fin de evaluar completitud). A su vez, se realizaron algunas consultas de prueba sobre algunos registros de cada tabla para determinar su correctitud. Es importante notar que, en todas las capturas de pantalla, puede apreciarse que corresponde a lo realizado por el grupo 6. Asimismo, el archivo con las sentencias lo encuentra en: https://github.com/alvarodpm/BI_Lab5/blob/main/queries.sql

- **Tabla Dimensión City:**

De acuerdo con el perfilamiento de los datos, y tras su preprocesamiento, hay 97 filas en el archivo. Para probar la cantidad de filas cargadas en la tabla City de PostgreSQL, se realizó la consulta de la figura 4.3, cuyo resultado puede verse en la caja roja en la esquina inferior izquierda. Note que la cantidad de filas es equivalente, por lo que podemos afirmar que los datos se cargaron completamente.

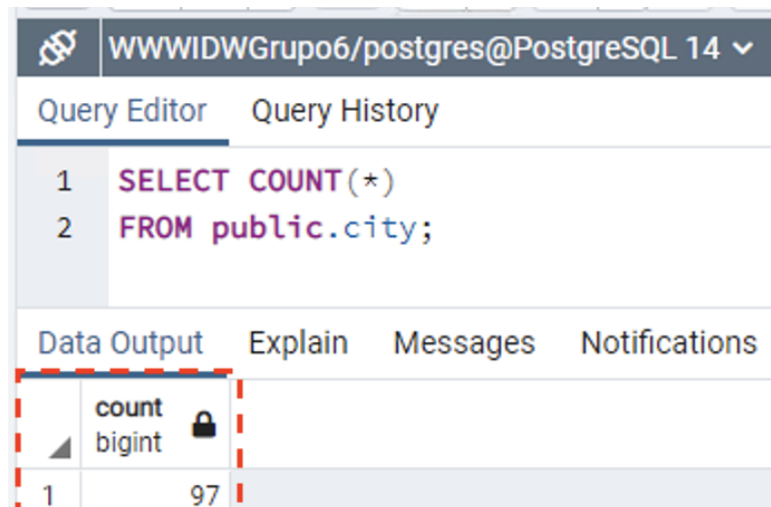


Figura 4.3: Query en PostgreSQL para contar la cantidad de filas de la tabla City.

Ahora, seleccionaremos todas las ciudades que pertenecen al estado de California:

The screenshot shows the PostgreSQL Query Editor interface. The query editor contains the following SQL query:

```
1 SELECT *
2 FROM public.city
3 WHERE state_province='California';
```

The Data Output tab is selected, showing the following result:

city_key [PK] integer	city character varying (150)	state_province character varying (150)	country character varying (150)	continent character varying (150)	sales_territory character varying (150)	region character varying (150)	subregion character varying (150)	latest_recorded_population integer
18	Carson	California	United States	North America	Far West	Americas	Northern America	91714
27	Cartago	California	United States	North America	Far West	Americas	Northern America	92
36	El Centro	California	United States	North America	Far West	Americas	Northern America	42598
37	El Cerrito	California	United States	North America	Far West	Americas	Northern America	23549
43	El Dorado Hills	California	United States	North America	Far West	Americas	Northern America	42108
46	El Granada	California	United States	North America	Far West	Americas	Northern America	5467
52	El Monte	California	United States	North America	Far West	Americas	Northern America	113475
55	El Nido	California	United States	North America	Far West	Americas	Northern America	330
62	El Paso de Robles	California	United States	North America	Far West	Americas	Northern America	29793
63	El Portal	California	United States	North America	Far West	Americas	Northern America	474
75	El Rio	California	United States	North America	Far West	Americas	Northern America	7198
78	El Segundo	California	United States	North America	Far West	Americas	Northern America	16654
79	El Sobrante	California	United States	North America	Far West	Americas	Northern America	12669
81	El Toro	California	United States	North America	Far West	Americas	Northern America	0
84	El Verano	California	United States	North America	Far West	Americas	Northern America	4123

Figura 4.4: Query en PostgreSQL para conocer todas las ciudades pertenecientes al estado de California.

Podemos ver que hemos obtenido todas las ciudades correspondientes al estado de California, y que todos los datos son correctos y completos.

- **Tabla dimensión Customer**

De nuevo, revisamos que haya 402 datos, como se encontró en el perfilamiento y en el preprocesamiento:

WWWIDWGrupo6/postgres@PostgreSQL 14	
Query Editor Query History	
1	SELECT COUNT(*)
2	FROM public.customer;
Data Output Explain Messages Notifications	
	count bigint
1	402

Figura 4.5: Query en PostgreSQL para contar la cantidad de registros de la tabla Customer.

