Flight Delays

September 24, 2025

1 Flight Delays ETL Pipeline Project (Lascaro Gianluca,Morbidelli Filippo,Trincia Elio)

Goal: This notebook implements a complete ETL (Extract, Transform, Load) pipeline for flight delay data. The objective is to process raw data from multiple CSV files, build a clean and consistent Star Schema, and load it into a MySQL database.

The project addresses real-world data engineering challenges, such as ensuring data quality, handling referential integrity, and loading large volumes of data efficiently.

[1]:
```
import pandas as pd
from sqlalchemy import create_engine
import sqlalchemy.exc

# ================================================================================
# --- CONFIGURATION PARAMETERS ---
# Please edit these settings to match your local environment.
# ================================================================================

# 1. Database Connection Settings
DB_USER = "root"
DB_PASSWORD = "Pirati2206#"  # put your password here
DB_HOST = "localhost"
DB_PORT = "3306"
DB_NAME = "flight_delays_db"

# 2. Input Data File Paths (relative to the script's location)
# Assumes a 'data' subfolder exists with the CSV files inside.
PATH_FLIGHTS = 'data/flights.csv'
PATH_AIRPORTS = 'data/airports.csv'
PATH_AIRLINES = 'data/airlines.csv'

# ================================================================================
# --- MAIN SCRIPT LOGIC ---
# ================================================================================

# --- 1. Setup: Load data from CSV files ---
print("--- Step 1: Loading Data ---")
```

```
try:
    flights_df = pd.read_csv(PATH_FLIGHTS, low_memory=False)
    airports_df = pd.read_csv(PATH_AIRPORTS)
    airlines_df = pd.read_csv(PATH_AIRLINES)
    print(" Data loaded successfully from CSV files.")
except FileNotFoundError as e:
    print(f" ERROR: A file was not found. Please check your paths and ensure␣
 ↪the 'data' folder exists.\nDetails: {e}")
```

```
--- Step 1: Loading Data ---
 Data loaded successfully from CSV files.
```

1.0.1 Step 2: Transform - Creating the Dimension Tables

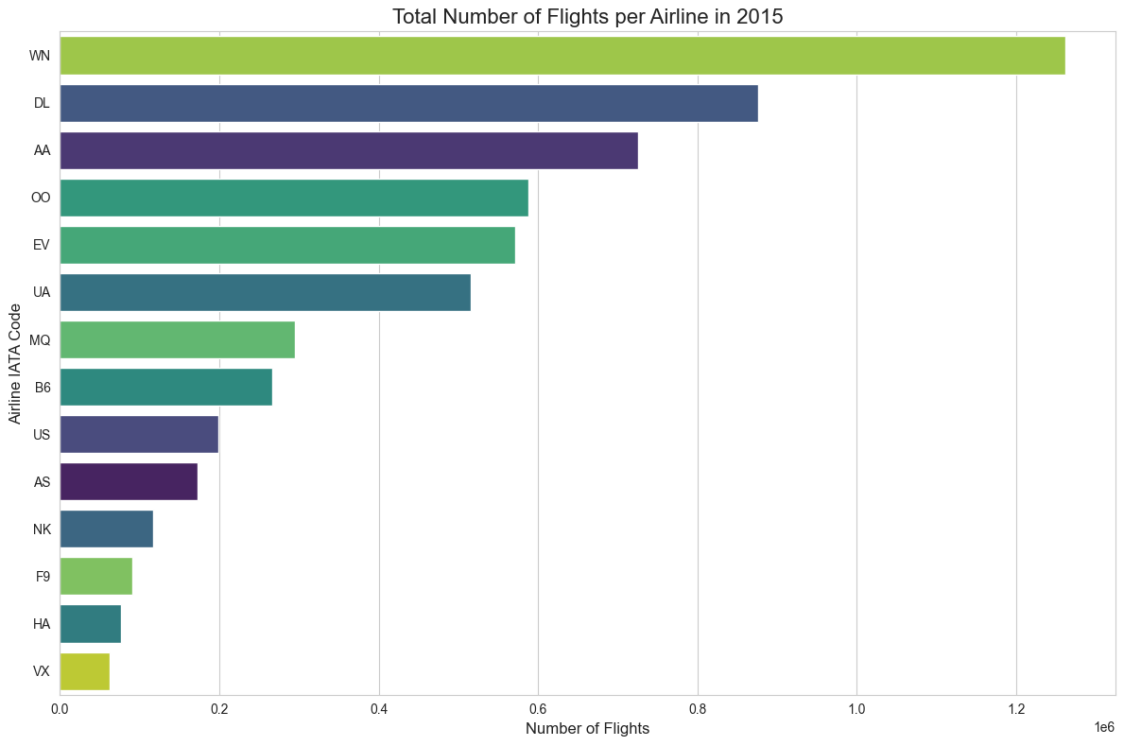
The first stage of the transformation process is to create our "lookup" tables, which are the dimensions of our Star Schema. These tables will hold the descriptive attributes for airlines, airports, and time. Each entity is assigned a unique, numeric ID (`_id`) to optimize database performance and ensure data stability.

