Data Quality Assessment & Cleaning Justification

Authors: Ayman EL ALASS and Abderaouf KHELFAOUI

Objective: In this notebook, we explore the raw CSV datasets to identify data quality issues before populating the database. This analysis justifies the cleaning rules implemented in our final `main.py` script.

In [3]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Configuration for cleaner plots
sns.set_theme(style="whitegrid")
plt.rcParams['figure.figsize'] = (14, 6)

print("Libraries loaded.")
```
```
Libraries loaded.
```

2. Flight Data Analysis

We start by analyzing the `flights.csv` dataset. Due to its size, we load a significant chunk to perform our quality checks.

2.1 Time Format Issues

SQL standard `TIME` format expects values between `00:00` and `23:59`. We suspect the raw dataset uses `2400` to represent midnight, which causes errors during SQL ingestion.

In [4]:
```
# Load a sample of the flights data (first 200,000 rows)
df_flights = pd.read_csv("flights.csv", nrows=200000, low_memory=False)

# Check for invalid time formats (>= 2400) in DEPARTURE_TIME
invalid_times = df_flights[df_flights['DEPARTURE_TIME'] >= 2400]

print(f"Total rows analyzed: {len(df_flights)}")
print(f"Rows with time format '2400' or higher: {len(invalid_times)}")

if len(invalid_times) > 0:
    print("\nSample of invalid times:")
    display(invalid_times[['FLIGHT_NUMBER', 'DEPARTURE_TIME']].head())
```
```
Total rows analyzed: 200000
Rows with time format '2400' or higher: 23

Sample of invalid times:
```

	FLIGHT_NUMBER	DEPARTURE_TIME
30657	333	2400.0
30682	745	2400.0
42698	4629	2400.0
60538	4513	2400.0
62119	5642	2400.0

> **Justification for `main.py` :** > The analysis confirms the presence of `2400` as a time value.
> **Decision:** In our Python script, we implemented a custom function `format_time(x)` to explicitly convert `2400` to `23:59:00` (or `00:00:00`) to ensure SQL compatibility.

2.2 Handling Missing Delays

We need to determine the strategy for `NULL` values in the `DEPARTURE_DELAY` column.

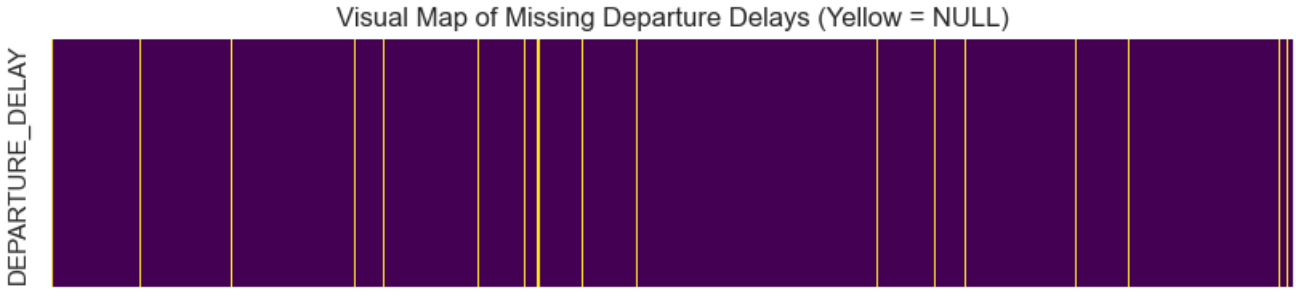
In [5]:

```
# Count missing values
missing_delays = df_flights['DEPARTURE_DELAY'].isna().sum()
total_records = len(df_flights)
percent_missing = (missing_delays / total_records) * 100

print(f"Missing Departure Delays: {missing_delays} ({percent_missing:.2f}%)")

# Visualize the missing data
plt.figure(figsize=(10, 2))
sns.heatmap(df_flights[['DEPARTURE_DELAY']].isnull().T, cbar=False, cmap='viridis', xticklabels=False)
plt.title("Visual Map of Missing Departure Delays (Yellow = NULL)")
plt.show()
```

```
Missing Departure Delays: 4868 (2.43%)
```



Justification for `main.py`: > Leaving delays as `NULL` makes aggregation queries difficult (e.g., calculating the average delay).
Decision: We apply `.fillna(0)` in the script. The business assumption is that if no delay is recorded, the flight departed on time.

2.3 Referential Integrity (Airports)

Our database schema links `FLIGHTS` to `AIRPORTS` via foreign keys. We must ensure all airports referenced in the flights dataset actually exist in our airports table.

In [6]:

```
# Load airports reference data
df_airports = pd.read_csv("airports.csv")
valid_iata_codes = set(df_airports['IATA_CODE'])

# Identify 'Orphan' flights (Origin Airport not in Airports table)
orphan_flights = df_flights[~df_flights['ORIGIN_AIRPORT'].isin(valid_iata_codes)]

print(f"Number of flights with unknown Origin Airport: {len(orphan_flights)}")

if not orphan_flights.empty:
    print(f"Unknown codes found: {orphan_flights['ORIGIN_AIRPORT'].unique()}")
```

```
Number of flights with unknown Origin Airport: 0
```

Justification for `main.py`: > Although this specific sample shows perfect referential integrity (0 orphan flights), we must guarantee this consistency across the entire dataset (millions of rows). **Decision:** We proactively apply a strict filter: `df_final = df_final[df_final['origin_airport'].isin(existing_airports)]`. This acts as a **safeguard** to prevent foreign key violations in SQLite if future data chunks contain unknown airport codes.

---

3. Weather Data Analysis

We analyze `temperature.csv` to ensure physical consistency. The raw data is in Kelvin and may contain sensor errors.

In [7]:

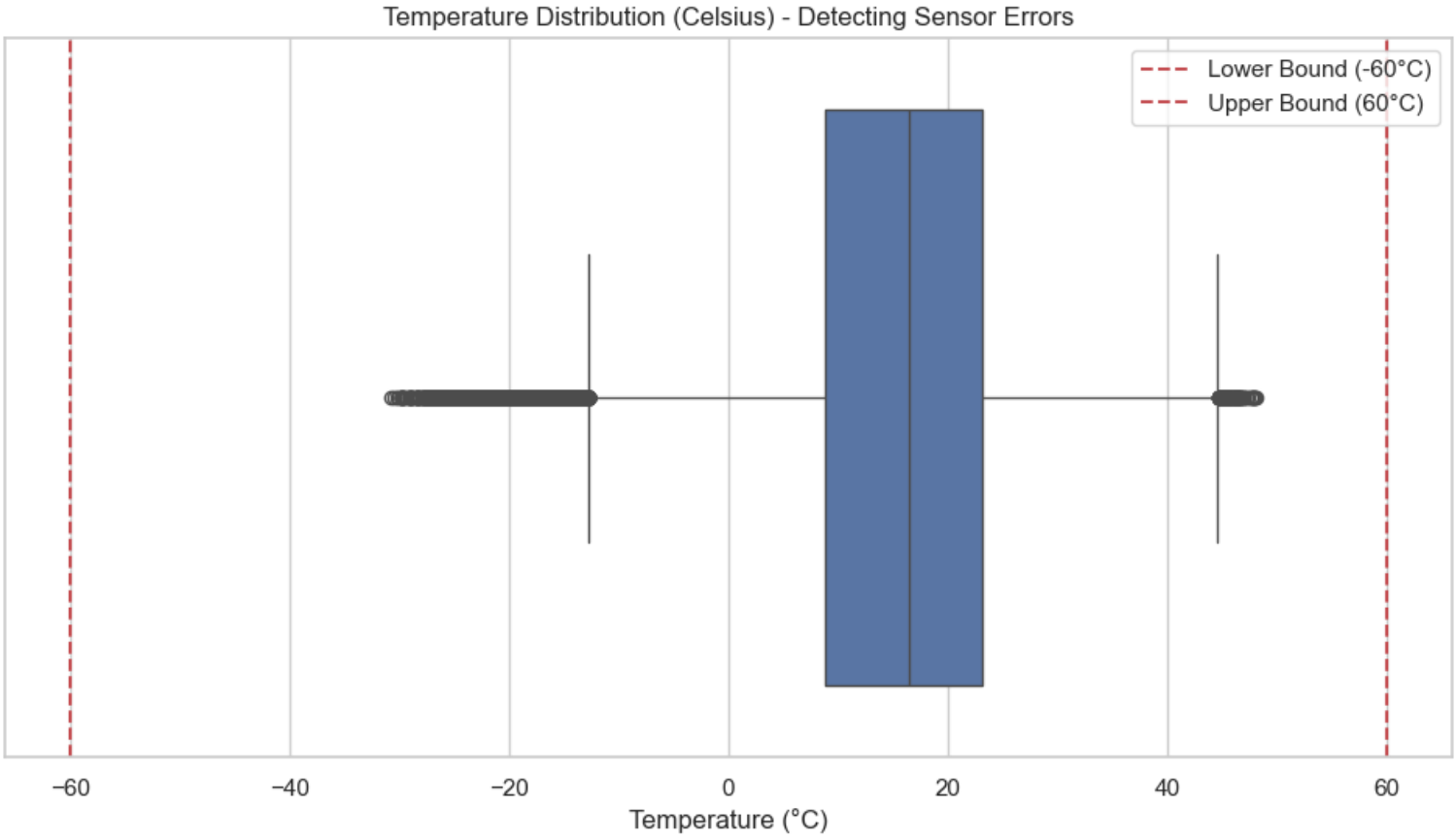
```
df_temp = pd.read_csv("temperature.csv")

# 1. Melt the dataframe to have a single 'temperature' column for analysis
df_temp_melted = pd.melt(df_temp, id_vars=['datetime'], var_name='City', value_name='temperature_k')

# 2. Convert to Celsius for easier interpretation
df_temp_melted['temperature_c'] = df_temp_melted['temperature_k'] - 273.15

# 3. Visualize distribution to spot outliers
plt.figure(figsize=(12, 6))
sns.boxplot(x=df_temp_melted['temperature_c'])
plt.title("Temperature Distribution (Celsius) - Detecting Sensor Errors")
plt.xlabel("Temperature (°C)")
plt.axvline(x=-60, color='r', linestyle='--', label='Lower Bound (-60°C)')
plt.axvline(x=60, color='r', linestyle='--', label='Upper Bound (60°C)')
plt.legend()
plt.show()

# Count extreme outliers
outliers = df_temp_melted[(df_temp_melted['temperature_c'] > 60) | (df_temp_melted['temperature_c'] < -60)]
print(f"Extreme outliers detected (outside -60 to 60 range): {len(outliers)}")
```




```
Extreme outliers detected (outside -60 to 60 range): 0
```

Justification for `main.py`: While the sample distribution looks consistent, we must anticipate sensor errors (e.g., negative wind speed, extreme temperatures) and data duplication common in weather datasets. **Decision:** To ensure the database quality, we implement a set of **defensive filtering rules**:

1. **Unit Conversion:** Convert temperatures to Celsius for readability.
2. **Physical Validity:** Enforce Temperature range $[-60, 60]$ and Wind Speed ≥ 0.
3. **Data Integrity:** Remove duplicates and ensure all weather records link to a valid Airport ID.

4. Summary of Data Cleaning Rules

Based on the analysis above, here is the comprehensive summary of rules implemented in our processing (`main.py`):

Dataset	Issue Identified	Cleaning Rule Applied
Flights	Invalid time format (`2400`)	Converted to `23:59:00`
Flights	Missing `DEPARTURE_DELAY`	Imputed with `0.0` (Assumed on time)
Flights	Unknown Airport Codes	Rows removed to satisfy Referential Integrity
Weather	Unit in Kelvin	Converted to Celsius ($K - 273.15$)
Weather	Sensor Errors (Temp)	Filtered range `[-60, 60]`
Weather	Impossible Values (Wind)	Filtered `wind_speed >= 0`
Weather	Mismatching Keys	Mapped City Names to IATA Codes manually
Weather	Duplicates	Removed based on `datetime + City`
Weather	Orphan Weather Data	Rows removed if Airport Code not in `AIRPORTS`

5. Final Implementation

In [8]:
```
import sqlite3
import os
import pandas as pd
import numpy as np

# --- Configuration ---
db_name = "project_database.db"
sql_script = "script.sql" # Ensure this file exists in your folder

# --- 5.1 Structure Creation (Tables) ---
def create_database():
    print("Creating database structure...")
    if os.path.exists(db_name):
        os.remove(db_name) # Start fresh to avoid duplicates

    conn = sqlite3.connect(db_name)
    cursor = conn.cursor()

    # Read external SQL script
    try:
        with open(sql_script, 'r') as f:
            sql_commands = f.read()
        cursor.executescript(sql_commands)
        conn.commit()
        print("Tables created successfully.")
    except FileNotFoundError:
        print(f"Error: {sql_script} not found.")
    finally:
        conn.close()

# --- 5.2 Loading and Cleaning Flights (Implementing rules defined above) ---
def populate_flights_and_others():
    conn = sqlite3.connect(db_name)
    print("Loading static data (Airlines, Airports)...")

    # 1. Airlines
```

```
    try:
        df_airlines = pd.read_csv("airlines.csv")[['IATA_CODE', 'AIRLINE']]
        df_airlines.columns = ['airline_code', 'airline_name']
        df_airlines.to_sql('AIRLINES', conn, if_exists='append', index=False)
    except Exception as e: print(f"Error Airlines: {e}")

    # 2. Airports
    try:
        df_airports = pd.read_csv("airports.csv")[['IATA_CODE','AIRPORT', 'CITY', 'STATE', 'LATITUDE', 'LONGITUDE']]
        df_airports.columns = ['iata_code','airport_name', 'city', 'state', 'latitude', 'longitude']
        df_airports.to_sql('AIRPORTS', conn, if_exists='append', index=False)
    except Exception as e: print(f"Error Airports: {e}")

    # 3. Flights (With Sampling and Cleaning)
    print("Loading flights (10% random sample)...")
    try:
        # Sampling logic to handle large CSV
        n = 5819079
        k = int(n * 0.1)
        skip_rows = sorted(set(range(1, n)) - set(np.random.choice(range(1, n), size=k, replace=False)))

        df_flights = pd.read_csv("flights.csv", low_memory=False, skiprows=skip_rows)

        # Rule: Time formatting (2400 -> 23:59)
        def format_time(x):
            if pd.isnull(x): return None
            s = str(int(x)).zfill(4)
            return '23:59:00' if s == '2400' else f"{s[:2]}:{s[2:]}:00"

        df_flights['dep_time'] = df_flights['DEPARTURE_TIME'].apply(format_time)
        df_flights['flight_date'] = pd.to_datetime(df_flights[['YEAR', 'MONTH', 'DAY']]).dt.date

        # Final Mapping
        df_final = pd.DataFrame({
            'flight_date': df_flights['flight_date'],
            'flight_number': df_flights['FLIGHT_NUMBER'],
            'dep_time': df_flights['dep_time'],
            'dep_delay': df_flights['DEPARTURE_DELAY'].fillna(0), # Rule: NULL -> 0
            'cancelled': df_flights['CANCELLED'],
            'airline_code': df_flights['AIRLINE'],
            'origin_airport': df_flights['ORIGIN_AIRPORT'],
            'dest_airport': df_flights['DESTINATION_AIRPORT']
        })

        # Rule: Referential Integrity (Remove flights with unknown airports)
        existing_airports = set(pd.read_sql("SELECT iata_code FROM AIRPORTS", conn)['iata_code'])
        df_final = df_final[df_final['origin_airport'].isin(existing_airports) & df_final['dest_airport'].isin(existing_airports)]

        df_final.to_sql('FLIGHTS', conn, if_exists='append', index=False)
        print(f"{len(df_final)} flights loaded into DB.")
    except Exception as e: print(f"Error Flights: {e}")
    conn.commit()
    conn.close()

# --- 5.3 Loading Weather Data ---
def process_weather_data():
    print("Processing weather data...")
    conn = sqlite3.connect(db_name)
    try:
        df_wind = pd.read_csv("wind_speed.csv")
        df_temp = pd.read_csv("temperature.csv")

        # Major cities mapping
        city_to_iata = {
            'New York': 'JFK', 'Los Angeles': 'LAX', 'Chicago': 'ORD',
            'Atlanta': 'ATL', 'Dallas': 'DFW', 'Denver': 'DEN',
            'San Francisco': 'SFO', 'Seattle': 'SEA', 'Miami': 'MIA',
            'Boston': 'BOS', 'Phoenix': 'PHX', 'Detroit': 'DTW',
            'Houston': 'IAH', 'Minneapolis': 'MSP', 'Philadelphia': 'PHL'
        }

        # Reshaping
        df_wind_melted = pd.melt(df_wind, id_vars=['datetime'], var_name='City', value_name='wind_speed')
```

```
        df_temp_melted = pd.melt(df_temp, id_vars=['datetime'], var_name='City', value_name='temperature_k')

        df_weather = pd.merge(df_wind_melted, df_temp_melted, on=['datetime', 'City'])

        # Filtering and Conversion
        df_weather = df_weather[df_weather['City'].isin(city_to_iata.keys())].copy().dropna()
        df_weather['airport_code'] = df_weather['City'].map(city_to_iata)

        # Rule: Kelvin to Celsius
        df_weather['temperature'] = df_weather['temperature_k'] - 273.15

        # Rule: Outlier filtering [-60, 60]
        df_weather = df_weather[(df_weather['temperature'] > -60) & (df_weather['temperature'] < 60)]
        df_weather = df_weather[df_weather['wind_speed'] >= 0]

        df_final_weather = pd.DataFrame({
            'reading_time': pd.to_datetime(df_weather['datetime']),
            'wind_speed': df_weather['wind_speed'],
            'temperature': df_weather['temperature'],
            'airport_code': df_weather['airport_code']
        })

        existing_airports = set(pd.read_sql("SELECT iata_code FROM AIRPORTS", conn)['iata_code'])
        df_final_weather = df_final_weather[df_final_weather['airport_code'].isin(existing_airports)]

        df_final_weather.to_sql('WEATHER', conn, if_exists='append', index=False)
        print(f"{len(df_final_weather)} weather records loaded.")
    except Exception as e: print(f"Error Weather: {e}")
    conn.commit()
    conn.close()

# --- Execution ---
if __name__ == "__main__":
    create_database()
    populate_flights_and_others()
    process_weather_data()
    print("Database is ready.")
```

```
Creating database structure...
---------------------------------------------------------------------------
PermissionError                           Traceback (most recent call last)
Cell In[8], line 143
    141 # --- Execution ---
    142 if __name__ == "__main__":
--> 143     create_database()
    144     populate_flights_and_others()
    145     process_weather_data()

Cell In[8], line 14, in create_database()
     12 print("Creating database structure...")
     13 if os.path.exists(db_name):
---> 14     os.remove(db_name) # Start fresh to avoid duplicates
     16 conn = sqlite3.connect(db_name)
     17 cursor = conn.cursor()

PermissionError: [WinError 32] Le processus ne peut pas accéder au fichier car ce fichier est utilisé par un autre processus: 'project_database.db'
```