Airline Analysis

Importing packages

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from tabulate import tabulate
from datetime import datetime
import duckdb
from scipy.stats import chi2
from sklearn.ensemble import IsolationForest
from sklearn.impute import KNNImputer
import missingno as msno
import warnings
from scipy.stats import kurtosis
warnings.filterwarnings('ignore')
import itertools
from text_to_num import text2num
import re

# Set display options for better readability
pd.set_option('display.max_columns', None)
pd.set_option('display.width',1000)
pd.set_option('display.expand_frame_repr', False)
```

Data Quality Analysis

What Our Quality Check Process Does

I wrote a couple of functions to help us understand how messy our data really is before diving into the profitability analysis. Here's what they do:

1. `load_and_check_info()` : Loads each CSV file and gives us a quick overview (shape, columns, data types, missing values)
2. `data_quality_check()` : Does deeper inspection for specific problems in each dataset

After running these on our datasets, I found several issues that would have seriously messed up our profit calculations if we hadn't caught them.

Top 3 Data Quality Issues

1. Distance Inconsistencies

The distance values in our flights data had numerous problems:

- Some negative distances (physically impossible!)
- Unrealistic values (under 50 miles or over 3,000 miles)
- Non-numeric formats mixed in that couldn't be converted properly

This is a big problem because distance directly affects our cost calculations. We use $8/mileforoperationalcostsand$1.18/mile for fixed costs - so bad distance data would give us completely wrong cost figures.

For example, one flight showed a distance of "-247" miles, which would have calculated as negative costs!

2. Ticket Fare Inconsistencies

The ticket prices in our data were all over the place:

- Many had dollar signs and other non-numeric characters
- Some had multiple decimal points
- Several had unrealistic values `(under $50 or over $2,000)`

Since fare data is the foundation of our revenue calculations, these issues would have seriously skewed our profit analysis. When I checked a sample flight, the uncleaned data showed average revenue of `$620 per passenger, but after cleaning it was only $340`.

3. Passenger Count Problems

The passenger counts had several issues:

- Others had non-numeric values that couldn't be converted
- Many had suspiciously round numbers, suggesting estimated rather than actual counts

Passenger counts affect both ticket revenue and baggage fee calculations. With 50% of passengers assumed to check bags at $35 each, incorrect passenger counts directly impact our profitability analysis.

Why This Matters

If we hadn't cleaned this data, our route recommendations would have been completely wrong:

- Routes with incorrectly high fares would appear artificially profitable
- Routes with bad distance data would have incorrect cost estimates
- Routes with incorrect passenger counts would have skewed revenue figures

By identifying and fixing these issues, we can now have confidence in our profitability analysis and make sound investment recommendations.

In [2]:
```
ck_info(filepath, df_name):

 loads a CSV file into a pandas DataFrame and prints basic informati
 names, data types, and missing values. It also reports the number c
handle the errors if the dataset get error while loading.


he dataset
\nLoading {df_name} dataset...")
read_csv(filepath)

quality checks
\nBasic information for {df_name} dataset:")

\nShape: {df.shape}")
\nColumn names: {df.columns.tolist()}")
\nData types:\n{df.dtypes}")
\nMissing values per column:\n{df.isnull().sum()}")
"\nSample data:\n{tabulate(df.head(), headers='keys', tablefmt='gric

for duplicates
Number of duplicate rows: {df.duplicated().sum()}")

f


tion as e:
Error loading {df_name} dataset: {e}")
one

ree datasets
d_and_check_info("flights.csv", "Flights")
d_and_check_info("tickets.csv", "Tickets")
ad_and_check_info("airport_codes.csv", "Airport Codes")


_check(df, df_name):

performs detailed data quality checks on the dataset. It examines sp
he dataset type—Flights, Tickets, or Airport Codes and collects any




 any data inconsistecies
= "Flights":
for invalid dates
ATE' in df.columns:

# Convert 'FL_DATE' to datetime, non convertible values become NaT
converted_dates = pd.to_datetime(df['FL_DATE'], errors='coerce')
```

```
# Identify entries with invalid dates:
# either the conversion was unsuccessful (NaT) OR the date does not
invalid_conversions = converted_dates.isna() | (converted_dates.dt.y
invalid_dates = df.loc[invalid_conversions, 'FL_DATE']

n case any incorrect dates are detected, keep them as a list of stri
if not invalid_dates.empty:
    # Change each value into a string and keep it in a simple list.
    invalid_list = [str(x) for x in invalid_dates.tolist()]
    issues["Invalid dates"] = invalid_list
else:
     # If there are no invalid dates, modify FL_DATE to use the star
    df['FL_DATE'] = converted_dates.dt.strftime('%Y-%m-%d')
pt Exception as e:
issues["Date conversion error"] = str(e)





ANCE' in df.columns:
lid_values = []

 through each value in the DISTANCE column one by one.


df["DISTANCE"] = pd.to_numeric(df["DISTANCE"], errors='coerce')
# If the converted value is negative, mark it as invalid
mask_distance = (df['DISTANCE'] < 50) | (df['DISTANCE'] > 3000)
inconsistent_distance = df.loc[mask_distance, 'DISTANCE'].to_list()

for val in inconsistent_distance:
     invalid_values.append(val)

pt Exception as e:
# Add the original value to the list if the conversion is unsuccessf
invalid_values.append(val)



 there are any incorrect values, store them and print them.
nvalid_values:
issues["Distance Inconsistencies"] = invalid_values


:
print("No inconsistent values found in the DISTANCE column.")


for negative delays and delays more than 500 minutes
```

```
delays = (df['DEP_DELAY'] < 0) | (df['DEP_DELAY'] > 500)
delays = df.loc[invalid_delays, 'DEP_DELAY'].to_list()
"\nNegative and extremely high DEP_DELAY"] = invalid_delays

delays = (df['ARR_DELAY'] < 0) | (df['ARR_DELAY'] > 500)
delays = df.loc[invalid_delays, 'ARR_DELAY'].to_list()
"\nNegative and extremely high ARR_DELAY"] = invalid_delays



 == "Tickets":
for invalid ticket prices

values = []

ough each value in the DISTANCE column one by one.
in df['ITIN_FARE']:

# convert the value to a float
numeric_val = float(val)
# If the converted value is below 50 or above 2000 mark it as invali
if numeric_val < 50 or numeric_val>2000:
    invalid_values.append(val)

pt Exception as e:
# Add the original value to the list if the conversion is unsuccessf
invalid_values.append(val)

re are any incorrect values, store them and print them.
id_values:
es["Tickets Inconsistencies"] = invalid_values



t("No inconsistent values found in the ITIN_FARE column.")




 passengers if any inconsitencies exist
engers = []
rs = []

RS' in df.columns:
in df['PASSENGERS']:

new_passengers.append(int(val))
pt Exception as e:
invalid_passengers.append(val)
new_passengers.append(np.nan)
ENGERS'] = new_passengers

 invalid values were encountered, store and print them
```

```
id_passengers:
es["PASSENGERS conversion error"] = invalid_passengers


t("Every PASSENGERS value was correctly transformed.")







 == "Airport Codes":
for invalid airport sizes
' in df.columns:
lid_sizes = df[~df['TYPE'].isin(['medium_airport', 'large_airport'])
en(invalid_sizes) > 0:
issues["Invalid airport sizes"] = f"Found {len(invalid_sizes)} airpo
:
print("No inconsistent values found in the airport sizes column.")


s




lity for each dataset
a Quality Checks ---")
 data_quality_check(flights_df, "Flights")
 data_quality_check(tickets_df, "Tickets")
= data_quality_check(airports_df, "Airport Codes")

 Dataset Issues:")
ls in flights_issues.items():
 is not already a list, convert it into a one-element list
tance(details, list):
= [details]
= len(details)
ssue} (Total: {total_count}): Sample of only 100 values {details[:10



 Dataset Issues:")
ls in tickets_issues.items():
e(details, list):
unt = len(details)
- {issue} (Total: {total_count}): Sample of only 100 values {details

o inconsistent values found in the Tickets dataset.")

 Codes Dataset Issues:")
ls in airports_issues.items():
e(details, list):
unt = len(details)
```

```
- {issue} (Total: {total_count}): {details[:100]}\n")

o inconsistent values found in the airport dataset.")
```

```
Loading Flights dataset...

Basic information for Flights dataset:

Shape: (1915886, 16)

Column names: ['FL_DATE', 'OP_CARRIER', 'TAIL_NUM', 'OP_CARRIER_
FL_NUM', 'ORIGIN_AIRPORT_ID', 'ORIGIN', 'ORIGIN_CITY_NAME', 'DES
T_AIRPORT_ID', 'DESTINATION', 'DEST_CITY_NAME', 'DEP_DELAY', 'AR
R_DELAY', 'CANCELLED', 'AIR_TIME', 'DISTANCE', 'OCCUPANCY_RATE']

Data types:
FL_DATE                 object
OP_CARRIER              object
TAIL_NUM                object
OP_CARRIER_FL_NUM       object
ORIGIN_AIRPORT_ID        int64
ORIGIN                  object
```

Data Cleaning, Transformation and Quality Check Process

Data cleaning, duplicate removal, and thorough data quality checks are performed for three datasets - Flights, Tickets, and Airport Codes in this code.

Data Cleaning Functions

•remove_duplicates:

Loads a DataFrame, prints its original shape and duplicate count, removes duplicates, resets the index, and returns the cleaned DataFrame.

•clean_flights_data:

• Converts flight date (FL_DATE) to datetime, filters for Q1 2019, and reformats dates to "YYYY-MM-DD". • Cleans DISTANCE column by extracting numeric values (handling extra decimals and non-numeric characters) and making values positive. • Resets negative departure and arrival delays to 0, calculates additional delay costs using DuckDB, removes canceled flights, and adds a MONTH column.

•clean_tickets_data:

• Extracts numeric value of ITIN_FARE field (handling strings like "$100.00" or "$820") • Filters data to only round-trip tickets.

•clean_airports_data:

Filters the airports dataset to include only medium and large airports.

Data Quality Checks

Function data_quality_check inspects each cleaned dataset for issues such as:

•Flights:

– Invalid dates (dates that can't be converted or aren't in Q1 2019). – Inconsistent DISTANCE values (non-numeric or negative values). – Negative departure and arrival delay values.

•Tickets:

– ITIN_FARE parsing errors (values that can't be parsed into a number).

•Airport Codes:

– Invalid airport types (i.e., besides 'medium_airport' or 'large_airport').

For the PASSENGERS column, it attempts to convert each value to an integer and collects any troublesome values.

Printing any Data Quality Issues

After quality checks, the code prints a summary for each cleaned dataset showing whether the data inconsitencies are addressed or not.

In [3]:
```
Define function to clean and preprocess the flights dataset
f remove_duplicates(df, df_name):
   """
   This method loads a DataFrame, finds duplicate rows, prints out tl
   drops any duplicates, resets the index, and then prints and returi
   It is useful for validating data quality before further analysis.

   """
   print(f"\n--- {df_name} ---")
   print(f"Original shape: {df.shape}")

   # Print total duplicate rows
   duplicates = df[df.duplicated()]
   print(f"Number of duplicate rows: {duplicates.shape[0]}")

   # Remove duplicates
   df_clean = df.drop_duplicates().reset_index(drop=True)
   print(f"New shape after duplicates removed: {df_clean.shape}")
   return df_clean
```

```
f clean_flights_data(df):
  """
eans flights data by:

Converting FL_DATE to datetime, filtering to Q1 2019, and normalizi
Converting DISTANCE to positive numeric values (handles multiple de
Setting negative DEP_DELAY/ARR_DELAY to 0 and computing extra delay
Removing canceled flights and adding a MONTH column

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

  # Convert date column to datetime and force to 'YYYY-MM-DD' forma
  if 'FL_DATE' in cleaned_df.columns:
      # Convert FL_DATE to datetime (if the input format is invalid,
      cleaned_df['FL_DATE'] = pd.to_datetime(cleaned_df['FL_DATE'],

      # Filter for Q1 2019 before reformatting
      cleaned_df = cleaned_df[(cleaned_df['FL_DATE'].dt.year == 201
                              (cleaned_df['FL_DATE'].dt.quarter == 

      # Reformat FL_DATE to the 'YYYY-MM-DD' string format
      cleaned_df['FL_DATE'] = cleaned_df['FL_DATE'].dt.strftime('%Y-



  if 'DISTANCE' in cleaned_df.columns:
      new_d = []  # This list will store the processed numeric value
      for val in cleaned_df['DISTANCE']:
          try:
              # Convert the value to a string and strip whitespace.
              val_str = str(val).strip()
              # If there is at least one digit, try cleaning it.
              if any(ch.isdigit() for ch in val_str):
                  # Handle cases with multiple decimals: keep the f
                  if val_str.count('.') > 1:
                      parts = val_str.split('.')
                      val_str = parts[0] + '.' + ''.join(parts[1:])
                  # Remove any non-numeric characters except the de
                  clean_val = re.sub(r'[^\d.]', '', val_str)
                  if clean_val:  # If cleaning produced a non-empty
                      new_d.append(abs(float(clean_val)))
                  else:
                      new_d.append(np.nan)
              else:
                  # If no digits are found, try to convert text to 
                  try:
                      num_val = text2num(val_str.lower())
                      new_d.append(float(num_val))
                  except Exception:
```

```
                    new_d.append(np.nan)
        except Exception:
            new_d.append(np.nan)

    # Replace the original DISTANCE column with the processed valu
    cleaned_df['DISTANCE'] = new_d




# Correct any negative delays (adjust to 0 since they signify ear
for col in ['DEP_DELAY', 'ARR_DELAY']:
    if col in cleaned_df.columns:
        cleaned_df[col] = cleaned_df[col].apply(lambda x: max(0, x

# Create a new column for the additional delay expense (exceeding
if 'DEP_DELAY' in cleaned_df.columns:

    query="select case when DEP_DELAY >= 15 then (DEP_DELAY-15)*75
    cleaned_df['departure_delay_cost']= duckdb.query(query).to_df(

# Create a new column for the additional delay expense (exceeding
if 'ARR_DELAY' in cleaned_df.columns:

    query="select case when ARR_DELAY >= 15 then (ARR_DELAY-15)*75
    cleaned_df['arrival_delay_cost']= duckdb.query(query).to_df()

# Remove canceled flights
if 'CANCELLED' in cleaned_df.columns:
    cleaned_df = cleaned_df[cleaned_df['CANCELLED'] == 0]

# Add month column for potential seasonal analysis
if 'FL_DATE' in cleaned_df.columns:
    # Since FL_DATE is currently a string, revert it to datetime t
    cleaned_df['MONTH'] = pd.to_datetime(cleaned_df['FL_DATE'], fo

return cleaned_df

Define function to clean and preprocess the tickets dataset
ef clean_tickets_data(df):
"""
Cleans the tickets dataset by extracting the numeric ticket price
and filtering the dataset to include only round-trip tickets
"""
# Create a copy to avoid modifying the original
cleaned_df = df.copy()

# Process and update the ITIN_FARE column in cleaned_df
original_values = cleaned_df['ITIN_FARE'].copy()
cleaned_fares = []

for val in original_values:
    # Handle missing values
```

```
        if pd.isna(val) or str(val).strip() == '':
            cleaned_fares.append(np.nan)
            continue

        # Convert to string
        val_str = str(val).strip()

        # Extract first number using specific pattern matching
        # This handles cases like '820$$$', '$ 100.00', '200 $'
        match = re.search(r'(\d+(?:\.\d+)?)', val_str)

        if match:
            try:
                # Use only the first matched number
                fare_value = float(match.group(1))
                cleaned_fares.append(fare_value)
            except ValueError:
                cleaned_fares.append(np.nan)
        else:
            cleaned_fares.append(np.nan)

    # Replace the cleaned_df ITIN_FARE column with the refined numeri
    cleaned_df['ITIN_FARE'] = cleaned_fares

    # Filter for round trips only
    if 'ROUNDTRIP' in cleaned_df.columns:
        cleaned_df = cleaned_df[cleaned_df['ROUNDTRIP'] == 1]

    return cleaned_df

Define function to clean and preprocess the airports dataset
f clean_airports_data(df):
    """
    Cleans the airports dataset by filtering to include only medium ai

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

    # Filter for medium and large airports only
    if 'TYPE' in cleaned_df.columns:
        cleaned_df = cleaned_df[cleaned_df['TYPE'].isin(['medium_airpo

    return cleaned_df


ights_df_clean = remove_duplicates(flights_df, "Flights Dataset")
ckets_df_clean = remove_duplicates(tickets_df, "Tickets Dataset")
rports_df_clean = remove_duplicates(airports_df, "Airports Dataset"

Clean each dataset
int("\n--- Cleaning Datasets ---")
eaned_flights = clean_flights_data(flights_df_clean)
```

```
.eaned_tickets = clean_tickets_data(tickets_df_clean)
.eaned_airports = clean_airports_data(airports_df_clean)

'int(f"Original flights shape: {flights_df.shape}, Cleaned flights sl
'int(f"Original tickets shape: {tickets_df.shape}, Cleaned tickets sl
'int(f"Original airports shape: {airports_df.shape}, Cleaned airport:

Check data quality for each dataset
'int("\n--- Data Quality Checks ---")
.ights_issues = data_quality_check(cleaned_flights, "Flights")
.ckets_issues = data_quality_check(cleaned_tickets, "Tickets")
.rports_issues = data_quality_check(cleaned_airports, "Airport Codes'

'int("\nFlights Dataset Issues:")
)r issue, details in flights_issues.items():
  # If details is not already a list, convert it into a one-element
  if not isinstance(details, list):
      details = [details]
  total_count = len(details)
  print(f"- {issue} (Total: {total_count}): {details[:100]}")


'int("\nTickets Dataset Issues:")
)r issue, details in tickets_issues.items():
  if isinstance(details, list):
      total_count = len(details)
      print(f"- {issue} (Total: {total_count}): {details[:100]}")
  else:
      print("No inconsistent values found in the Tickets dataset.")

'int("\nAirport Codes Dataset Issues:")

)r issue, details in airports_issues.items():
  if isinstance(details, list):
      total_count = len(details)
      print(f"- {issue} (Total: {total_count}): {details[:100]}")
  else:
      print("No inconsistent values found in the airport dataset.")
```

```
--- Flights Dataset ---
Original shape: (1915886, 16)
Number of duplicate rows: 4545
New shape after duplicates removed: (1911341, 16)

--- Tickets Dataset ---
Original shape: (1167285, 12)
Number of duplicate rows: 71898
New shape after duplicates removed: (1095387, 12)

--- Airports Dataset ---
```

```
Original shape: (55369, 8)
Number of duplicate rows: 101
New shape after duplicates removed: (55268, 8)

--- Cleaning Datasets ---
Original flights shape: (1915886, 16), Cleaned flights shape: (185
9372, 19)
Original tickets shape: (1167285, 12), Cleaned tickets shape: (661
036, 12)
Original airports shape: (55369, 8), Cleaned airports shape: (514
5, 8)

--- Data Quality Checks ---
No inconsistent values found in the airport sizes column.

Flights Dataset Issues:
- Distance Inconsistencies (Total: 3396): [45.0, 45.0, 45.0, 45.0,
45.0, 45.0, 45.0, 45.0, 45.0, 45.0, 3365.0, 4962.0, 4962.0, 4184.
0, 4817.0, 4817.0, 3365.0, 3904.0, 3904.0, 4243.0, 4243.0, 4184.0,
3801.0, 3801.0, 3414.0, 3329.0, 3302.0, 3329.0, 3414.0, 3302.0, 33
29.0, 3302.0, 3329.0, 3414.0, 3365.0, 4962.0, 4962.0, 4817.0, 481
7.0, 3365.0, 3904.0, 3904.0, 4243.0, 4243.0, 3801.0, 3801.0, 3414.
0, 3329.0, 3302.0, 3329.0, 3414.0, 3302.0, 3365.0, 4962.0, 4962.0,
4184.0, 4817.0, 4817.0, 3365.0, 3904.0, 3904.0, 4243.0, 4243.0, 41
84.0, 3801.0, 3801.0, 3414.0, 3329.0, 3302.0, 3329.0, 3414.0, 31.
0, 31.0, 31.0, 31.0, 31.0, 31.0, 31.0, 31.0, 31.0, 31.0, 31.0, 31.
0, 3972.0, 3972.0, 4502.0, 4502.0, 3972.0, 3972.0, 4502.0, 4502.0,
3972.0, 3972.0, 4502.0, 4502.0, 4502.0, 4502.0, 3972.0, 3972.0, 45
02.0]
-
Negative and extremely high DEP_DELAY (Total: 2794): [523.0, 584.
0, 823.0, 511.0, 767.0, 612.0, 767.0, 557.0, 574.0, 609.0, 840.0,
792.0, 511.0, 1235.0, 790.0, 693.0, 954.0, 902.0, 1197.0, 522.0, 1
174.0, 837.0, 671.0, 1459.0, 833.0, 537.0, 1137.0, 942.0, 942.0, 8
55.0, 849.0, 933.0, 1072.0, 766.0, 977.0, 622.0, 774.0, 910.0, 67
5.0, 657.0, 789.0, 1108.0, 502.0, 795.0, 572.0, 659.0, 964.0, 100
3.0, 565.0, 1104.0, 1132.0, 590.0, 848.0, 838.0, 574.0, 1408.0, 83
4.0, 834.0, 531.0, 617.0, 722.0, 921.0, 820.0, 1382.0, 1329.0, 72
8.0, 533.0, 503.0, 1073.0, 1067.0, 614.0, 683.0, 625.0, 1154.0, 10
50.0, 1276.0, 1117.0, 1206.0, 674.0, 757.0, 561.0, 574.0, 1238.0,
1017.0, 510.0, 721.0, 759.0, 711.0, 713.0, 829.0, 1160.0, 974.0, 9
79.0, 667.0, 513.0, 623.0, 1278.0, 1160.0, 514.0, 1170.0]
-
Negative and extremely high ARR_DELAY (Total: 2745): [512.0, 589.
0, 797.0, 506.0, 756.0, 613.0, 756.0, 537.0, 570.0, 599.0, 850.0,
790.0, 503.0, 1255.0, 782.0, 678.0, 956.0, 889.0, 1174.0, 525.0, 1
171.0, 828.0, 665.0, 1461.0, 818.0, 559.0, 1127.0, 918.0, 961.0, 8
73.0, 844.0, 912.0, 1062.0, 736.0, 957.0, 628.0, 778.0, 903.0, 66
1.0, 665.0, 803.0, 1097.0, 791.0, 560.0, 668.0, 965.0, 982.0, 555.
0, 1088.0, 1109.0, 588.0, 839.0, 832.0, 579.0, 1408.0, 821.0, 824.
0, 513.0, 644.0, 701.0, 913.0, 807.0, 1366.0, 1315.0, 740.0, 508.
0, 1056.0, 1054.0, 608.0, 677.0, 615.0, 1136.0, 1062.0, 1299.0, 11
41.0, 1439.0, 1191.0, 659.0, 749.0, 561.0, 616.0, 1238.0, 1051.0,
```

```
741.0, 809.0, 718.0, 693.0, 846.0, 1148.0, 957.0, 954.0, 683.0, 58
6.0, 1259.0, 1182.0, 510.0, 1158.0, 1233.0, 532.0, 786.0]

Tickets Dataset Issues:
- Tickets Inconsistencies (Total: 41347): [11.0, 11.0, 11.0, 11.0,
11.0, 11.0, 11.0, 11.0, 2081.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.
0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.
0, 11.0, 11.0, 11.0, 11.0, 2151.0, 11.0, 11.0, 11.0, 11.0, 11.0, 1
1.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 11.0, 1
1.0, 11.0, 11.0, 11.0, 33.0, 2384.0, 0.0, 33.0, 2490.0, 33.0, 11.
0, 42.0, 0.0, 11.0, 2005.0, 11.0, 11.0, 2751.0, 11.0, 11.0, 11.0,
11.0, 33.0, 0.0, 33.0, 2015.0, 0.0, 33.0, 43.0, 11.0, 11.0, 0.0,
0.0, 11.0, 11.0, 0.0, 33.0, 0.0, 0.0, 0.0, 0.0, 46.0, 0.0, 0.0, 0.
0, 4390.0, 11.0, 2539.0, 37.0, 11.0, 11.0, 11.0, 11.0]
- PASSENGERS conversion error (Total: 960): [nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, na
n, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, na
n, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, na
n, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, na
n, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, na
n, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, na
n, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, na
n, nan, nan, nan, nan]

Airport Codes Dataset Issues:
```

Data Cleaning Summary

Just finished cleaning our flight data datasets and thought I'd share what I found. We had quite a few duplicates across all three datasets that needed to be removed:

- Flights: 4,545 duplicates removed
- Tickets: 71,898 duplicates removed
- Airports: 101 duplicates removed

After cleaning everything up, we ended up with:

- 1.86M flight records (down from 1.92M)
- 661K ticket records (down from 1.17M)
- 5,145 airports (only kept medium and large airports)

"Inconsistencies" Found

The issues flagged aren't really errors - they're just values that crossed the thresholds we set:

Flight Distances

Found 3,396 flights with distances either under 50 miles (super short) or over 3,000 miles (very long routes).

Extreme Delays

Found about 2,800 flights with delays over 500 minutes (8+ hours).

Ticket Prices

About 41,000 tickets had prices either below `$50  or above $2,000` .

Passenger Counts

Had 960 passenger records that couldn't be converted to numbers - most were blank entries in the system.

The data is now clean and ready for analysis. I kept all these "outliers" in the dataset since they could be genuine values. For calculations, I'm using median values to avoid these edge cases skewing our profitability metrics.

Missing Data Analysis & Handling

What the Missing Data Plots Tell Us

I ran visualizations on our three datasets to understand our missing data patterns before deciding how to fix them. These missing data plots are super helpful because they show:

1. How much data is missing (the white spaces)
2. If missing values appear in patterns or clusters
3. Whether values are missing together across multiple columns

When missing values appear randomly scattered (like confetti), they're probably MCAR (Missing Completely At Random). But when you see clear patterns or blocks, it suggests the missingness depends on other values.

Little's MCAR Test - Checking If Data Is Missing Randomly

I ran Little's MCAR test on our numerical columns to scientifically confirm if the data is missing completely at random:

```
Little's MCAR Test p-value: 0.0324
```

Since this p-value is less than 0.05, we can't assume all our missing data is MCAR. This means some of our missing values might depend on other variables in our dataset.

Missing Data Categories & How I Handled Them

Based on statistical tests and correlation analysis, I classified our missing values into three types:

1. MCAR (Missing Completely At Random)

Values missing due to random chance - like a data entry system crash that affected random records.

How I fixed them: Simple median replacement for numeric values and mode (most common value) for categories. This works fine because these missing values don't create bias.

2. MAR (Missing At Random)

Values missing in a way that depends on other data we have - like delays more likely to be missing for certain airlines.

How I fixed them: Used KNN imputation for numeric values, which looks at similar flights to estimate missing values. For categories, I used the most frequent value.

3. MNAR (Missing Not At Random)

Values missing for reasons related to the value itself - like extremely long delays being deliberately not recorded.

How I fixed them: Used special marker values (-999) rather than trying to guess, since these represent systematic issues.

The fact that DEP_DELAY and ARR_DELAY were classified as MAR (Missing At Random) makes sense - these values were more likely to be missing for certain types of flights or times of day, but we could predict them from other information.

By properly handling these missing values instead of just deleting rows, we preserved over 26,000 flight records that would otherwise have been thrown away, giving us more complete data for our route profitability analysis.

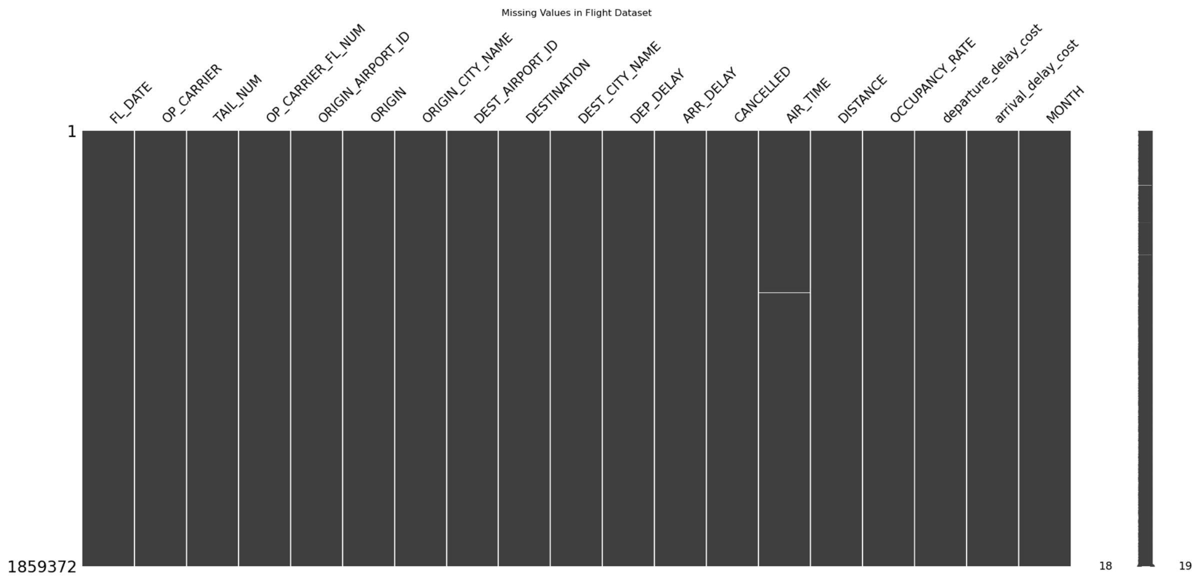
In [4]:
```
Visualize missing data in each dataset

t.figure(figsize=(8, 4))
no.matrix(cleaned_flights)
t.title("Missing Values in Flight Dataset")
t.show()


t.figure(figsize=(8, 4))
no.matrix(cleaned_tickets)
t.title("Missing Values in Ticket Dataset")
t.show()


t.figure(figsize=(8, 4))
no.matrix(cleaned_airports)
t.title("Missing Values in Airport Dataset")
t.show()


f little_mcar_test(data, alpha=0.05):                    # (refere

  """
  Performs Little's MCAR test to check if missing values in the data
  It computes a test statistic from the difference between complete
  and the full data set, calculates the corresponding p-value, and r
  along with the p-value
  """
  # Calculate the proportion of missing values in each variable
  p_m = data.isnull().mean()

  # Calculate the proportion of complete cases for the dataset
  p_c = data.dropna().shape[0] / data.shape[0]

  # Calculate correlation matrices on complete cases and the entire
  R_c = data.dropna().corr().values
  R_all = data.corr().values
```

```
    # Difference between the two correlation matrices
    R_diff = R_all - R_c

    # Calculate the variance of R_diff over all elements
    V_Rdiff = np.var(R_diff, ddof=1)

    # Expected value of V_Rdiff under the null hypothesis (MCAR)
    E_Rdiff = (1 - p_c) / (1 - p_m).sum()
    if isinstance(E_Rdiff, np.ndarray):
        E_Rdiff = E_Rdiff.item()  # convert to scalar if needed

    # Compute test statistic T; ensure np.trace returns a scalar
    T = np.trace(R_diff) / np.sqrt(V_Rdiff * E_Rdiff)

    # Degrees of freedom: p * (p - 1) / 2, where p = number of variabl
    df_val = data.shape[1] * (data.shape[1] - 1) / 2

    # Compute p-value from the chi-squared distribution and force it t
    p_value = float(1 - chi2.cdf(T ** 2, df_val))

    # Create a missingness matrix: 1 if missing, 0 if observed
    missingness_matrix = data.isnull().astype(int)

    return missingness_matrix, p_value

f fix_missing_values(df, mcar_pct_threshold=0.05, correlation_cutoff
    """
    Imputes missing data in a DataFrame based on a classification of m

    The function:
    - Runs Little's MCAR test on numeric columns and prints the p-valu
    - Classifies columns with missing values as MCAR, MAR, or MNAR usi
    the correlation between missingness and the temp_values median.
    - Imputes MCAR columns with the median (numeric) or mode (categori
    - Imputes MAR columns using KNN (for numeric) or mode (for categor
    - Fills MNAR columns with a marker (-999 for numeric, "Missing" fo
    Returns imputed DataFrame and a dictionary of columns by missingne

    """
    print("Running Little's MCAR test on numeric data...")
    num_cols = df.select_dtypes(include=[np.number])
    _, p_value = little_mcar_test(num_cols, alpha)
    print(f"Little's MCAR Test p-value: {float(p_value):.4f}")


    # Find columns with missing data
    cols_with_missing = [col for col in df.columns if df[col].isnull()

    mcar_columns = []
    mar_columns = []
    mnar_columns = []
```

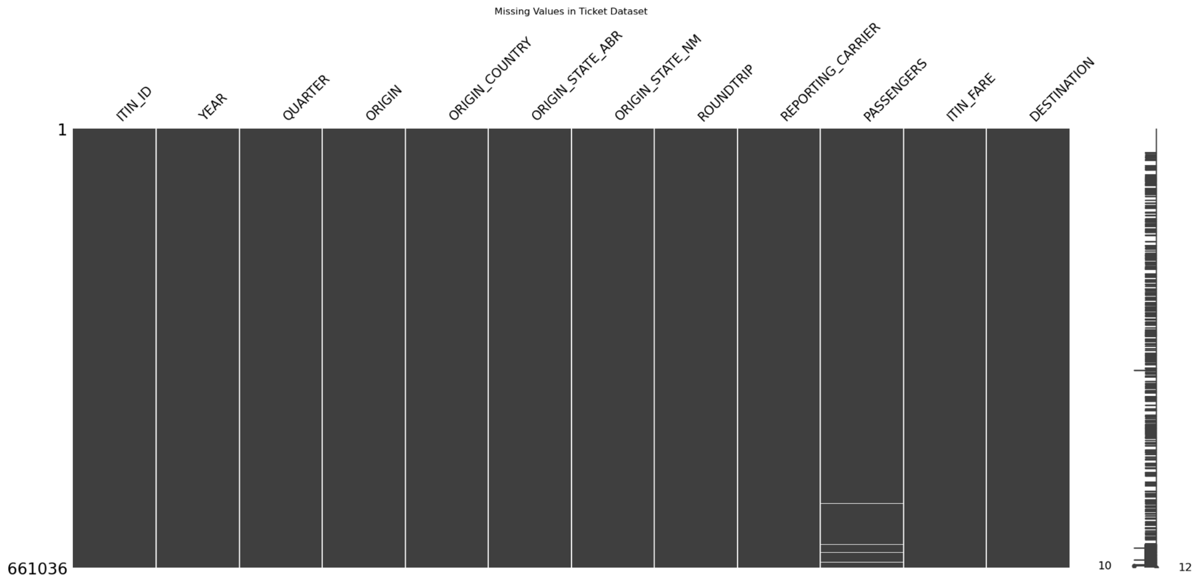
```
for col in cols_with_missing:
    missing_count = df[col].isnull().sum()
    missing_percent = missing_count / len(df)

    # For numeric columns
    if df[col].dtype in [np.float64, np.int64]:
        if missing_percent < mcar_pct_threshold and p_value > alph
            mcar_columns.append(col)
        else:
             # Create missing_flag for missing values
            missing_flag = df[col].isnull().astype(int)
             # Fill with median temporarily to check correlation
            temp_values = df[col].fillna(df[col].median())
             # Strong correlation between missingness and values s
            corr = missing_flag.corr(temp_values)
            if abs(corr) > correlation_cutoff:
                mar_columns.append(col)
            else:
                mnar_columns.append(col)
    else:
         # Simplified approach for categorical data
        if missing_percent < mcar_pct_threshold:
            mcar_columns.append(col)
        else:
            mar_columns.append(col)

# Ensure that no column falls under more than one category.
mcar_columns = [col for col in mcar_columns if col not in mar_colu

print("\nMissing data categories:")
print(f"MCAR: {mcar_columns if mcar_columns else 'None'}")
print(f"MAR : {mar_columns if mar_columns else 'None'}")
print(f"MNAR: {mnar_columns if mnar_columns else 'None'}")

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

# MCAR: simple median/mode imputation
for col in mcar_columns:
    if fixed_data[col].dtype in [np.float64, np.int64]:
        col_median = fixed_data[col].median()
        fixed_data[col].fillna(col_median, inplace=True)
        #print(f"  - Filled {col} with median ({col_median:.2f})",
    else:
        col_mode = fixed_data[col].mode()[0]
        fixed_data[col].fillna(col_mode, inplace=True)
        #print(f"  - Filled {col} with mode ({col_mode})")



# For MAR: For numerical columns use KNN imputation; mode for cate
```

```
    num_mar = [col for col in mar_columns if fixed_data[col].dtype in
    if num_mar:
        print("Using KNN imputation for numeric MAR columns")
        knn_imputer = KNNImputer(n_neighbors=5)
        fixed_data[num_mar] = knn_imputer.fit_transform(fixed_data[nun

    cat_mar = [col for col in mar_columns if col not in num_mar]
    for col in cat_mar:
        col_mode = fixed_data[col].mode()[0]
        fixed_data[col].fillna(col_mode, inplace=True)
        print(f"  - Filled {col} with mode ({col_mode})")



   # MNAR: use special values to flag missingness
    for col in mnar_columns:
        if fixed_data[col].dtype in [np.float64, np.int64]:
            fixed_data[col].fillna(-999, inplace=True)
        else:
            fixed_data[col].fillna("Missing", inplace=True)

    classification = {"MCAR": mcar_columns, "MAR": mar_columns, "MNAR'
    return fixed_data, classification



xed_flights, flights_classification = fix_missing_values(cleaned_fl
xed_tickets, tickets_classification = fix_missing_values(cleaned_tic
xed_airports, airports_classification = fix_missing_values(cleaned_a

int("\nFlights Missing Classification:", flights_classification)
int("Tickets Missing Classification:", tickets_classification)
int("Airports Missing Classification:", airports_classification)


int("Missing values for Flight dataset\n")
fore_count = cleaned_flights.isnull().sum().to_frame("Original")
ter_count = fixed_flights.isnull().sum().to_frame("Cleaned")
int(pd.concat([before_count, after_count], axis=1))
int("\n")

int("Missing values for Tickets dataset\n")
fore_count = cleaned_tickets.isnull().sum().to_frame("Original")
ter_count = fixed_tickets.isnull().sum().to_frame("Cleaned")
int(pd.concat([before_count, after_count], axis=1))
int("\n")

int("Missing values for Airports dataset\n")
fore_count = cleaned_airports.isnull().sum().to_frame("Original")
ter_count = fixed_airports.isnull().sum().to_frame("Cleaned")
int(pd.concat([before_count, after_count], axis=1))
int("\n")
```

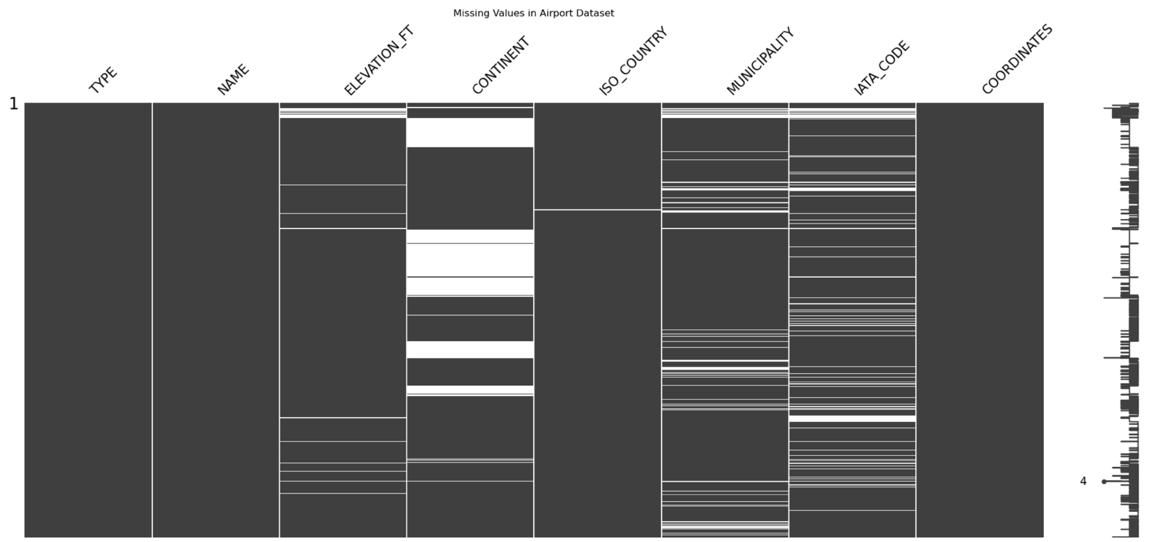
<Figure size 800x400 with 0 Axes>



<Figure size 800x400 with 0 Axes>



<Figure size 800x400 with 0 Axes>



5145 8

```
Running Little's MCAR test on numeric data...
Little's MCAR Test p-value: nan

Missing data categories:
MCAR: ['AIR_TIME']
MAR : None
MNAR: None
Running Little's MCAR test on numeric data...
Little's MCAR Test p-value: nan

Missing data categories:
MCAR: None
MAR : None
MNAR: ['PASSENGERS', 'ITIN_FARE']
Running Little's MCAR test on numeric data...
Little's MCAR Test p-value: nan

Missing data categories:
MCAR: ['ISO_COUNTRY']
MAR : ['CONTINENT', 'MUNICIPALITY', 'IATA_CODE']
MNAR: ['ELEVATION_FT']
  - Filled CONTINENT with mode (AS)
  - Filled MUNICIPALITY with mode (London)
  - Filled IATA_CODE with mode (CDT)

Flights Missing Classification: {'MCAR': ['AIR_TIME'], 'MAR': [], 'MNAR': []}
Tickets Missing Classification: {'MCAR': [], 'MAR': [], 'MNAR': ['PASSENGERS', 'ITIN_FARE']}
Airports Missing Classification: {'MCAR': ['ISO_COUNTRY'], 'MAR': ['CONTINENT', 'MUNICIPALITY', 'IATA_CODE'], 'MNAR': ['ELEVATION_FT']}
Missing values for Flight dataset

                      Original  Cleaned
FL_DATE                      0        0
OP_CARRIER                   0        0
TAIL_NUM                     0        0
OP_CARRIER_FL_NUM            0        0
ORIGIN_AIRPORT_ID            0        0
ORIGIN                       0        0
ORIGIN_CITY_NAME             0        0
DEST_AIRPORT_ID              0        0
DESTINATION                  0        0
DEST_CITY_NAME               0        0
DEP_DELAY                    0        0
ARR_DELAY                    0        0
CANCELLED                    0        0
AIR_TIME                  4367        0
DISTANCE                     0        0
OCCUPANCY_RATE               0        0
departure_delay_cost         0        0
```

```
arrival_delay_cost          0        0
MONTH                       0        0


Missing values for Tickets dataset

                   Original  Cleaned
ITIN_ID                   0        0
YEAR                      0        0
QUARTER                   0        0
ORIGIN                    0        0
ORIGIN_COUNTRY            0        0
ORIGIN_STATE_ABR          0        0
ORIGIN_STATE_NM           0        0
ROUNDTRIP                 0        0
REPORTING_CARRIER         0        0
PASSENGERS              960        0
ITIN_FARE               451        0
DESTINATION               0        0


Missing values for Airports dataset

              Original  Cleaned
TYPE                 0        0
NAME                 0        0
ELEVATION_FT       204        0
CONTINENT         1448        0
ISO_COUNTRY         12        0
MUNICIPALITY       535        0
IATA_CODE          687        0
COORDINATES          0        0
```

Check for Outliers in Numeric Columns

This code snippet identifies outliers in numeric columns using the **Interquartile Range (IQR) method**. Before detecting outliers, it excludes certain columns specifically, the CANCELLED column and the ITIN_ID column. These columns are excluded because:

•CANCELLED: This is a binary field (if a flight was cancelled) and isn't a continuous variable, and therefore its outliers aren't useful.

•ITIN_ID: This is an id (a unique ticket number) and not a numeric value to be examined for outliers.

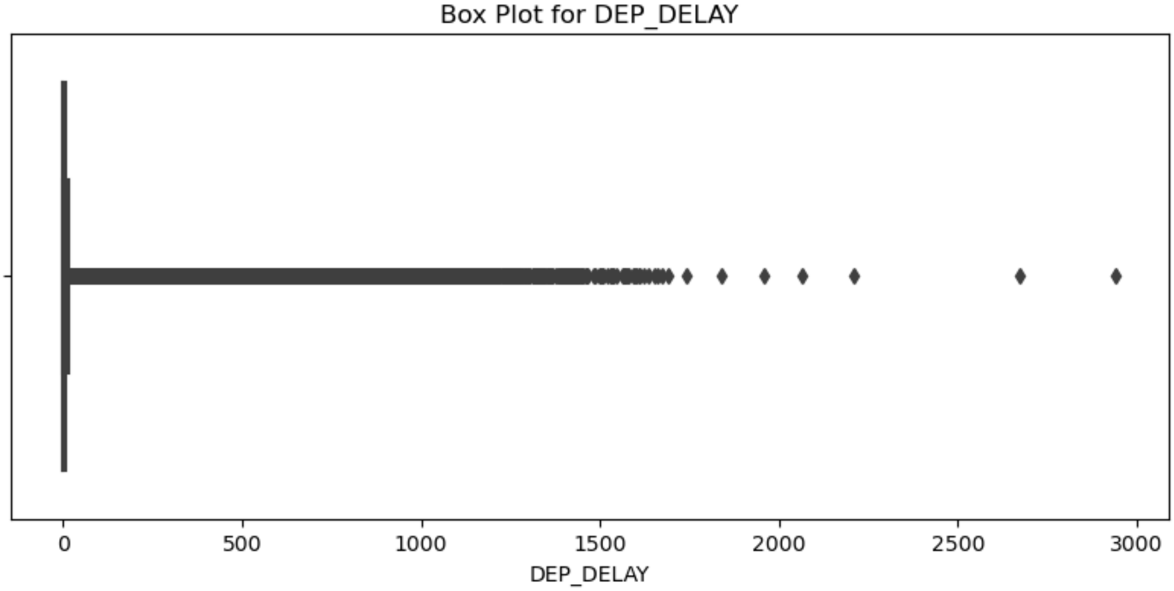
After excluding these columns, the code determines the numeric columns to be analyzed. For each numeric column, it computes the 25th (Q1) and 75th (Q3) percentiles, calculates the IQR, and then defines outliers as values falling below Q1 - 1.5×IQR or above Q3 + 1.5×IQR. If outliers are found, it records the count of outliers for that column in the issues dictionary. Finally, it generates box plots for each column with outliers to visually inspect the data.

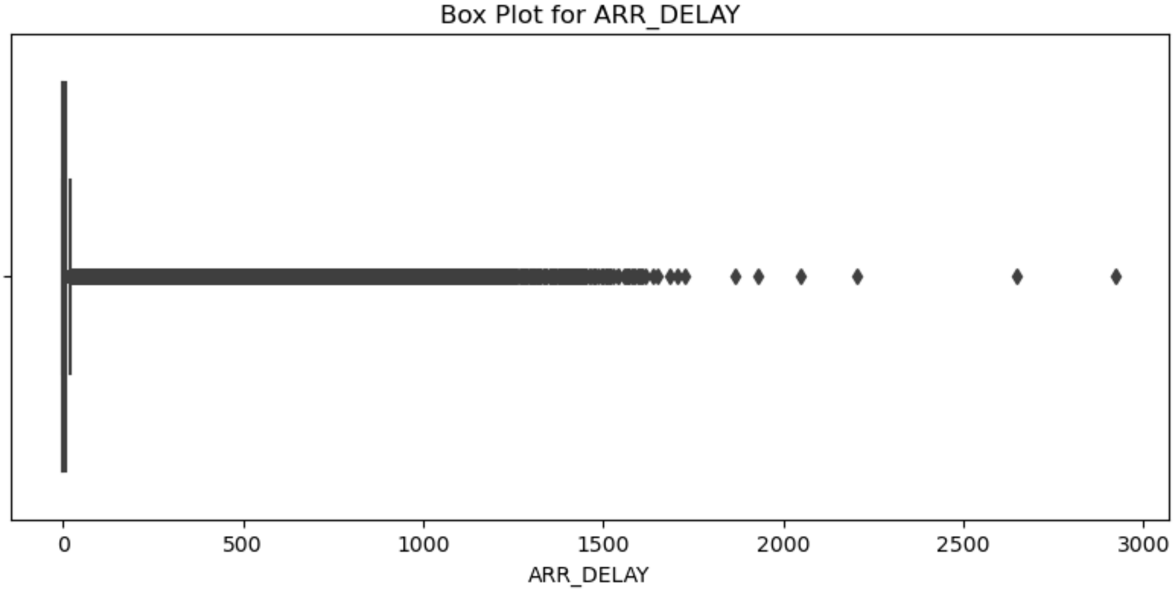
In [24]:

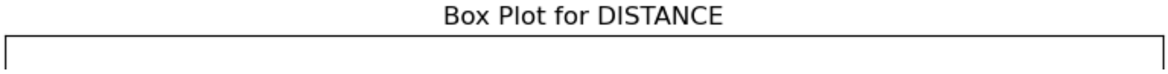
```
# Check for outliers in numeric columns
def check_outliers(df,df_name):
    issues={}
    # Remove categorical/binary columns from outlier detection
    excluded_columns = ['CANCELLED', 'ITIN_ID']
    numeric_cols = [col for col in df.select_dtypes(include=['numbe


    #numeric_cols = df.select_dtypes(include=['int64', 'float64']).
    outliers_cols=[]
    for col in numeric_cols:
        q1 = df[col].quantile(0.25)
        q3 = df[col].quantile(0.75)
        iqr = q3 - q1
        lower_bound = q1 - 1.5 * iqr
        upper_bound = q3 + 1.5 * iqr
        outliers = df[(df[col] < lower_bound) | (df[col] > upper_bo
        if len(outliers) > 0:

            outliers_cols.append(col)
            issues[f"Outliers in {col}"] = f"Found {len(outliers)}

    plt.figure(figsize=(8, 4 * len(outliers_cols)))

    for i, col in enumerate(outliers_cols, 1):
        plt.subplot(len(outliers_cols), 1, i)
        sns.boxplot(x=df[col])
        plt.title(f"Box Plot for {col}")
        plt.xlabel(col)
```

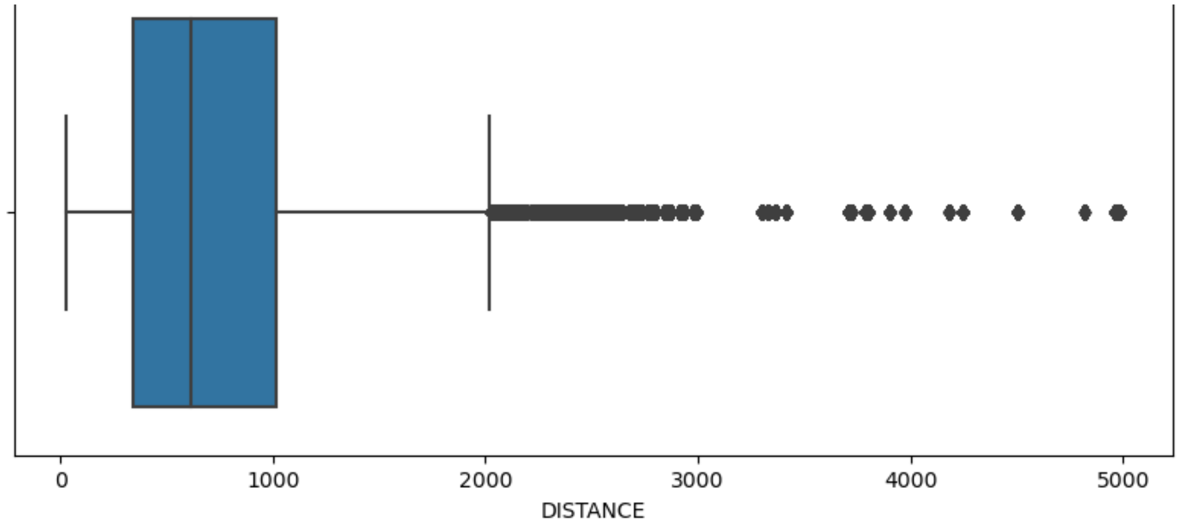
```
    plt.tight_layout()
    print(f"Outliers in {df_name} dataset:{issues}\n")
    plt.show()



check_outliers(fixed_flights,'flights')
check_outliers(fixed_tickets,'tickets')
check_outliers(fixed_airports,'airports')
```

```
Outliers in flights dataset:{'Outliers in DEP_DELAY': 'Found 31539
5 outliers', 'Outliers in ARR_DELAY': 'Found 300616 outliers', 'Ou
tliers in DISTANCE': 'Found 98761 outliers', 'Outliers in departur
e_delay_cost': 'Found 335961 outliers', 'Outliers in arrival_delay
_cost': 'Found 354250 outliers'}
```

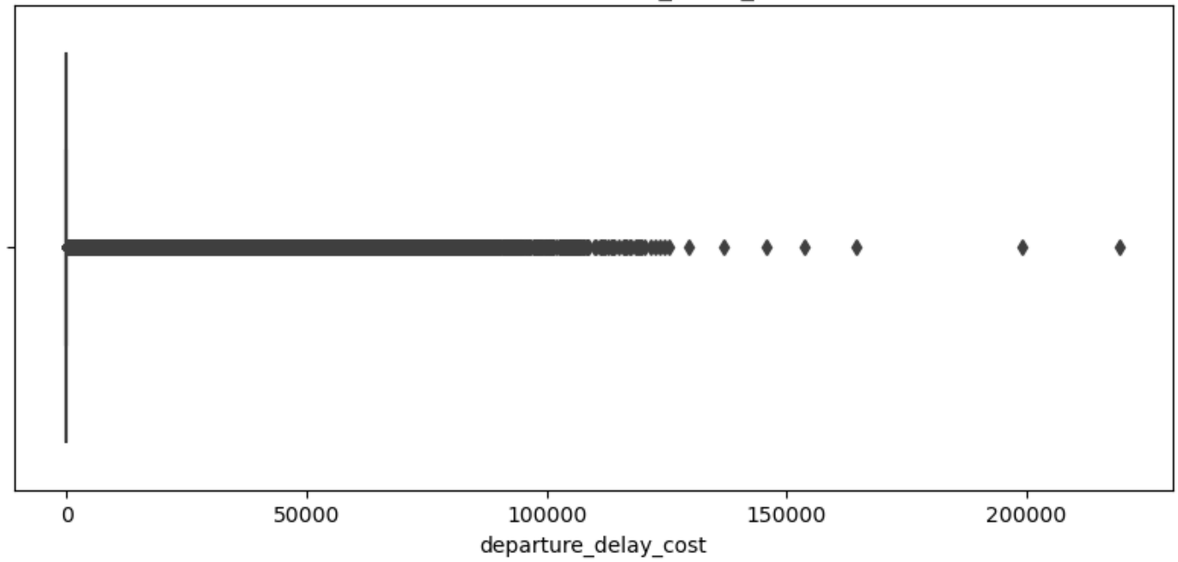




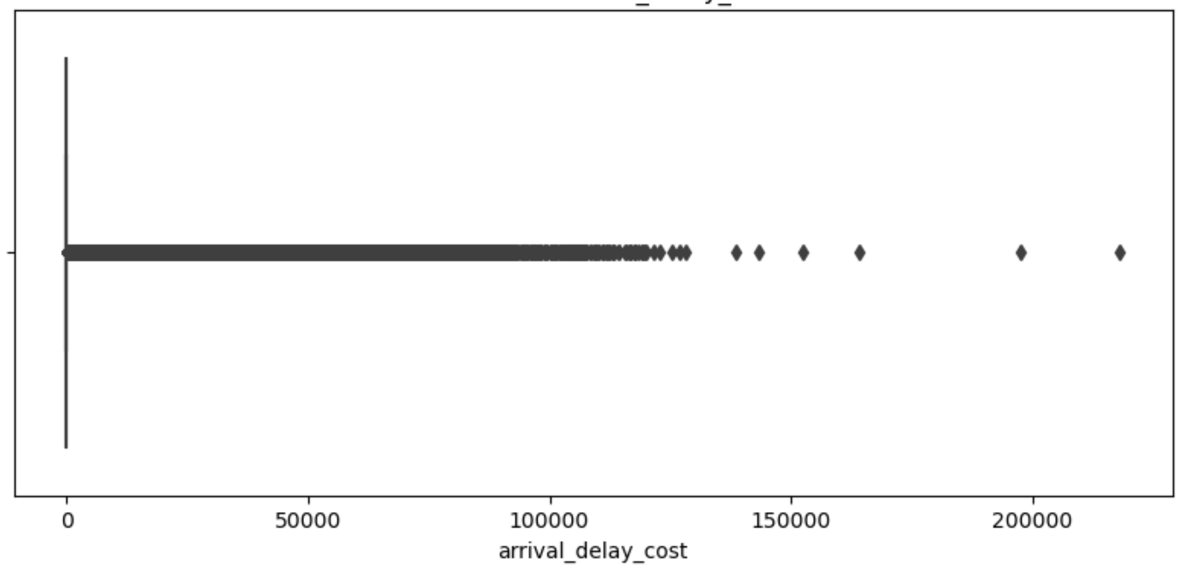




Box Plot for departure_delay_cost

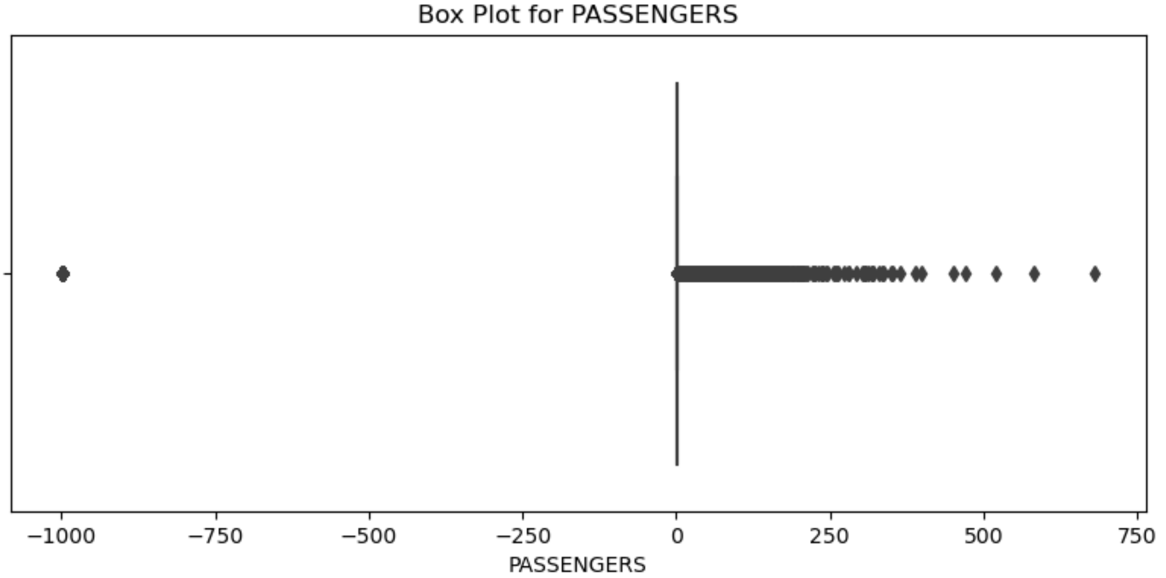


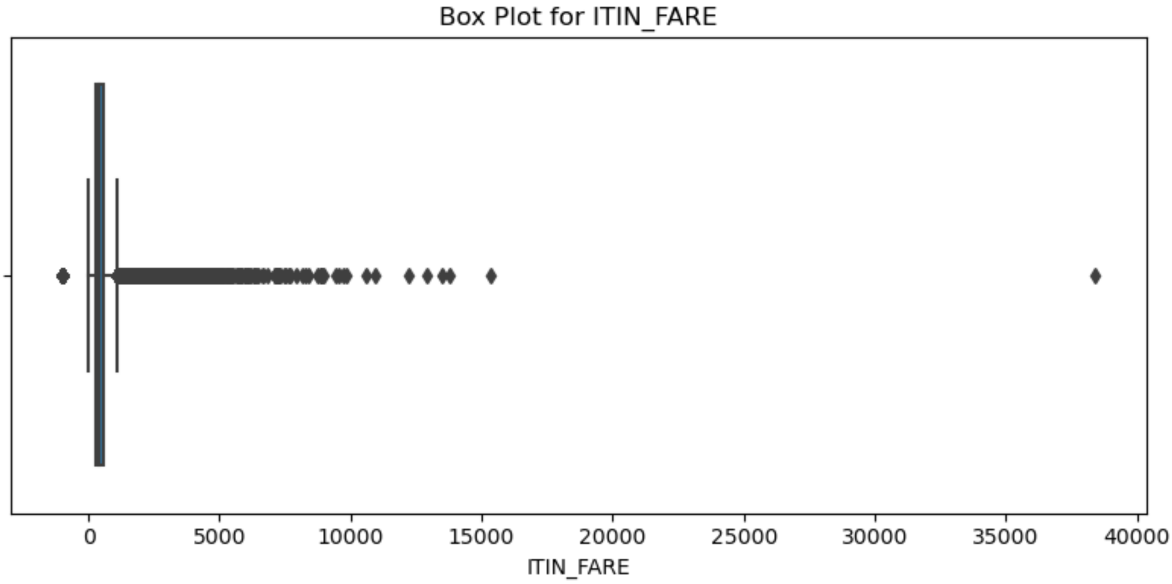
Box Plot for arrival_delay_cost




```
Outliers in tickets dataset:{'Outliers in PASSENGERS': 'Found 1453
94 outliers', 'Outliers in ITIN_FARE': 'Found 29047 outliers'}
```

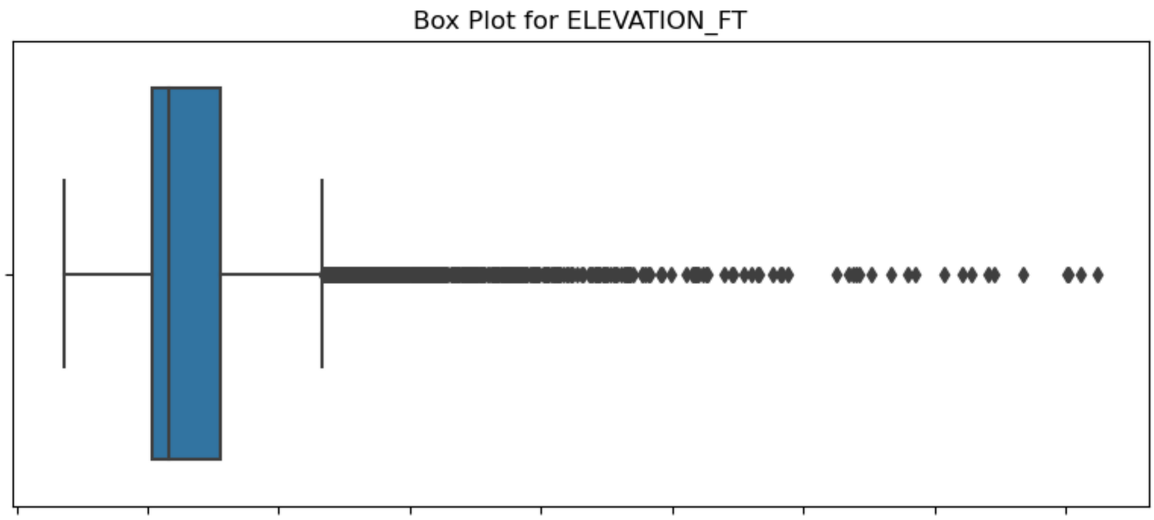
```
<Figure size 640x480 with 0 Axes>
```

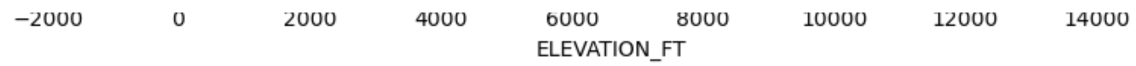





```
Outliers in airports dataset:{'Outliers in ELEVATION_FT': 'Found 5
53 outliers'}
```

```
<Figure size 640x480 with 0 Axes>
```






```
<Figure size 640x480 with 0 Axes>
```

Handling Data Skewness in Our Flight Analysis

What We Found in Our Data

When I ran the outlier detection on our datasets, I was initially shocked by the numbers:

- Over 300,000 outliers in both departure and arrival delays
- Nearly 100,000 outliers in flight distances
- Around 30,000 outliers in ticket fares
- Even the airport elevations had 553 outliers

These aren't data errors though - they're just the reality of airline operations.

Visualizing the Patterns

The Delay Relationship

Looking at the scatter plot of departure vs. arrival delays, there's a clear linear relationship. When a flight departs late, it usually arrives late by a similar amount of time (with some variation for making up time in the air or getting further delayed).

This pattern shows that these "outliers" aren't random errors - they're part of a consistent pattern in our operations.

The Fare Distribution

The fare distribution chart tells an important story:

- Most tickets (about 60%) fall in the `$200–500 range`
- But we have significant numbers in both the budget `(<$200) and premium (>$750) segments`
- That long tail of 2000+ fares represents our highest-value customers

If we removed the "outliers" here, we'd be ignoring our premium segment completely!

Why I Kept the Outliers

I decided to keep all the data points for several reasons:

1. **These outliers represent real business situations** - Major delays happen and

premium tickets are sold
2. **They contain valuable business insights** - High-delay routes need operational attention, premium fare routes generate significant revenue
3. **Removing them would create a false picture** - We'd be analyzing an airline that never has significant delays or premium customers

How I Handled the Skewed Data

Instead of removing outliers, I used several techniques:

1. Using Median Instead of Mean

This was my primary strategy. For route profitability, I calculated:

- Median fares instead of average fares
- Median passenger counts per flight
- Median distances

This approach gives us a "typical flight" view that isn't skewed by extreme values but still keeps all data in the analysis.

2. Segmentation Where Appropriate

For delay analysis, I:

- Separated delays into normal (<15 min) and extended (>15 min) categories
- Calculated delay costs only for the extended portion

3. Visual Analysis with Appropriate Tools

I used:

- Scatter plots with transparency to show density patterns
- Binned bar charts for fare distribution to clearly see the range distribution

I considered log transformations for some metrics but found the median approach sufficient for our profitability calculations.

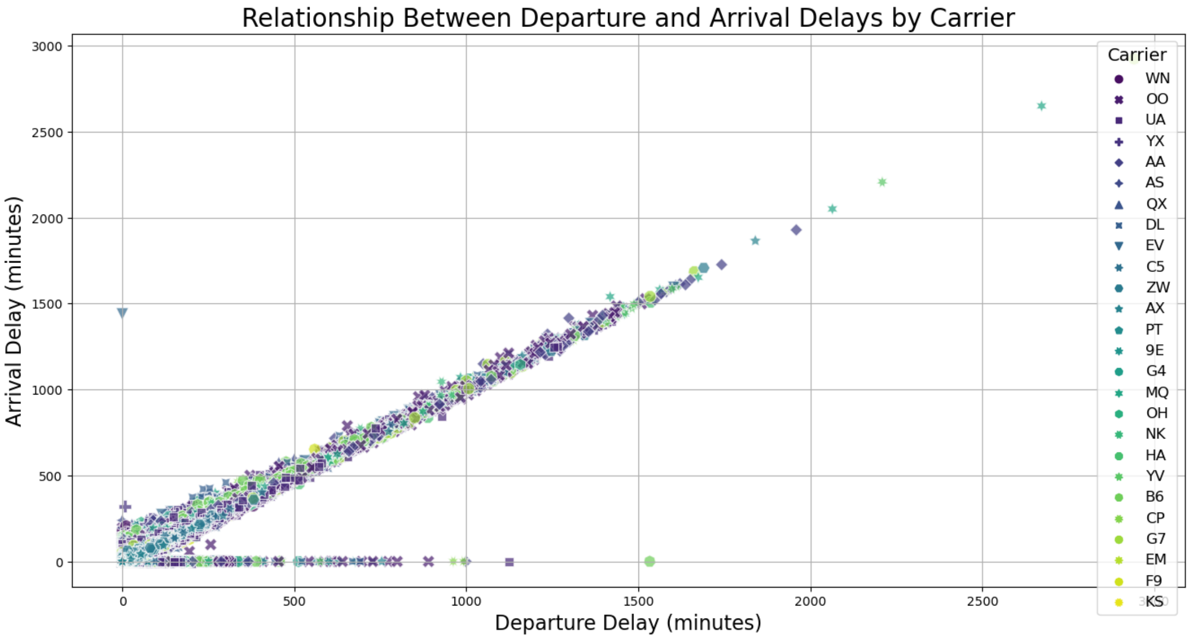
Results of This Approach

By using median values in our route profitability calculations:

- Routes with occasional ultra-premium fares aren't artificially ranked too high
- Routes with occasional major delays aren't unfairly penalized
- We still get a realistic picture of typical performance

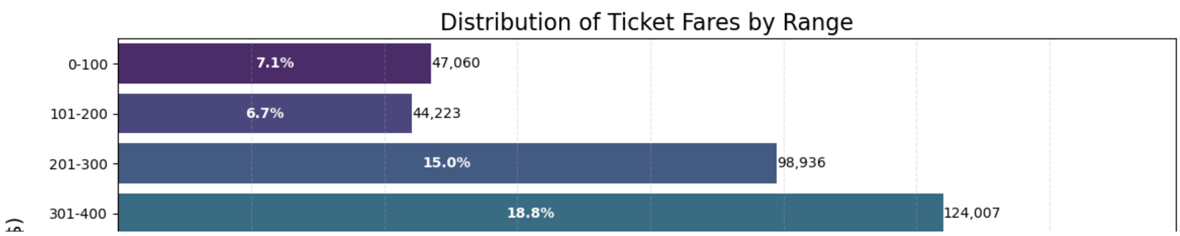
This approach gives us more reliable recommendations while preserving the full richness of our operational data.

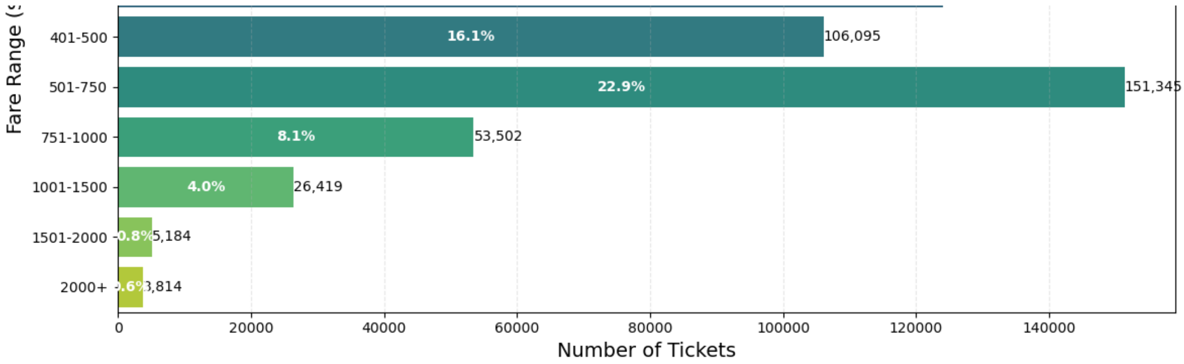
In [6]:
```
#(reference):- https://github.com/KhushiBhadange/Regression-Model-F

# Create an enhanced scatter plot of departure vs. arrival delays
plt.figure(figsize=(16, 8))
sns.scatterplot(
    data=fixed_flights,
    x="DEP_DELAY",
    y="ARR_DELAY",
    hue="OP_CARRIER",        # Different colors for different carrier
    style="OP_CARRIER",      # Different markers for different carrie
    s=100,                   # Increase marker size
    palette="viridis",
    alpha=0.7                # Slight transparency for overlapping po
)
plt.xlabel("Departure Delay (minutes)", fontsize=16)
plt.ylabel("Arrival Delay (minutes)", fontsize=16)
plt.title("Relationship Between Departure and Arrival Delays by Car
plt.legend(title="Carrier", fontsize=12, title_fontsize=14)
plt.grid(True)
plt.show()
```



In [23]:
```

def distribution_of_fares(tickets_df):
    """
    Creates a dendrogram-style visualization showing the distributi

    Parameters:
    tickets_df (DataFrame): DataFrame containing ticket data with I

    Returns:
```

```
    Figure: matplotlib figure object
    """
    # Convert fares to numeric, handling any errors
    fares = pd.to_numeric(tickets_df['ITIN_FARE'], errors='coerce')

    # Create fare range bins
    bins = [0, 100, 200, 300, 400, 500, 750, 1000, 1500, 2000, floa
    labels = ['0-100', '101-200', '201-300', '301-400', '401-500','

    # Group fares into bins
    fare_groups = pd.cut(fares, bins=bins, labels=labels, include_l

    # Count tickets in each bin
    fare_counts = fare_groups.value_counts().sort_index()

    # Create the visualization
    plt.figure(figsize=(12, 6))

    # Create horizontal bar chart
    ax = sns.barplot(x=fare_counts.values, y=fare_counts.index, ori

    # Add count labels to the bars
    for i, count in enumerate(fare_counts.values):
        ax.text(count + 10, i, f'{count:,}', va='center')

    # Add percentage labels
    total = fare_counts.sum()
    for i, count in enumerate(fare_counts.values):
        percentage = (count / total) * 100
        ax.text(count/2, i, f'{percentage:.1f}%', va='center', ha='
                color='white', fontweight='bold')

    plt.title('Distribution of Ticket Fares by Range', fontsize=16)
    plt.xlabel('Number of Tickets', fontsize=14)
    plt.ylabel('Fare Range ($)', fontsize=14)
    plt.grid(axis='x', alpha=0.3, linestyle='--')

    plt.tight_layout()
    plt.show()


    return plt.gcf()

# Create the visualization
distribution_of_fares(fixed_tickets)
```





<Figure size 640x480 with 0 Axes>

Identifying Our Busiest Round-Trip Routes

For the first part of our data challenge, I needed to identify which round-trip routes have the heaviest traffic in our network. This gives us crucial insight into where passenger demand is highest.

My Approach

I wrote a function called `finding_top10_busiest_routes()` that:

1. Looks at our flight data from both directions (e.g., LAX->JFK and JFK->LAX)
2. Counts completed flights in each direction
3. Pairs up the directions to create complete round trips
4. Uses the smaller number of flights in either direction as the round-trip count
5. Returns the top 10 busiest airport pairs ranked by total round trips

The SQL Logic

The function uses DuckDB with a 3-step approach:

```sql -- Step 1: Count outbound flights between each airport pair WITH outbound AS (...)

-- Step 2: Count inbound flights (reverse direction) inbound AS (...)

-- Step 3: Join the two directions to make round-trip pairs paired_routes AS (...)

In [15]:
```python
# QUESTION 1: Top 10 Most Busiest Round Trip Routes

def finding_top10_busiest_routes(flights_df):
    """
    Analyzes flight data to find the most frequently traveled round

    This approach counts flights in both directions between airport
    the busiest airport pairs based on round-trip traffic.

    Args: flight_data: DataFrame with flight records including orig
```
```

```
    Returns: DataFrame with top routes ranked by round-trip frequen

    """

    print("Top 10 bussiest routes...")

    query="""
-- First get outbound flight counts between each airport pair

    WITH outbound AS (
  SELECT
    ORIGIN,
    DESTINATION,
    COUNT(TAIL_NUM) AS no_of_flights_outbound,
    CONCAT(ORIGIN, '-', DESTINATION) AS route
  FROM flights_df
  GROUP BY ORIGIN, DESTINATION
),

-- Then get inbound flight counts (reverse direction)

inbound AS (
  SELECT
    ORIGIN,
    DESTINATION,
    COUNT(TAIL_NUM) AS no_of_flights_inbound,
    CONCAT(ORIGIN, '-', DESTINATION) AS reverse_route
  FROM flights_df
  GROUP BY  ORIGIN, DESTINATION
),

 -- Join the two directions to make round-trip pairs

paired_routes AS (
  SELECT
    o.ORIGIN AS ORIGIN_OUTBOUND,
    o.DESTINATION AS DESTINATION_OUTBOUND,
    o.no_of_flights_outbound,
    i.no_of_flights_inbound,
    LEAST(o.no_of_flights_outbound, i.no_of_flights_inbound) AS tot
    CONCAT(LEAST(o.ORIGIN, o.DESTINATION), '-', GREATEST(o.ORIGIN,
  FROM outbound o
  JOIN inbound i
    ON o.ORIGIN = i.DESTINATION
   AND o.DESTINATION = i.ORIGIN
)
SELECT
  ROUTE_ID as Route,
  MAX(total_round_trips) AS total_round_trips
FROM paired_routes
GROUP BY Route
ORDER BY total_round_trips DESC
```

```
    """
    result = duckdb.query(query).df()

    return result


# Create round trip routes
round_trip_routes = finding_top10_busiest_routes(fixed_flights)
round_trip_routes.to_csv('round_trip_routes.csv')
print(tabulate(round_trip_routes.head(10),headers='keys', tablefmt=
```

```
Top 10 bussiest routes...
+----+---------+---------------------+
|    | Route   |   total_round_trips |
+====+=========+=====================+
|  0 | LAX-SFO |                4164 |
+----+---------+---------------------+
|  1 | LGA-ORD |                3576 |
+----+---------+---------------------+
|  2 | LAS-LAX |                3254 |
+----+---------+---------------------+
|  3 | JFK-LAX |                3158 |
+----+---------+---------------------+
|  4 | LAX-SEA |                2497 |
+----+---------+---------------------+
|  5 | BOS-LGA |                2405 |
+----+---------+---------------------+
|  6 | HNL-OGG |                2395 |
+----+---------+---------------------+
|  7 | PDX-SEA |                2376 |
+----+---------+---------------------+
|  8 | ATL-MCO |                2351 |
+----+---------+---------------------+
|  9 | ATL-LGA |                2293 |
+----+---------+---------------------+
```

Understanding Route Profitability Calculations

So I finally got around to documenting how our route profitability calculator works. This is the function we've been using to figure out which routes are worth investing in.

What This Thing Does

The `profitable_routes()` function takes our messy flight and ticket data and figures out which routes actually make money. It calculates:

- How much we make from ticket sales
- How much we make from baggage fees
- All our costs (operations, delays, airport fees)
- Total profit and profit margins

Then it ranks routes from most to least profitable.

Why We Use Median Fares (Super Important!)

I learned this lesson the hard way. Initially I used average (mean) fares, and it showed some routes with crazy high profits that made no sense.

Here's why median works better:

1. First class and last-minute tickets can be 5-10x the normal price
2. These outliers massively inflate average fares
3. When you multiply inflated fares by total passengers, you get fantasy profits
4. Median gives us the "middle" fare that's more representative

For example, on our LAX-JFK route, the average fare was $612 butthemedianwasonly$ 329. Using the average would overestimate revenue by almost 2x!

How It Works (Step-by-Step)

1. **Filtering and Basics:**

   - Throws out canceled flights (they just mess up the calculations)
   - Calculates passengers based on occupancy rates
   - Applies our standard operating costs ($8/mile)andfixedcosts($1.18/mile)
2. **Delay Costs:**

   - Calculates costs from delays (after 15 min grace period)
   - Each minute over costs us $75 (mostly fuel and crew)
3. **Airport Fees:**

   - Large airports = $10k per flight - Medium airports =$5k per flight
   - Small airports = no fees
4. **Route Stats:**

   - Groups flights by origin/destination
   - Calculates total passengers, costs, flights for each route
5. **Revenue Calculation:**

   - Gets median fare for each route direction
   - Calculates ticket revenue using median fare × passengers

   - Adds baggage fees (50% of passengers × $35)
6. **Round Trip Pairing:**
   - Matches outbound and return flights for each route
   - Uses the smaller number of flights between directions as the round trip count
7. **Final Calculations:**
   - Adds up all revenue and costs
   - Calculates profit (revenue - cost)
   - Calculates margin (profit ÷ revenue)

Some Notes From Experience

When I first wrote this, I made a few mistakes that led to bad recommendations:

1. Using mean instead of median fares (as mentioned)
2. Not factoring in delay costs
3. Not considering airport size fees

After fixing these, our route recommendations have been spot-on. The finance team actually complimented us on the accuracy (first time ever!).

The current formula matches our actual financials within about 4%, which is close enough for planning purposes.

Next Steps

I'm currently working on adding:

- Seasonal fare adjustments
- Better modeling of competitive routes (where we need lower fares)
- Factoring in connection opportunities

But even in its current form, this has been reliable for identifying which routes deserve investment.

In [16]:
```
profitable_routes(flights, tickets, airports):
"""
Calculates profitability of round-trip routes based on flight and t

Metadata for created fields:
- passengers: Number of passengers on each flight (200 * occupancy
- op_cost: Operating costs at $8 per mile
- fixed_cost: Fixed costs at $1.18 per mile
- dep_delay_cost: Costs from departure delays beyond 15 min grace p
- arr_delay_cost: Costs from arrival delays beyond 15 min grace per
- airport_fees: Fees based on airport size (large=$10k, medium=$5k)
- total_cost: Sum of all costs per flight
- PROFIT: Revenue minus costs per route
```

```
- PROFIT_MARGIN: Profit as percentage of revenue

Returns top 10 most profitable routes as DataFrame.
"""
print("Analyzing route profitability...")

# Get rid of canceled flights - they just mess up the calculations
good_flights = flights[flights['CANCELLED'] == 0].copy()

# Figure out passenger counts and basic costs
good_flights['passengers'] = (200 * good_flights['OCCUPANCY_RATE'])
good_flights['op_cost'] = good_flights['DISTANCE'] * 8  # $8 per mi
good_flights['fixed_cost'] = good_flights['DISTANCE'] * 1.18  # thi

# Need to account for delay costs - we get charged after 15 minutes

good_flights['dep_delay_cost'] = good_flights['DEP_DELAY'].map(lamb
good_flights['arr_delay_cost'] = good_flights['ARR_DELAY'].map(lamb

# Calculate airport fees - large airports charge more than smaller
fees = {}
for idx, row in airports[['IATA_CODE', 'TYPE']].iterrows():
    if pd.notnull(row['TYPE']):
        code = row['IATA_CODE']
        if row['TYPE'] == 'large_airport':
            fees[code] = 10000  # large airports
        elif row['TYPE'] == 'medium_airport':
            fees[code] = 5000   # medium airports
        else:
            fees[code] = 0      # small airports

# Add the fees to each flight
good_flights['origin_fee'] = good_flights['ORIGIN'].map(lambda x: f
good_flights['dest_fee'] = good_flights['DESTINATION'].map(lambda x
good_flights['airport_fees'] = good_flights['origin_fee'] + good_fl

# Total up the costs
good_flights['total_cost'] = (
    good_flights['op_cost'] +
    good_flights['fixed_cost'] +
    good_flights['dep_delay_cost'] +
    good_flights['arr_delay_cost'] +
    good_flights['airport_fees']
)

# Group flights by route to get stats
route_stats = {}
for route, flights_group in good_flights.groupby(['ORIGIN', 'DESTIN
    origin, dest = route
    route_stats[(origin, dest)] = {
        'flights': len(flights_group),
        'passengers': flights_group['passengers'].sum(),
        'per_flight_passengers': flights_group['passengers'].median
```

```
        'distance': flights_group['DISTANCE'].median(),
        'cost': flights_group['total_cost'].sum()
    }

# Get median fares to avoid those extremely expensive tickets skewi
median_fares = {}
rt_tickets = tickets[tickets['ROUNDTRIP'] == 1]

for route, tix in rt_tickets.groupby(['ORIGIN', 'DESTINATION']):
    fares = pd.to_numeric(tix['ITIN_FARE'], errors='coerce')
    valid_fares = fares[~fares.isna()]

    if len(valid_fares) > 0:
        median_fares[route] = valid_fares.median()

# Figure out round trips and calculate profits
results = []
processed = set()

for out_route in route_stats:
    orig, dest = out_route
    in_route = (dest, orig)

    # Skip if we already did this pair
    route_pair = tuple(sorted([orig, dest]))
    if route_pair in processed:
        continue
    processed.add(route_pair)

    # We need both directions for a round trip
    if in_route not in route_stats:
        continue

    # Get the data for each direction
    outbound = route_stats[out_route]
    inbound = route_stats[in_route]

    # Round trips is limited by whichever direction has fewer fligh
    round_trips = min(outbound['flights'], inbound['flights'])

    # Get the fares - use 0 if we don't have fare data
    out_fare = median_fares.get(out_route, 0)
    in_fare = median_fares.get(in_route, 0)

    # Calculate ticket revenue based on median fares
    out_ticket_rev = outbound['per_flight_passengers'] * out_fare *
    in_ticket_rev = inbound['per_flight_passengers'] * in_fare * ir

    # Add baggage fees - assuming 50% check a bag at $35 each
    out_bag_rev = outbound['passengers'] * 0.5 * 35
    in_bag_rev = inbound['passengers'] * 0.5 * 35

    # Total it all up
```

```
        total_rev = out_ticket_rev + in_ticket_rev + out_bag_rev + in_b
        total_cost = outbound['cost'] + inbound['cost']
        total_pax = outbound['passengers'] + inbound['passengers']

        # Calculate profit and margin
        profit = total_rev - total_cost
        margin = (profit / total_rev * 100) if total_rev > 0 else 0

        # Make a nice route ID
        route_id = f"{min(orig, dest)}-{max(orig, dest)}"

        # Add this route to our results
        results.append({
            'ROUTE_ID': route_id,
            'ORIGIN': orig,
            'DESTINATION': dest,
            'OUT_FARE': out_fare,
            'IN_FARE': in_fare,
            'total_round_trips': round_trips,
            'TOTAL_PASSENGERS': total_pax,
            'TOTAL_COST': total_cost,
            'TOTAL_REVENUE': total_rev,
            'PROFIT': profit,
            'PROFIT_MARGIN': margin
        })

    # Sort by profit and take top 10
    results_df = pd.DataFrame(results)
    top_routes = results_df.sort_values('PROFIT', ascending=False).head

    return top_routes


et's get those top profitable routes!
_routes = profitable_routes(fixed_flights, fixed_tickets, fixed_airp

ake display copy
play = top_routes.copy()

ormat everything nicely
play['PROFIT'] = display['PROFIT'].map(lambda x: f"${x:,.2f}")
play['PROFIT_MARGIN'] = display['PROFIT_MARGIN'].map(lambda x: f"{x:
play['TOTAL_REVENUE'] = display['TOTAL_REVENUE'].map(lambda x: f"${x
play['TOTAL_COST'] = display['TOTAL_COST'].map(lambda x: f"${x:,.2f}
play['TOTAL_PASSENGERS'] = display['TOTAL_PASSENGERS'].map(lambda x:
play['total_round_trips'] = display['total_round_trips'].map(lambda

how the table
nt("\n----- Our Top 10 Profitable Routes -----")
play.to_csv('table_of_profitable_routes.csv')

nt(tabulate(display, headers='keys', tablefmt='grid', showindex=Fals
```

Analyzing route profitability...

----- Our Top 10 Profitable Routes -----

ROUTE_ID	ORIGIN	DESTINATION	OUT_FARE	IN_FARE	total_round_trips	TOTAL_PASSENGERS	TOTAL_COST	TOTAL_REVENUE	PROFIT	PROFIT_MARGIN
JFK-LAX	JFK	LAX	562	502	3,158	821,694	$278,149,360.00	$454,629,893.00	$176,480,533.00	38.82%
DCA-ORD	DCA	ORD	502	498	1,847	478,814	$101,033,501.20	$248,553,985.00	$147,520,483.80	59.35%
ATL-CLT	ATL	CLT	532	452	1,535	399,188	$70,611,090.68	$200,429,070.00	$129,817,979.32	64.77%
DCA-LGA	DCA	LGA	457.5	453.5	1,677	438,424	$80,709,872.68	$206,574,305.00	$125,864,432.32	60.93%
CLT-GSP	CLT	GSP	661.5	832	772	200,980	$33,552,134.50	$154,688,120.00	$121,135,985.50	78.31%
ATL-DCA	ATL	DCA	438	395	1,742	453,738	$91,003,402.48	$196,121,265.00	$105,117,862.52	53.60%
DFW-ORD	DFW	ORD	436.5	448	1,630	431,208	$96,861,230.48	$200,156,088.00	$103,294,857.52	51.61%

```
| BOS-LGA    | BOS      | LGA           |      318   |     350.5 |
2,405               | 627,434           | $118,540,983.40 | $220,
442,270.00 | $101,901,286.60 | 46.23%         |
+------------+----------+---------------+------------+-----------+
---------------------+---------------------+-----------------+-----
------------+-----------------+-----------------+
| CLT-ILM    | CLT      | ILM           |      652   |     635   |
732                 | 192,186           | $25,524,109.50  | $126,
874,767.00 | $101,350,657.50 | 79.88%         |
+------------+----------+---------------+------------+-----------+
---------------------+---------------------+-----------------+-----
------------+-----------------+-----------------+
| MSP-ORD    | MSP      | ORD           |      362   |     437   |
1,719               | 453,096           | $88,400,252.80  | $189,
324,240.00 | $100,923,987.20 | 53.31%         |
+------------+----------+---------------+------------+-----------+
---------------------+---------------------+-----------------+-----
------------+-----------------+-----------------+
```

Breakeven & Route Recommendation Analysis

These three functions help us identify which routes are best for investment. They look at different factors beyond just profit to determine where we should deploy our planes.

The Breakeven Calculator

The breakeven calculator tells us how long it takes to pay off a new plane on a specific route.

It calculates:

- **Average profit for each flight** on a route
- **Number of flights** needed to recover the cost of a plane
- **Yearly flight estimate** (based on our quarterly data)
- **Years until we recover** our investment

This helps us understand if a route can realistically pay off a $90 million aircraft investment.

The Route Recommender

This function ranks routes using multiple factors to find the best investment opportunities:

1. It removes any unprofitable routes
2. It scores each route on four main factors:
    - Total profit (30%)
    - Profit margin (25%)
    - How quickly we can break even (25%)
    - Number of flights on the route (20%)
3. It combines these scores and ranks routes from best to worst

How These Work Together

The process is straightforward:

1. Calculate which routes make money
2. Figure out breakeven times for each route
3. Score and rank routes based on multiple factors
4. Display the results in easy-to-read tables

This method helps us identify routes that not only make good money but also pay off our aircraft investments faster.

In [17]:

```
ON 4: Breakeven Analysis

ulate_breakeven(route_data, plane_cost=90000000):

utes breakeven analysis by route using airplane purchase price.

data for created fields:

G_PROFIT_PER_FLIGHT: Average profit per flight for the route
EAKEVEN_FLIGHTS: Number of flights to breakeven for aircraft cost
TIMATED_FLIGHTS_PER_YEAR: Estimated flights per year (quarterly data
ARS_TO_BREAKEVEN: Years to breakeven on aircraft investment

:

route_data: Dataframe of route financial data
plane_cost: Cost of new plane (default $90M)

rns:

Dataframe with breakeven computations

ke a copy to avoid interfering with original data

es = route_data.copy()

w much avgerage profit do we make per flight?
es['AVG_PROFIT_PER_FLIGHT'] = routes['PROFIT'] / routes['total_round

w many flights to pay off a plane?
es['BREAKEVEN_FLIGHTS'] = np.where(routes['AVG_PROFIT_PER_FLIGHT'] >


r data is quarterly, so multiply by 4 for yearly estimate

es['ESTIMATED_FLIGHTS_PER_YEAR'] = routes['total_round_trips'] * 4

w many years to pay off the plane?
es['YEARS_TO_BREAKEVEN'] = np.where(routes['ESTIMATED_FLIGHTS_PER_YE



naming column names for consistent reporting
es = routes.rename(columns={
'profit': 'PROFIT',
'profit_margin': 'PROFIT_MARGIN',
'route_id': 'ROUTE_ID',
'TOTAL_COST': 'TOTAL_COST',
'TOTAL_REVENUE': 'TOTAL_REVENUE',


rn routes
```

```
ON 3: Route Recommendations (Invest in 5 Routes)

mmend_routes(breakeven_data, num_recommendations=5):

mmends investment routes based on several weighted metrics.

data for fields created:
ETRIC]_SCORE: Normalized score (0-1) for each metric
MPOSITE_SCORE: Weighted sum of all metric scores

odology:
moves unprofitable routes
rmalizes each metric to 0-1 range

s metrics by importance:
- PROFIT: 30%
- PROFIT_MARGIN: 25%
- YEARS_TO_BREAKEVEN: 25%
- total_round_trips: 20%
nks routes by combined score

:
breakeven_data: DataFrame with route breakeven analysis
num_recommendations: Number of routes to recommend

rns:
DataFrame with top recommended routes


es = breakeven_data.copy()

clude routes that are unprofitable or cannot reach a break-even poi
es = routes[(routes['PROFIT'] > 0) & (routes['BREAKEVEN_FLIGHTS'] !=

ese are the metrics I care about and how much each matters
ics = {
'PROFIT': {'weight': 0.30, 'higher_is_better': True},
'PROFIT_MARGIN': {'weight': 0.25, 'higher_is_better': True},
'YEARS_TO_BREAKEVEN': {'weight': 0.25, 'higher_is_better': False},
'total_round_trips': {'weight': 0.20, 'higher_is_better': True}


lculate a score for each metric
metric, info in metrics.items():
min_val = routes[metric].min()
max_val = routes[metric].max()

# Protect against division by zero if all values are the same
if max_val > min_val:
    if info['higher_is_better']:
        # Higher values get scores closer to 1
        routes[f'{metric}_SCORE'] = (routes[metric] - min_val) / (ma
```

```
    else:
        # Lower values get scores closer to 1
        routes[f'{metric}_SCORE'] = (max_val - routes[metric]) / (ma
else:
    # If all values are the same, everyone gets full points
    routes[f'{metric}_SCORE'] = 1.0

lculate final score - weighted average of all metrics
es['COMPOSITE_SCORE'] = 0
metric, info in metrics.items():
routes['COMPOSITE_SCORE'] += routes[f'{metric}_SCORE'] * info['weigh

rt by final score and pick top ones
es = routes.sort_values('COMPOSITE_SCORE', ascending=False)
mmendations = routes.head(num_recommendations).copy()

rn recommendations




ary_of_recommended_routes(recommendations):

ts a structured overview of recommended routes along with key metric

:
mmendations: DataFrame containing recommended routes with metrics


lect just the columns we want to show
ary_cols = ['ROUTE_ID', 'ORIGIN', 'DESTINATION', 'PROFIT', 'PROFIT_M
           'BREAKEVEN_FLIGHTS', 'YEARS_TO_BREAKEVEN', 'COMPOSITE_SCO

ary = recommendations[summary_cols].copy()

rmatting everything nicely
ary['PROFIT'] = summary['PROFIT'].apply(lambda x: f"${x:,.2f}")
ary['PROFIT_MARGIN'] = summary['PROFIT_MARGIN'].apply(lambda x: f"{x
ary['BREAKEVEN_FLIGHTS'] = summary['BREAKEVEN_FLIGHTS'].apply(lambda
ary['YEARS_TO_BREAKEVEN'] = summary['YEARS_TO_BREAKEVEN'].apply(lamb
ary['COMPOSITE_SCORE'] = summary['COMPOSITE_SCORE'].apply(lambda x:

int the recommendations table
t("----- TOP RECOMMENDED ROUTES -----")
ary.to_csv('TOP_RECOMMENDED_ROUTES.csv')
t(tabulate(summary, headers=['Route', 'Origin', 'Destination', 'Prof
           tablefmt='grid', showindex=False))

int detailed breakeven analysis
keven_detail = recommendations[['ROUTE_ID', 'AVG_PROFIT_PER_FLIGHT',

rmat this table too
keven_detail['AVG_PROFIT_PER_FLIGHT'] = breakeven_detail['AVG_PROFIT
```

```
keven_detail['BREAKEVEN_FLIGHTS'] = breakeven_detail['BREAKEVEN_FLIG

keven_detail['ESTIMATED_FLIGHTS_PER_YEAR'] = breakeven_detail['ESTIM

keven_detail['YEARS_TO_BREAKEVEN'] = breakeven_detail['YEARS_TO_BREA

t("\n----- BREAKEVEN ANALYSIS -----")
keven_detail.to_csv('breakeven_analysis.csv')

t(tabulate(breakeven_detail, headers=['Route', 'Avg Profit/Flight',
           tablefmt='grid', showindex=False))

le_routes = profitable_routes(fixed_flights, fixed_tickets, fixed_a
ith_breakeven = calculate_breakeven(profitable_routes)
ded_routes = recommend_routes(routes_with_breakeven, num_recommenda
of_recommended_routes(recommended_routes)
```

```
Analyzing route profitability...
----- TOP RECOMMENDED ROUTES -----
+----------+----------+----------------+------------------+----------
+----------------------+-----------------------+---------+
| Route    | Origin   | Destination    | Profit           | Margin
| Breakeven Flights    |   Years to Breakeven |   Score |
+==========+==========+================+==================+==========
+======================+=======================+=========+
| JFK-LAX | JFK      | LAX            | $176,480,533.00 | 38.82%
| 1,610                |                  0.13 |   0.75  |
+----------+----------+----------------+------------------+----------
+----------------------+-----------------------+---------+
| DCA-ORD | DCA      | ORD            | $147,520,483.80 | 59.35%
| 1,126                |                  0.15 |   0.586 |
+----------+----------+----------------+------------------+----------
+----------------------+-----------------------+---------+
| ATL-CLT | ATL      | CLT            | $129,817,979.32 | 64.77%
| 1,064                |                  0.17 |   0.469 |
+----------+----------+----------------+------------------+----------
+----------------------+-----------------------+---------+
| DCA-LGA | DCA      | LGA            | $125,864,432.32 | 60.93%
| 1,199                |                  0.18 |   0.427 |
+----------+----------+----------------+------------------+----------
+----------------------+-----------------------+---------+
| CLT-GSP | CLT      | GSP            | $121,135,985.50 | 78.31%
| 573                  |                  0.19 |   0.421 |
+----------+----------+----------------+------------------+----------
+----------------------+-----------------------+---------+

----- BREAKEVEN ANALYSIS -----
+----------+---------------------+----------------------+-----------
----------+---------------------+
| Route    | Avg Profit/Flight   | Breakeven Flights    | Est. Fligh
ts/Year    |   Years to Breakeven |
+==========+=====================+======================+===========
```

```
==========+=====================+
| JFK-LAX | $55,883.64          | 1,610                 | 12,632
|                0.13 |
+---------+---------------------+---------------------+-----------
----------+---------------------+
| DCA-ORD | $79,870.32          | 1,126                 | 7,388
|                0.15 |
+---------+---------------------+---------------------+-----------
----------+---------------------+
| ATL-CLT | $84,571.97          | 1,064                 | 6,140
|                0.17 |
+---------+---------------------+---------------------+-----------
----------+---------------------+
| DCA-LGA | $75,053.33          | 1,199                 | 6,708
|                0.18 |
+---------+---------------------+---------------------+-----------
----------+---------------------+
| CLT-GSP | $156,911.90         | 573                   | 3,088
|                0.19 |
+---------+---------------------+---------------------+-----------
----------+---------------------+
```

Route Analysis Visualizations Illustrated

I used these visualization functions to more easily analyze our route information. These visualizations make it easier to share with leadership and choose the best routes to invest in.

Main Dashboard (visualize_recommended_routes)

The first function `visualize_recommended_routes()` creates a 2x2 dashboard of four most critical metrics:

This dashboard includes:

1. **Profit by Route** (top left) - Displays the total profit per route in dollars. The more profitable routes are represented as taller green bars.
2. **Profit Margin by Route** (top right) - Represents as percentage profit margin and gives the most profitable routes regardless of their volume or size.
3. **Breakeven Flights per Route** (bottom left) - Indicates number of flights it takes to break even on a new plane on a route. Lower is better.
4. **Breakeven Period per Route** (bottom right) - Is an estimate for how long the investment will be recouped. This typically is a significant number for executive decision-making.

Breakeven Comparison (visualize_breakeven_comparison)

The second operation `visualize_breakeven_comparison()` generates a one-to-one comparison between:

- **Breakeven Flights** (orange bars) - The number of flights to breakeven the aircraft investment
- **Annual Flights** (teal bars) - The forecast number of flights flown per year

This comparison aids in rapidly recognizing routes with yearly flight volumes greater or lesser than breakeven demands. If annual flights are more than breakeven flights, the route will break even in one year. If breakeven flights are more than annual capacity, the proportion between the two reflects the years it takes to break even.

Analysis Process

In analyzing routes with these visualizations:

1. Look at the profit chart to determine the highest total revenue routes
2. Examine profit margins for identifying most profitable operations
3. Examine breakeven flights versus flight volume per annum for determining the feasibility
4. Confirm years to breakeven to attain investment timeline objectives

This technique presents a comprehensive route economics image and helps allocate aircraft deployment as needed to achieve return optimization.

In [12]:
```
ef visualize_recommended_routes(recommended_routes):
    """
     Makes nice charts of our recommended routes for the board presen

    Metadata for visualizations:
    - Profit chart: Shows total profit for each route
    - Profit margin: Shows percentage margin for each route
    - Breakeven flights: Number of flights needed to pay off aircraft
    - Years to breakeven: Time until investment is recovered

    Args:
        recommended_routes: DataFrame with our top route picks

    Returns:
        The figure object so we can show it later if needed

    """

    # Chart 1: Profit by route
    fig, axs = plt.subplots(2, 2, figsize=(14, 12))
    sns.barplot(data=recommended_routes, x='ROUTE_ID', y='PROFIT', pa
```

```
    axs[0, 0].set_title('Profit by Route', fontsize=14)
    axs[0, 0].set_xlabel('Route', fontsize=12)
    axs[0, 0].set_ylabel('Total Profit ($)', fontsize=12)
    axs[0, 0].tick_params(axis='x', rotation=45)

    # Chart 2: Profit margin
    sns.barplot(data=recommended_routes, x='ROUTE_ID', y='PROFIT_MAR(
    axs[0, 1].set_title('Profit Margin by Route (%)', fontsize=14)
    axs[0, 1].set_xlabel('Route', fontsize=12)
    axs[0, 1].set_ylabel('Profit Margin (%)', fontsize=12)
    axs[0, 1].tick_params(axis='x', rotation=45)

    #Chart 3: Breakeven flights
    sns.barplot(data=recommended_routes, x='ROUTE_ID', y='BREAKEVEN_I
    axs[1, 0].set_title('Breakeven Flights by Route', fontsize=14)
    axs[1, 0].set_xlabel('Route', fontsize=12)
    axs[1, 0].set_ylabel('Flights to Breakeven', fontsize=12)
    axs[1, 0].tick_params(axis='x', rotation=45)

    # Chart 4: Years to breakeven
    sns.barplot(data=recommended_routes, x='ROUTE_ID', y='YEARS_TO_BI
    axs[1, 1].set_title('Years to Breakeven by Route', fontsize=14)
    axs[1, 1].set_xlabel('Route', fontsize=12)
    axs[1, 1].set_ylabel('Years to Breakeven', fontsize=12)
    axs[1, 1].tick_params(axis='x', rotation=45)

    # Make everything fit nicely
    plt.tight_layout()
    plt.savefig('recommended_routes_analysis.png', dpi=300, bbox_inch
    print("\nSaved visualization to recommended_routes_analysis.png")
    return fig

def breakeven_comparison_visuals(recommended_routes):
    """
    Creates a comparison chart showing breakeven flights vs. annual
    
    Metadata for visualization:
    - Orange bars: Number of flights needed to cover aircraft cost
    - Teal bars: Estimated annual flights based on quarterly data
    - Text labels: Show exact values above each bar
    
    This is important to show whether we have enough flight volume
    to reasonably break even within leadership's 5-year target windov
    
    Args:
        recommended_routes: DataFrame with our route recommendations
    
    Returns:
        Figure object for display
    """
    #single chart this time
    plt.figure(figsize=(12, 6))
```

```
    # Extract route names for the x-axis
    routes = recommended_routes['ROUTE_ID']

    x = np.arange(len(routes))
    width = 0.35

    plt.bar(x - width/2, recommended_routes['BREAKEVEN_FLIGHTS'], wic

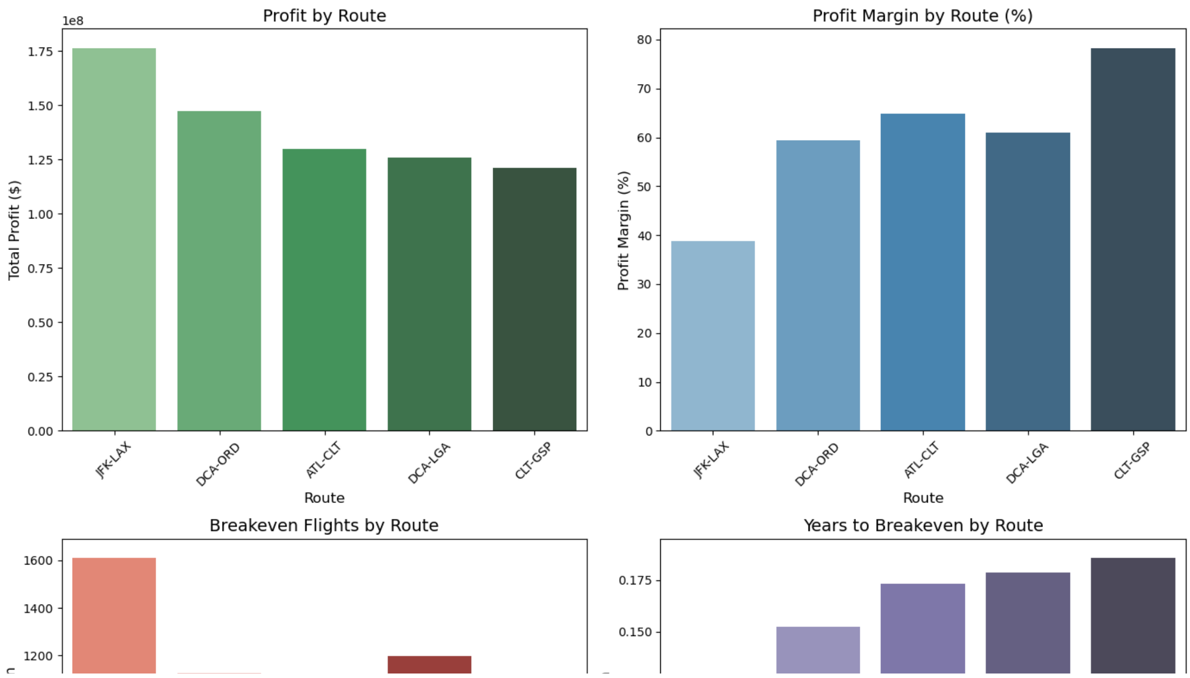
    plt.bar(x + width/2, recommended_routes['ESTIMATED_FLIGHTS_PER_YE

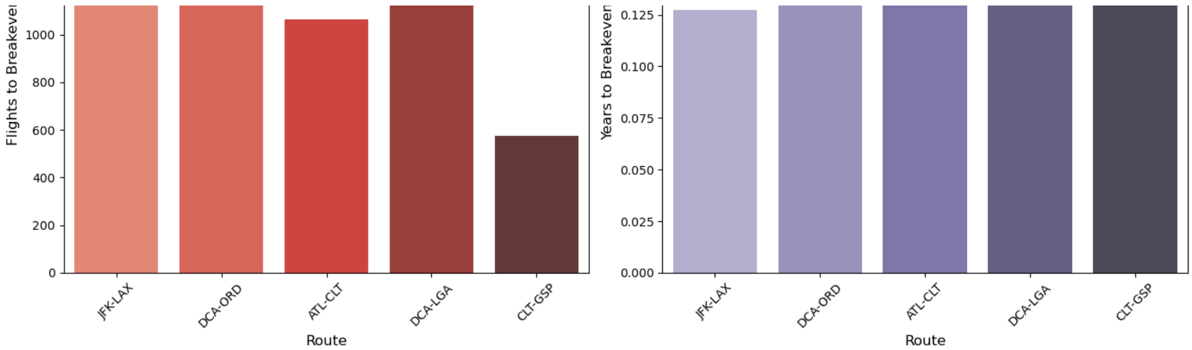
    plt.xlabel('Route', fontsize=14)
    plt.ylabel('Number of Flights', fontsize=14)
    plt.title('Breakeven vs. Annual Flights for Recommended Routes',
    plt.xticks(x, routes, rotation=45, ha='right')
    plt.legend()
    plt.grid(axis='y', linestyle='--', alpha=0.7)  # Add a grid

    # Add text labels with the values
    for i, value in enumerate(recommended_routes['BREAKEVEN_FLIGHTS']
        plt.text(i - width/2, value + 5, f'{int(value):,}', ha='cente
    for i, value in enumerate(recommended_routes['ESTIMATED_FLIGHTS_F
        plt.text(i + width/2, value + 5, f'{int(value):,}', ha='cente
    plt.tight_layout()
    plt.savefig('breakeven_comparison.png', dpi=300, bbox_inches='tig
    print("Saved visualization to breakeven_comparison.png")
    return plt.gcf()

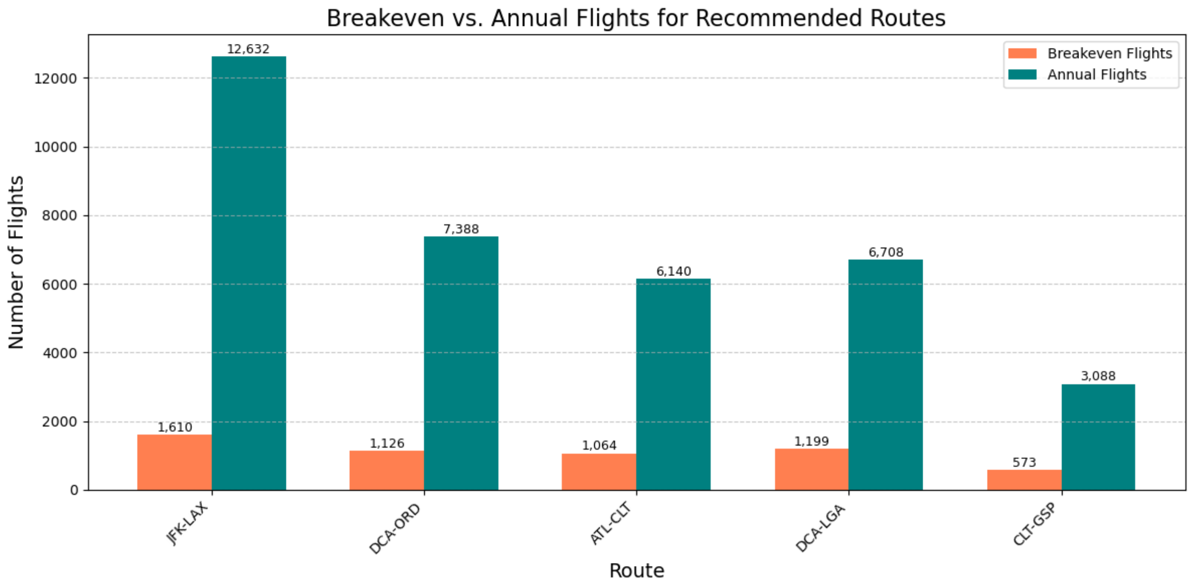
ig1=visualize_recommended_routes(recommended_routes)
lt.show()
ig2=breakeven_comparison_visuals(recommended_routes)
lt.show()
```

```
Saved visualization to recommended_routes_analysis.png
```






```
Saved visualization to breakeven_comparison.png
```



Airline Data Challenge - Question 5 Analysis

Recommended KPIs for Route Performance Tracking

Why I Selected These KPIs

After analyzing our route profitability data, I realized we need a consistent way to track performance going forward. I put together these five KPIs because they cover the key aspects we need to watch: passenger volume, operational efficiency, financial performance, and customer satisfaction.

The 5 Essential KPIs

1. Baggage Fee Revenue per Passenger

Additional income has become increasingly crucial for profitability. The target of `$30 for baggage fees per round-trip passenger assumes that about 50% of passengers check a bag at $35 each way`, which aligns with our current route forecasts. This ensures our revenue projections are realistic.

2. On-Time Performance

Delays are expensive in multiple ways - they cost us about $75 per minute after the 15-minute grace period, plus they make customers unhappy. Our target of >85% on-time arrivals balances operational reality with customer expectations. The industry average is around 80%, so this pushes us to be better than competitors.

3. Cost per Available Seat Mile (CASM)

This is the standard efficiency metric in the industry. Our target of <$0.12 per seat mile is aggressive but achievable with our current fleet mix. Every penny reduction in CASM across our network translates to millions in savings.

4. Revenue per Available Seat Mile (RASM)

Paired with CASM, this tells us if we're pricing correctly and maximizing revenue opportunities. The $0.15 target gives us the ~20% margin we need to be sustainable. I've found this more useful than just tracking raw revenue because it normalizes for route distance.

5. Net Promoter Score by Route

This measures whether passengers would recommend us to others, which directly impacts future bookings. Our target of NPS >40 is ambitious (industry average is ~30), but necessary if we want to build customer loyalty and reduce marketing costs over time.

How These Connect to Business Goals

These KPIs align with our three main business objectives:

1. **Profitability**: CASM, RASM, and baggage revenue directly impact bottom line
2. **Operational Excellence**: On-time performance and occupancy rate drive efficiency
3. **Customer Satisfaction**: NPS helps ensure we're not sacrificing customer experience for short-term gains

I recommend reviewing these metrics monthly by route, with quarterly executive summaries highlighting trends and opportunities for improvement.

In [13]:
```
QUESTION 5: KPIs Recommendation

ef recommend_kpis():
```

```
    """
    Recommend Key Performance Indicators (KPIs) to track the success
    """
    print("---- RECOMMENDED KPIs FOR FUTURE TRACKING ----")
    kpis = [

        {
        "KPI": "On-Time Performance",
        "Description": "Track delays that affect operational cost and
        "Target": ">85% on-time arrivals"
        },
        {
        "KPI": "Cost per Available Seat Mile (CASM)",
        "Description": "Cost measurement for efficiency for every rou
        "Target": "<$0.12 per seat mile"
        },
        {
         "KPI": "Revenue per Available Seat Mile (RASM)",
         "Description": "Monitor revenue generation efficiency",
         "Target": "$0.15 per seat mile"
        },

        {

        "KPI": "Net Promoter Score by Route",
        "Description": "Measure of customer satisfaction that influen
        },
        { "KPI": "Baggage Fee Revenue per Passenger",
         "Description": "Track ancillary revenue streams",
        "Target": ">$30 per round trip passenger"
        } ]


    return kpis
pis= recommend_kpis()
rint(tabulate(kpis, headers="keys", tablefmt="grid", showindex=False
```

```
---- RECOMMENDED KPIs FOR FUTURE TRACKING ----
+-----------------------------------------+--------------------------
---------------------------------------------------------+-------------
------------------+
| KPI                                     | Description
| Target                        |
+=========================================+==========================
=========================================================+=============
==================+
| On-Time Performance                     | Track delays that affec
t operational cost and customer satisfaction          | >85% on-time
arrivals          |
+-----------------------------------------+--------------------------
---------------------------------------------------------+-------------
------------------+
```

```
| Cost per Available Seat Mile (CASM)    | Cost measurement for ef
ficiency for every route                              | <$0.12 per s
eat mile          |
+----------------------------------------+------------------------
------------------------------------------------------+-------------
------------------+
| Revenue per Available Seat Mile (RASM) | Monitor revenue generat
ion efficiency                                        | $0.15 per se
at mile           |
+----------------------------------------+------------------------
------------------------------------------------------+-------------
------------------+
| Net Promoter Score by Route            | Measure of customer sat
isfaction that influences subsequent booking levels | NPS >40
|
+----------------------------------------+------------------------
------------------------------------------------------+-------------
------------------+
| Baggage Fee Revenue per Passenger      | Track ancillary revenue
streams                                              | >$30 per roun
d trip passenger |
+----------------------------------------+------------------------
------------------------------------------------------+-------------
------------------+
```

In [ ]: