TABLE OF CONTENTS

1. Introduction

This project develops a comprehensive data product to generate valuable business insights for Takeda Pharmaceuticals, focusing on Alunbrig, an oncology drug. The analysis compares Alunbrig's market performance with four competing products and evaluates the effectiveness of contracts with insurance companies in U.S. pharmaceutical market. The analysis examines whether contracted and non-contracted payers impact Alunbrig's performance.

It establishes an end-to-end analytics environment involving database design, data implementation, synthetic data generation, and business reporting. Using SQLite for management and querying, along with visualisation techniques, the project accurately models the U.S. healthcare insurance and pharmaceutical landscape to support strategic decision-making and enhance Takeda's competitive positioning.

2. Business Context and Database Design

2.1. Business Context Definition

Takeda Pharmaceuticals, a global leader in biopharmaceuticals, operates in over 80 countries. The focus of this project is on Alunbrig, a targeted therapy for NSCLC, whose main competitors are Xalkori, Alecensa, Zykadia, and Avastin. Considering that external factors such as insurance coverage, prescriber incentives, and formulary restrictions influence prescription trends, the project explores whether insurance-related barriers contribute to stagnant Alunbrig sales in the commercial channel in 2024, despite its clinical efficacy. A comparative study of contracted versus non-contracted payers helps assess whether strategic adjustments are needed (Takeda, (no date)).

2.2. Database Design

2.2.1. Overview of Database Structure

The database reflects the U.S. healthcare insurance landscape, incorporating payers, insurance plans, and formularies. It captures data on plan coverage, formulary restrictions, drug sales, payer relationships, and covered lives. This structure ensures data integrity and supports efficient querying to generate insights on Alunbrig's market dynamics.

Assumptions:

- Drug Prices: Prices are sourced from industry-standard sites and normalised on a per-unit basis. Strength and dosage are assumed to be constant across drugs.
- Lives Covered: The number of covered lives is well below the U.S. population, reflecting healthcare plan enrolment rates.

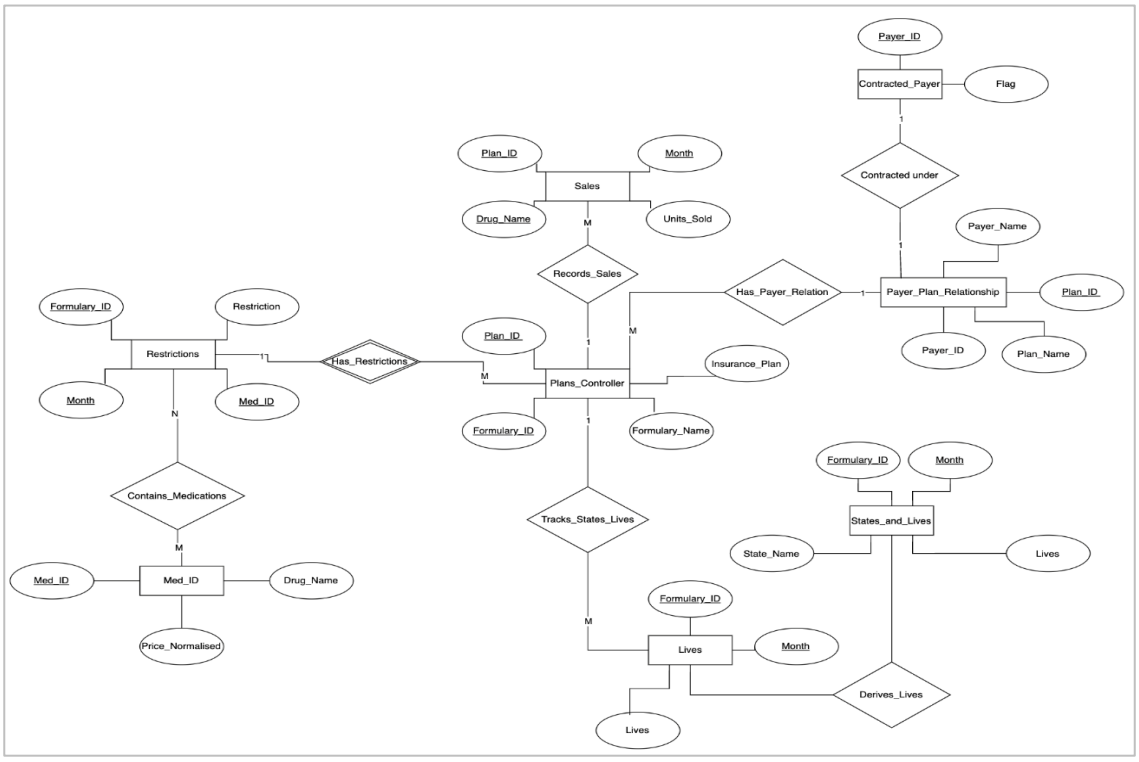
- Units Sold: Data is randomly generated based on SME insights, accounting for limited patient uptake in cancer markets.

2.2.2. Entity-Relationship (ER) Diagram

The central entity in the database is Plans_Controller, which is connected to several other entities to form the overall structure of the database. These include Restrictions, Sales, Medications, States_and_Lives, Lives, Contracted_Payer, and Payer_Plan_Relationship (Refer Appendix 1).

Key relationships:

- **Plans_Controller to Restrictions:** One-to-many. Each plan, associated with multiple formularies, can have varying or identical restrictions across different months of the year.
- **Plans_Controller to Sales:** One-to-many. A single plan can have multiple sales records for each drug across various months.
- **Plans_Controller to States_and_Lives:** One-to-many. Plans, through associated formularies, cover multiple states, impacting the number of covered lives.
- **Plans_Controller to Payer_Plan_Relationship:** Many-to-one. Multiple plans can be linked to a single payer, indicating payer control over various plans.
- **Med_ID to Restrictions:** Many-to-many. A medication can be subject to multiple restrictions and vice versa depending on the formulary and month it is associated with.
- **Contracted_Payer to Payer_Plan_Relationship:** One-to-one. Each contracted payer corresponds to a unique entry in the payer-plan relationship, indicating whether it is under contract.



2.2.3. Normalisation Considerations

The schema adheres to normalisation rules (1NF, 2NF, and 3NF) to ensure that the database is free from redundancy and maintains data integrity.

3. Synthetic Data Generation

To simulate a realistic pharmaceutical market, a synthetic dataset was generated to mirror the U.S. healthcare insurance and pharmaceutical landscape. Created through sequential ChatGPT prompts, the process ensured consistency and coherence across payers, insurance plans, formularies, and drug sales for 2024.

Step 1: Plans Controller Dataset

- A foundational table was created including 112 plans of the largest U.S. insurance companies, each assigned a unique 5-digit Plan_ID.

  Formularies per Plan

- Each plan was assigned 2 to 7 hypothetical formulary names, representing drug coverage structures within plans.

Unique ID Structuring

- Random 5-digit Plan_IDs and Formulary_IDs were generated to ensure uniqueness per company, avoiding duplication across entities.

```
Top 10 rows of Plans_Controller table:
['Plan_ID', 'Insurance_Plan', 'Formulary_ID', 'Formulary_Name']
(20000, 'Premier Advantage Option', 30000, 'Exclusive Prescription Plan')
(20000, 'Premier Advantage Option', 30001, 'Comprehensive Prescription List')
(20000, 'Premier Advantage Option', 30002, 'Comprehensive Prescription Option')
(20000, 'Premier Advantage Option', 30003, 'Enhanced Pharmacy Plan')
(20000, 'Premier Advantage Option', 30004, 'Standard Prescription Formulary')
(20001, 'Essential Advantage Plan', 30005, 'Advanced Prescription List')
(20001, 'Essential Advantage Plan', 30006, 'Preferred Prescription Option')
(20002, 'Smart Wellness Program', 30007, 'Enhanced Health Option')
(20002, 'Smart Wellness Program', 30008, 'Enhanced Medication List')
(20002, 'Smart Wellness Program', 30009, 'Basic Medication Program')
```

Step 2: Payer Plan Relationship Dataset

- Considering that each payer can have multiple plans, a new table was created in which the 112 plans were randomly assigned to Payer_IDs.

```
Top 10 rows of Payer_plan_relationship table:
['Plan_ID', 'Plan_Name', 'Payer_ID', 'Payer_Name']
(20038, 'Comprehensive Care Benefit', 10000, 'UnitedHealth Group')
(20021, 'Comprehensive Coverage Benefit', 10001, 'Berkshire Hathaway (GEICO)')
(20026, 'Comprehensive Coverage Option', 10002, 'Elevance Health (formerly Anthem)')
(20061, 'Comprehensive Coverage Plan', 10003, 'Centene Corporation')
(20017, 'Comprehensive Coverage Plus', 10004, 'Humana')
(20085, 'Comprehensive Coverage Program', 10005, 'CVS Health (Aetna)')
(20051, 'Comprehensive Health Option', 10006, 'Cigna')
(20068, 'Comprehensive Health Plan', 10007, 'MetLife')
(20005, 'Comprehensive Protection Program', 10008, 'Prudential Financial')
(20063, 'Comprehensive Shield Package', 10009, 'Allstate')
```

Step 3: States and Lives Dataset

- A dataset was created for each Formulary_ID, showing monthly lives covered for every month in 2024 across US states.

```
Top 10 rows of States_and_Lives table:
['Formulary_ID', 'Month', 'State_Code', 'State_Name', 'Lives']
(30000, '1012024', 'MD', 'Maryland', 263501)
(30001, '1012024', 'MI', 'Michigan', 2917482)
(30002, '1012024', 'OK', 'Oklahoma', 2160087)
(30003, '1012024', 'HI', 'Hawaii', 3154351)
(30004, '1012024', 'NJ', 'New Jersey', 2853235)
(30005, '1012024', 'LA', 'Louisiana', 1640154)
(30006, '1012024', 'KY', 'Kentucky', 206462)
(30007, '1012024', 'ND', 'North Dakota', 3447815)
(30008, '1012024', 'IN', 'Indiana', 3373423)
(30009, '1012024', 'AK', 'Alaska', 1213073)
```

- Lives data was extracted and imported in a separate table since geographic insights are not in the scope of this project.

```
Top 10 rows of Lives table:
['Formulary_ID', 'Month', 'Lives']
(30000, 1012024, 263501)
(30000, 1022024, 2217587)
(30000, 1032024, 1259309)
(30000, 1042024, 441585)
(30000, 1052024, 2062434)
(30000, 1062024, 1572063)
(30000, 1072024, 1891156)
(30000, 1082024, 1152539)
(30000, 1092024, 277679)
(30000, 1102024, 305944)
```

Step 4: Plan Sales Dataset

- A table was generated simulating monthly drug sales data for each Plan_ID in 2024.

```
Top 10 rows of Sales table:
['Plan_ID', 'Month', 'Drug_Name', 'Units_Sold']
(20000, 1012024, 'Alunbrig', 140)
(20000, 1012024, 'Alecensa', 181)
(20000, 1012024, 'Zykadia', 190)
(20000, 1012024, 'Xalkori', 118)
(20000, 1012024, 'Avastin', 197)
(20000, 1022024, 'Alunbrig', 120)
(20000, 1022024, 'Alecensa', 135)
(20000, 1022024, 'Zykadia', 117)
(20000, 1022024, 'Xalkori', 89)
(20000, 1022024, 'Avastin', 185)
```

Step 5: Medications Dataset

- A master reference dataset was created randomly assigning 6-digit Med_IDs to each of the 5 drugs.
- Additionally, the current unitary market price of the medications was assigned to the corresponding Med_ID.

```
Top 10 rows of Med_ID table:
['Med_ID', 'Drug_Name', 'Price_Normalized']
(244042, 'Alunbrig', 725)
(312386, 'Alecensa', 84)
(675243, 'Zykadia', 85)
(267912, 'Xalkori', 293)
(474795, 'Avastin', 210)
```

Step 6: Formulary Restrictions Dataset

- A table was developed simulating formulary restrictions randomly applied to each Med_ID under each Formulary_ID in 2024.

- Each Formulary_ID was randomly assigned 1 to 5 Med_IDs, each tagged with a restriction level (Low, Medium, High).

```
Top 10 rows of Restrictions table:
['Formulary_ID', 'Month', 'Med_ID', 'Restriction']
(30000, 1012024, 244042, 'Medium')
(30000, 1012024, 312386, 'Medium')
(30000, 1012024, 675243, 'Low')
(30000, 1012024, 267912, 'Low')
(30000, 1022024, 244042, 'Medium')
(30000, 1022024, 312386, 'Medium')
(30000, 1022024, 675243, 'High')
(30000, 1022024, 267912, 'High')
(30000, 1032024, 244042, 'Medium')
(30000, 1032024, 312386, 'Low')
```

Step 7: Contracts Dataset

- A subset of Payer_IDs was randomly chosen and assigned flag 1 to indicate they were currently in contract with Takeda.

```
Top 10 rows of Contracted_Payer table:
['Payer_ID', 'Flag']
(10002, 1)
(10004, 1)
(10008, 1)
(10010, 1)
(10013, 1)
(10023, 1)
(10025, 1)
(10027, 1)
(10028, 1)
(10029, 1)
```

(Refer Appendix 2)

4. Data Integrity & Consistency Measures

Throughout the process, the following controls were implemented to improve data quality:

- Unique ID enforcement across entities.
- Non-repetition of Plan/Formulary names and IDs per company.
- Logical alignment between sales data, medications data, and insurance structures.
- Total lives distributed to match realistic U.S. demographics (~340 million lives).
- Time-series consistency ensured for all 2024 datasets.

This synthetic data forms the foundation for analysing Alunbrig's sales performance against competitors and assessing the impact of insurance-related access barriers. The structured process ensures robust, relational datasets aligned with the project's goals.

(Refer Appendix 2)

5. Database Creation and Data Import

To aggregate and analyse sales data through various insurance companies (payers) SQLite has been used. The goal is to generate an aggregated dataset for further analysis and insights generation. The entire process was carried out using Python.

5.1. Database Setup

Tools Used:

- **SQLite** for database creation and querying.
- **pandas** for exporting the final dataset to CSV.
- **csv module** for importing data from CSV files.

A SQLite database named Takeda_sales.db was created with Write-Ahead Logging (WAL) mode enabled for improved concurrency.

5.2. Table Definitions and Creation

1. **Restrictions Table**: Stores information about formulary restrictions for different drugs across various months.
   Primary Key (Composite): Formulary_ID, Month, Med_ID
2. **Contracted_Payer Table**: Stores information about payers and their contracted status.
   *Primary Key***:** Payer_ID
3. **Med_ID Table**: Stores drug names with their corresponding IDs and normalised prices for 1 unit of drug.
   *Primary Key***:** Med_ID
4. **Plans_Controller Table**: Maps formularies to their respective plans.
   Primary Key (Composite): (Plan_ID, Formulary_ID)
5. **Lives Table**: Contains data about covered lives for specific formularies and months. This is derived from the States_and_Lives Table
   Primary Key (Composite): (Formulary_ID, Month)
6. **Sales Table**: Sales data related to units sold for each drug under different plans and months.
   Primary Key (Composite): (Plan_ID, Month, Drug_Name)
7. **Payer_plan_relationship Table**: Contains relationships between insurance plans and their corresponding payers. This is a hierarchy table.
   Primary Key: Plan_ID

(Refer Appendix 3)

5.3. Importing Data from CSV Files

Data from CSV files is imported into relevant tables using a batch insert approach for efficiency. The import function ensures duplicates are ignored.

(Refer Appendix 4)

5.4. Data Normalisation

The States_and_Lives table was used to extract the lives in order to normalise the data and improve query efficiency.

(Refer Appendix 4)

5.5. Data Aggregation

A complex SQL query was executed to join tables, aggregate data, and rank records based on total lives for each drug under specific plans.

Explanation of the Query -

1. **Ranked Data (CTE):** The ranked_data CTE calculates the rank of each record by ordering the total lives per month, drug name, and plan name in ascending order.
    a. **Join tables**: Join Sales, Plans_Controller, Payer_plan_relationship, Plans_Controller, Contracted_Payer, Med_ID, and Restrictions to get all relevant information.
    b. **Calculate revenue**: Multiply Units_Sold by Price_Normalized to obtain the revenue for each drug through each plan and month.
    c. **Aggregate data**: Summarise lives for each drug for each plan because Lives are available at a formulary level which needs to be pivoted up to a plan level.
2. **Rank the data:** The ROW_NUMBER() function ranks rows based on Total_Lives for each month, drug, and plan name, ordered from largest to smallest. Since each formulary lists one restriction, the most relevant restriction per plan is selected by identifying the formulary with the highest enrolled lives. This approach ensures accurate data representation at plan level, reflecting the most commonly applicable restriction for each plan.
3. **Final_DataSet (CTE):** Joins the ranked data with payer information and contractual status to identify which plans have contracted payers. This also adds a Contracted_Payer_Flag to indicate if the payer is contracted with currently.
4. **Output:** The final result includes month-wise sales, total lives, sales revenue, and payer information. The output is saved as a CSV file named aggregated_data.csv.

(Refer Appendix 5)

The project posed challenges, including data generation limitations due to inaccessible real-world data, requiring randomised assumptions and multiple iterations for realistic demographics. For example, sales were assumed between 80-200 units, considering the oncology market's niche nature, and randomly allocated lives to plans, ensuring they didn't exceed the population. ID hierarchy complexity required bridge tables, and state-level analysis was excluded due to data limitations. Ensuring ID consistency, removing duplicates, and preparing data for real-world integration pose future challenges.

This process demonstrates a systematic approach to creating, populating, and querying a SQLite database, enabling further analysis like sales trends and market share evaluation.

(Complete code available in Appendix 8)

6. Business Insights

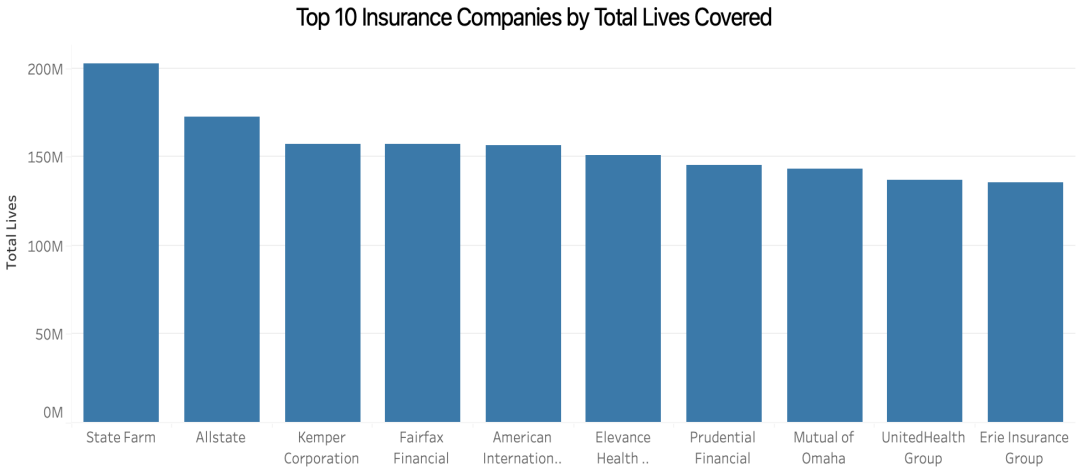
The development of business insights began with exploratory data analysis (EDA) to identify market trends and anomalies. This step ensured the data was clean, well-structured, and aligned with business objectives, providing a solid foundation for deeper analysis.

6.1. Approach:

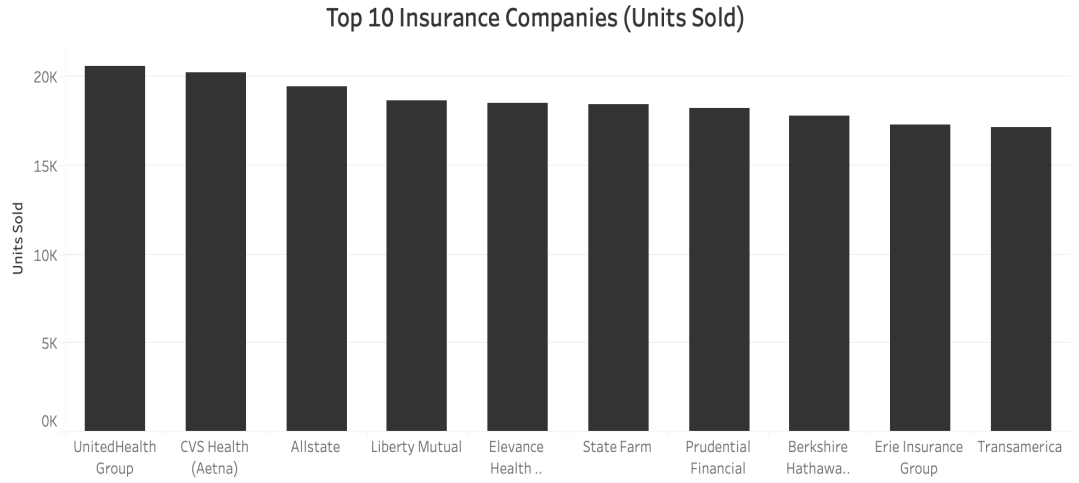
- **Data Aggregation**: The data was aggregated monthly to evaluate the performance of each drug, payer, and restriction type. This provided insights into trends and changes over time.
- **Exploratory Data Analysis**: EDA was performed initially to understand the basic market dynamics. (Refer Appendix 6)
- **Key SQL Queries**: SQL queries were designed to generate metrics such as monthly sales, revenue per drug per month and payer share, facilitating the identification of key patterns and insights.