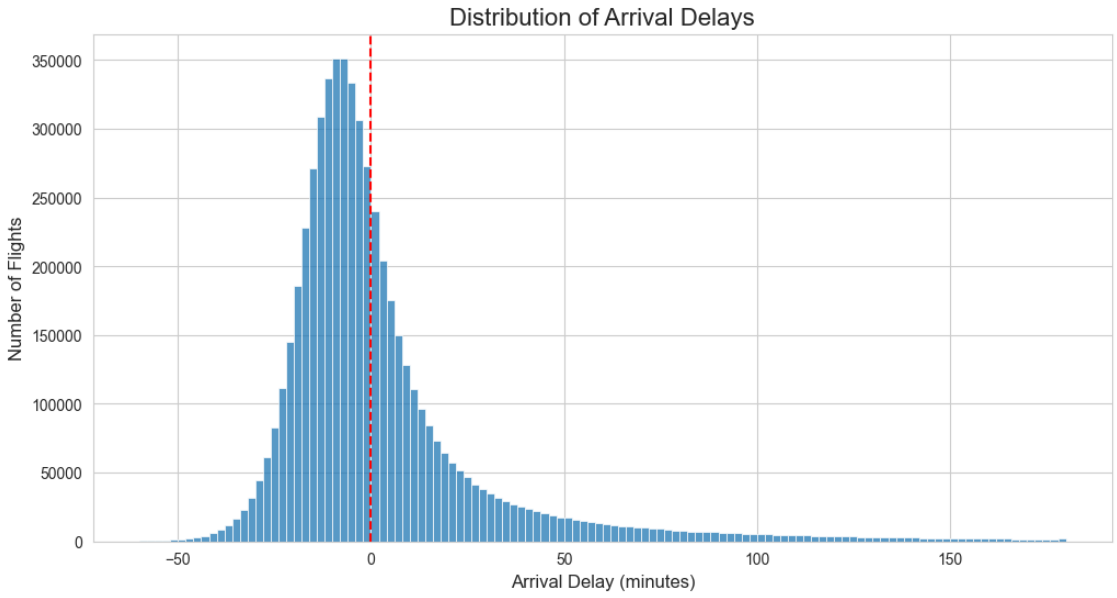
[2]:
```
# Initial Exploratory Data Analysis (EDA)
# This step helps to understand the raw data before transformation.

import matplotlib.pyplot as plt
import seaborn as sns

print("\n--- Performing Initial Exploratory Data Analysis ---")
sns.set_style("whitegrid")

# Plot 1: Show the number of flights per airline
plt.figure(figsize=(12, 8))
sns.countplot(
    y="AIRLINE",
    data=flights_df,
    order=flights_df['AIRLINE'].value_counts().index,
    hue="AIRLINE",
    legend=False,
    palette="viridis"
)
plt.title('Total Number of Flights per Airline in 2015', fontsize=16)
plt.xlabel('Number of Flights', fontsize=12)
plt.ylabel('Airline IATA Code', fontsize=12)
plt.tight_layout()
plt.show()

# Plot 2: Show the distribution of arrival delays
plt.figure(figsize=(12, 6))
```

```
# We focus on a reasonable range of delays (-60 to 180 mins) to make the plot␣
 ↪readable
sns.histplot(flights_df['ARRIVAL_DELAY'].dropna(), bins=120, kde=False,␣
 ↪binrange=(-60, 180))
plt.title('Distribution of Arrival Delays', fontsize=16)
plt.xlabel('Arrival Delay (minutes)', fontsize=12)
plt.ylabel('Number of Flights', fontsize=12)
plt.axvline(0, color='red', linestyle='--') # Add a line at 0 for reference
plt.show()
```

```
C:\Users\gianluca\AppData\Local\Programs\Python\Python313\Lib\site-
packages\seaborn\_statistics.py:32: UserWarning: A NumPy version >=1.23.5 and
<2.3.0 is required for this version of SciPy (detected version 2.3.3)
  from scipy.stats import gaussian_kde
```

```
--- Performing Initial Exploratory Data Analysis ---
```





1.0.2 Step 3: Transform - Preparing the Fact Table

This is the most critical phase of the pipeline. In this step, we clean the main flights dataset and connect it to the dimensions we just created. A key focus here is on data quality: we will identify and remove any inconsistent records to ensure the final database has perfect referential integrity.

[3]:
```
# --- 2. Data Transformation: Create Dimension Tables ---
print("\n--- Step 2: Creating Dimension Tables ---")

# Dimension 1: Airlines
dim_airlines = airlines_df.copy()
dim_airlines['airline_id'] = dim_airlines.index + 1
print("  - Airlines dimension created.")

# Dimension 2: Airports
dim_airports = airports_df.copy()
dim_airports['airport_id'] = dim_airports.index + 1
print("  - Airports dimension created.")

# Dimension 3: Time
flights_df['date'] = pd.to_datetime(flights_df[['YEAR', 'MONTH', 'DAY']])
unique_dates = flights_df['date'].unique()
dim_time = pd.DataFrame({'date': unique_dates})
dim_time['year'] = dim_time['date'].dt.year
dim_time['month'] = dim_time['date'].dt.month
dim_time['day'] = dim_time['date'].dt.day
dim_time['day_of_week'] = dim_time['date'].dt.dayofweek
```

```
dim_time['is_weekend'] = dim_time['day_of_week'].isin([5, 6])
dim_time = dim_time.sort_values(by='date').reset_index(drop=True)
dim_time['time_id'] = dim_time.index + 1
print("  - Time dimension created.")
print("  All dimension tables are ready.")
```

```
--- Step 2: Creating Dimension Tables ---
  - Airlines dimension created.
  - Airports dimension created.
  - Time dimension created.
 All dimension tables are ready.
```

[4]:
```
# --- 3. Data Transformation: Create the Fact Table ---
print("\n--- Step 3: Preparing the Fact Table ---")

# Select only the columns we need
columns_to_keep = [
    'date', 'AIRLINE', 'ORIGIN_AIRPORT', 'DESTINATION_AIRPORT',
    'DEPARTURE_DELAY', 'ARRIVAL_DELAY', 'AIR_TIME', 'DISTANCE', 'CANCELLED',␣
 ↪'DIVERTED'
]
fact_flights = flights_df[columns_to_keep].copy()

# Clean data by removing rows with missing essential values
fact_flights.dropna(subset=['ARRIVAL_DELAY', 'DEPARTURE_DELAY', 'AIR_TIME'],␣
 ↪inplace=True)

# Merge with dimension tables to get the numeric foreign keys
fact_flights = pd.merge(fact_flights, dim_time[['date', 'time_id']], on='date',␣
 ↪how='left')
fact_flights = pd.merge(fact_flights, dim_airlines[['IATA_CODE',␣
 ↪'airline_id']], left_on='AIRLINE', right_on='IATA_CODE', how='left')
fact_flights = pd.merge(fact_flights, dim_airports[['IATA_CODE',␣
 ↪'airport_id']], left_on='ORIGIN_AIRPORT', right_on='IATA_CODE', how='left')
fact_flights.rename(columns={'airport_id': 'origin_airport_id'}, inplace=True)
fact_flights = pd.merge(fact_flights, dim_airports[['IATA_CODE',␣
 ↪'airport_id']], left_on='DESTINATION_AIRPORT', right_on='IATA_CODE',␣
 ↪how='left')
fact_flights.rename(columns={'airport_id': 'destination_airport_id'},␣
 ↪inplace=True)

# Select and order the final columns for the fact table
final_columns = [
    'time_id', 'airline_id', 'origin_airport_id', 'destination_airport_id',
    'ARRIVAL_DELAY', 'DEPARTURE_DELAY', 'AIR_TIME', 'DISTANCE', 'CANCELLED',␣
 ↪'DIVERTED'
```

```
]
fact_flights = fact_flights[final_columns]

#  Remove rows where a merge failed (resulting in a null ID) to ensure foreign␣
 ↪key integrity
initial_rows = len(fact_flights)
fact_flights.dropna(inplace=True)
print(f"  - Cleaned data: Removed {initial_rows - len(fact_flights)} rows with␣
 ↪inconsistent foreign keys.")

#  Ensure data types in Pandas match the SQL table schema
fact_flights['time_id'] = fact_flights['time_id'].astype(int)
fact_flights['airline_id'] = fact_flights['airline_id'].astype(int)
fact_flights['origin_airport_id'] = fact_flights['origin_airport_id'].
 ↪astype(int)
fact_flights['destination_airport_id'] = fact_flights['destination_airport_id'].
 ↪astype(int)
fact_flights['ARRIVAL_DELAY'] = fact_flights['ARRIVAL_DELAY'].astype(int)
fact_flights['DEPARTURE_DELAY'] = fact_flights['DEPARTURE_DELAY'].astype(int)
print("  - Data types corrected to match SQL schema.")

print(" Fact table is now clean and ready.")
```

```
--- Step 3: Preparing the Fact Table ---
  - Cleaned data: Removed 482878 rows with inconsistent foreign keys.
  - Data types corrected to match SQL schema.
 Fact table is now clean and ready.
```