Efectivamente, la cantidad de filas coincide. Asimismo, revisamos los registros asociados al código postal 90683 que, de acuerdo con el Pandas profiling, es el que más filas asociadas tiene:

WWWIDWGrupo6/postgres@PostgreSQL 14	
Query Editor Query History	
1	SELECT *
2	FROM public.customer
3	WHERE postal_code=90683;
Data Output Explain Messages Notifications	
	customer_key [PK] integer customer character varying (150) bill_to_customer character varying (150) category character varying (150) buying_group character varying (150) primary_contact character varying (150) postal_code integer
1	46 Tailspin Toys (Jemison- AL) Tailspin Toys (Head Office) Novelty Shop Tailspin Toys Didem ozCelik 90683
2	241 Wingtip Toys (Asher- OK) Wingtip Toys (Head Office) Novelty Shop Wingtip Toys Kadir Usenuly 90683
3	355 Wingtip Toys (Hollandsburg- IN) Wingtip Toys (Head Office) Novelty Shop Wingtip Toys Ondrej Tomek 90683
4	369 Wingtip Toys (Ovilla- TX) Wingtip Toys (Head Office) Novelty Shop Wingtip Toys Linda Ledezma 90683
5	383 Wingtip Toys (Montoya- NM) Wingtip Toys (Head Office) Novelty Shop Wingtip Toys Pavel Bohuslav 90683

Figura 4.6: Query en PostgreSQL para conocer los registros asociados al código postal 90683.

Vemos que los datos son completos y correctos.

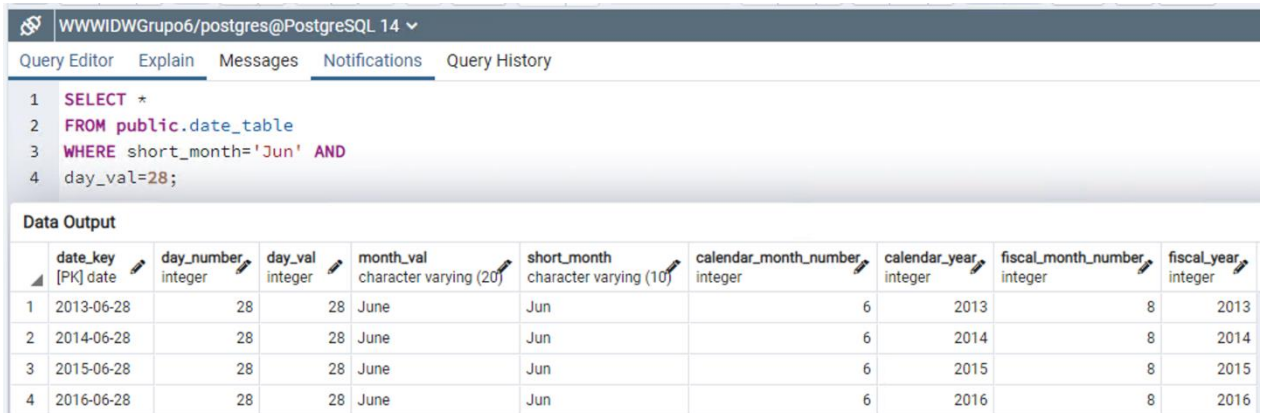
- **Tabla dimensión Date:**

Lo primero que revisamos es que haya 1461 datos, correspondientes a la totalidad de días que hay en 4 años (incluido el año bisiesto – 2016). Esto se cumple, como se ve en la figura 4.7:

WWWIDWGrupo6/postgres@PostgreSQL 14	
Query Editor Query History	
1	SELECT COUNT(*)
2	FROM public.date_table;
Data Output Explain Messages Notifications	
	count bigint
1	1461

Figura 4.7: Query en PostgreSQL para contar la cantidad de registros de días.

