DWH project report

Members:

Shahmeer Khan - 25156

Uzair Nadeem - 24928

Schema URL: https://github.com/GziXnine/Hospital_Management_System

Github URL: https://github.com/shahmeerkm10/DWH_project

Video URL: https://youtu.be/DaL45NUHpUc?si=cpU6hLxrvlSXqgdT

Base Idea:

We opted for a healthcare dataset warehouse to define a data mart.

Tech stack:

- **Airflow** for pipelining
- **Python** for scripts
- **Colab** for defining scripts via python notebooks
- **WSL Ubuntu** via Windows Powershell for managing Airflow
- **Snowflake** for DB and dashboarding

Rough Work:

This section contains images of the rough work conducted when deciding our approach to the warehousing problem.

Date:
M T W T F S S

DWH Project

DWH Rough Work

Uzair Nadeem
24928

Problem Statement: Analysis of appointments held at the hospital where patients visit doctors and are prescribed medicines.

Shahmeer Khan
25156

Grain: One row of the fact table ~~x~~ represents information about one appointment taken by one patient from a doctor in which they were prescribed a specific medicine.

Useful Tables
- Patients
- Doctors
- Pharmacy
- Prescription
- ~~room assignment~~
- ~~rooms~~
- medicines
- medical records
- medical records medicine - Dosage column from this not to be used ~~for~~ to get dosage as a fact. Potentially used to make a dimension table attribute
- ~~Am~~ Appointment
- Billing

Date:
M T W T F S S

Possible Dimensions

Patients

Doctors

Appointments

~~Medic~~ Prescription - will also have medicine and pharmacy attributes

Date - Appointment date (will have date hierarchy

Medical Record - Diagnosis, Treatment, and info from medical records medicine.

Potential Facts

Appointment amount

Medicine Quantity

Medicine Amount (~~$~~Unit price ~~$~~ x Quantity)

~~B~~ Dosage

Frequency_in_days

Date:
M T W T F S S

Schema Selection:

Select online schema related to healthcare and extract relevant tables from it to create our own schema.

Data Preparation

. Create generalized functions to populate df columns and make dfs for each table and apply the functions ~~&~~ on the dfs.

~~Data~~ Creating Problems in the data:

. Create generalized functions for problems and apply them on df columns one by one.

The following functions will be [redacted] created:

- fill column with missing values
- fill column values with special characters (textual columns)
- ~~&~~ add duplicate rows
- change date column's date formatting
- function that changes a specific value in a textual column to ~~som~~ a different value to add ~~&~~ spelling mistakes in certain values.

Rough work

Date: M T W T F S S

Dat~~a~~a Cleaning:

- Clean dfs relevant for our star schema step by step.
- Generalized functions for ~~remov~~:
  - removing special characters
  - Date format issue → Two functions → Convert to datetime and fix formatting.
  - Function to fill missing values with "N/A" in cases where no other value can be filled.
  - Function to count ~~missing values~~ values with special characters in a column → Diff functions for numerical columns (such as phone number)
  - Function to remove special characters from the columns

~~In D~~

Dimensional Modelling:

- Use cleaned dfs to build dimension table dfs and fact table df.

Date: M T W T F S S

Dimension Table Attributes

Patient: Patient ID, Name, dob, age, gender, contact no, address, email, blood type

Doctor: Doctor ID, Name, Contact No, Email, Position, ~~Started At~~ Specialty, Experience In Years

Experience In Years will be calculated using ~~starting~~ Started At column.

Appointment: Appointment ID, Purpose, Status, Appointment hour, Appointment Part of Day

Appointment hour and part of day will be added using Appointment Time Column.

Prescription: Prescription ID, Medication Name, Dosage, Frequency, Duration (Days), ~~quanti~~ Medication Brand, Medication Type

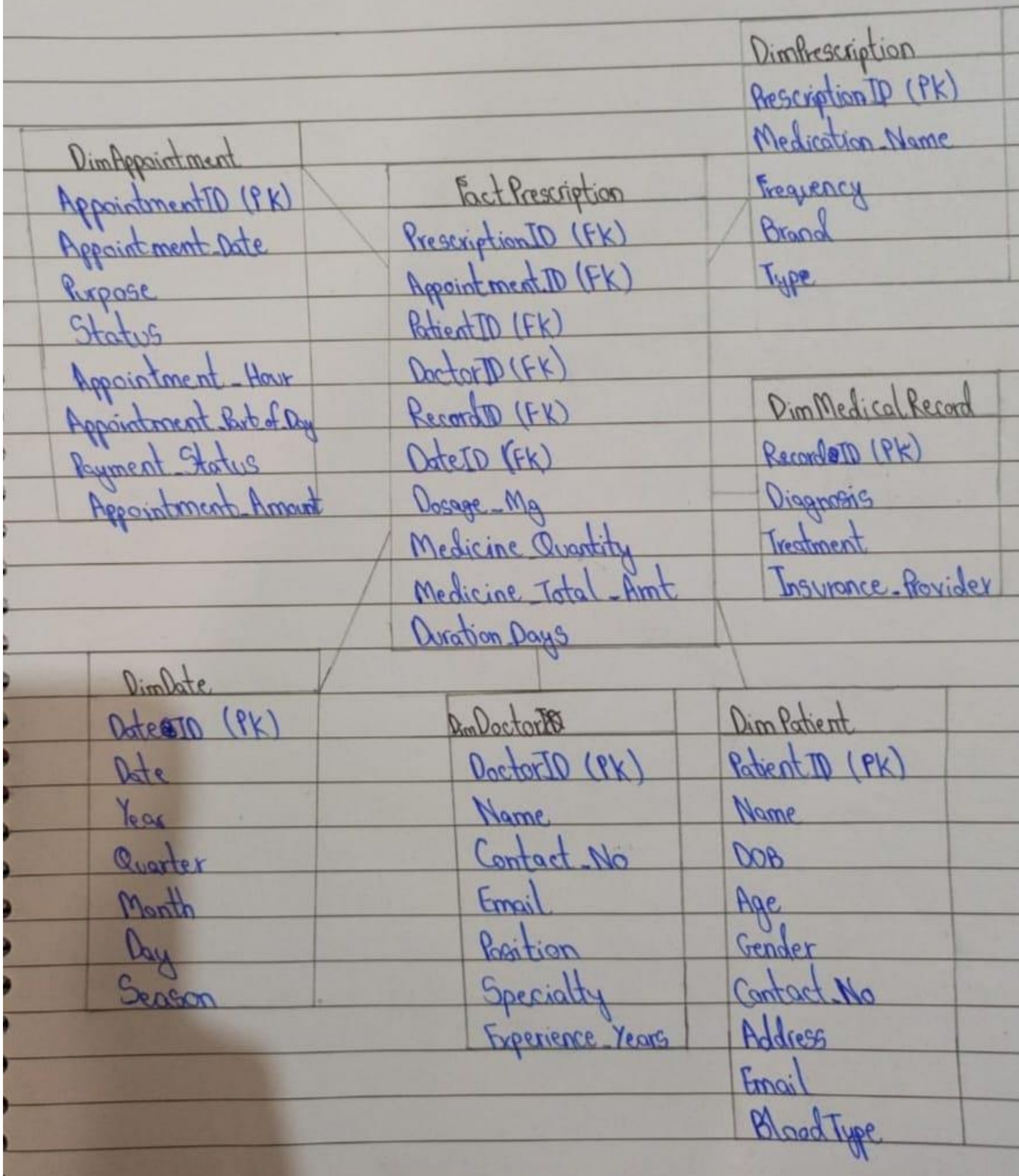
~~Use prescri~~

Date: Date ID, Year, ~~Month~~, Quarter, Month, Day

Appointment Date will be used to fill the date dimension table.

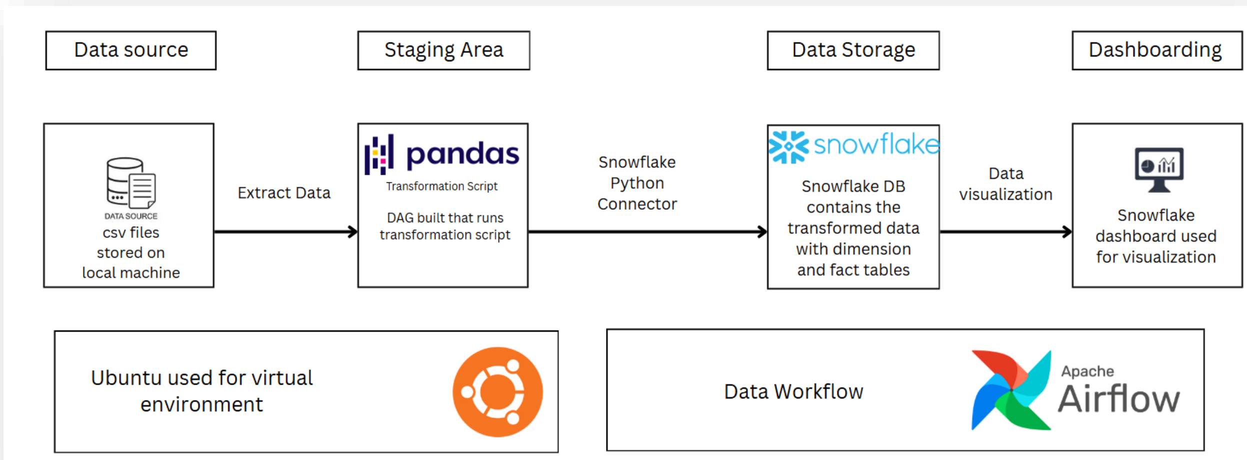
Medical Records: Record ID, Diagnosis, Treatment, Medicine Names, Dosage

Star Schema:



DimPrescription
PrescriptionID (PK)
Medication_Name
Frequency
Brand
Type
DimAppointment
AppointmentID (PK)
Appointment_Date
Purpose
Status
Appointment_Hour
Appointment_Part of Day
Payment_Status
Appointment_Amount
FactPrescription
PrescriptionID (FK)
AppointmentID (FK)
PatientID (FK)
DoctorID (FK)
RecordID (FK)
DateID (FK)
Dosage_Mg
Medicine Quantity
Medicine_Total_Amt
Duration Days
DimMedicalRecord
RecordID (PK)
Diagnosis
Treatment
Insurance_Provider
DimDate
DateID (PK)
Date
Year
Quarter
Month
Day
Season
DimDoctor
DoctorID (PK)
Name
Contact_No
Email
Position
Specialty
Experience_Years
DimPatient
PatientID (PK)
Name
DOB
Age
Gender
Contact_No
Address
Email
BloodType

System Architecture Diagram:



Data Generation:

Tables were populated via Faker library; Schema was defined using the ERD in the Schema URL above (not all tables were used).

Faker was used for majority columns; some were given arrays like blood types and faker was made to choose random values from the array to populate the relevant table.

```
[ ]  import pandas as pd
     from faker import Faker
     import random
     from datetime import datetime

     fake = Faker()

     def generate_patient_data(num_records):

         blood_types = ['A+', 'A-', 'B+', 'B-', 'AB+', 'AB-', 'O+', 'O-']

         data = []
         for _ in range(num_records):
             first_name = fake.first_name()
             last_name = fake.last_name()
             full_name = f"{first_name} {last_name}"
             dob = fake.date_of_birth(minimum_age=1, maximum_age=100)
             age = (datetime.now().date() - dob).days // 365

             # Generate name-matching email
             email_pattern = random.choice([
                 f"{first_name[0]}{last_name}@example.com",
                 f"{first_name}.{last_name}@example.com",
                 f"{first_name}_{last_name}@example.com",
                 f"{first_name}{last_name[:3]}@example.com"
             ]).lower().replace(" ", "")
```

Across tables a few were given bonus rows like the 3k here for patients, these will late be used as additional data to update the DB and dashboards.

```
# Generate 15000 patient records (3k for addition)
patients_df = generate_patient_data(15000)

# Display sample
print(patients_df.head())
```

In some cases, like appointments table, which depended on doctor and patient ids to generate appointments, these ids were taken from the relevant tables. This approach was implemented across all tables that had foreign key dependencies.

```
data = []
for _ in range(num_records):
    # Generate realistic datetime (future appointments more likely)
    appt_date = fake.date_between(start_date='-6m', end_date='+3m')
    appt_time = fake.time(pattern='%H:%M', end_datetime=None)

    # 80% chance status is Completed for past appointments
    if appt_date < datetime.now().date():
        status = "Completed" if random.random() > 0.2 else random.choice(["Cancelled", "No-show"])
    else:
        status = random.choice(["Scheduled", "Rescheduled"])

    data.append({
        'Appointment id': fake.unique.random_number(digits=8),
        'Patient id': random.choice(patient_ids),
        'Doctor id': random.choice(doctor_ids),
        'Appointment date': fake.date_between(start_date='-2y', end_date='today').strftime('%Y-%m-%d'),
        'Appointment time': appt_time,
        'Purpose': random.choice(purposes),
        'Status': status
    })
```

Tables generated via Faker (not injected with data issues):

Patients – 15k rows (3k extra)

Doctors – 800 rows

Appointments – 10k rows (3k extra)

Prescriptions – 15k rows (3k extra)

Medicines – 800 rows

Ambulances – 200 rows

Ambulance logs – 600 rows

Rooms – 200 rows

Medical records – 10k rows (3k extra)

Pharmacy – 15k rows (3k extra)

Room assignments – 7k rows

Medical record medicine – 15k rows (3k extra)

Billing – 10k rows (3k extra)

Data issues functions:

These functions were applied at random across all the data to generate data issues

```
[28] import pandas as pd
import random
import numpy as np
import string
from datetime import datetime

def inject_data_issues(df, issues_config):
    """
    Injects data quality issues into a copy of the DataFrame.

    issues_config: {
        'missing_values': {'columns': [...], 'percentage': float},
        'date_format_issues': {'columns': [...], 'percentage': float},
        'duplicates': {'percentage': float},
        'text_corruption': {'columns': [...], 'percentage': float}
    }
    """
    dirty_df = df.copy()
    total_rows = len(dirty_df)

    # Inject missing values
    if 'missing_values' in issues_config:
        columns = issues_config['missing_values']['columns']
        pct = issues_config['missing_values']['percentage']
        n = int(pct * total_rows)
```

```
        for col in columns:
            indices = random.sample(range(total_rows), n)
            dirty_df.loc[indices, col] = np.nan

    # Inject inconsistent date formats
    if 'date_format_issues' in issues_config:
        columns = issues_config['date_format_issues']['columns']
        pct = issues_config['date_format_issues']['percentage']
        n = int(pct * total_rows)

        for col in columns:
            indices = random.sample(range(total_rows), n)
            for idx in indices:
                original_date = dirty_df.loc[idx, col]
                if pd.isnull(original_date):
                    continue
                try:
                    date_obj = pd.to_datetime(original_date)
                    dirty_df.loc[idx, col] = random.choice([
                        date_obj.strftime('%d-%m-%Y'),
                        date_obj.strftime('%m/%d/%Y'),
                        date_obj.strftime('%b %d, %Y'),
                        date_obj.strftime('%Y.%m.%d'),
                    ])
                except Exception:
                    continue
```

```
    # Inject duplicate rows
    if 'duplicates' in issues_config:
        pct = issues_config['duplicates']['percentage']
        n = int(pct * total_rows)
        dup_rows = dirty_df.sample(n=n, replace=False)
        dirty_df = pd.concat([dirty_df, dup_rows], ignore_index=True)

    # Inject text corruption (special characters)
    if 'text_corruption' in issues_config:
        columns = issues_config['text_corruption']['columns']
        pct = issues_config['text_corruption']['percentage']
        n = int(pct * total_rows)
        special_chars = list("!@#$%^&*()_+=[]{}|:;<>,.?/~`")

        for col in columns:
            indices = random.sample(range(total_rows), n)
            for idx in indices:
                val = dirty_df.loc[idx, col]
                if pd.isnull(val) or not isinstance(val, str):
                    continue
                insert_pos = random.randint(0, len(val))
                num_chars = random.randint(1, 3)
                corruption = ''.join(random.choices(special_chars, k=num_chars))
                corrupted_val = val[:insert_pos] + corruption + val[insert_pos:]
                dirty_df.loc[idx, col] = corrupted_val

    return dirty_df
```

```python
def change_unique_value(df, column_name, original_value, new_value, error_percentage):

    df = df.copy()  # Create a copy to avoid modifying the original DataFrame

    # Get the indices of rows containing the original value
    original_value_indices = df.index[df[column_name] == original_value].tolist()

    # Calculate the number of rows to modify
    num_rows_to_modify = int(len(original_value_indices) * error_percentage)

    # Randomly select rows to modify
    rows_to_modify = random.sample(original_value_indices, num_rows_to_modify)

    # Update the selected rows with the new value
    df.loc[rows_to_modify, column_name] = new_value

    return df
```

Functions applied as such

```python
dirty_ambulances_df = inject_data_issues(
    df=ambulances_df,
    issues_config={
        'missing_values': {
            'columns': ['Availability'],
            'percentage': 0.15
        },
        'date_format_issues': {
            'columns': ['Last service date'],
            'percentage': 0.15
        },
        'duplicates': {
            'percentage': 0.13
        },
        'text_corruption': {
            'columns': ['Availability'],
            'percentage': 0.15
        }
    }
)

print(dirty_ambulances_df.sample(10).to_string(index=False))
```

Data splits for extra data:

We did this logically where we split the patients table into 3k and 12k rows, then applied random data issues to both to ensure no duplicates exist among the splits, using the 3k patients, we then split the other tables in order, so appointments for those 3k, prescriptions for those 3k and so on, for extra data.

```python
patients_df_extra = patients_df.iloc[:3000].reset_index(drop=True)
patients_df_org = patients_df.iloc[3000:].reset_index(drop=True)

dirty_patients_df_extra = inject_data_issues(
    df=patients_df_extra,
    issues_config={
        'missing_values': {'columns': ['address', 'email', 'blood type'], 'percentage': 0.15},
        'date_format_issues': {'columns': ['dob'], 'percentage': 0.17},
        'duplicates': {'percentage': 0.09},
        'text_corruption': {'columns': ['contact no'], 'percentage': 0.13}
    }
)

print(dirty_patients_df_extra.sample(10).to_string(index=False))

dirty_patients_df_org = inject_data_issues(
    df=patients_df_org,
    issues_config={
        'missing_values': {'columns': ['address', 'email', 'blood type'], 'percentage': 0.15},
        'date_format_issues': {'columns': ['dob'], 'percentage': 0.17},
        'duplicates': {'percentage': 0.09},
        'text_corruption': {'columns': ['contact no'], 'percentage': 0.13}
    }
)

print(dirty_patients_df_extra.sample(10).to_string(index=False))
```

```python
# Extra data split for later use (prescriptions)
extra_patient_ids = patients_df_extra['Patient id'].tolist()
prescriptions_df_extra = prescriptions_df[prescriptions_df['Patient id'].isin(extra_patient_ids)]
prescriptions_df_extra = prescriptions_df_extra.reset_index(drop=True)  # Reset index
prescriptions_df_org = prescriptions_df[~prescriptions_df['Patient id'].isin(extra_patient_ids)]
prescriptions_df_org = prescriptions_df_org.reset_index(drop=True)  # Reset index
```

CSV Downloads

Converted the data-frames to CSV for original and extra data and downloaded them onto our local systems for use later on.

```python
# For original dirty data tables

from google.colab import files
dirty_dfs = {
    'ambulances_org': dirty_ambulances_df,
    'ambulance_logs_org': dirty_ambulance_logs_df,
    'rooms_org': dirty_rooms_df,
    'room_assignments_org': dirty_room_assignments_df,
    'medical_record_medicines_org': dirty_medical_record_medicine_df_org,
    'billing_org': dirty_billing_df_org
}

for table_name, df in dirty_dfs.items():
    csv_file_name = f"{table_name}.csv"
    df.to_csv(csv_file_name, index=False)
    files.download(csv_file_name)
```


```python
# For original dirty data tables

from google.colab import files
dirty_dfs = {
    'ambulances_org': dirty_ambulances_df,
    'ambulance_logs_org': dirty_ambulance_logs_df,
    'rooms_org': dirty_rooms_df,
    'room_assignments_org': dirty_room_assignments_df,
    'medical_record_medicines_org': dirty_medical_record_medicine_df_org,
    'billing_org': dirty_billing_df_org
}

for table_name, df in dirty_dfs.items():
    csv_file_name = f"{table_name}.csv"
    df.to_csv(csv_file_name, index=False)
    files.download(csv_file_name)
```

```python
# For original dirty data tables

from google.colab import files
dirty_dfs = {
    'patients_extra': dirty_patients_df_extra,
    'medical_records_extra': dirty_medical_records_df_extra,
    'prescriptions_extra': dirty_prescriptions_df_extra,
    'appointments_extra': dirty_appointments_df_extra,
    'billing_extra': dirty_billing_df_extra,
    'pharmacy_extra': dirty_pharmacy_df_extra,
}

for table_name, df in dirty_dfs.items():
    csv_file_name = f"{table_name}.csv"
    df.to_csv(csv_file_name, index=False)
    files.download(csv_file_name)
```

Data transformation:

After downloading the data, the csvs were read and then cleaned before dimensional modelling.

Data Cleaning

Generic functions for data cleaning:

Function that counts duplicate values

```python
# Counting duplicate values in each df

def count_duplicate_rows(df, df_name):
    duplicates = df[df.duplicated(keep=False)]
    num_duplicates = len(duplicates)
    if num_duplicates > 0:
        print(f"Number of duplicate rows in {df_name}: {num_duplicates}")
        #print(duplicates) #Uncomment if you want to see the actual duplicate rows.
    else:
        print(f"No duplicate rows found in {df_name}")
```

Function to replace missing vals with N/A, this was used a lot as logically, if a patient's diagnoses was missing, u can't use a modal value to fill it, similarly several columns with missing vals made no logical sense to fill except for with N/A.

```python
def replace_missing_with_na(df, column_name):

    if column_name in df.columns:
        df_copy = df.copy()
        # Condition to check for both null and "nan" values
        df_copy[column_name] = df_copy[column_name].fillna('N/A')
        # Fill "nan" string values with "N/A"
        df_copy[column_name] = df_copy[column_name].replace('nan', 'N/A')
        return df_copy
    else:
        print(f"Warning: Column '{column_name}' not found in DataFrame.")
        return df
```

These two functions worked to remove special characters, they were used to deal with text corruption, the one with num in it was specifically used for contact numbers etc.

```python
def remove_special_characters(df, column_name):
  if column_name not in df.columns:
    print(f"Warning: Column '{column_name}' not found in DataFrame.")
    return df

  df_copy = df.copy()

  df_copy[column_name] = df_copy[column_name].astype(str).apply(lambda x: re.sub(r'[!@#$%^&*()_+=\[\]{}|:;<>,.?/~`]', '', x))
  return df_copy

# Function to handle special characters in numerical cols

def remove_special_characters_num(df, column_name):

  if column_name not in df.columns:
    print(f"Warning: Column '{column_name}' not found in DataFrame.")
    return df

  df_copy = df.copy()

  # Function to process each value
  def process_value(value):
      # Extract digits using regex
      digits = re.findall(r'\d', str(value))
      # Re-concatenate digits
      return ''.join(digits) if digits else value

  # Apply the function to the column
  df_copy[column_name] = df_copy[column_name].apply(process_value)
  return df_copy
```

A more logical function that replaced column values based on modes in correlation to a different column, for example, used to get doctor specialty based on position

```python
# Function that replaces missing values/nan values in column B based on the modal value of column B for the corresponding value in column A

def fill_missing_with_modal(df, column_a, column_b):

    # Create a copy of the DataFrame to avoid modifying the original
    df_copy = df.copy()

    # Group by column_a and get the modal value for column_b
    modal_values = df_copy.groupby(column_a)[column_b].agg(lambda x: x.mode()[0] if not x.mode().empty else None)

    # Fill missing values in column_b based on the modal value for each group
    for group, modal_value in modal_values.items():
        # Condition to check for both null and "nan" values
        df_copy.loc[(df_copy[column_a] == group) & (df_copy[column_b].isnull() | (df_copy[column_b] == 'nan')), column_b] = modal_value

    return df_copy
```

Function to handle date formatting issues, changes datatype to date before running the secondary function

```python
# Function to convert a columns data type to date

def convert_to_date(df, column_name, format='%Y-%m-%d'):
    if column_name in df.columns:
        try:
            # Convert to datetime with errors='raise'
            df[column_name] = pd.to_datetime(df[column_name], format=format, errors='raise')
            # Convert to date only for valid dates
            df[column_name] = df[column_name].dt.date
        except ValueError:
            # Skip invalid dates and do nothing
            pass
    else:
        print(f"Warning: Column '{column_name}' not found in DataFrame.")
    return df

doctor_df = convert_to_date(doctor_df, 'Started At')
doctor_df.dtypes

doctor_df.head()
```

Function to count invalid dates i.e. dates that don't follow the format yyyy-mm-dd

```python
# Code to count the values in a date column where format isnt xxxx-xx-xx

def count_invalid_dates(df, column_name):

    # Regular expression to match 'YYYY-MM-DD' format
    date_pattern = r'^\d{4}-\d{2}-\d{2}$'

    # Filter out 'N/A' values before counting
    filtered_df = df[df[column_name] != 'N/A']

    # Count values in the filtered DataFrame that don't match the pattern
    invalid_dates_count = filtered_df[~filtered_df[column_name].astype(str).str.match(date_pattern)].shape[0]

    return invalid_dates_count

invalid_count = count_invalid_dates(doctor_df, 'Started At')
print(f"Number of invalid dates in 'Started At' column: {invalid_count}")
```

Function to fix invalid date formats

```python
# Function to fix invalid dates

def convert_invalid_dates(df, column_name):

    date_pattern = r'^\d{4}-\d{2}-\d{2}$'
    # Filter out 'N/A' values before checking for invalid dates
    invalid_dates = df[~df[column_name].astype(str).str.match(date_pattern) & (df[column_name] != 'N/A')][column_name]

    for index, value in invalid_dates.items():
        try:
            # Try to convert using pandas to_datetime (handles various formats)
            converted_date = pd.to_datetime(value, errors='raise').strftime('%Y-%m-%d')  # Convert to datetime
            df.loc[index, column_name] = pd.to_datetime(converted_date).date()  # Convert to date
        except ValueError:
            try:
                # If pandas conversion fails, try extracting numbers and formatting
                numbers = re.findall(r'\d+', str(value))
                if len(numbers) >= 3:
                    year = numbers[0][:4]
                    month = numbers[1][:2] if len(numbers[1]) >= 2 else numbers[1].zfill(2)
                    day = numbers[2][:2] if len(numbers[2]) >= 2 else numbers[2].zfill(2)
                    converted_date = f"{year}-{month}-{day}"
                    df.loc[index, column_name] = pd.to_datetime(converted_date).date()  # Convert to date
                else:
                    print(f"Warning: Could not extract enough numbers from '{value}' at index {index}")
            except (ValueError, TypeError) as e:
                print(f"Warning: Error processing '{value}' at index {index}: {e}")

    return df
```

Function to count column values with special characters, used to look for text corruption

```python
# Function to count rows with special characters

def count_special_characters(df, column_name, special_characters="!@#$%^&*()_+=\[\]{}|:;<>,.?/~-`"):

    # Create a pattern to match any of the special characters
    pattern = f"[{re.escape(special_characters)}]"

    # Filter out rows with "N/A" values
    filtered_df = df[df[column_name] != "N/A"]

    # Count rows where the column contains any special character
    count = filtered_df[filtered_df[column_name].astype(str).str.contains(pattern, na=False)].shape[0]

    return count
```

Function to count invalid contact details

```python
def count_invalid_contact_numbers(df, column_name):

    # Regular expression to match the desired format
    contact_pattern = r'^\d{3}-\d{3}-\d{4}$'

    # Get invalid contact numbers
    invalid_contacts = df[~df[column_name].astype(str).str.match(contact_pattern)][column_name]

    # Count invalid contact numbers
    invalid_count = invalid_contacts.shape[0]

    return invalid_count
```

Partial function to clean diagnoses column in medical records

```python
# Function to fix diagnoses column issues

def clean_diagnosis_column(df, column_name):

    if column_name not in df.columns:
        print(f"Warning: Column '{column_name}' not found in DataFrame.")
        return df

    df_copy = df.copy()

    def clean_text(text):
        # Remove text within parentheses
        text = re.sub(r'\(.*?\)', '', str(text))
        # Remove special characters
        text = re.sub(r'[!@#$%^&*()_+=\[\]{}|:;<>,.?/~`-]', '', text)  # Updated pattern
        return text.strip()

    df_copy[column_name] = df_copy[column_name].apply(clean_text)
    return df_copy
```

Diagnoses column had several issues with text corruption, needed column mapping to deal with some issues

```python
# Create mapping dictionary
diagnosis_mapping = {
    'Type 2 Diabetes': 'Type 2 Diabetes',
    'Type 2 D': 'Type 2 Diabetes',
    'Type 2 Di': 'Type 2 Diabetes',
    'Type 2 Diabet': 'Type 2 Diabetes',
    'Type': 'Type 2 Diabetes',

    'Hyperlipidemia': 'Hyperlipidemia',
    'Hyperlipidemia E785': 'Hyperlipidemia',
    'Hype': 'Hyperlipidemia',
    'Hyp': 'Hyperlipidemia',

    'UTI': 'UTI',
    'UTI 90': 'UTI',
    'U': 'UTI',
    'UTI 390': 'UTI',

    'Major Depressive Disorder': 'Major Depressive Disorder',
    'Major': 'Major Depressive Disorder',
    'Major Depr': 'Major Depressive Disorder',
```

```python
def map_diagnosis(text):
    if pd.isna(text):
        return None
    text = text.lower()
    if 'type 2 di' in text:
        return 'Type 2 Diabetes'
    elif 'hyperlip' in text:
        return 'Hyperlipidemia'
    elif 'uti' in text:
        return 'UTI'
    elif 'major dep' in text or 'majo' in text:
        return 'Major Depressive Disorder'
    elif 'covid' in text or 'co' in text:
        return 'COVID-19'
    elif 'asthma' in text or 'asth' in text:
        return 'Asthma'
    elif 'hypertens' in text or 'hyp' in text:
        return 'Hypertension'
    elif 'back pain' in text or 'bac' in text or text == 'back':
        return 'Back Pain'
    elif 'migraine' in text or 'migra' in text or 'mig' in text or text == 'mi':
        return 'Migraine'
    elif 'acute bron' in text or 'bro' in text or 'acu' in text or text == 'ac':
        return 'Acute Bronchitis'
    else:
        return text  # or None to mark unmapped
```

Used the following functions as such to clean data, done for all dirty data tables based on the data issues in them

```
invalid_count = count_invalid_dates(billing_df, 'Payment date')
print(f"Number of invalid dates in 'Payment date' column: {invalid_count}")

billing_df = convert_to_date(billing_df, 'Payment date')
billing_df.dtypes

billing_df = replace_missing_with_na(billing_df, 'Payment date')
billing_df.isnull().sum()

billing_df = convert_invalid_dates(billing_df, 'Payment date')
invalid_count = count_invalid_dates(billing_df, 'Payment date')
print(f"Number of invalid dates in 'Payment date' column: {invalid_count}")

billing_df.isnull().sum()

billing_df = replace_missing_with_na(billing_df, 'Insurance provider')
billing_df.isnull().sum()
```

Problem statement:

A data mart that facilitates analysis of medicine prescriptions that were given to patients for a specific appointment

Grain:

One row of the fact table represents information about one prescription given to a patient in one appointment by one doctor for a specific medicine

Dimensional modelling

Dimension tables:

- DimDoctor
- DimPtient
- DimAppointment
- DimPrescription
- DimMedicalRecord
- DimDate

Fact Table:

- FactPrescription

Dimension modelling was done via several merges; some new columns were created using previously existing columns as well.

Creating experience years column using started at date for doctordf.

```
# Assuming your DataFrame is called doctor_df

# Step 1: Convert 'Started At' to datetime
DimDoctordf['Started At'] = pd.to_datetime(doctor_df['Started At'])

# Step 2: Extract the starting year
DimDoctordf['Start Year'] = DimDoctordf['Started At'].dt.year

# Step 3: Calculate experience in years as of 2025
DimDoctordf['Experience Years'] = 2025 - DimDoctordf['Start Year']

DimDoctordf.head()

DimDoctordf = DimDoctordf.drop(columns=['Started At', 'Start Year'])
DimDoctordf.head()
```

In some cases, entire cleaned tables were copied into relevant dimension table, columns were then dropped when not needed.

```
DimAppointmentdf = appointment_df.copy()
DimAppointmentdf.head()
```

New column for DimAppointment

```
# Function to classify part of day
def get_part_of_day(hour):
    if pd.isna(hour):
        return np.nan
    if 5 <= hour < 12:
        return 'Morning'
    elif 12 <= hour < 17:
        return 'Afternoon'
    elif 17 <= hour < 21:
        return 'Evening'
    else:
        return 'Night'

# Apply function
DimAppointmentdf['Appointment Part of Day'] = DimAppointmentdf['Appointment_Hour'].apply(get_part_of_day)
```

Merge functions to copy relevant columns using ids, done for all dimension table and fact table

```
# Step 1: Merge prescriptions with pharmacy to get Medicine id
prescription_with_medicine = DimPrescriptiondf.merge(
    pharmacy_df[['Prescription id', 'Medicine id']],
    on='Prescription id',
    how='left'
)

# Step 2: Merge with medicine_df to get Brand and Type
prescription_with_medicine = prescription_with_medicine.merge(
    medicine_df[['Medicine id', 'Brand', 'Type']],
    on='Medicine id',
    how='left'
)
```

Defining a new season column for DimDate

```
# Define function to map month to season
def get_season(month):
    if month in [12, 1, 2]:
        return 'Winter'
    elif month in [3, 4, 5]:
        return 'Spring'
    elif month in [6, 7, 8]:
        return 'Summer'
    else:  # 9, 10, 11
        return 'Autumn'
```

Defining a hierarchy for DimDate and adding season column

```
# Create DimDatedf
DimDatedf = pd.DataFrame({
    'Date id': range(1, len(date_range) + 1),
    'Date': date_range,
    'Year': date_range.year,
    'Quarter': date_range.quarter,
    'Month': date_range.month,
    'Day': date_range.day,
})

# Add Season column
DimDatedf['Season'] = DimDatedf['Month'].apply(get_season)
```

Defining fact table foreign keys and facts (quantity, dosage and duration in days)

```
# Initialize FactPrescriptiondf directly with selected columns
FactPrescriptiondf = DimPrescriptiondf[['Prescription id','Appointment id', 'Patient id', 'Doctor Id', 'Medicine id', 'Dosage', 'Duration days']].copy()

# Check the result
FactPrescriptiondf.head()

pharmacy_df.head()

# Merge using both Prescription id and Medicine id to avoid duplicates
FactPrescriptiondf = FactPrescriptiondf.merge(
    pharmacy_df[['Prescription id', 'Medicine id', 'Quantity']],
    on=['Prescription id', 'Medicine id'],
    how='left'
)

# Rename the column
FactPrescriptiondf.rename(columns={'Quantity': 'Medicine Quantity'}, inplace=True)
```

Medicine amount as a fact using medicine price and quantity

```
# Check uniqueness of Medicine id in medicine_df
if medicine_df['Medicine id'].is_unique:
    # Step 1: Merge to bring Price from medicine_df
    FactPrescriptiondf = FactPrescriptiondf.merge(
        medicine_df[['Medicine id', 'Price']],
        on='Medicine id',
        how='left'
    )

    # Step 2: Calculate total amount
    FactPrescriptiondf['Medicine_Total_Amt'] = FactPrescriptiondf['Price'] * FactPrescriptiondf['Medicine Quantity']

    # Optional: Drop 'Price' column if not needed
    FactPrescriptiondf.drop(columns=['Price'], inplace=True)
else:
    print("Warning: Medicine id is not unique in medicine_df. Merge might cause duplicates.")
```

Dosage as a fact, a function was needed to convert all dosage vals into mg

```python
# Replace "N/A" with NaN
df['Dosage'] = df['Dosage'].replace('N/A', pd.NA)

# Conversion helper function
def convert_to_mg(dosage):
    if pd.isna(dosage):
        return pd.NA
    dosage = dosage.strip().lower()

    # Match patterns like '90mcg', '90mcg/inh', '90 mcg/ml', etc.
    match = re.match(r"([\d\.]+)\s*mcg", dosage)
    if match:
        mcg_val = float(match.group(1))
        mg_val = mcg_val / 1000  # 1 mg = 1000 mcg
        return round(mg_val, 5)

    # Match already in mg
    match = re.match(r"([\d\.]+)\s*mg", dosage)
    if match:
        return round(float(match.group(1)), 5)

    # Try just numeric string
    try:
        return round(float(dosage), 5)
    except ValueError:
        return pd.NA  # Can't interpret the value
```

Converting dosage column to numerical for aggregation

```python
# Apply the conversion
df['Dosage_mg'] = df['Dosage'].apply(convert_to_mg)
# Function to get numeric mode per group
def get_numeric_mode(series):
    non_na_values = series.dropna()
    if non_na_values.empty:
        return pd.NA
    mode_vals = non_na_values.mode()
    for val in mode_vals:
        if pd.notna(val):
            return val
    return pd.NA

# Get mode mapping
mode_map = df.groupby('Medicine id')['Dosage_mg'].apply(get_numeric_mode)

# Fill missing Dosage_mg values
def fill_missing_dosage(row):
    if pd.isna(row['Dosage_mg']):
        return mode_map.get(row['Medicine id'], pd.NA)
    return row['Dosage_mg']

df['Dosage_mg'] = df.apply(fill_missing_dosage, axis=1)
FactPrescriptiondf1 = df

# Drop 'Dosage' column
FactPrescriptiondf1 = FactPrescriptiondf1.drop(columns=['Dosage'])

# Reorder columns: insert 'Dosage (mg)' at position 5 (6th column, index starts at 0)
cols = list(FactPrescriptiondf1.columns)
cols.remove('Dosage_mg')
cols.insert(5, 'Dosage_mg')  # insert at position 6
```

Dropping irrelevant columns as such after dimensional modelling was completed, done for all dimension table and fact table

```
DimPrescriptiondf = DimPrescriptiondf.drop(columns=['Appointment id','Patient id','Doctor Id'])
DimPrescriptiondf.head()
```

Function to standardize column names for the tables, lowercase and _ instead of " "

```
def standardize_column_names(df):
    # Remove all non-alphanumeric characters except underscores
    new_columns = [re.sub(r'[^\w]', '_', col.lower()).strip('_') for col in df.columns]
    df.columns = new_columns
    return df


DimDatedf = standardize_column_names(DimDatedf)
DimDatedf.head()
```

Snowflake connection

We connected to the snowflake DB with a connection string as such, we also used a pandas package write_pandas to populate our snowflake database by directly uploading our data-frames to the relevant tables.

Overwrite was kept as true for doctors and date as they had to be reintegrated in the second run as well, appending would cause duplicates in snowflake.

```python
import pandas as pd
import snowflake.connector
from snowflake.connector.pandas_tools import write_pandas

# Snowflake connection
conn = snowflake.connector.connect(
    user='shahmeerkm',
    password='[redacted]',
    account='[redacted]',
    warehouse='PROJECT_WAREHOUSE',
    database='HOSPITAL_PRESCRIPTIONS',
    schema='STARSCHEMA'
)

# Upload each DataFrame to its respective table
write_pandas(conn,DimAppointmentdf,'DimAppointment',schema='STARSCHEMA')
write_pandas(conn, DimDatedf, 'DimDate',overwrite=True, schema='STARSCHEMA')
write_pandas(conn, DimDoctordf, 'DimDoctor', overwrite=True,schema='STARSCHEMA')
write_pandas(conn, DimMedicalRecorddf, 'DimMedicalRecord',schema='STARSCHEMA')
write_pandas(conn, DimPrescriptiondf, 'DimPrescription',schema='STARSCHEMA')
write_pandas(conn, DimPatientdf, 'DimPatient',schema='STARSCHEMA')
write_pandas(conn, FactPrescriptiondf, 'FactPrescription',schema='STARSCHEMA')
```

Snowflake implementation

Defined a warehouse (medium sized) and a database to load the trasnsformed data into

```sql
SHOW WAREHOUSES;
USE WAREHOUSE PROJECT_WAREHOUSE;
SHOW DATABASES;
USE HOSPITAL_PRESCRIPTIONS;
SHOW SCHEMAS;
USE SCHEMA STARSCHEMA;
```

Defining schemas for every table within snowflake

Column names should match the ones on the data-frames, otherwise errors are possible, defining these schemas ensures upload into snowflake is successful

```sql
CREATE OR REPLACE TABLE STARSCHEMA."DimAppointment" (
    "appointment_id" INT PRIMARY KEY,
    "appointment_date" TIMESTAMP_NTZ,
    "purpose" VARCHAR(255),
    "status" VARCHAR(50),
    "appointment_hour" INT,
    "appointment_part_of_day" VARCHAR(50),
    "payment_status" VARCHAR(50),
    "appointment_amount" FLOAT
);
```

```sql
CREATE OR REPLACE TABLE STARSCHEMA."DimDate" (
    "date_id" INT PRIMARY KEY,
    "date" VARCHAR(100),
    "year" INT,
    "quarter" INT,
    "month" INT,
    "day" INT,
    "season" VARCHAR(20)
);
```

```sql
CREATE OR REPLACE TABLE STARSCHEMA."DimDoctor" (
    "doctor_id" INT PRIMARY KEY,
    "name" VARCHAR(100),
    "contact_no" VARCHAR(20),
    "email" VARCHAR(100),
    "position" VARCHAR(50),
    "specialty" VARCHAR(100),
    "experience_years" INT
);
```

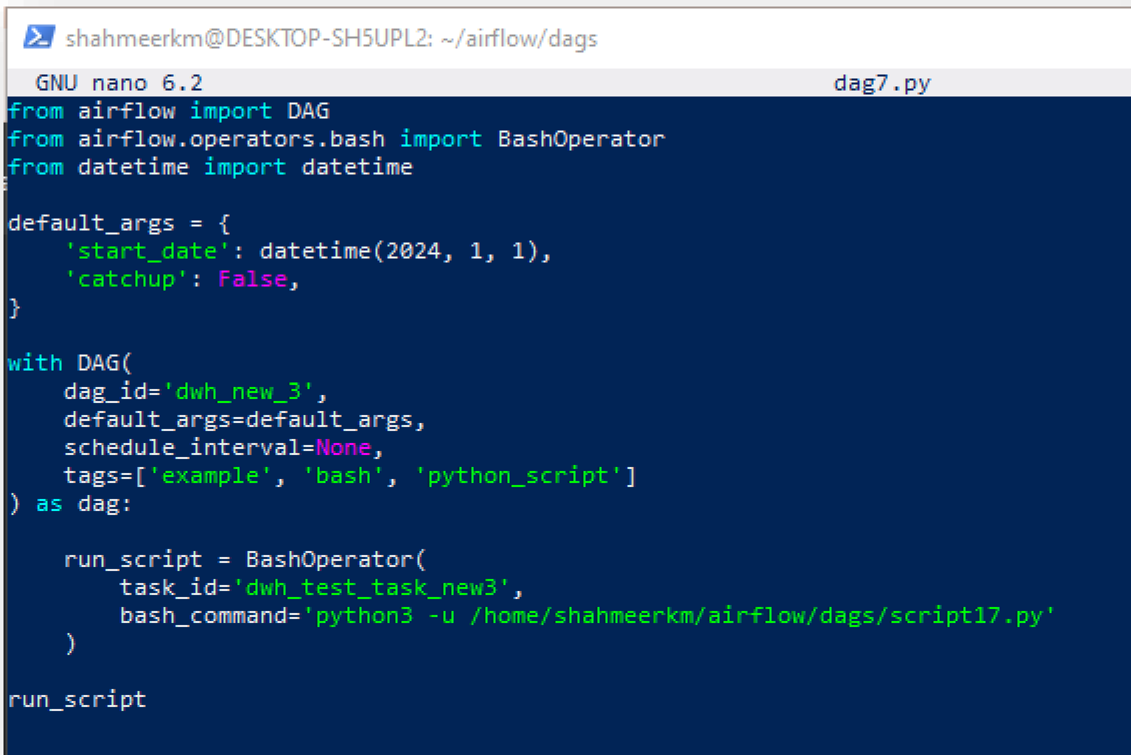
```sql
CREATE OR REPLACE TABLE STARSCHEMA."DimMedicalRecord" (
    "record_id" INT PRIMARY KEY,
    "diagnosis" VARCHAR(200),
    "treatment" VARCHAR(200),
    "insurance_provider" VARCHAR(100)
);

CREATE OR REPLACE TABLE STARSCHEMA."DimPatient" (
    "patient_id" VARCHAR(50) PRIMARY KEY,
    "name" VARCHAR(100),
    "dob" VARCHAR(20),
    "age" INT,
    "gender" VARCHAR(10),
    "contact_no" VARCHAR(20),
    "address" VARCHAR(200),
    "email" VARCHAR(100),
    "blood_type" VARCHAR(10)
);
```

```sql
CREATE OR REPLACE TABLE STARSCHEMA."DimPrescription" (
    "prescription_id" INT PRIMARY KEY,
    "medication_name" VARCHAR(100),
    "frequency" VARCHAR(50),
    "brand" VARCHAR(100),
    "type" VARCHAR(50)
);

CREATE OR REPLACE TABLE STARSCHEMA."FactPrescription" (
    "prescription_id" INT,
    "appointment_id" INT,
    "patient_id" VARCHAR(50),
    "doctor_id" INT,
    "record_id" INT,
    "date_id" INT,
    "dosage_mg" FLOAT,
    "duration_days" INT,
    "medicine_quantity" INT,
    "medicine_total_amt" FLOAT
);
```

DAG Implementation

For the Dag implementation in airflow, we used the BashOperator library, this enabled us to directly execute scripts using the Dag.

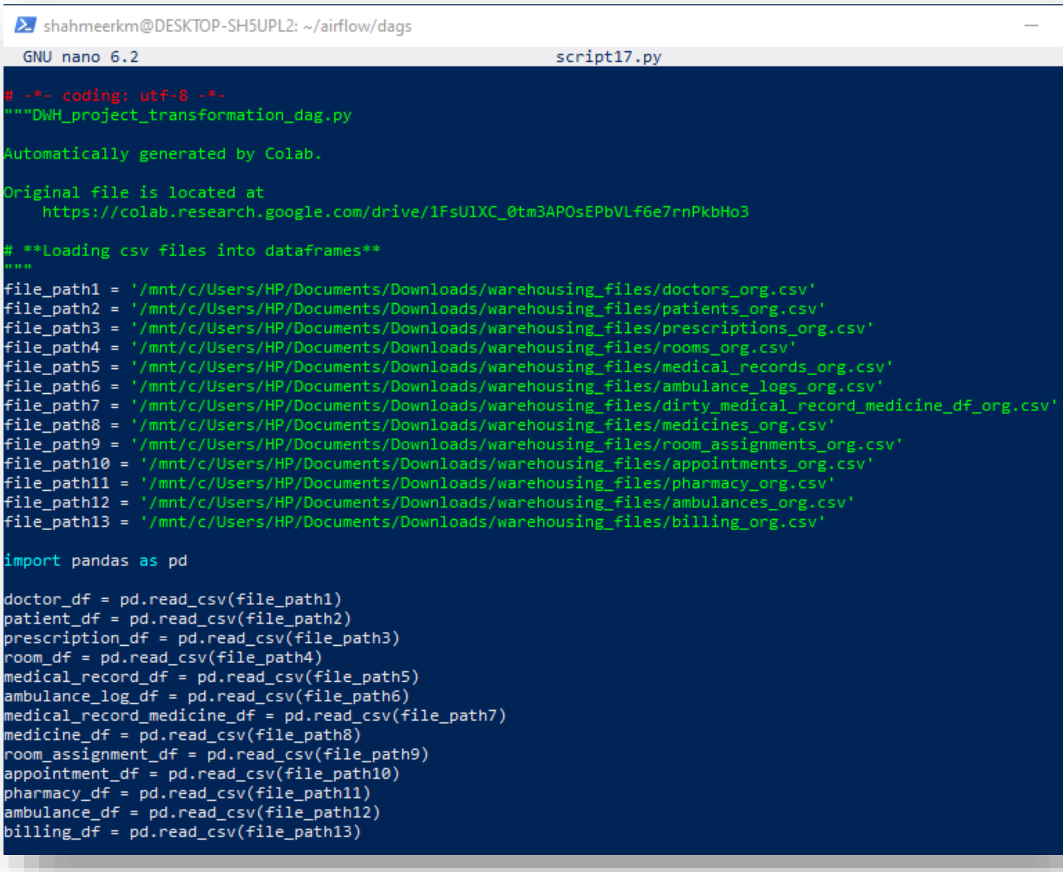
Here we define the basic configuration for the Dag, the bas command executes your script as if it were a simple command line command, the script is stored in the same Dags folder for surety. The name of the dag on airflow will be dwh_new_3.



```
shahmeerkm@DESKTOP-SH5UPL2: ~/airflow/dags
  GNU nano 6.2                                        dag7.py
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime

default_args = {
    'start_date': datetime(2024, 1, 1),
    'catchup': False,
}

with DAG(
    dag_id='dwh_new_3',
    default_args=default_args,
    schedule_interval=None,
    tags=['example', 'bash', 'python_script']
) as dag:

    run_script = BashOperator(
        task_id='dwh_test_task_new3',
        bash_command='python3 -u /home/shahmeerkm/airflow/dags/script17.py'
    )

run_script
```

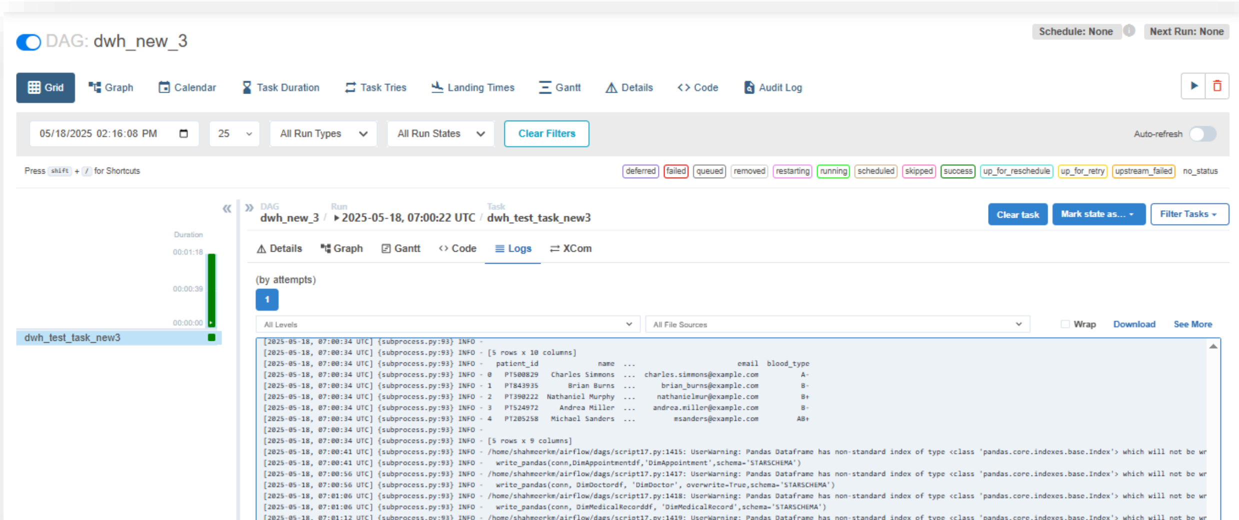
The script was as defined above in the transformation function, only difference being the file paths, as we're using linux in a virtual env, we need to mount the files accordingly.

/mnt mounts the files into linux for use.




```
# -*- coding: utf-8 -*-
"""DWH_project_transformation_dag.py

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1FsUlXC_0tm3APOsEPbVLf6e7rnPkbHo3

# **Loading csv files into dataframes**
"""
file_path1 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/doctors_org.csv'
file_path2 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/patients_org.csv'
file_path3 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/prescriptions_org.csv'
file_path4 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/rooms_org.csv'
file_path5 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/medical_records_org.csv'
file_path6 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/ambulance_logs_org.csv'
file_path7 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/dirty_medical_record_medicine_df_org.csv'
file_path8 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/medicines_org.csv'
file_path9 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/room_assignments_org.csv'
file_path10 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/appointments_org.csv'
file_path11 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/pharmacy_org.csv'
file_path12 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/ambulances_org.csv'
file_path13 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/billing_org.csv'

import pandas as pd

doctor_df = pd.read_csv(file_path1)
patient_df = pd.read_csv(file_path2)
prescription_df = pd.read_csv(file_path3)
room_df = pd.read_csv(file_path4)
medical_record_df = pd.read_csv(file_path5)
ambulance_log_df = pd.read_csv(file_path6)
medical_record_medicine_df = pd.read_csv(file_path7)
medicine_df = pd.read_csv(file_path8)
room_assignment_df = pd.read_csv(file_path9)
appointment_df = pd.read_csv(file_path10)
pharmacy_df = pd.read_csv(file_path11)
ambulance_df = pd.read_csv(file_path12)
billing_df = pd.read_csv(file_path13)
```

By accessing the airflow server on localhost we viewed and executed our Dag, here we see the successful execution of our Dag.

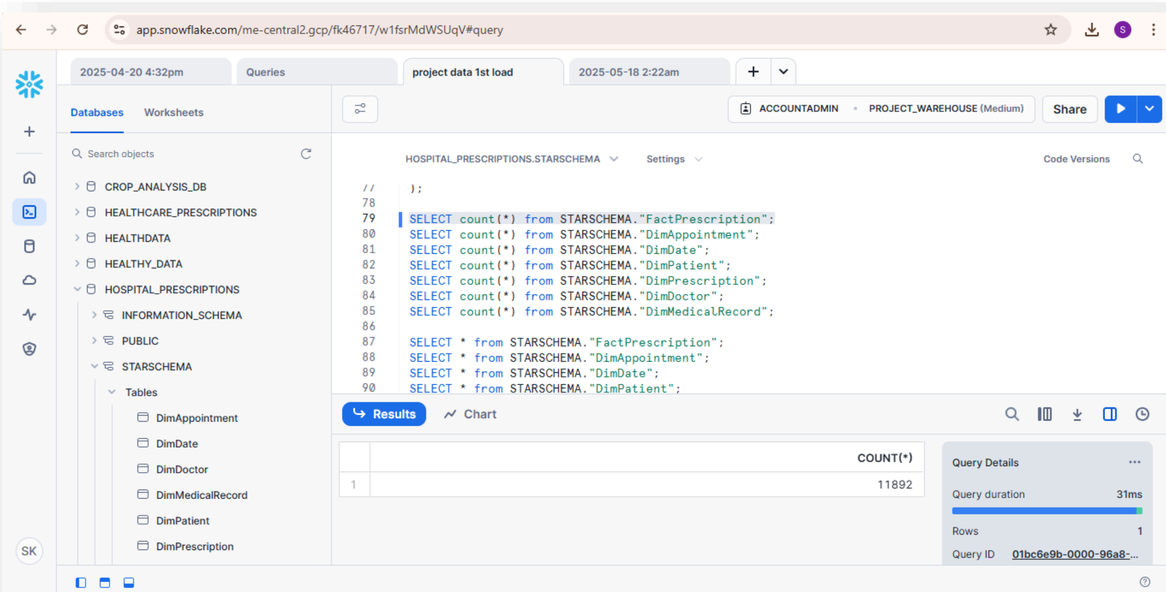


A closer look into the logs shows us that data was successfully loaded into snowflake. As we can see here, the Dag executed the entire python transformation script creating the star schema as requested (DimPatient printed in the logs below), it also uploaded the data-frames into relevant snowflake tables as needed via write_pandas.

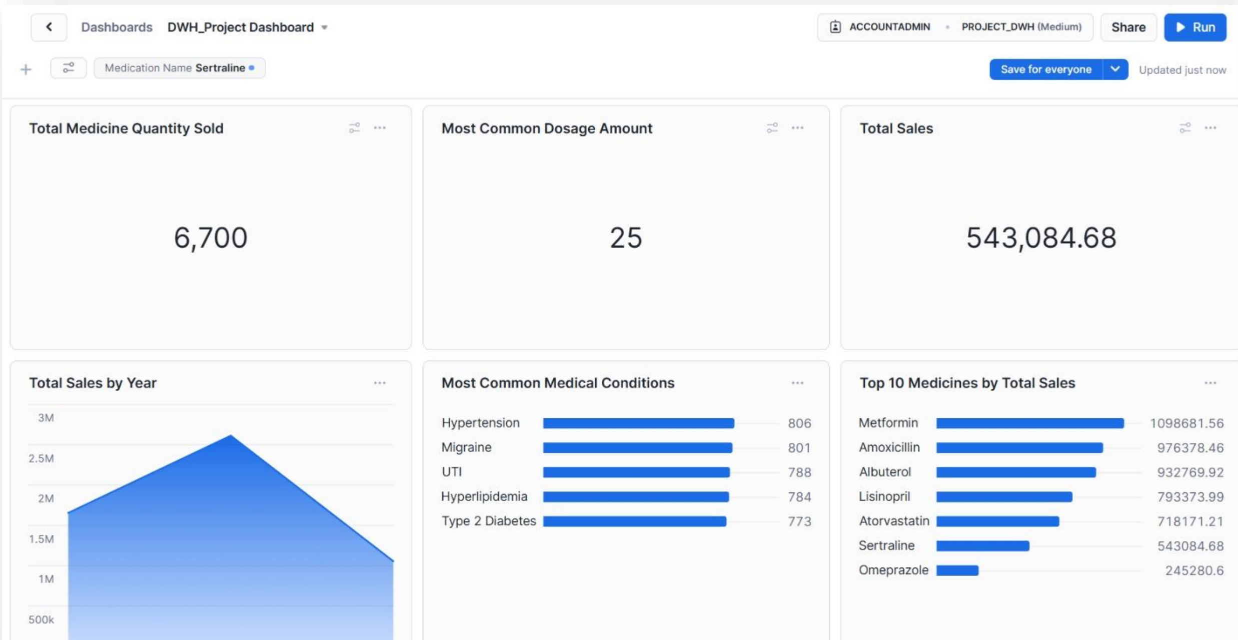
```
[2025-05-18, 07:00:34 UTC] {subprocess.py:93} INFO -
[2025-05-18, 07:00:34 UTC] {subprocess.py:93} INFO - [5 rows x 10 columns]
[2025-05-18, 07:00:34 UTC] {subprocess.py:93} INFO -   patient_id              name  ...                     email  blood_type
[2025-05-18, 07:00:34 UTC] {subprocess.py:93} INFO - 0   PT500829   Charles Simmons  ...  charles.simmons@example.com          A-
[2025-05-18, 07:00:34 UTC] {subprocess.py:93} INFO - 1   PT843935       Brian Burns  ...     brian_burns@example.com          B-
[2025-05-18, 07:00:34 UTC] {subprocess.py:93} INFO - 2   PT390222  Nathaniel Murphy  ...     nathanielmur@example.com          B+
[2025-05-18, 07:00:34 UTC] {subprocess.py:93} INFO - 3   PT524972     Andrea Miller  ...    andrea.miller@example.com          B-
[2025-05-18, 07:00:34 UTC] {subprocess.py:93} INFO - 4   PT205258   Michael Sanders  ...        msanders@example.com         AB+
[2025-05-18, 07:00:34 UTC] {subprocess.py:93} INFO -
[2025-05-18, 07:00:34 UTC] {subprocess.py:93} INFO - [5 rows x 9 columns]
[2025-05-18, 07:00:41 UTC] {subprocess.py:93} INFO - /home/shahmeerkm/airflow/dags/script17.py:1415: UserWarning: Pandas Dataframe has
[2025-05-18, 07:00:41 UTC] {subprocess.py:93} INFO -   write_pandas(conn,DimAppointmentdf,'DimAppointment',schema='STARSCHEMA')
[2025-05-18, 07:00:56 UTC] {subprocess.py:93} INFO - /home/shahmeerkm/airflow/dags/script17.py:1417: UserWarning: Pandas Dataframe has
[2025-05-18, 07:00:56 UTC] {subprocess.py:93} INFO -   write_pandas(conn, DimDoctordf, 'DimDoctor', overwrite=True,schema='STARSCHEMA')
[2025-05-18, 07:01:06 UTC] {subprocess.py:93} INFO - /home/shahmeerkm/airflow/dags/script17.py:1418: UserWarning: Pandas Dataframe has
[2025-05-18, 07:01:06 UTC] {subprocess.py:93} INFO -   write_pandas(conn, DimMedicalRecorddf, 'DimMedicalRecord',schema='STARSCHEMA')
[2025-05-18, 07:01:12 UTC] {subprocess.py:93} INFO - /home/shahmeerkm/airflow/dags/script17.py:1419: UserWarning: Pandas Dataframe has
[2025-05-18, 07:01:12 UTC] {subprocess.py:93} INFO -   write_pandas(conn, DimPrescriptiondf, 'DimPrescription',schema='STARSCHEMA')
[2025-05-18, 07:01:22 UTC] {subprocess.py:93} INFO - /home/shahmeerkm/airflow/dags/script17.py:1420: UserWarning: Pandas Dataframe has
[2025-05-18, 07:01:22 UTC] {subprocess.py:93} INFO -   write_pandas(conn, DimPatientdf, 'DimPatient',schema='STARSCHEMA')
[2025-05-18, 07:01:33 UTC] {subprocess.py:93} INFO - /home/shahmeerkm/airflow/dags/script17.py:1421: UserWarning: Pandas Dataframe has
[2025-05-18, 07:01:33 UTC] {subprocess.py:93} INFO -   write_pandas(conn, FactPrescriptiondf, 'FactPrescription',schema='STARSCHEMA')
[2025-05-18, 07:01:40 UTC] {subprocess.py:97} INFO - Command exited with return code 0
[2025-05-18, 07:01:40 UTC] {taskinstance.py:1138} INFO - Marking task as SUCCESS. dag_id=dwh_new_3, task_id=dwh_test_task_new3, executi
[2025-05-18, 07:01:40 UTC] {local_task_job_runner.py:234} INFO - Task exited with return code 0
[2025-05-18, 07:01:40 UTC] {taskinstance.py:3280} INFO - 0 downstream tasks scheduled from follow-on schedule check
```

Work on snowflake (Dashboarding)

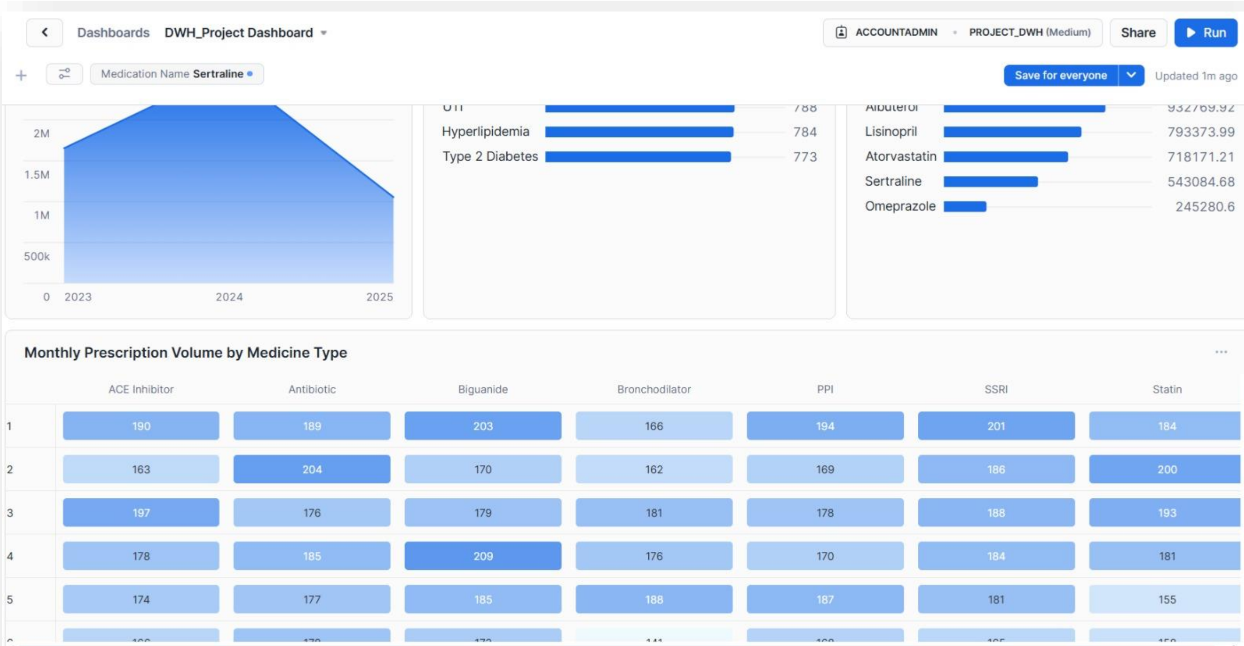
Here we can see that after the Dag's execution, the snowflake tables were populated



Dashboards:



Dashboards
DWH_Project Dashboard
ACCOUNTADMIN
PROJECT_DWH (Medium)
Share
Run
Medication Name Sertraline
Save for everyone
Updated just now
Total Medicine Quantity Sold
6,700
Most Common Dosage Amount
25
Total Sales
543,084.68
Total Sales by Year
3M
2.5M
2M
1.5M
1M
500k
Most Common Medical Conditions
Hypertension 806
Migraine 801
UTI 788
Hyperlipidemia 784
Type 2 Diabetes 773
Top 10 Medicines by Total Sales
Metformin 1098681.56
Amoxicillin 976378.46
Albuterol 932769.92
Lisinopril 793373.99
Atorvastatin 718171.21
Sertraline 543084.68
Omeprazole 245280.6



Dashboards
DWH_Project Dashboard
ACCOUNTADMIN
PROJECT_DWH (Medium)
Share
Run
Medication Name Sertraline
Save for everyone
Updated 1m ago
2M
1.5M
1M
500k
0
2023
2024
2025
Hyperlipidemia 784
Type 2 Diabetes 773
Lisinopril 793373.99
Atorvastatin 718171.21
Sertraline 543084.68
Omeprazole 245280.6
Monthly Prescription Volume by Medicine Type
ACE Inhibitor
Antibiotic
Biguanide
Bronchodilator
PPI
SSRI
Statin
1 190 189 203 166 194 201 184
2 163 204 170 162 169 186 200
3 197 176 179 181 178 188 193
4 178 185 209 176 170 184 181
5 174 177 185 188 187 181 155

Dashboard queries for charts:

Bar chart for Top Ten Medicines By Total Sales:

Calculates total medication sales based on medication names

```
SHOW WAREHOUSES;
USE WAREHOUSE PROJECT_WAREHOUSE;
SHOW DATABASES;
USE HOSPITAL_PRESCRIPTIONS;
SHOW SCHEMAS;
USE SCHEMA STARSCHEMA;

SELECT
    dp."medication_name",
    SUM(fp."medicine_total_amt") AS total_medicine_amount
FROM STARSCHEMA."FactPrescription" fp
JOIN STARSCHEMA."DimPrescription" dp ON fp."prescription_id" = dp."prescription_id"
GROUP BY dp."medication_name"
ORDER BY total_medicine_amount DESC
LIMIT 10;
```

Bar chart for Most Common Medical Conditions

Calculates the top ten patient counts by diagnosis

```
SHOW WAREHOUSES;
USE WAREHOUSE PROJECT_WAREHOUSE;
SHOW DATABASES;
USE HOSPITAL_PRESCRIPTIONS;
SHOW SCHEMAS;
USE SCHEMA STARSCHEMA;

SELECT
  DMR."diagnosis",
  COUNT(DISTINCT FP."patient_id") AS "avg_patients"
FROM
  STARSCHEMA."FactPrescription" FP
JOIN
  STARSCHEMA."DimMedicalRecord" DMR
    ON FP."record_id" = DMR."record_id"
WHERE
  DMR."diagnosis" <> 'N/A'
GROUP BY
  DMR."diagnosis"
ORDER BY
  "avg_patients" DESC
LIMIT 10;
```

Line chart for total sales by year:

Calculates total sales per year

```
SHOW WAREHOUSES;
USE WAREHOUSE PROJECT_WAREHOUSE;
SHOW DATABASES;
USE HOSPITAL_PRESCRIPTIONS;
SHOW SCHEMAS;
USE SCHEMA STARSCHEMA;

SELECT
    D."year",
    D."month",
    SUM(FP."medicine_total_amt") AS total_sales
FROM
    STARSCHEMA."FactPrescription" FP
JOIN
    STARSCHEMA."DimDate" D ON FP."date_id" = D."date_id"
GROUP BY
    D."year", D."month"
ORDER BY
    D."year", D."month";
```

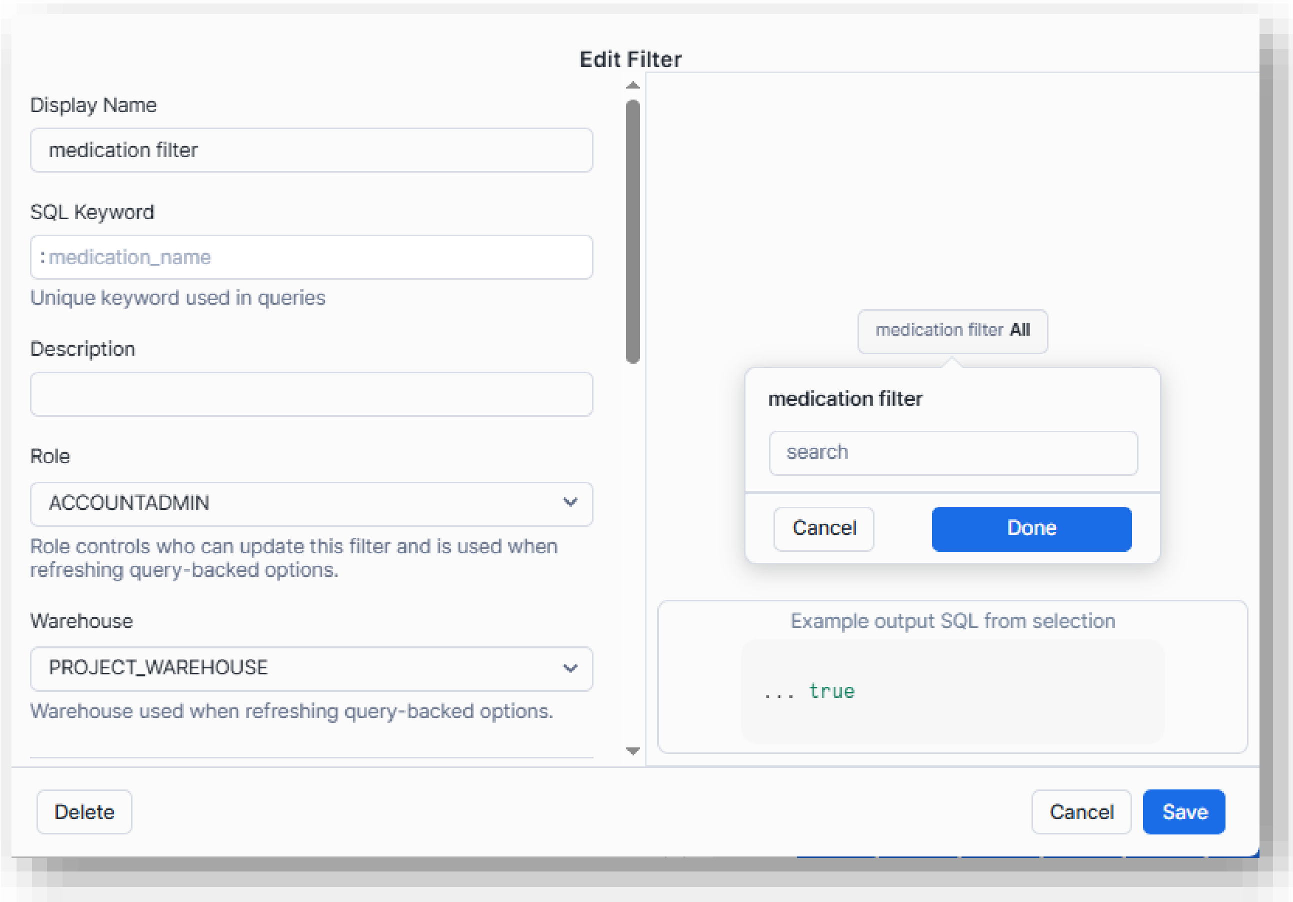
Heatmap chart for Monthly Prescription Volume By Medicine Type:

Calculates prescription counts per month for every medication

```
SHOW WAREHOUSES;
USE WAREHOUSE PROJECT_WAREHOUSE;
SHOW DATABASES;
USE HOSPITAL_PRESCRIPTIONS;
SHOW SCHEMAS;
USE SCHEMA STARSCHEMA;

SELECT
  DD."month",
  DP."type",
  COUNT(*) AS prescription_count
FROM
  STARSCHEMA."FactPrescription" FP
JOIN
  STARSCHEMA."DimDate" DD ON FP."date_id" = DD."date_id"
JOIN
  STARSCHEMA."DimPrescription" DP ON FP."prescription_id" = DP."prescription_id"
GROUP BY
  DD."month",
  DP."type"
ORDER BY
  DD."month", DP."type";
```

Medication name filter for KPIs

```sql
SELECT "DimPrescription"."medication_name" from "STARSCHEMA"."DimPrescription"
```



Dashboard queries for KPIs using medication name filter:

Total sales:

Calculates the total sales based on medication name

```sql
SELECT
  SUM(FP."medicine_total_amt") AS total_sales
FROM
  STARSCHEMA."FactPrescription" FP
JOIN
  STARSCHEMA."DimPrescription" DP
    ON FP."prescription_id" = DP."prescription_id"
WHERE
  DP."medication_name" = :medication_name;
```

Total Medicine Quantity Sold:

Calculates the total quantity based on medication name

```sql
SELECT
  SUM(FP."medicine_quantity") AS total_quantity
FROM
  STARSCHEMA."FactPrescription" FP
JOIN
  STARSCHEMA."DimPrescription" DP ON FP."prescription_id" = DP."prescription_id"
WHERE
  DP."medication_name" = :medication_name;
```

Most Common Dosage Amount:

Calculates the modal dosage based on medication name

```sql
SELECT
  FP."dosage_mg"
FROM
  STARSCHEMA."FactPrescription" FP
JOIN
  STARSCHEMA."DimPrescription" DP
    ON FP."prescription_id" = DP."prescription_id"
WHERE
  DP."medication_name" = :medication_name
GROUP BY
FP."dosage_mg"
ORDER BY
  COUNT(*) DESC
LIMIT 1;
```

Problems faced

This section highlights all the issues faced throughout this project

Data generation:

- Deciding on and generating proper functions for making the data dirty, several functions gave errors.
- Logically splitting the data-frames into original and extra tables, dependencies caused issues.
- We first ran our data issues functions on the main files and then split them, but this would cause problems after splits as duplicates mean the same record could exist in both splits, so instead injected data issues after split.

Data transformation:

- In certain tables the data issue functions really ruined our data so defining the proper generalized functions to work across all tables became a challenge.
- Came across several new issues with formatting that we hadn't even injected into our data so had to deal with those as well.
- Column mapping with snowflake caused a lot of hurdles so had to redefine several column names to match the snowflake system as () are not allowed in snowflake columns but were used in several columns of ours.

Dimensional modelling:

- Had to spend a lot of time deciding the grain, the dimension tables, figuring out facts etc as we went through multiple possible data marts to land on the prescription data mart in the end.

Dags:

- Initially used PythonOperator, but for that we had to wrap the whole script in a function which was too inconvenient and caused several issues.
- We then went for BashOperator which was also really difficult to get a hold of and understand its implementation and execute it.

Snowflake Connection:

- Initially we opted for SQLAlchemy to push our data into snowflake however that kept giving us cursor errors as such

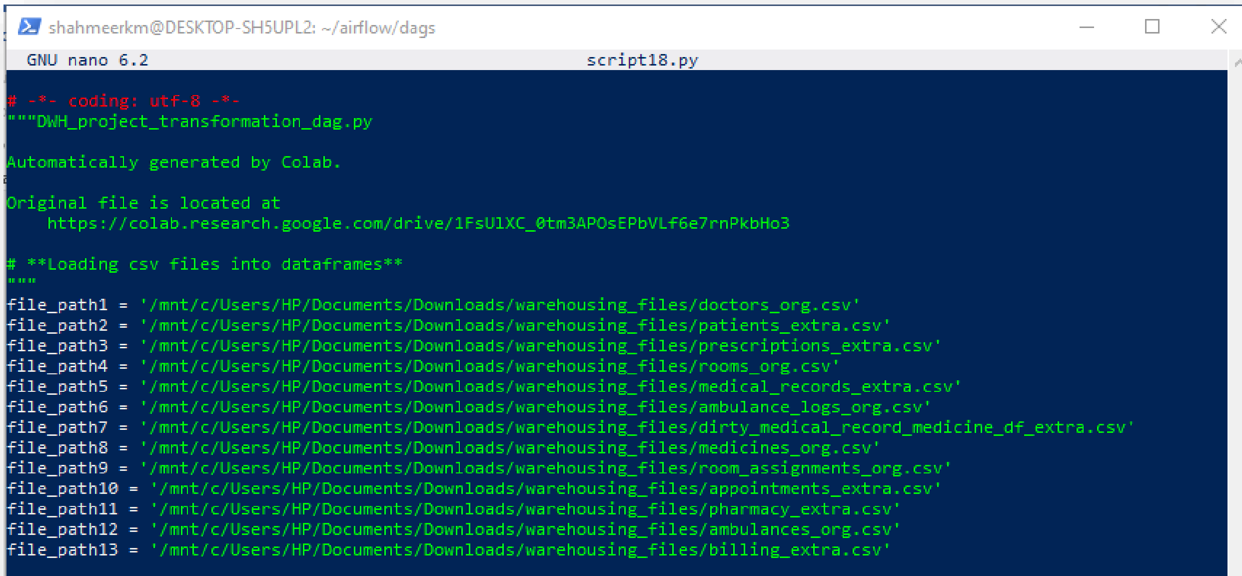
```
17, 19:32:04 UTC] {subprocess.py:93} INFO -     table.create()
17, 19:32:04 UTC] {subprocess.py:93} INFO -   File "/mnt/c/Windows/system32/airflow_venv/lib/python3.10/site-packag
17, 19:32:04 UTC] {subprocess.py:93} INFO -     if self.exists():
17, 19:32:04 UTC] {subprocess.py:93} INFO -   File "/mnt/c/Windows/system32/airflow_venv/lib/python3.10/site-packag
17, 19:32:04 UTC] {subprocess.py:93} INFO -     return self.pd_sql.has_table(self.name, self.schema)
17, 19:32:04 UTC] {subprocess.py:93} INFO -   File "/mnt/c/Windows/system32/airflow_venv/lib/python3.10/site-packag
17, 19:32:04 UTC] {subprocess.py:93} INFO -     return len(self.execute(query, [name]).fetchall()) > 0
17, 19:32:04 UTC] {subprocess.py:93} INFO -   File "/mnt/c/Windows/system32/airflow_venv/lib/python3.10/site-packag
17, 19:32:04 UTC] {subprocess.py:93} INFO -     cur = self.con.cursor()
17, 19:32:04 UTC] {subprocess.py:93} INFO - AttributeError: 'Engine' object has no attribute 'cursor'
17, 19:32:04 UTC] {subprocess.py:97} INFO - Command exited with return code 1
17, 19:32:04 UTC] {taskinstance.py:2698} ERROR - Task failed with exception
```

- We later went for pandas_write as an alternative to SQLAclhemy, that however gave other new errors due to column names as mentioned before.
- Table names also caused issues due to case sensitivity, snowflake expects table names to either be in all caps or in "", due to this it wasn't recognizing our table names from the dfs as such, fixed this by having "" for every table and column name in our snowflake schema design.

```
| {subprocess.py:93} INFO -   File "/mnt/c/Windows/system32/airflow_venv/lib/pythor
| {subprocess.py:93} INFO -     cursor.errorhandler(connection, cursor, error_class
| {subprocess.py:93} INFO -   File "/mnt/c/Windows/system32/airflow_venv/lib/pythor
| {subprocess.py:93} INFO -     raise error_class(
| {subprocess.py:93} INFO - snowflake.connector.errors.ProgrammingError: 001757 (42
| {subprocess.py:93} INFO - Table '"DimAppointment"' does not exist
| {subprocess.py:97} INFO - Command exited with return code 1
| {taskinstance.py:2698} ERROR - Task failed with exception
.l last):
:tem32/airflow_venv/lib/python3.10/site-packages/airflow/models/taskinstance.py", l
ible(context=context, **execute_callable_kwargs)
:tem32/airflow_venv/lib/python3.10/site-packages/airflow/operators/bash.py", line 2
```

DB Update Task:

We mentioned before that we prepared separate data for uploading, we simply ran this through the same pipeline as before using the same Dag and the same script with a minor change, updated file paths to this time take the new data instead. Here, the file paths now represent the extra data as well as some of the original data which will be needed for dimensional modelling of the extra data, this however has been ensured to not be duplicated via overwrite=true as mentioned before.


shahmeerkm@DESKTOP-SH5UPL2: ~/airflow/dags

GNU nano 6.2 script18.py

```
# -*- coding: utf-8 -*-
"""DWH_project_transformation_dag.py

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1FsU1XC_0tm3APOsEPbVLf6e7rnPkbHo3

# **Loading csv files into dataframes**
"""
file_path1 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/doctors_org.csv'
file_path2 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/patients_extra.csv'
file_path3 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/prescriptions_extra.csv'
file_path4 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/rooms_org.csv'
file_path5 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/medical_records_extra.csv'
file_path6 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/ambulance_logs_org.csv'
file_path7 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/dirty_medical_record_medicine_df_extra.csv'
file_path8 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/medicines_org.csv'
file_path9 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/room_assignments_org.csv'
file_path10 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/appointments_extra.csv'
file_path11 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/pharmacy_extra.csv'
file_path12 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/ambulances_org.csv'
file_path13 = '/mnt/c/Users/HP/Documents/Downloads/warehousing_files/billing_extra.csv'
```

Once this data was uploaded to the same DB, the dashboard automatically updated as shown in the video tutorial.

Issues faced here were that we initially didn't understand how to prevent duplication of data however that was dealt with via the overwrite command.