6.2. Key Insights Generated:

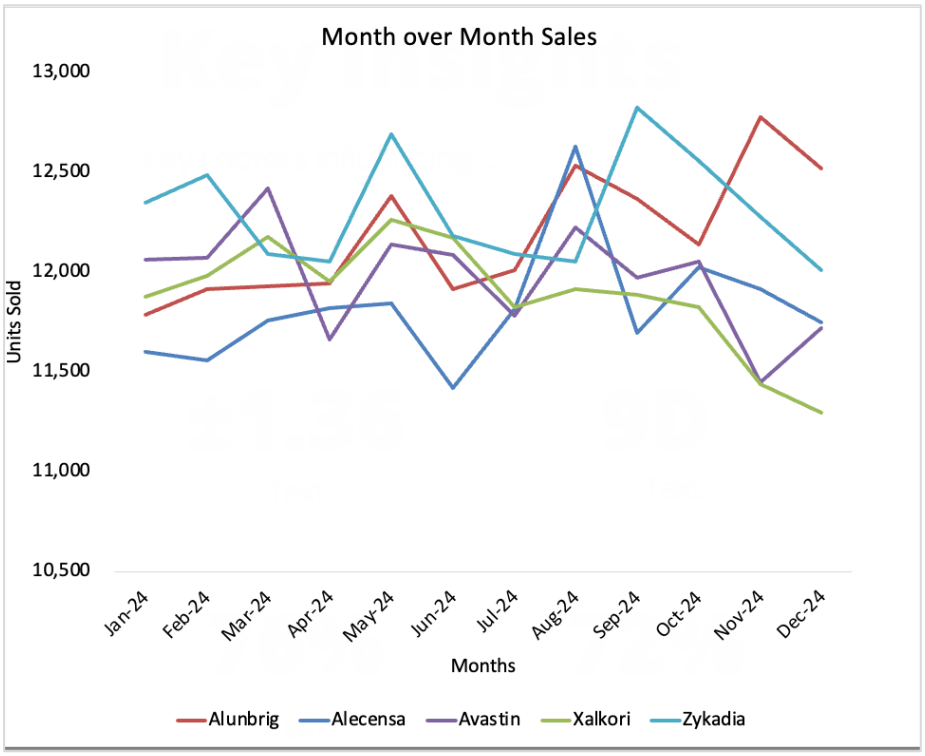
1. **Top Payer by Lives Covered**: Identified the top 10 payers with the highest number of people enrolled in 2024.



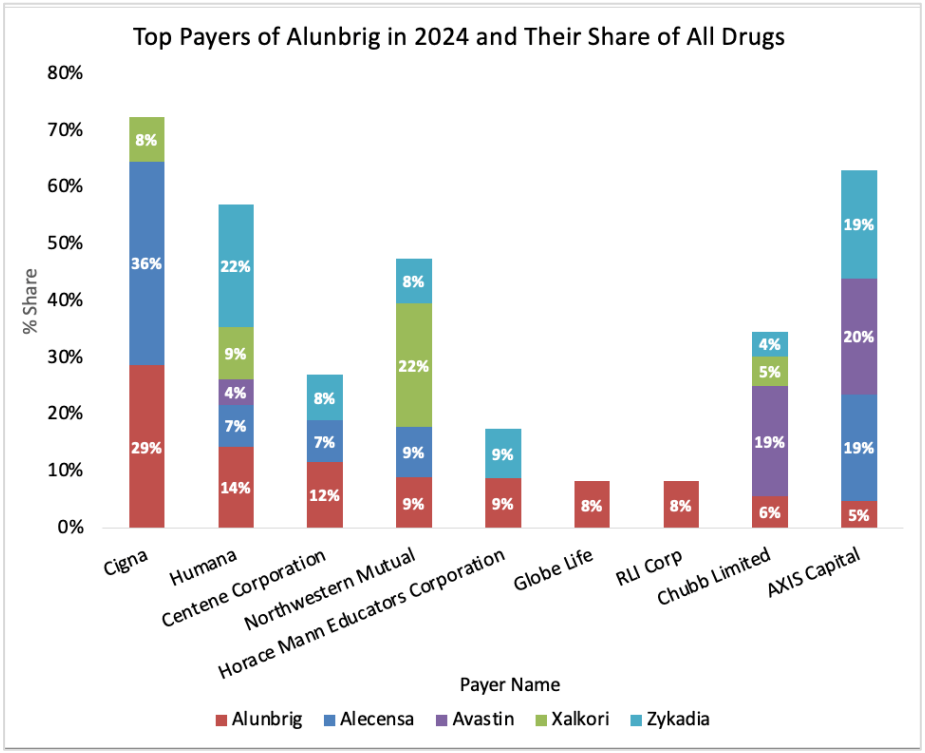
2. **Top Payer by Units Sold**: Identified the top 10 payers driving the most sales for all drug on a monthly basis in 2024.



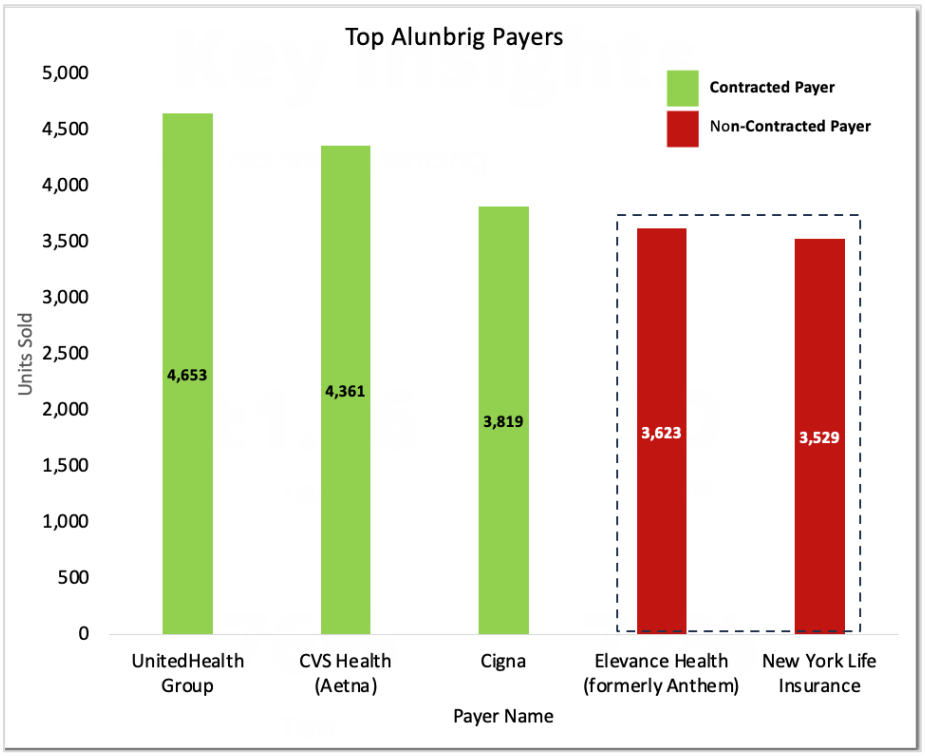
3. **Month-over-Month Sales Trends**: Sales data aggregated monthly to identify growth and decline patterns for Alunbrig and other drugs.



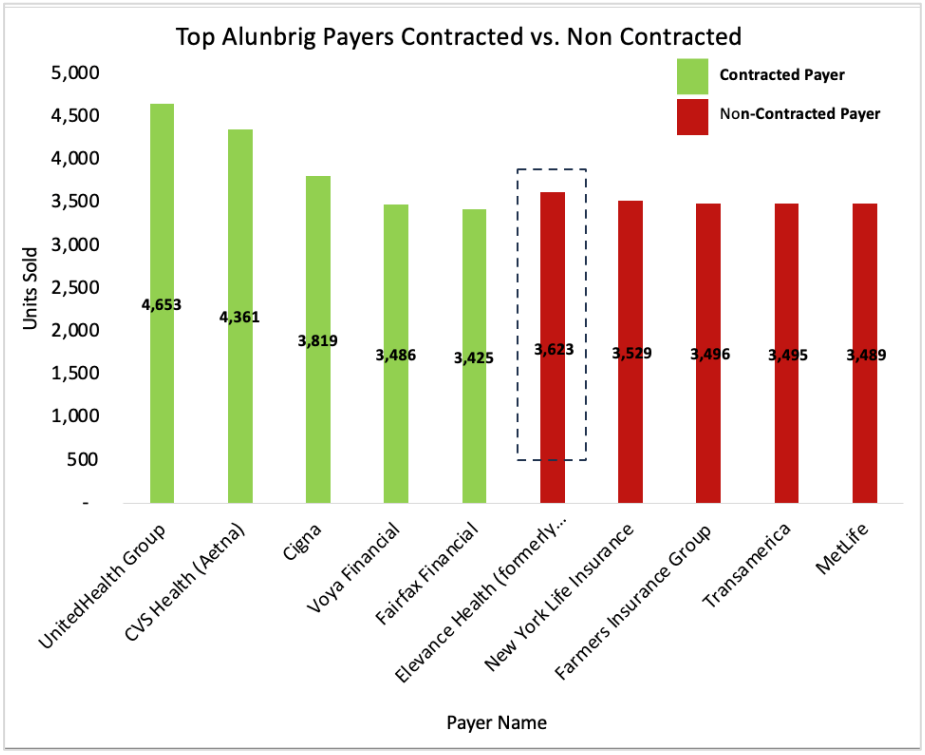
4. **Market Share Analysis**: Examined the share contribution of top payers for Alunbrig and other competing products in 2024.



5. **Top 5 Payers for Alunbrig Sales**: Analysis highlighted the top 5 payers responsible for the highest volume of Alunbrig units sold.



6. **Contracted vs Non-Contracted Payers**: Comparison of contracted versus non-contracted payers to determine which group contributed more to Alunbrig's sales.



(Refer Appendix 7)

By utilising SQLite queries and performing EDA, a comprehensive view of Alunbrig's sales landscape was developed. The insights revealed key trends in drug sales, payer behaviour, and revenue generation, providing valuable information to support future strategic decisions.

7. Conclusion

The analysis successfully developed a comprehensive data product that simulated U.S. healthcare insurance and pharmaceutical landscape, focusing on Alunbrig and its competitors. The process involved designing a robust database, generating synthetic data, and leveraging SQLite for efficient querying. Through detailed querying and visualisation, valuable business insights were generated to support strategic decision-making. The findings highlighted key trends in sales performance, payer behaviour, and revenue generation, revealing differences between contracted and non-contracted payers. Additionally, the analysis provided insights into restriction impacts, top-selling drugs, and market share dynamics. These insights offer a clear understanding of market patterns and payer strategies, guiding efforts to optimise contracts and enhance market positioning. While the current analysis offers actionable insights, further exploration of demographic factors and competitor strategies could enhance the data product's value and provide a more comprehensive foundation for strategic decisions.

8. Bibliography

- ALUNBRIG (no date) *ALUNBRIG® (brigatinib)*. Available at: https://www.alunbrig.com/ (Accessed: 15 March 2025).
- Cancer Research UK (2025) *Brigatinib (Alunbrig)*. Available at: https://www.cancerresearchuk.org/about-cancer/treatment/drugs/brigatinib (Accessed: 15 March 2025).
- Drugs.com (2025a) *Alunbrig prices, coupons, copay cards & patient assistance*, *Drugs.com*. Available at: https://www.drugs.com/price-guide/alunbrig (Accessed: 15 March 2025).
- Drugs.com (2025b) *Alecensa prices, coupons, copay cards & patient assistance*, *Drugs.com*. Available at: https://www.drugs.com/price-guide/alecensa (Accessed: 15 March 2025).
- Drugs.com (2025c) *Avastin prices, coupons, copay cards & patient assistance*, *Drugs.com*. Available at: https://www.drugs.com/price-guide/avastin (Accessed: 15 March 2025).
- Drugs.com (2025d) *Xalkori prices, coupons, copay cards & patient assistance*, *Drugs.com*. Available at: https://www.drugs.com/price-guide/xalkori (Accessed: 15 March 2025).
- Drugs.com (2025e) *Zykadia prices, coupons, copay cards & patient assistance*, *Drugs.com*. Available at: https://www.drugs.com/price-guide/zykadia (Accessed: 15 March 2025).
- Medical News Today (no date). *Alunbrig: Alternatives, side effects, dosage, and more.* Available at: https://www.medicalnewstoday.com/articles/drugs-alunbrig#generic (Accessed: 15 March 2025).
- Takeda (no date) *Takeda Pharmaceuticals: Global Homepage*. Available at: https://www.takeda.com/ (Accessed: 15 March 2025).
- United States Government (2025) *Population clock*, *United States Census Bureau*. Available at: https://www.census.gov/popclock/ (Accessed: 15 March 2025).

9. Appendices

Appendix 1

Tables and Relationships

Table Name	Primary Key	Foreign Keys	Purpose
Plans_Controller	Plan_ID	None	Central table for insurance plans
Restrictions	(Formulary_ID, Month, Med_ID)	Formulary_ID (Plans_Controller), Med_ID	Defines drug restrictions per formulary
Medications	Med_ID	None	Stores drug information
Sales	(Plan_ID, Month, Drug_Name)	Plan_ID (Plans_Controller)	Tracks sales data per plan and drug
States_and_Lives	(Formulary_ID, Month, State_Name)	Formulary_ID (Plans_Controller)	Tracks covered lives per formulary and state
Lives	(Formulary_ID, Month)	Formulary_ID (Plans_Controller)	Simplified table derived from States_and_Lives
Contracted_Payer	Payer_ID	None	Stores payer contract details
Payer_Plan_Relationship	Plan_ID	Plan_ID (Plans_Controller), Payer_ID	Links insurance plans to payers

Appendix 2

Data Generation Prompts

- Step 1: “Consider US's 50 largest Insurance companies. Assign each company to a distinct 5-digit Payer_ID. Generate a Plans_Controller table file with four columns: Payer_ID, Insurance_Plan, Formulary_ID and Company_Name. For each of these 50 US insurance companies, please produce between 2 and 7 random names for insurance plans. For each of these 50 US insurance companies with varied plans, please produce between 2 and 7 random names for formulary names within each plan. Ensure that Plan IDs and Formulary IDs are unique to each insurance company.”
- Step 2: “Consider that each payer can have multiple plans. Generate a new table in which the 112 plans are randomly assigned to Payer_IDs. This file should include the following columns: Plan_ID, Plan_Name, Payer_ID and Payer_Name.”
- Step 3: “For each of the already generated Formulary_IDs, generate a new States_and_Lives table with columns: Formulary_ID, Month, State_Code, State_Name and Lives. The month data should cover all months in 2024, it should always refer to the first day of each month and be formatted as DDMMYYYY. Randomly assign US states to the formularies. The Lives column should have 300 million lives in total across all formularies so split them randomly but their total should be 300M. Please generate a new file.”
- Step 4: “For each of the already generated Plan IDs, generate a new Sales table with columns: Plan_ID, Month, Drug_Name, Units_Sold. The month data should cover all months in 2024, it should always refer to the first day of each month and be formatted as DDMMYYYY. The Drug Names should be Alunbrig, Alecensa, Zykadia, Xalkori and Avastin. Randomly assign units sold values to each drug ranging from 80 to 200.”
- Step 5: “Generate a Med_ID table including the following columns: Med_ID, Drug_Name and Price_Normalized. Randomly generate a unique 6-digit Med_ID for each of the drugs: Alunbrig, Alecensa, Avastin, Xalkori and Zykadia. Also assign the normalised price of each drug as 725 for Alunbrig, 84 for Alecensa, 85 for Zykadia, 293 for Xalkori, and 210 for Avastin.”
- Step 6: “Generate a Restrictions table with: Formulary_ID, Month, Med_ID, and Restriction. The month data should cover all months in 2024, it should always refer to the first day of each month and be formatted as DDMMYYYY. For the Med ID column use the data previously generated. For the Formulary_ID column use the data previously generated. Randomly assign between 1 to 5 Med_ID values to each Formulary_ID. For the Restriction column assign random values ranging between Low, Medium and High for each Med_ID.”

- Step 7: "Generate a new Contract_Data table with columns: Payer_ID and Flag. Randomly select 20 Payer_IDs and assign them with a flag of value 1."

Appendix 3

Create Tables

```
import sqlite3
import csv

# Create a connection to the database with a timeout to avoid locking issues
conn = sqlite3.connect('Takeda_sales.db', timeout=10)
cursor = conn.cursor()

# Set WAL (Write-Ahead Logging) mode for better concurrency
cursor.execute("PRAGMA journal_mode = WAL;")
conn.commit()

# Creating Tables
cursor.execute('''
CREATE TABLE IF NOT EXISTS Restrictions (
    Formulary_ID INT NOT NULL,
    Month INT NOT NULL,
    Med_ID INT NOT NULL,
    Restriction VARCHAR(10) NOT NULL,
    PRIMARY KEY (Formulary_ID, Month, Med_ID)
);
''')

cursor.execute('''
CREATE TABLE IF NOT EXISTS Contracted_Payer (
    Payer_ID INT PRIMARY KEY,
    Flag INT NOT NULL
);
''')

cursor.execute('''
CREATE TABLE IF NOT EXISTS Med_ID (
    Med_ID INT PRIMARY KEY,
    Drug_Name VARCHAR(255) NOT NULL,
    Price_Normalized INT
```

```
);
'')

cursor.execute('''
CREATE TABLE IF NOT EXISTS Plans_Controller (
    Plan_ID INT,
    Insurance_Plan VARCHAR(255) NOT NULL,
    Formulary_ID INT,
    Formulary_Name VARCHAR(255) NOT NULL,
    PRIMARY KEY (Plan_ID, Formulary_ID)
);
'')

cursor.execute('''
CREATE TABLE IF NOT EXISTS States_and_Lives (
    Formulary_ID INT,
    Month VARCHAR(10),
    State_Code VARCHAR(10),
    State_Name VARCHAR(255),
    Lives INT,
    PRIMARY KEY (Formulary_ID, Month, State_Code)
);
'')

cursor.execute('''
CREATE TABLE IF NOT EXISTS Sales (
    Plan_ID INT,
    Month INT,
    Drug_Name VARCHAR(255),
    Units_Sold INT,
    PRIMARY KEY (Plan_ID, Month, Drug_Name)
);
'')

cursor.execute('''
CREATE TABLE IF NOT EXISTS Lives (
```

```
    Formulary_ID INT,
    Month INT,
    Lives INT,
    PRIMARY KEY (Formulary_ID, Month)
);
''')

cursor.execute('''
CREATE TABLE IF NOT EXISTS Payer_plan_relationship (
    Plan_ID INT,
    Plan_Name VARCHAR(255),
    Payer_ID INT,
    Payer_Name VARCHAR(255),
    PRIMARY KEY (Plan_ID)
);
''')

# Commit table creation
conn.commit()

# Inserting data into Payer_plan_relationship table (Example)

# Print success message
print("Database and tables created successfully!")

# Closing the connection
conn.close()
```

Appendix 4

Importing Data from CSV and Extracting Lives

```
import sqlite3
import csv

# Establish connection and cursor for executing queries
conn = sqlite3.connect('Takeda_sales.db', timeout=10)
cursor = conn.cursor()

# Set WAL (Write-Ahead Logging) mode for better concurrency
cursor.execute("PRAGMA journal_mode = WAL;")
conn.commit()

# Function to import CSV data into the table
def import_csv_to_table(csv_file, table_name, ignore_duplicates=False, batch_size=1000):
    data = []
    try:
        with open(csv_file, 'r', encoding='utf-8') as file:
            csv_reader = csv.reader(file)
            next(csv_reader)  # Skip the header row if present
            for row in csv_reader:
                data.append(row)
                if len(data) >= batch_size:
                    placeholders = ', '.join(['?' for _ in row])
                    sql = f"INSERT OR IGNORE INTO {table_name} VALUES ({placeholders})"
                    cursor.executemany(sql, data)  # Batch insert
                    conn.commit()  # Commit every batch
                    data = [] # Reset the data list after committing

            if data:  # Insert any remaining rows
                placeholders = ', '.join(['?' for _ in data[0]])
                sql = f"INSERT OR IGNORE INTO {table_name} VALUES ({placeholders})"
                cursor.executemany(sql, data)
                conn.commit()  # Final commit
    except Exception as e:
        print(f"Error importing data from {csv_file} into {table_name}: {e}")
        conn.rollback()  # Rollback changes in case of an error
```

```
# Import data from CSV files into the relevant table with `IGNORE` for duplicates
try:
    import_csv_to_table('Formulary_Restrictions_2024.csv', 'Restrictions',
ignore_duplicates=True)
    import_csv_to_table('Contract_Data.csv', 'Contracted_Payer', ignore_duplicates=True)
    import_csv_to_table('Med_ID_Data.csv', 'Med_ID', ignore_duplicates=True)
    import_csv_to_table('US_Insurance_Companies_With_Unique_IDs.csv', 'Plans_Controller',
ignore_duplicates=True)
    import_csv_to_table('Plan_Sales_Distribution_2024.csv', 'Sales', ignore_duplicates=True)
    import_csv_to_table('States_and_Lives_2024.csv', 'States_and_Lives',
ignore_duplicates=True)
    import_csv_to_table('Payer_Plan_Bridge.csv', 'Payer_plan_relationship',
ignore_duplicates=True)
    print("Data imported successfully!")
except Exception as e:
    print(f"An error occurred during import: {e}")
    conn.rollback()  # Rollback changes if an error occurred

# Extract the Lives data into 'Lives' tables
try:

    # Inserting into Lives table
    cursor.execute('''
    INSERT INTO Lives (Formulary_ID, Month, Lives)
    SELECT Distinct Formulary_ID, Month, Lives
    FROM States_and_Lives;
    ''')
    # Commit the changes
    conn.commit()
    print("Data split into Lives tables successfully!")
except Exception as e:
    print(f"An error occurred during data split: {e}")
    conn.rollback()  # Rollback changes if an error occurred

# Finally, close the connection after all operations
conn.close(
```

Appendix 5

Database Creation

```
import sqlite3
import pandas as pd

# Create a connection to the SQLite database (modify the path to your actual database
file)
conn = sqlite3.connect('Takeda_sales.db')

# SQL Query to join tables and aggregate data
sql_query = """
With ranked_data as (
  Select
    *,
   ROW_NUMBER() OVER (
     PARTITION BY Month,
     Drug_Name,
     Plan_Name
     ORDER BY
       Total_Lives DESC
   ) AS row_num
  from
   (
     Select
       a.Month,
       b.Plan_ID,
       a.Plan_Name,
       a.Drug_Name,
       Sum(a.Lives) as Total_Lives,

       a.Restriction,
       b.Units_Sold,
       b.Price_Normalized,
       b.Sales_Revenue
     from
       (
```

```
        Select
          a.Month,
          b.Formulary_ID,
          b.Formulary_Name,
          b.Plan_ID,
          b.Insurance_Plan as Plan_Name,
          a.Drug_Name,
          b.Lives,

          a.Restriction
        from
          (
            select
              *
            from
              restrictions r
              left join (Select Drug_name,Med_ID from Med_ID) m on r.med_id = m.med_id
          ) a
          Inner Join (
            Select
              *
            from
              Lives l
              left Join plans_controller pc on pc.Formulary_ID = l.Formulary_ID

          ) b on a.formulary_id = b.formulary_id
          and a.month = b.month
      ) a
      Inner Join (
       Select a.*, b.Price_Normalized, Units_sold * Price_Normalized AS Sales_Revenue
from(
        Select
          *
        from
          Sales
          ) a
```

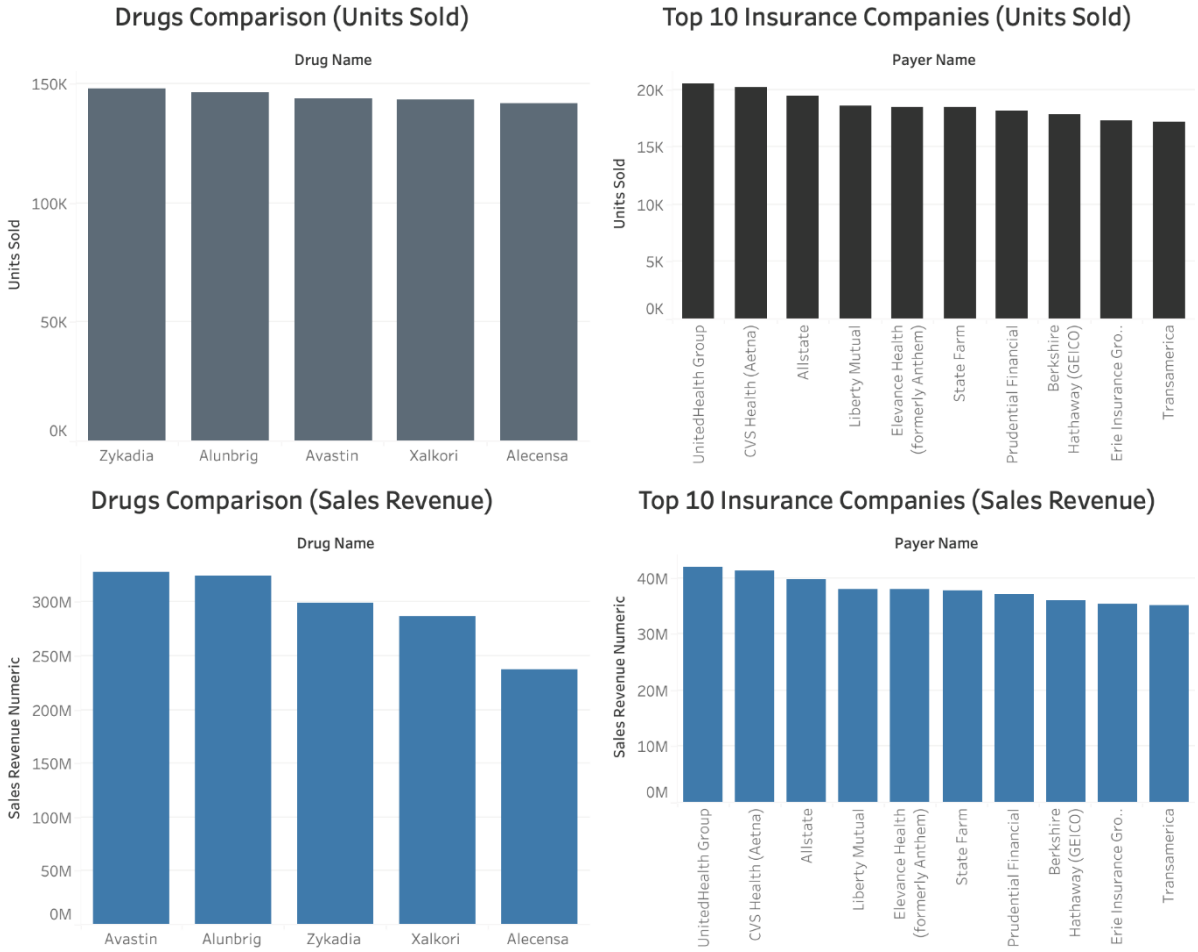
```
        Left join (Select Drug_Name, Price_Normalized from Med_ID) b on a.Drug_Name
= b.Drug_Name

      ) b on a.Plan_ID = b.Plan_ID
      and a.Month = b.Month
      and a.Drug_Name = b.Drug_Name
    group by
      a.Month,
      b.Plan_ID,
      a.Plan_Name,
      a.Drug_Name,

      a.Restriction,
      b.Units_Sold,
      b.Price_Normalized,
      b.Sales_Revenue
  )
),
Final_DataSet as (
 Select
  a.*,
  CASE WHEN c.Flag IS NULL THEN 0 ELSE c.Flag END AS Contracted_Payer_Flag
 FROM
  (
   SELECT
    a.Month,
    a.Plan_ID,
    a.Plan_Name,
    b.Payer_ID,
    b.Payer_Name,
    a.Drug_Name,
    a.Total_Lives,

    a.Restriction,
    a.Units_Sold,
    a.Price_Normalized,
```

```
        a.Sales_Revenue
      FROM
       (
        SELECT
          Month,
          Plan_ID,
          Plan_Name,
          Drug_Name,
          Total_Lives,

          Restriction,
          Units_Sold,
          Price_Normalized,
          Sales_Revenue
        FROM
          ranked_data
        WHERE
          row_num = 1
       ) a
       LEFT JOIN (
        SELECT
          Plan_ID,
          Payer_ID,
          Payer_Name
        FROM
          Payer_plan_relationship
       ) b ON a.Plan_ID = b.Plan_ID
    ) a
    Left Join(
     Select
       *
     from
       Contracted_Payer
    ) c on a.Payer_ID = c.Payer_ID
)
Select
```

```
  *
from
  Final_DataSet;
"""

# Execute the SQL query and load the result into a DataFrame
Final_data_df = pd.read_sql_query(sql_query, conn)

print(Final_data_df)
# Save the DataFrame to a CSV file
Final_data_df.to_csv('aggregated_data.csv', index=False)

# Print confirmation
print("CSV file has been created successfully!")

# Close the connection
conn.close()
```

Appendix 6

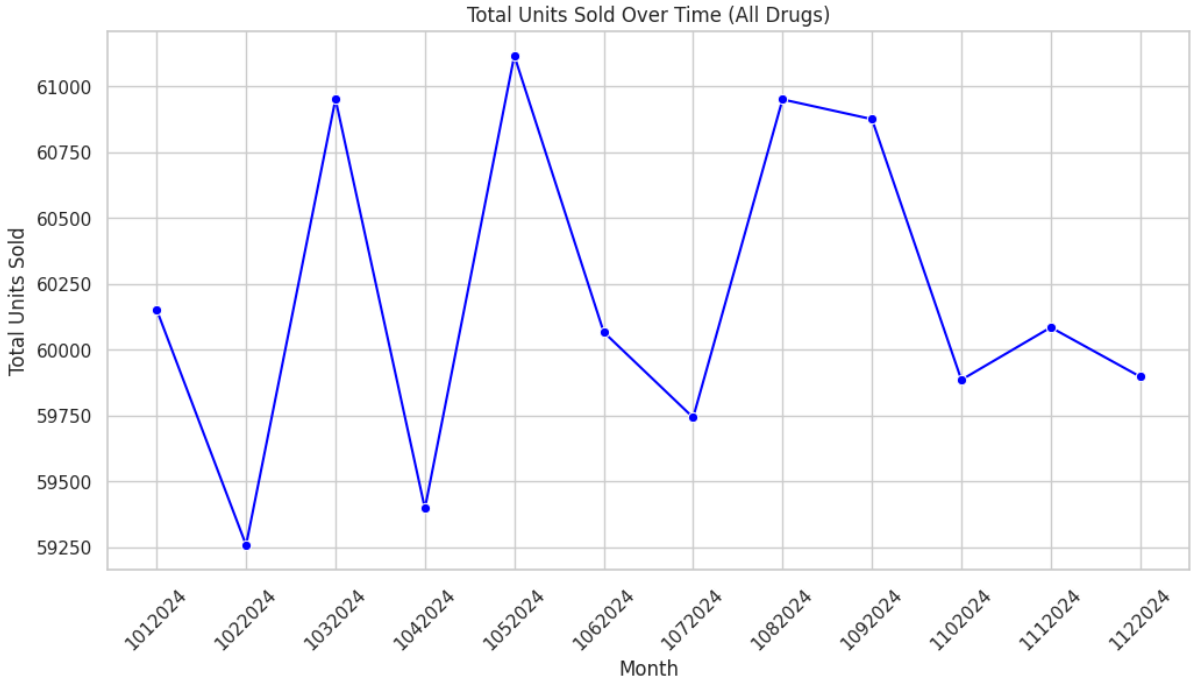
EDA

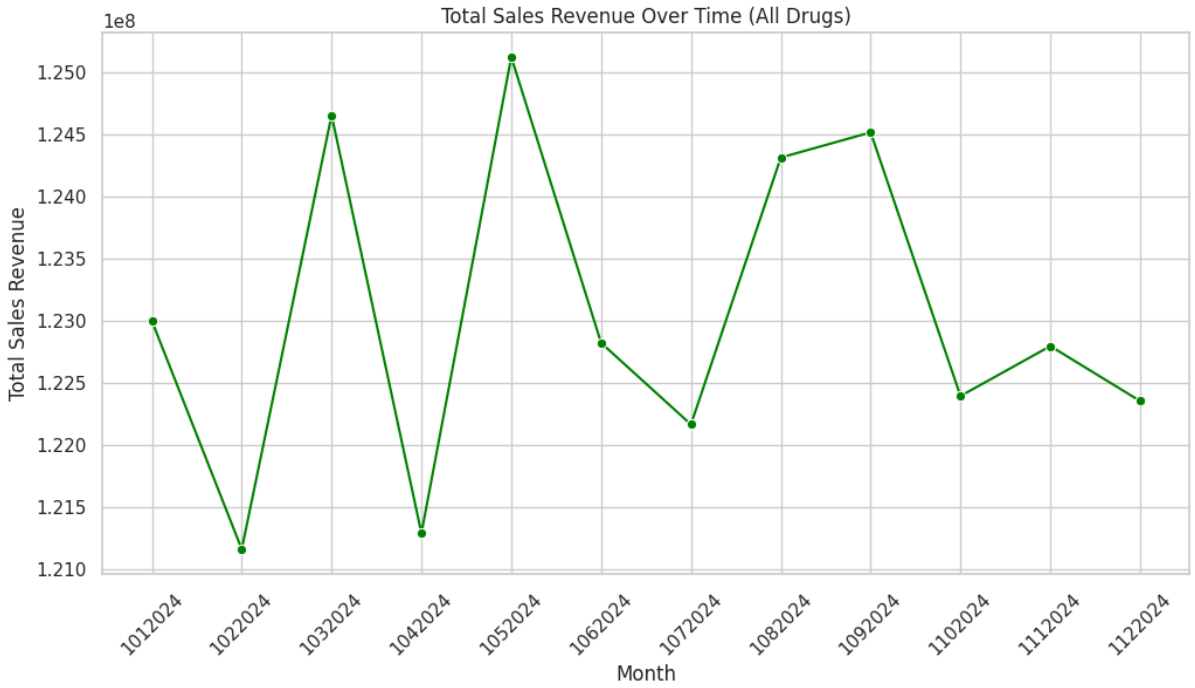


These four charts provide a clear breakdown of how different drugs and insurance companies perform in terms of units sold and revenue. The top-left chart compares drug sales volume, showing that Zykadia, Alunbrig, Avastin, Xalkori, and Alecensa are all selling at similar levels, suggesting strong competition. The top-right chart ranks the top 10 insurance companies by units sold, with UnitedHealth Group and CVS Health (Aetna) leading the market. The bottom-left chart shifts the focus to revenue, where Avastin and Alunbrig generate the highest earnings, indicating that having a higher level of sales does not always mean higher revenue. Lastly, the bottom-right chart highlights the top insurance companies by revenue, where UnitedHealth Group again dominates, reinforcing its strong presence in both sales volume and earnings. Overall, these insights help us understand which drugs are in demand and which insurers are driving the most business.

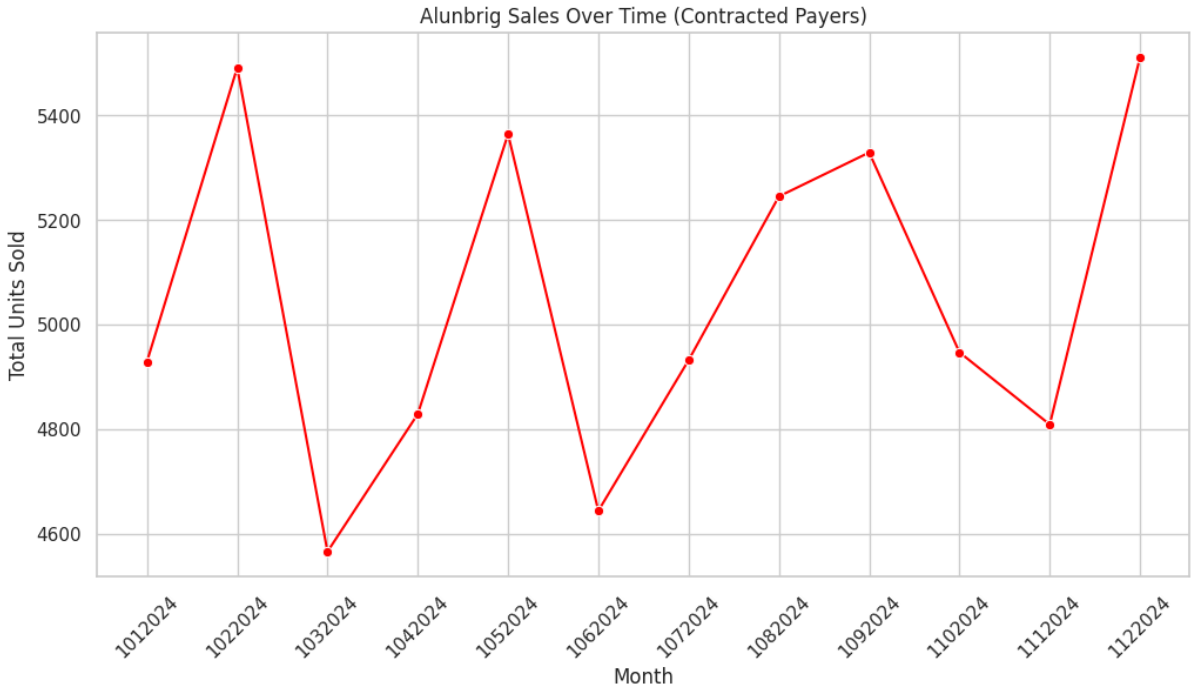
Synthetic data summary:

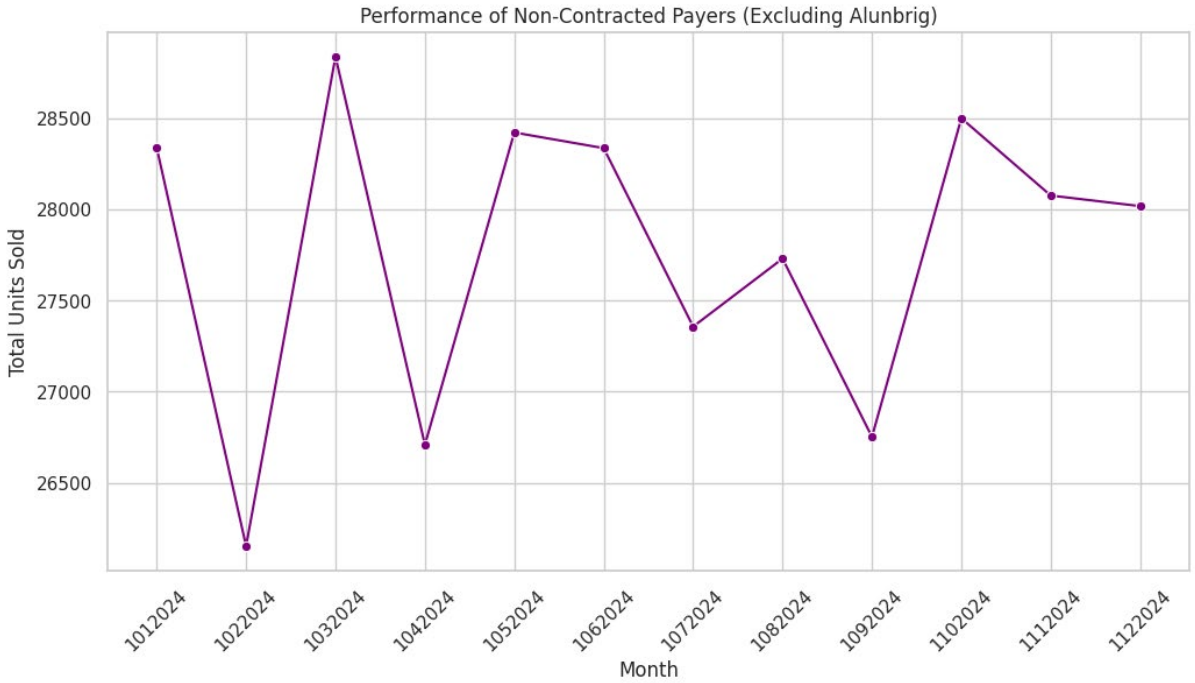
Metric	Value
Month Range	Jan 2024 - Dec 2024
Total Unique Plans	112
Total Unique Payers	50
Total Unique Drugs	5
Total Unique Restrictions	3 (Low, Medium, High)
Contracted Payers	20
Non-Contracted Payers	30
Total Lives covered in a month (Alunbrig)	317 Million (US Population)





The first two charts illustrate how total units sold and total sales revenue fluctuate over time for all drugs combined. The first chart (blue) shows the total number of units sold per month, revealing noticeable peaks and dips, suggesting seasonal demand or external factors influencing sales. The second chart (green) presents total sales revenue over time, which follows a similar fluctuating pattern, implying that revenue is largely dependent on volume rather than drastic price changes. Both charts indicate a cyclical trend, which could be due to insurance coverage cycles, promotional efforts, or prescribing patterns.


Alunbrig Sales Over Time (Contracted Payers)
Total Units Sold
5400
5200
5000
4800
4600
1012024
1022024
1032024
1042024
1052024
1062024
1072024
1082024
1092024
1102024
1112024
1122024
Month


Performance of Non-Contracted Payers (Excluding Alunbrig)
Total Units Sold
28500
28000
27500
27000
26500
1012024
1022024
1032024
1042024
1052024
1062024
1072024
1082024
1092024
1102024
1112024
1122024
Month

Appendix 7

Business Insights generation

```
# Insights Generation
import sqlite3
import pandas as pd

# Reconnect to the SQLite database
conn = sqlite3.connect('Takeda_sales.db')
cursor = conn.cursor()

# Load the saved data into a DataFrame
Final_data_df = pd.read_csv('/content/aggregated_data.csv')

# Save the DataFrame into a temporary table in SQLite
Final_data_df.to_sql('Final_DataSet', conn, if_exists='replace', index=False)

#Month over Month sale of each drug
query = """
   SELECT
      Month,
      Drug_Name,
      SUM(Units_Sold) AS Total_Units_Sold_Drug
   FROM
      Final_DataSet
   GROUP BY
      Drug_Name, Month
      Order by Month, Total_Units_Sold_Drug ;
"""

result_df = pd.read_sql_query(query, conn)
result_df.to_csv('MoM_Sales.csv', index=False)
print(result_df)

# Which restriction has most Lives for each and
#how much revenue they generate for each drugs
query = """
```

```
SELECT Restriction,Drug_Name, AVG(Total_Lives) AS Avg_Lives_Covered,
SUM(CAST(REPLACE(Sales_Revenue, ',', '') AS INTEGER)) AS Total_Revenue
FROM Final_DataSet
GROUP BY  Restriction, Drug_Name
ORDER BY  Avg_Lives_Covered DESC;
"""

result_df = pd.read_sql_query(query, conn)
result_df.to_csv('restriction_vs_lives_revenue.csv', index=False)
print(result_df)

#Most Drug sold in 2024 by Units and their revenue generated
query = """
SELECT Drug_Name, SUM(Units_Sold) AS Total_Units,
SUM(CAST(REPLACE(Sales_Revenue, ',', '') AS INTEGER)) AS Total_Revenue
FROM Final_DataSet
GROUP BY Drug_Name
ORDER BY Total_Units DESC
;
"""
result_df = pd.read_sql_query(query, conn)
result_df.to_csv('top_drugs_units.csv', index=False)
print(result_df)

# Share for each drug within each payer Month over Month
query = """
WITH Total_units_by_Drug AS (
   SELECT
      Month,
      Payer_ID,
      Payer_Name,
      Drug_Name,
      Contracted_Payer_Flag,
      SUM(Units_Sold) AS Total_Units_Sold_Drug
   FROM
      Final_DataSet
```

```
    GROUP BY
        Payer_ID, Payer_Name, Drug_Name, Month, Contracted_Payer_Flag
),
Total_units AS (
    SELECT
        Month,
        Payer_ID,
        Payer_Name,
        SUM(Units_Sold) AS Total_Units_Sold_Payer,
        Contracted_Payer_Flag
    FROM
        Final_DataSet
    GROUP BY
        Payer_ID, Payer_Name, Contracted_Payer_Flag, Month
)
SELECT
    a.*,
    b.Total_Units_Sold_Payer,
    CAST(Total_Units_Sold_Drug AS FLOAT) / CAST(Total_Units_Sold_Payer AS FLOAT)
AS Drug_share
FROM
    Total_units_by_Drug a
    LEFT JOIN Total_units b
    ON a.Payer_ID = b.Payer_ID
    AND a.Month = b.Month
    Order by Month, Drug_share DESC
    ;
"""
result_df = pd.read_sql_query(query, conn)
result_df.to_csv('MoM_Share.csv', index=False)
print(result_df)

# Top payer with highest share for every brand in each month
query = """
WITH Total_units_by_Drug AS (
    SELECT
```

```
        Month,
        Payer_ID,
        Payer_Name,
        Drug_Name,
        Contracted_Payer_Flag,
        SUM(Units_Sold) AS Total_Units_Sold_Drug
    FROM Final_DataSet
    GROUP BY Payer_ID, Payer_Name, Drug_Name, Month, Contracted_Payer_Flag
),
Total_units AS (
    SELECT
        Month,
        Payer_ID,
        Payer_Name,
        SUM(Units_Sold) AS Total_Units_Sold_Payer,
        Contracted_Payer_Flag
    FROM Final_DataSet
    GROUP BY Payer_ID, Payer_Name, Contracted_Payer_Flag, Month
)
SELECT Month, Payer_ID, Payer_Name, Drug_Name, Contracted_Payer_Flag,
Total_Units_Sold_Drug,
Drug_share
 FROM (
    SELECT
        a.*,
        b.Total_Units_Sold_Payer,
        CAST(Total_Units_Sold_Drug AS FLOAT) / CAST(Total_Units_Sold_Payer AS
FLOAT) AS Drug_share,
        ROW_NUMBER() OVER (
            PARTITION BY a.Month, a.Drug_Name
            ORDER BY
                CAST(Total_Units_Sold_Drug AS FLOAT) / CAST(Total_Units_Sold_Payer AS
FLOAT) DESC
        ) AS row_num
    FROM Total_units_by_Drug a
    LEFT JOIN Total_units b
```

```
        ON a.Payer_ID = b.Payer_ID
        AND a.Month = b.Month
) result
WHERE row_num = 1
ORDER BY Month, Drug_share DESC;
"""
result_df = pd.read_sql_query(query, conn)
result_df.to_csv('Top_payer_each_Month.csv', index=False)
print(result_df)

# 5 Payer with Most Alunbrig Units Sold
query = """
SELECT Payer_ID,Payer_Name, SUM(Units_Sold) AS Total_Units,
Contracted_Payer_Flag
FROM Final_DataSet
Where Drug_Name = "Alunbrig"
GROUP BY Payer_ID,Payer_Name, Contracted_Payer_Flag
ORDER BY Total_Units DESC
LIMIT 5;
"""

result_df = pd.read_sql_query(query, conn)
result_df.to_csv('top_payer_units_Alunbrig.csv', index=False)
print(result_df)

# Top 5 Contracted vs Non-Contracted Payers by Total Units Sold of Alunbrig
query = """
WITH Payer_Sales AS (
    SELECT
        Payer_ID,
        Payer_Name,
        Contracted_Payer_Flag,
        SUM(Units_Sold) AS Total_Units
    FROM Final_DataSet
    Where Drug_Name = "Alunbrig"
    GROUP BY Payer_ID, Payer_Name, Contracted_Payer_Flag
```

```
)
SELECT * FROM (
    SELECT Payer_ID,Payer_Name, Contracted_Payer_Flag, Total_Units
    FROM Payer_Sales
    WHERE Contracted_Payer_Flag = 1
    ORDER BY Total_Units DESC
    LIMIT 5
)
UNION ALL
SELECT * FROM (
    SELECT Payer_ID,Payer_Name, Contracted_Payer_Flag, Total_Units
    FROM Payer_Sales
    WHERE Contracted_Payer_Flag = 0
    ORDER BY Total_Units DESC
    LIMIT 5
);
"""
result_df = pd.read_sql_query(query, conn)
result_df.to_csv('top_payers_Contracted_vs_NC.csv', index=False)
print(result_df)

# Close the connection when done
conn.close()
```

Appendix 8

Complete Code

```
# -*- coding: utf-8 -*-
"""GROUP 17 DM CODE.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1EpEpQly1_ehzdOLOzx5LhtVwCFv0gVF9
"""

import sqlite3
import csv

# Create a connection to the database with a timeout to avoid locking issues
conn = sqlite3.connect('Takeda_sales.db', timeout=10)
cursor = conn.cursor()

# Set WAL (Write-Ahead Logging) mode for better concurrency
cursor.execute("PRAGMA journal_mode = WAL;")
conn.commit()

# Creating Tables
cursor.execute('''
CREATE TABLE IF NOT EXISTS Restrictions (
    Formulary_ID INT NOT NULL,
    Month INT NOT NULL,
    Med_ID INT NOT NULL,
    Restriction VARCHAR(10) NOT NULL,
    PRIMARY KEY (Formulary_ID, Month, Med_ID)
);
''')

cursor.execute('''
CREATE TABLE IF NOT EXISTS Contracted_Payer (
    Payer_ID INT PRIMARY KEY,
    Flag INT NOT NULL
```

```
);
''')

cursor.execute('''
CREATE TABLE IF NOT EXISTS Med_ID (
    Med_ID INT PRIMARY KEY,
    Drug_Name VARCHAR(255) NOT NULL,
    Price_Normalized INT
);
''')

cursor.execute('''
CREATE TABLE IF NOT EXISTS Plans_Controller (
    Plan_ID INT,
    Insurance_Plan VARCHAR(255) NOT NULL,
    Formulary_ID INT,
    Formulary_Name VARCHAR(255) NOT NULL,
    PRIMARY KEY (Plan_ID, Formulary_ID)
);
''')

cursor.execute('''
CREATE TABLE IF NOT EXISTS States_and_Lives (
    Formulary_ID INT,
    Month VARCHAR(10),
    State_Code VARCHAR(10),
    State_Name VARCHAR(255),
    Lives INT,
    PRIMARY KEY (Formulary_ID, Month, State_Code)
);
''')

cursor.execute('''
CREATE TABLE IF NOT EXISTS Sales (
    Plan_ID INT,
    Month INT,
    Drug_Name VARCHAR(255),
```

```
    Units_Sold INT,
    PRIMARY KEY (Plan_ID, Month, Drug_Name)
);
''')

cursor.execute('''
CREATE TABLE IF NOT EXISTS Lives (
    Formulary_ID INT,
    Month INT,
    Lives INT,
    PRIMARY KEY (Formulary_ID, Month)
);
''')

cursor.execute('''
CREATE TABLE IF NOT EXISTS Payer_plan_relationship (
    Plan_ID INT,
    Plan_Name VARCHAR(255),
    Payer_ID INT,
    Payer_Name VARCHAR(255),
    PRIMARY KEY (Plan_ID)
);
''')

# Commit table creation
conn.commit()



# Print success message
print("Database and tables created successfully!")

# Closing the connection
conn.close()

import sqlite3
import csv
```

```
# Establish connection and cursor for executing queries
conn = sqlite3.connect('Takeda_sales.db', timeout=10)
cursor = conn.cursor()

# Set WAL (Write-Ahead Logging) mode for better concurrency
cursor.execute("PRAGMA journal_mode = WAL;")
conn.commit()

# Function to import CSV data into the table
def import_csv_to_table(csv_file, table_name, ignore_duplicates=False, batch_size=1000):
    data = []
    try:
        with open(csv_file, 'r', encoding='utf-8') as file:
            csv_reader = csv.reader(file)
            next(csv_reader)  # Skip the header row if present
            for row in csv_reader:
                data.append(row)
                if len(data) >= batch_size:
                    placeholders = ', '.join(['?' for _ in row])
                    sql = f"INSERT OR IGNORE INTO {table_name} VALUES ({placeholders})"
                    cursor.executemany(sql, data)  # Batch insert
                    conn.commit()  # Commit every batch
                    data = []  # Reset the data list after committing

            if data:  # Insert any remaining rows
                placeholders = ', '.join(['?' for _ in data[0]])
                sql = f"INSERT OR IGNORE INTO {table_name} VALUES ({placeholders})"
                cursor.executemany(sql, data)
                conn.commit()  # Final commit
    except Exception as e:
        print(f"Error importing data from {csv_file} into {table_name}: {e}")
        conn.rollback()  # Rollback changes in case of an error

# Import data from CSV files into the relevant table with `IGNORE` for duplicates
try:
    import_csv_to_table('Formulary_Restrictions_2024.csv', 'Restrictions',
ignore_duplicates=True)
```

```
    import_csv_to_table('Contract_Data.csv', 'Contracted_Payer', ignore_duplicates=True)
    import_csv_to_table('Med_ID_Data.csv', 'Med_ID', ignore_duplicates=True)
    import_csv_to_table('US_Insurance_Companies_With_Unique_IDs.csv',
'Plans_Controller', ignore_duplicates=True)
    import_csv_to_table('Plan_Sales_Distribution_2024.csv', 'Sales', ignore_duplicates=True)
    import_csv_to_table('States_and_Lives_2024.csv', 'States_and_Lives',
ignore_duplicates=True)
    import_csv_to_table('Payer_Plan_Bridge.csv', 'Payer_plan_relationship',
ignore_duplicates=True)
    print("Data imported successfully!")
except Exception as e:
    print(f"An error occurred during import: {e}")
    conn.rollback()  # Rollback changes if an error occurred

# Extract the Lives data into 'Lives' tables
try:

    # Inserting into Lives table
    cursor.execute('''
    INSERT INTO Lives (Formulary_ID, Month, Lives)
    SELECT Distinct Formulary_ID, Month, Lives
    FROM States_and_Lives;
    ''')

    # Commit the changes
    conn.commit()
    print("Data split into Lives tables successfully!")
except Exception as e:
    print(f"An error occurred during data split: {e}")
    conn.rollback()  # Rollback changes if an error occurred

# Finally, close the connection after all operations
conn.close()

import sqlite3

# Create a connection to the database
```

```
conn = sqlite3.connect('Takeda_sales.db')
cursor = conn.cursor()

# Function to fetch and display first 10 rows along with column headers from each table
def fetch_table_head(table_name):
    cursor.execute(f"PRAGMA table_info({table_name});")
    headers = [column[1] for column in cursor.fetchall()]  # Fetch column headers
    cursor.execute(f"SELECT * FROM {table_name} LIMIT 10;")
    rows = cursor.fetchall()

    print(f"\nTop 10 rows of {table_name} table:")
    print(headers)  # Print headers
    for row in rows:
        print(row)

# List of table names to fetch data from
tables = [
    "Restrictions",
    "Contracted_Payer",
    "Med_ID",
    "Plans_Controller",
    "States_and_Lives",
    "Sales",
    "Lives",
    "Payer_plan_relationship"
]

# Fetch and display top 10 rows and headers from each table
for table in tables:
    fetch_table_head(table)

# Closing the connection
conn.close()

import sqlite3
import pandas as pd
```

```
# Create a connection to the SQLite database (modify the path to your actual database file)
conn = sqlite3.connect('Takeda_sales.db')

# SQL Query to join tables and aggregate data
sql_query = """
With ranked_data as (
  Select
    *,
    ROW_NUMBER() OVER (
      PARTITION BY Month,
      Drug_Name,
      Plan_Name
      ORDER BY
        Total_Lives DESC
    ) AS row_num
  from
    (
      Select
        a.Month,
        b.Plan_ID,
        a.Plan_Name,
        a.Drug_Name,
        Sum(a.Lives) as Total_Lives,

        a.Restriction,
        b.Units_Sold,
        b.Price_Normalized,
        b.Sales_Revenue
      from
        (
          Select
            a.Month,
            b.Formulary_ID,
            b.Formulary_Name,
            b.Plan_ID,
            b.Insurance_Plan as Plan_Name,
            a.Drug_Name,
```

```
          b.Lives,

          a.Restriction
        from
          (
            select
              *
            from
              restrictions r
              left join (Select Drug_name,Med_ID from Med_ID) m on r.med_id = m.med_id
          ) a
          Inner Join (
            Select
              *
            from
              Lives l
              left Join plans_controller pc on pc.Formulary_ID = l.Formulary_ID

          ) b on a.formulary_id = b.formulary_id
          and a.month = b.month
      ) a
      Inner Join (
       Select a.*, b.Price_Normalized, Units_sold * Price_Normalized AS Sales_Revenue
from(
        Select
          *
        from
          Sales
          ) a
          Left join (Select Drug_Name, Price_Normalized from Med_ID) b on a.Drug_Name =
b.Drug_Name



      ) b on a.Plan_ID = b.Plan_ID
      and a.Month = b.Month
      and a.Drug_Name = b.Drug_Name
    group by
```

```
        a.Month,
        b.Plan_ID,
        a.Plan_Name,
        a.Drug_Name,

        a.Restriction,
        b.Units_Sold,
        b.Price_Normalized,
        b.Sales_Revenue
    )
),
Final_DataSet as (
  Select
    a.*,
    CASE WHEN c.Flag IS NULL THEN 0 ELSE c.Flag END AS Contracted_Payer_Flag
  FROM
    (
      SELECT
        a.Month,
        a.Plan_ID,
        a.Plan_Name,
        b.Payer_ID,
        b.Payer_Name,
        a.Drug_Name,
        a.Total_Lives,

        a.Restriction,
        a.Units_Sold,
        a.Price_Normalized,
        a.Sales_Revenue
      FROM
        (
          SELECT
            Month,
            Plan_ID,
            Plan_Name,
            Drug_Name,
```

```
            Total_Lives,
            Restriction,
            Units_Sold,
            Price_Normalized,
            Sales_Revenue
          FROM
            ranked_data
          WHERE
            row_num = 1
        ) a
        LEFT JOIN (
          SELECT
            Plan_ID,
            Payer_ID,
            Payer_Name
          FROM
            Payer_plan_relationship
        ) b ON a.Plan_ID = b.Plan_ID
    ) a
    Left Join(
      Select
        *
      from
        Contracted_Payer
    ) c on a.Payer_ID = c.Payer_ID
)
Select
  *
from
  Final_DataSet;
"""


# Execute the SQL query and load the result into a DataFrame
Final_data_df = pd.read_sql_query(sql_query, conn)

print(Final_data_df)
```

```
# Save the DataFrame to a CSV file
Final_data_df.to_csv('aggregated_data.csv', index=False)

# Print confirmation
print("CSV file has been created successfully!")

# Close the connection
conn.close()

# Insights Generation
import sqlite3
import pandas as pd

# Reconnect to the SQLite database
conn = sqlite3.connect('Takeda_sales.db')
cursor = conn.cursor()

# Load the saved data into a DataFrame
Final_data_df = pd.read_csv('/content/aggregated_data.csv')

# Save the DataFrame into a temporary table in SQLite
Final_data_df.to_sql('Final_DataSet', conn, if_exists='replace', index=False)

#Month over Month sale of each drug
query = """
   SELECT
      Month,
      Drug_Name,
      SUM(Units_Sold) AS Total_Units_Sold_Drug
   FROM
      Final_DataSet
   GROUP BY
      Drug_Name, Month
      Order by Month, Total_Units_Sold_Drug ;
"""
result_df = pd.read_sql_query(query, conn)
result_df.to_csv('MoM_Sales.csv', index=False)
```

```
print(result_df)

# Which restriction has most Lives for each and how much revenue they generate for each
drugs
query = """
SELECT Restriction,Drug_Name, AVG(Total_Lives) AS Avg_Lives_Covered,
SUM(CAST(REPLACE(Sales_Revenue, ',', '') AS INTEGER)) AS Total_Revenue
FROM Final_DataSet
GROUP BY  Restriction, Drug_Name
ORDER BY  Avg_Lives_Covered DESC;
"""

result_df = pd.read_sql_query(query, conn)
result_df.to_csv('restriction_vs_lives_revenue.csv', index=False)
print(result_df)



#Most Drug sold in 2024 by Units and their revenue generated
query = """
SELECT Drug_Name, SUM(Units_Sold) AS Total_Units,
SUM(CAST(REPLACE(Sales_Revenue, ',', '') AS INTEGER)) AS Total_Revenue
FROM Final_DataSet
GROUP BY Drug_Name
ORDER BY Total_Units DESC
;
"""
result_df = pd.read_sql_query(query, conn)
result_df.to_csv('top_drugs_units.csv', index=False)
print(result_df)

# Share for each drug within each payer Month over Month
query = """
WITH Total_units_by_Drug AS (
   SELECT
      Month,
      Payer_ID,
      Payer_Name,
```

```
        Drug_Name,
        Contracted_Payer_Flag,
        SUM(Units_Sold) AS Total_Units_Sold_Drug
    FROM
        Final_DataSet
    GROUP BY
        Payer_ID, Payer_Name, Drug_Name, Month, Contracted_Payer_Flag
),
Total_units AS (
    SELECT
        Month,
        Payer_ID,
        Payer_Name,
        SUM(Units_Sold) AS Total_Units_Sold_Payer,
        Contracted_Payer_Flag
    FROM
        Final_DataSet
    GROUP BY
        Payer_ID, Payer_Name, Contracted_Payer_Flag, Month
)
SELECT
    a.*,
    b.Total_Units_Sold_Payer,
    CAST(Total_Units_Sold_Drug AS FLOAT) / CAST(Total_Units_Sold_Payer AS FLOAT)
AS Drug_share
FROM
    Total_units_by_Drug a
    LEFT JOIN Total_units b
    ON a.Payer_ID = b.Payer_ID
    AND a.Month = b.Month
    Order by Month, Drug_share DESC
    ;
"""
result_df = pd.read_sql_query(query, conn)
result_df.to_csv('MoM_Share.csv', index=False)
print(result_df)
```

```
# Top payer with highest share for every brand in each month
query = """
WITH Total_units_by_Drug AS (
    SELECT
        Month,
        Payer_ID,
        Payer_Name,
        Drug_Name,
        Contracted_Payer_Flag,
        SUM(Units_Sold) AS Total_Units_Sold_Drug
    FROM Final_DataSet
    GROUP BY Payer_ID, Payer_Name, Drug_Name, Month, Contracted_Payer_Flag
),
Total_units AS (
    SELECT
        Month,
        Payer_ID,
        Payer_Name,
        SUM(Units_Sold) AS Total_Units_Sold_Payer,
        Contracted_Payer_Flag
    FROM Final_DataSet
    GROUP BY Payer_ID, Payer_Name, Contracted_Payer_Flag, Month
)
SELECT Month, Payer_ID, Payer_Name, Drug_Name, Contracted_Payer_Flag,
Total_Units_Sold_Drug,
Drug_share
 FROM (
    SELECT
        a.*,
        b.Total_Units_Sold_Payer,
        CAST(Total_Units_Sold_Drug AS FLOAT) / CAST(Total_Units_Sold_Payer AS
FLOAT) AS Drug_share,
        ROW_NUMBER() OVER (
            PARTITION BY a.Month, a.Drug_Name
            ORDER BY
                CAST(Total_Units_Sold_Drug AS FLOAT) / CAST(Total_Units_Sold_Payer AS
FLOAT) DESC
```

```
        ) AS row_num
    FROM Total_units_by_Drug a
    LEFT JOIN Total_units b
        ON a.Payer_ID = b.Payer_ID
        AND a.Month = b.Month
) result
WHERE row_num = 1
ORDER BY Month, Drug_share DESC;
"""
result_df = pd.read_sql_query(query, conn)
result_df.to_csv('Top_payer_each_Month.csv', index=False)
print(result_df)



# 5 Payer with Most Alunbrig Units Sold
query = """
SELECT Payer_ID,Payer_Name, SUM(Units_Sold) AS Total_Units,
Contracted_Payer_Flag
FROM Final_DataSet
Where Drug_Name = "Alunbrig"
GROUP BY Payer_ID,Payer_Name, Contracted_Payer_Flag
ORDER BY Total_Units DESC
LIMIT 5;
"""

result_df = pd.read_sql_query(query, conn)
result_df.to_csv('top_payer_units_Alunbrig.csv', index=False)
print(result_df)



# Top 5 Contracted vs Non-Contracted Payers by Total Units Sold of Alunbrig
query = """
WITH Payer_Sales AS (
    SELECT
        Payer_ID,
        Payer_Name,
        Contracted_Payer_Flag,
```

```
        SUM(Units_Sold) AS Total_Units
    FROM Final_DataSet
    Where Drug_Name = "Alunbrig"
    GROUP BY Payer_ID, Payer_Name, Contracted_Payer_Flag
)
SELECT * FROM (
    SELECT Payer_ID,Payer_Name, Contracted_Payer_Flag, Total_Units
    FROM Payer_Sales
    WHERE Contracted_Payer_Flag = 1
    ORDER BY Total_Units DESC
    LIMIT 5
)
UNION ALL
SELECT * FROM (
    SELECT Payer_ID,Payer_Name, Contracted_Payer_Flag, Total_Units
    FROM Payer_Sales
    WHERE Contracted_Payer_Flag = 0
    ORDER BY Total_Units DESC
    LIMIT 5
);
"""
result_df = pd.read_sql_query(query, conn)
result_df.to_csv('top_payers_Contracted_vs_NC.csv', index=False)
print(result_df)


# Close the connection when done
conn.close()
```