1.0.3 Step 4: Load - Populating the Database

With the clean and transformed DataFrames ready, we now load them into the pre-defined SQL tables. For the large `f_flights` fact table, we use a technique called "chunking" to break the data into smaller batches. This ensures a robust and efficient loading process that avoids common issues like database timeouts.

[5]:
```
# --- 4. Load Data into Database ---
print("\n--- Step 4: Connecting to and Loading the Database ---")

# Build the database connection string from the configuration parameters
db_connection_str = f'mysql+mysqlconnector://{DB_USER}:{DB_PASSWORD}@{DB_HOST}:
 ↪{DB_PORT}/{DB_NAME}'

try:
    engine = create_engine(db_connection_str)
    print("  - Connection to database successful.")

    # Load data into SQL tables
```

```
    print("  - Loading dimension tables...")
    dim_airlines.to_sql('d_airlines', con=engine, if_exists='append',␣
 ↪index=False)
    dim_airports.to_sql('d_airports', con=engine, if_exists='append',␣
 ↪index=False)
    dim_time.to_sql('d_time', con=engine, if_exists='append', index=False)

    print("  - Loading fact table (this may take a few minutes)...")
    # Use chunksize for large data to prevent timeouts
    fact_flights.to_sql(
        'f_flights',
        con=engine,
        if_exists='append',
        index=False,
        chunksize=50000
    )
    print("  All data has been successfully loaded into the database!")

except Exception as e:
    print(f"  An error occurred during database operations. Details: {e}")
```

```
--- Step 4: Connecting to and Loading the Database ---
  - Connection to database successful.
  - Loading dimension tables…
  - Loading fact table (this may take a few minutes)…
  All data has been successfully loaded into the database!
```

1.0.4 Step 5: Verification and Value Demonstration

To confirm that the entire pipeline was successful and to demonstrate the value of the newly created data model, we will run several analytical queries directly against the populated database. Each query is designed to answer one of the key business questions defined for this project, showcasing the power and simplicity of the Star Schema for analysis.

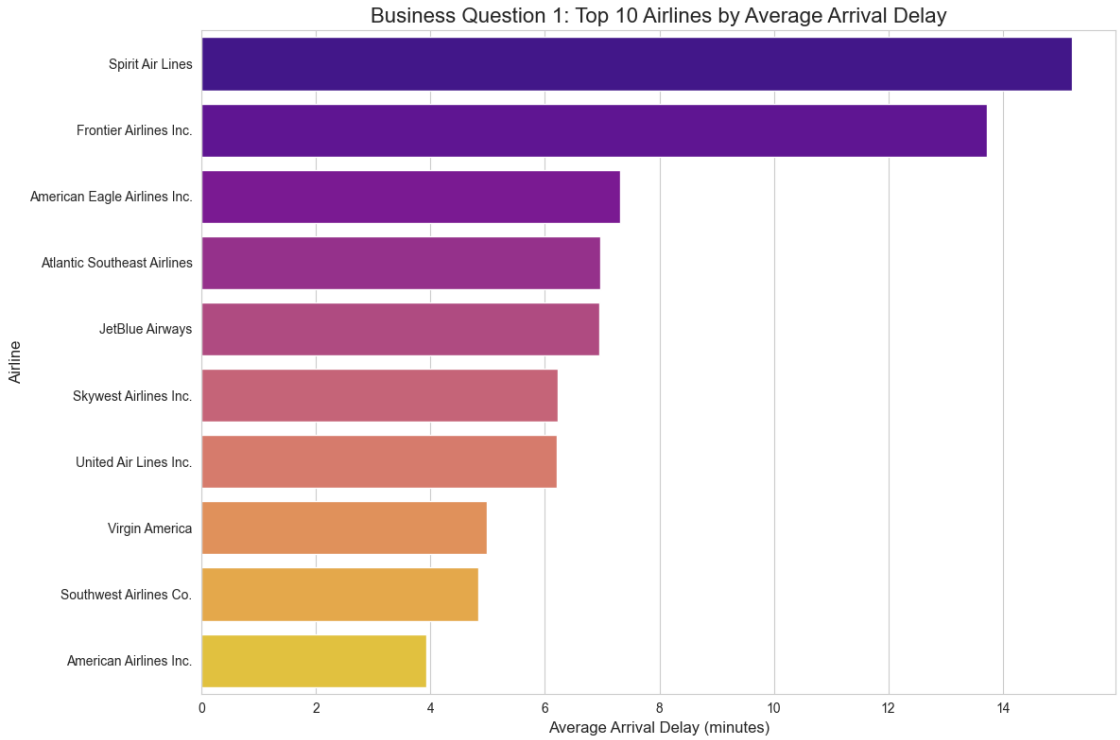
[6]:
```
# --- 5. Verification: Demonstrating the Power of the Data Model ---
import matplotlib.pyplot as plt
import seaborn as sns

print("\n--- Step 5: Running Verification Queries on the Database ---")
sns.set_style("whitegrid")

# ================================================================================
# Query 1: Which airlines have the highest average arrival delays?
# This query answers the "Airline Performance" business question.
# ================================================================================
query_1 = """
SELECT
```

```
    da.AIRLINE AS airline_name,
    AVG(ff.ARRIVAL_DELAY) AS average_delay
FROM f_flights ff
JOIN d_airlines da ON ff.airline_id = da.airline_id
GROUP BY da.AIRLINE
ORDER BY average_delay DESC
LIMIT 10;
"""

try:
    print("\nExecuting Query 1: Top 10 Airlines by Average Delay...")
    df1 = pd.read_sql(query_1, engine)

    plt.figure(figsize=(12, 8))

    sns.barplot(x='average_delay', y='airline_name', data=df1,␣
 ↪palette="plasma", hue='airline_name', legend=False)

    plt.title('Business Question 1: Top 10 Airlines by Average Arrival Delay',␣
 ↪fontsize=16)
    plt.xlabel('Average Arrival Delay (minutes)', fontsize=12)
    plt.ylabel('Airline', fontsize=12)
    plt.tight_layout()
    plt.show()

except Exception as e:
    print(f"  Query 1 failed. Details: {e}")

# ============================================================================
# Query 2: Which major airports have the worst departure delays?
# This query answers the "Airport Hotspots" business question.
# ============================================================================
query_2 = """
SELECT
    da.AIRPORT AS airport_name,
    COUNT(ff.flight_id) AS total_flights,
    AVG(ff.DEPARTURE_DELAY) AS average_departure_delay
FROM f_flights ff
JOIN d_airports da ON ff.origin_airport_id = da.airport_id
GROUP BY da.AIRPORT
HAVING COUNT(ff.flight_id) > 50000 -- Filter for major airports only
ORDER BY average_departure_delay DESC
LIMIT 10;
"""

try:
```