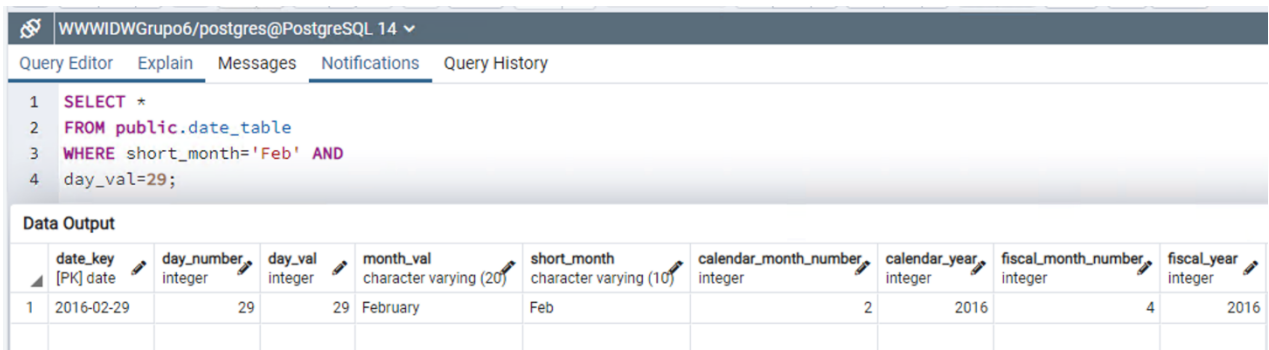
Esperemos entonces que, por cada día del año, haya 4 registros (correspondientes a 4 años), como se ve en la figura 4.8, excepto para el 29 de febrero, para el que esperamos 1 solo registro, como se ve en la figura 4.9:



```
1 SELECT *
2 FROM public.date_table
3 WHERE short_month='Jun' AND
4 day_val=28;
```

	date_key [PK] date	day_number integer	day_val integer	month_val character varying (20)	short_month character varying (10)	calendar_month_number integer	calendar_year integer	fiscal_month_number integer	fiscal_year integer
1	2013-06-28	28	28	June	Jun	6	2013	8	2013
2	2014-06-28	28	28	June	Jun	6	2014	8	2014
3	2015-06-28	28	28	June	Jun	6	2015	8	2015
4	2016-06-28	28	28	June	Jun	6	2016	8	2016

Figura 4.8: Query en PostgreSQL para ver los registros asociados al 28 de junio de los 4 años.



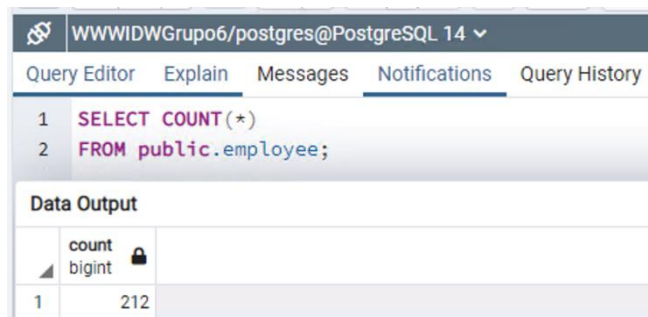
```
1 SELECT *
2 FROM public.date_table
3 WHERE short_month='Feb' AND
4 day_val=29;
```

	date_key [PK] date	day_number integer	day_val integer	month_val character varying (20)	short_month character varying (10)	calendar_month_number integer	calendar_year integer	fiscal_month_number integer	fiscal_year integer
1	2016-02-29	29	29	February	Feb	2	2016	4	2016

Figura 4.9: Query en PostgreSQL para ver los registros asociados al 29 de febrero de los 4 años. Como vemos, solo hay un registro, correspondiente al año bisiesto 2016.

- **Tabla dimensión Employee:**

Primero, revisamos que, en efecto, haya 212 registros, como se observó en la fase de perfilamiento y preprocesamiento. Esta confirmación se evidencia en la figura 4.10:

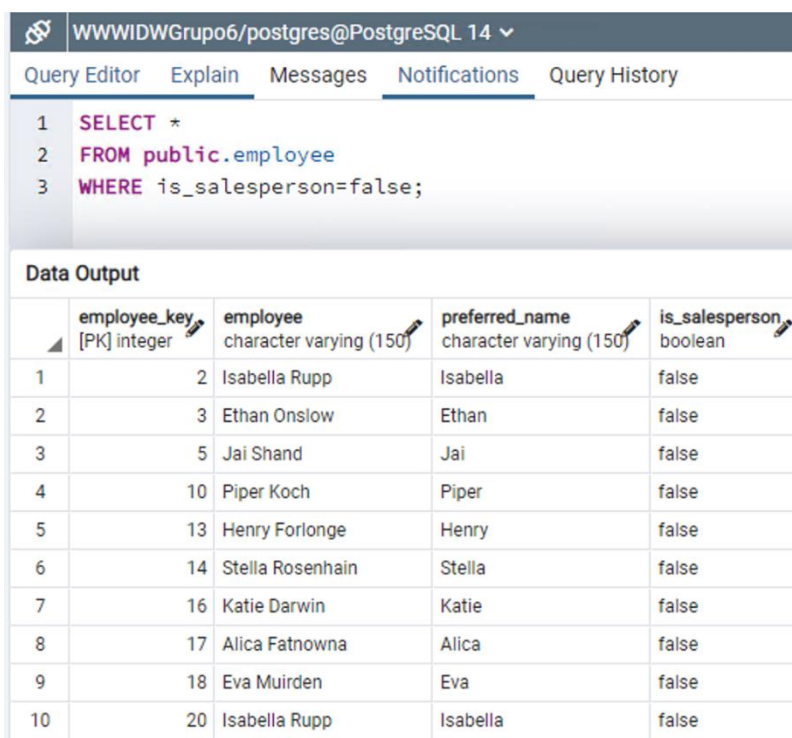


```
1 SELECT COUNT(*)
2 FROM public.employee;
```

	count bigint
1	212

Figura 4.10: Query en PostgreSQL para ver la cantidad de registros de la tabla Employee.

Asimismo, con el fin de revisar la correctitud de los datos cargados, veamos, por ejemplo, los 10 primeros registros asociados a aquellos empleados que no son vendedores (i.e. donde el atributo `is_salesperson=false`). El resultado puede verse en la figura 4.11:



The screenshot shows a PostgreSQL Query Editor interface. At the top, the title bar reads 'WWWIDWGrupo6/postgres@PostgreSQL 14'. Below the title bar are tabs for 'Query Editor', 'Explain', 'Messages', 'Notifications', and 'Query History'. The 'Query Editor' tab is active, displaying a SQL query:

```
1 SELECT *
2 FROM public.employee
3 WHERE is_salesperson=false;
```

Below the query editor is a section titled 'Data Output' which displays a table of results. The table has four columns: 'employee_key' (integer), 'employee' (character varying (150)), 'preferred_name' (character varying (150)), and 'is_salesperson' (boolean). The table contains 10 rows of data, numbered 1 through 10 in the first column.

	employee_key [PK] integer	employee character varying (150)	preferred_name character varying (150)	is_salesperson boolean
1	2	Isabella Rupp	Isabella	false
2	3	Ethan Onslow	Ethan	false
3	5	Jai Shand	Jai	false
4	10	Piper Koch	Piper	false
5	13	Henry Forlonge	Henry	false
6	14	Stella Rosenhain	Stella	false
7	16	Katie Darwin	Katie	false
8	17	Alica Fatnowna	Alica	false
9	18	Eva Muirden	Eva	false
10	20	Isabella Rupp	Isabella	false

Figura 4.11: Query en PostgreSQL para ver los resultados de los 10 primeros empleados que no son vendedores.

- **Tabla dimensión Stock item:**

La última tabla de la base de datos que fue poblada con los csv fue Stock item. Esta fue la que más transformaciones y preprocesamiento requirió. Se esperaban 671 registros, que fue la misma cantidad cargada en la base de datos por el proceso ETL, como se ve en la figura 4.12:

WWWIDWGrupo6/postgres@PostgreSQL 14	
Query Editor	Explain Messages Notifications Query History
1	SELECT COUNT(*)
2	FROM public.stockitem;
Data Output	
	count bigint
1	671

Figura 4.12: Query en PostgreSQL para contar la cantidad de registros de la tabla de Stock item.

Asimismo, para revisar la correctitud, tomamos los 8 primeros resultados que resultan al visualizar los registros con el atributo `Is_chiller_stock=true`. Estos se pueden ver en la figura 4.13:

WWWIDWGrupo6/postgres@PostgreSQL 14

Query Editor

Explain

Messages

Notifications

Query History

1

2

3

SELECT *

FROM public.stockitem

WHERE Is_chiller_stock=true;

Data Output

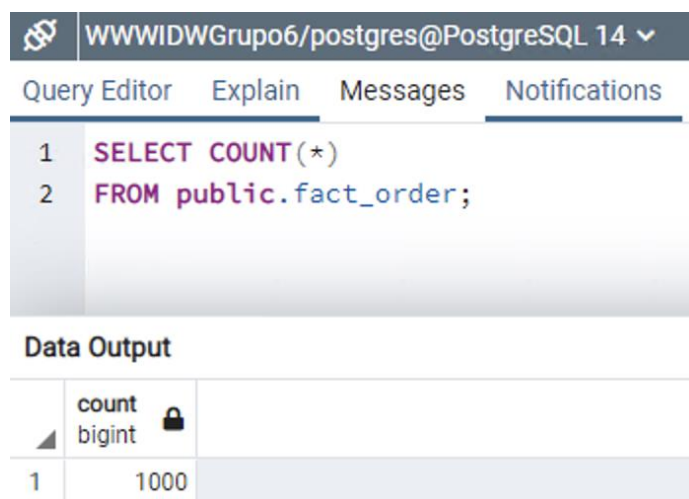
	stock_item_key [PK] integer	stock_item character varying (200)	color character varying (50)	selling_package character varying (50)	buying_package character varying (50)	brand character varying (50)	size_val character varying (50)
1	220	Novelty chilli chocolates 500g	nan	Bag	Carton	nan	500g
2	221	Novelty chilli chocolates 250g	nan	Bag	Carton	nan	250g
3	222	Chocolate echidnas 250g	nan	Bag	Carton	nan	250g
4	223	Chocolate beetles 250g	nan	Bag	Carton	nan	250g
5	224	Chocolate sharks 250g	nan	Bag	Carton	nan	250g
6	225	Chocolate frogs 250g	nan	Bag	Carton	nan	250g
7	226	White chocolate moon rocks 250g	nan	Bag	Carton	nan	250g
8	227	White chocolate snow balls 250g	nan	Bag	Carton	nan	250g

lead_time_days integer	quantity_per_outer integer	is_chiller_stock boolean	tax_rate numeric	unit_price numeric	recommended_retail_price numeric	typical_weight_per_unit numeric
3	12	true	3.0	14.5	20.74	0.5
3	24	true	3.0	8.55	12.23	0.25
3	24	true	3.0	8.55	12.23	0.25
3	24	true	3.0	8.55	12.23	0.25
3	24	true	3.0	8.55	12.23	0.25
3	24	true	3.0	8.55	12.23	0.25
3	24	true	3.0	8.55	12.23	0.25
3	24	true	3.0	8.55	12.23	0.25

Figura 4.13: Query en PostgreSQL para ver los primeros registros con el atributo `Is_chiller_stock = true`. Debido a que esta tabla tiene muchas columnas, la imagen se encuentra partida en dos.

- **Tabla dimensión Fact order:**

Después de crear y poblar todas las tablas, el proceso ETL crea y puebla la tabla de hechos. Este es el último paso del proceso. La tabla de hechos tiene mil registros, como se ve en la figura 4.13:



The screenshot shows the PostgreSQL Query Editor interface. The query editor contains the following SQL query:

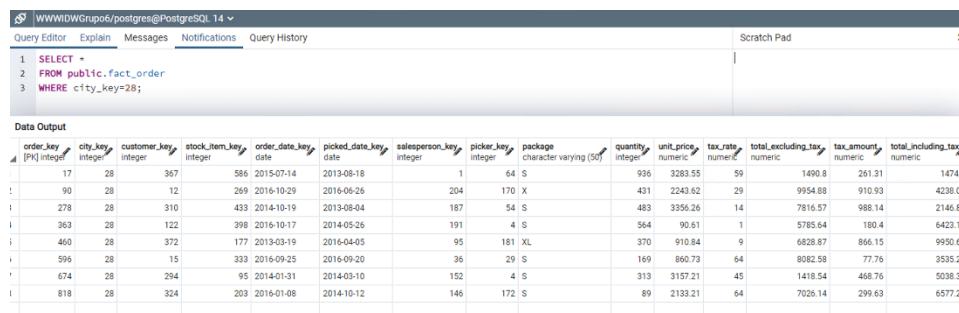
```
1 SELECT COUNT(*)
2 FROM public.fact_order;
```

Below the query editor, the 'Data Output' section shows the result of the query:

	count bigint
1	1000

Figura 4.14: Query en PostgreSQL para contar la cantidad de registros de la tabla de hechos.

Para visualizar la correctitud de esta tabla, veamos por ejemplo los registros asociados a la ciudad con id 28, como muestra la figura 4.15:



The screenshot shows the PostgreSQL Query Editor interface. The query editor contains the following SQL query:

```
1 SELECT *
2 FROM public.fact_order
3 WHERE city_key=28;
```

Below the query editor, the 'Data Output' section shows the result of the query. The table has 14 columns: order_key, city_key, customer_key, stock_item_key, order_date_key, picked_date_key, salesperson_key, picker_key, package, quantity, unit_price, tax_rate, total_excluding_tax, tax_amount, and total_including_tax. The data is filtered for city_key=28.

order_key	city_key	customer_key	stock_item_key	order_date_key	picked_date_key	salesperson_key	picker_key	package	quantity	unit_price	tax_rate	total_excluding_tax	tax_amount	total_including_tax
17	28	367	586	2015-07-14	2013-08-18		1	64 S	936	3283.55	59	1490.8	261.31	1474.3
90	28	12	269	2016-10-29	2016-09-26	204	170 X		431	2243.62	29	9954.88	910.93	4238.04
278	28	310	433	2014-10-19	2013-08-04	187	54 S		483	3356.26	14	7816.57	988.14	2146.81
363	28	122	398	2016-10-17	2014-05-26	191	4 S		564	90.61	1	5785.64	180.4	6423.15
460	28	372	177	2013-03-19	2016-04-05	95	181 XL		370	910.84	9	6828.87	866.15	9950.61
595	28	15	333	2016-09-25	2016-09-20	36	29 S		169	860.73	64	8062.58	77.76	3535.29
674	28	294	95	2014-01-31	2014-03-10	152	4 S		313	3157.21	45	1418.54	468.76	5038.37
818	28	324	203	2016-01-08	2014-10-12	146	172 S		89	2133.21	64	7026.14	299.63	6577.24

Figura 4.14: Query en PostgreSQL para ver las filas de la tabla de hechos asociadas a la ciudad 28.

Como vemos, todas las tablas fueron correctamente pobladas y las filas de la tabla de hechos tienen sentido con lo visto previamente y la información cargada en las otras tablas.

Preguntas:

1. Explique a fondo los siguientes conceptos de airflow: Task, Operator, DAG.
 - Task: es la unidad básica de ejecución en Airflow. El orden de estas tareas se organiza en DAGs (concepto que se explica más adelante). Las tasks tienen dependencias entre ellas para expresar el orden de ejecución.

- Operator: Es una plantilla para una tarea (task) predefinida, esta declaración puede ir dentro del DAG. Existen varios tipos de operadores, como los que se ven en la figura a continuación:

- **SimpleHttpOperator**
- **MySQLOperator**
- **PostgresOperator**
- **MsSqlOperator**
- **OracleOperator**
- **JdbcOperator**
- **DockerOperator**
- **HiveOperator**
- **S3FileTransformOperator**
- **PrestoToMySQLOperator**
- **SlackAPIOperator**

Algunos operadores admitidos en Airflow. Tomado de [2].

En en caso particular de este laboratorio, se usó el operador PostgresOperator.

- DAG: es un grafo acíclico dirigido, esto quiere decir que es una manera lógica de organizar las tareas, donde cada una tiene sus relaciones y dependencias (por ejemplo, para hacer la tarea 2, se debe completar la tarea 1). El hecho de que se acíclico significa que no se puede ejecutar dos veces una tarea. El DAG dicta la forma en la que deben ejecutarse las tareas (tasks). Un ejemplo básico de un DAG, de acuerdo con la documentación de Airflow, se puede ver a continuación:



Ejemplo de DAG. Tomado de [3].

2. ¿Por qué se utiliza el comando "IF NOT EXISTS" en la sentencia de creación de tablas, en el contexto del proceso de ETL?

Principalmente, este comando se realiza para que no se vuelva a crear la tabla si ya existe. En la documentación de PostgreSQL la sentencia CREATE TABLE crea una nueva vista de la BD. Ahora, en caso de que la sentencia IF NOT EXIST no estuviera, generará un error. Este error impedirá que esta tarea finalice y como consecuencia, las demás tareas, que dependen de esta no podrán correr, por lo que el flujo se detendría en su totalidad.

En otras palabras, IF NOT EXISTS lo que busca es crear la tabla en caso de que esta no exista. En caso de exista, impide que se lance un error (pasando por alto la creación de la tabla) y permitiendo que continúe el flujo de ejecución.

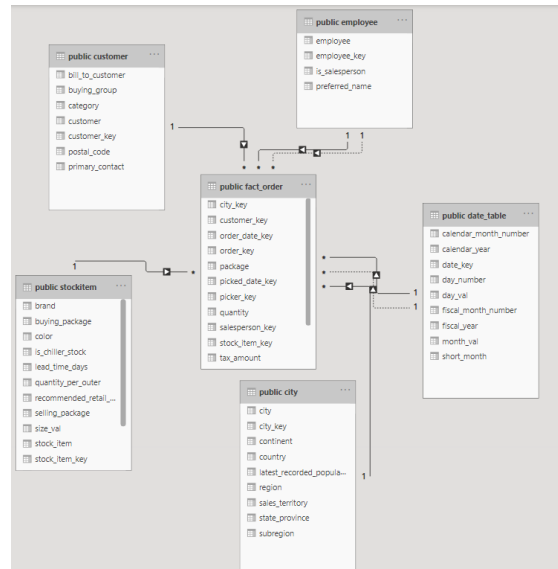
3. ¿Por qué para la columna de día se utiliza el nombre "day_val" y no "day"?

Para facilitar el análisis, es recomendable que la dimensión fecha pueda ser separada en diferentes atributos, por ejemplo, día, mes, día de la semana, etc. Un posible análisis que puede requerir el cliente es que desee saber en qué mes se vende menos, o qué día registra las mayores ventas. Con esto, nombrar este atributo (columna) solo como day, puede llegar a generar confusiones sobre los datos guardados en este campo.

Adicionalmente, la palabra day es una palabra reservada en SQL, por lo que, si se llega a colocar este nombre en la columna de la tabla, generará un error ya que se confundiría con la función DAY, la cual recibe un String con la fecha.

4. ¿De dónde se obtiene la información sobre las columnas que hay que crear en la tabla?

Esta información se obtiene principalmente del modelo multidimensional que WWI desea obtener, el cual puede verse en la imagen a continuación.



Ejemplo de modelo multidimensional. Tomado de [4].

Adicional a esto, los tipos de datos y en sí, los datos que irían en estas columnas se obtienen directamente de los csv (luego del preprocesamiento) que nos brindó la empresa.

5. ¿Por qué es necesario un flujo de ejecución de las tareas en Airflow?

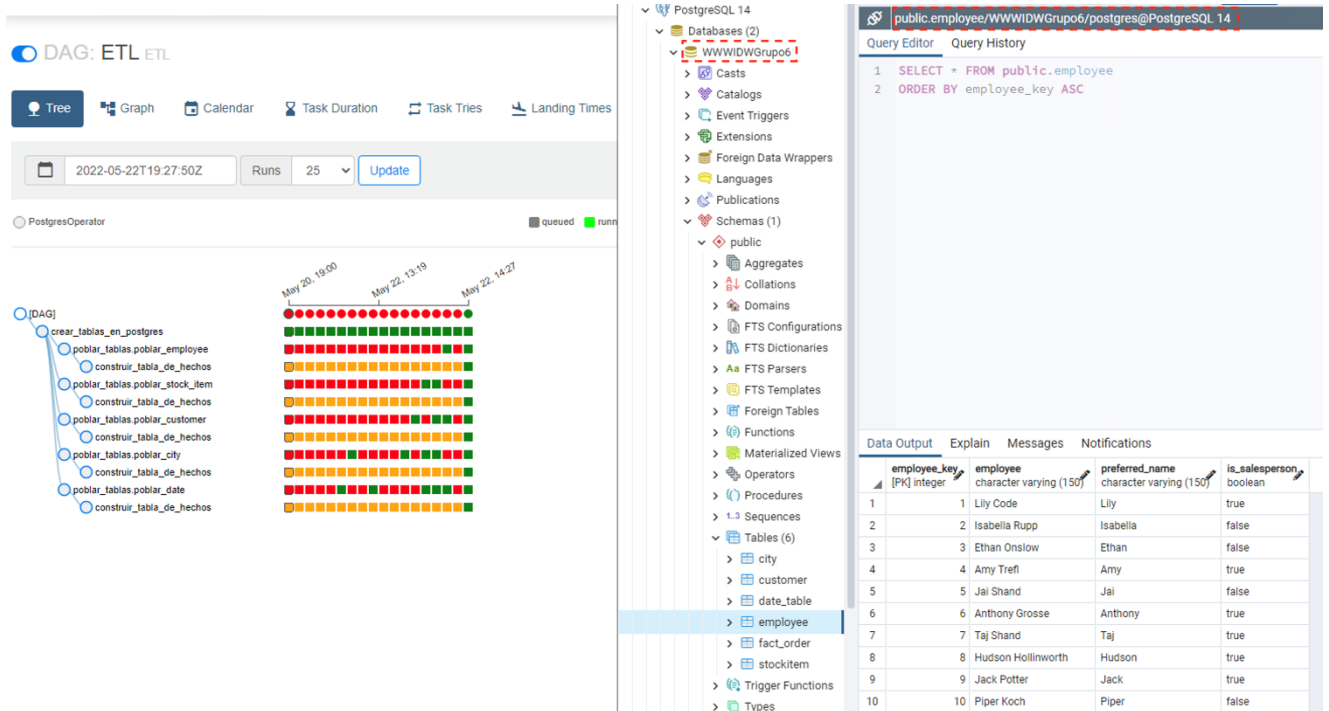
Airflow, al ser una herramienta para la automatización y planificación del trabajo, ayuda en el manejo de datos que se obtienen del sistema transaccional. El data Warehouse va a ejecutar cada una de las tareas para procesar la información que se establezca. Este es un proceso automático, por lo que, el usuario o la empresa deja establecido el lapso de tiempo para volver a ejecutar la tarea de manera automática, evitando así errores humanos. En este flujo de ejecución de las tareas se puede ver si alguna de ellas envió algún error, almacena la información e incluso se puede llegar a hacer el manejo de historias si así lo solicita el cliente.

5. BONO:

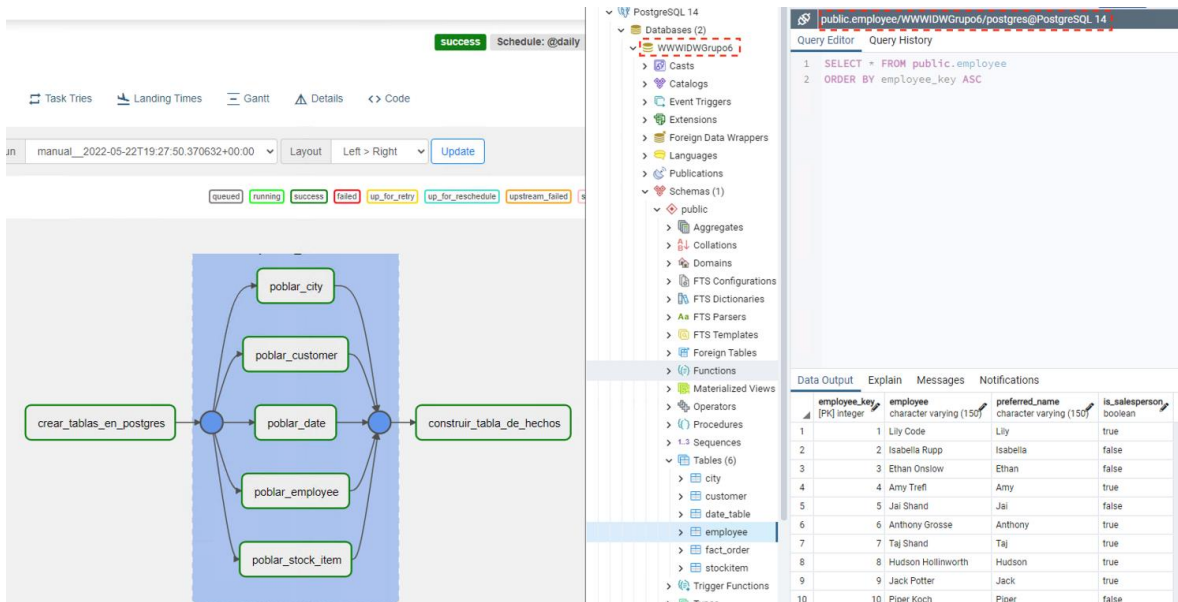
Para ver la implementación del bono, con sus respectivos archivos y ejecución, remítase al siguiente repositorio: https://github.com/sofiaalvarezlopez/BI_bono_lab5. Decidimos hacerlo aparte de esta entrega para no dañar el trabajo ya realizado.

6. Anexos

Resultado del árbol de ejecución donde a la derecha puede encontrarse que corresponde a la base de datos del grupo 6:



Resultado del grafo de ejecución donde a la derecha puede encontrarse que corresponde a la base de datos del grupo 6:



7. Bibliografía

[1] MLA (7th ed.) Kimball, Ralph, and Margy Ross. The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling. New York: Wiley, 2002.

[2] <https://airflow.apache.org/docs/apache-airflow/stable/concepts/operators.html>

[3] <https://airflow.apache.org/docs/apache-airflow/stable/concepts/dags.html>

[4] <https://gitlab.virtual.uniandes.edu.co/ISIS3301/laboratorios/blob/patch-3/202210/Laboratorio%205/enunciado.md>