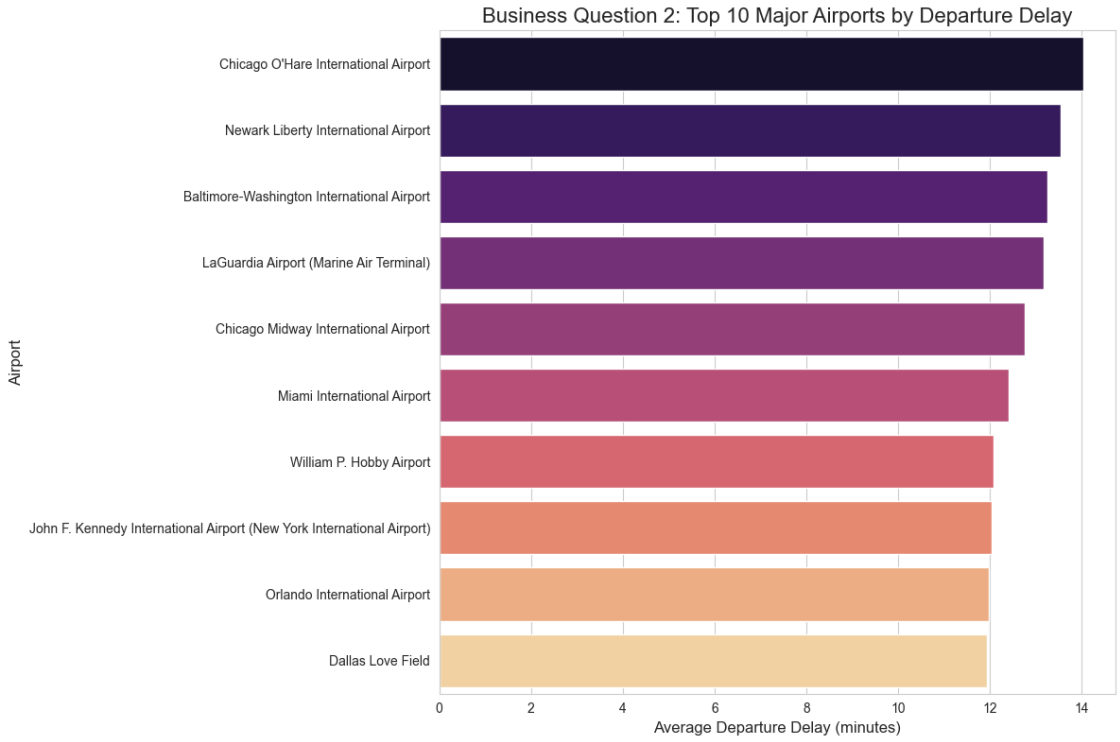
```
    print("\nExecuting Query 2: Top 10 Major Airports by Average Departure␣
 ↪Delay...")
    df2 = pd.read_sql(query_2, engine)

    plt.figure(figsize=(12, 8))
    sns.barplot(x='average_departure_delay', y='airport_name', data=df2,␣
 ↪palette="magma", hue='airport_name', legend=False)

    plt.title('Business Question 2: Top 10 Major Airports by Departure Delay',␣
 ↪fontsize=16)
    plt.xlabel('Average Departure Delay (minutes)', fontsize=12)
    plt.ylabel('Airport', fontsize=12)
    plt.tight_layout()
    plt.show()

except Exception as e:
    print(f"  Query 2 failed. Details: {e}")

# ============================================================================
# Query 3: How do delays vary between weekdays and weekends?
# This query answers the "Temporal Patterns" business question.
# ============================================================================
query_3 = """
SELECT
    dt.is_weekend,
    AVG(ff.ARRIVAL_DELAY) AS average_arrival_delay
FROM f_flights ff
JOIN d_time dt ON ff.time_id = dt.time_id
GROUP BY dt.is_weekend
ORDER BY dt.is_weekend;
"""

try:
    print("\nExecuting Query 3: Average Delay on Weekdays vs. Weekends...")
    df3 = pd.read_sql(query_3, engine)

    # Map the boolean 'is_weekend' to readable string labels for the plot
    df3['day_type'] = df3['is_weekend'].apply(lambda x: 'Weekend' if x else␣
 ↪'Weekday')

    plt.figure(figsize=(8, 6))
    sns.barplot(x='day_type', y='average_arrival_delay', data=df3,␣
 ↪palette="cividis", hue='day_type', legend=False)

    plt.title('Business Question 3: Average Arrival Delay - Weekday vs.␣
 ↪Weekend', fontsize=16)
```

```
    plt.xlabel('Day Type', fontsize=12)
    plt.ylabel('Average Arrival Delay (minutes)', fontsize=12)
    plt.tight_layout()
    plt.show()

except Exception as e:
    print(f"  Query 3 failed. Details: {e}")
```

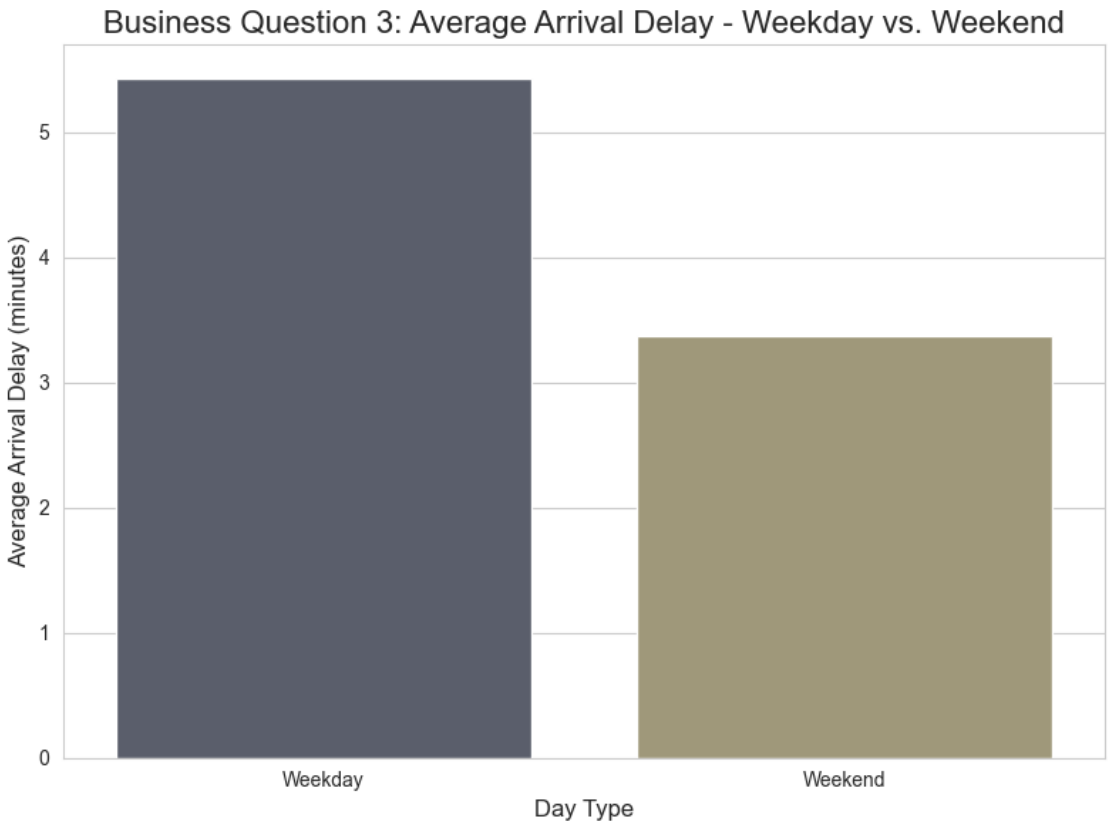
```
--- Step 5: Running Verification Queries on the Database ---

Executing Query 1: Top 10 Airlines by Average Delay…
```




```
Executing Query 2: Top 10 Major Airports by Average Departure Delay…
```




```
Executing Query 3: Average Delay on Weekdays vs. Weekends…
```


Business Question 3: Average Arrival Delay - Weekday vs. Weekend
Average Arrival Delay (minutes)
0
1
2
3
4
5
Weekday
Weekend
Day Type

[ ]: