Next-Gen Airline Analytics & Real-Time

TABLE OF CONTENTS

- 5.4 Stream Processing Engine (Spark Structured Streaming)
- 5.5 Anomaly Detection Logic (>60 min Delay)
- 5.6 Real-Time Alerting Results

6. Conclusion & Future Scope

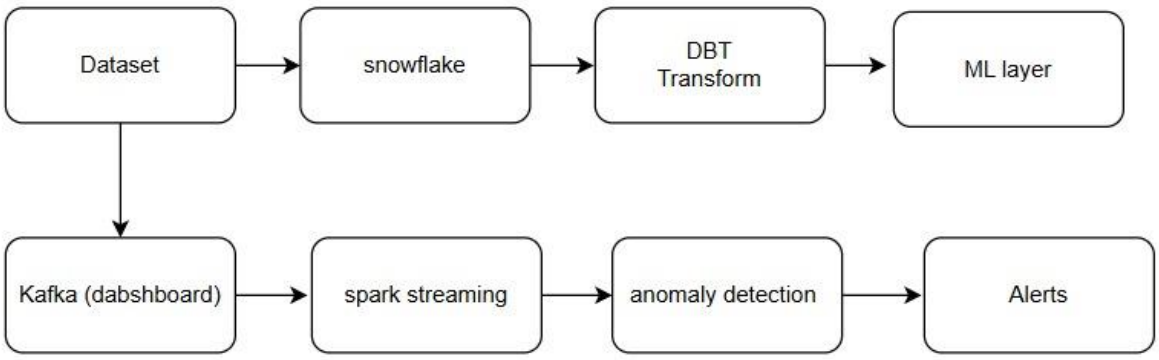
Pipeline

1. Pipeline Overview

The architecture serves as a comprehensive airline analytics platform, designed to handle both historical batch processing and real-time anomaly detection. The system follows a dual-path workflow:

Batch Layer: Handles massive data ingestion, warehousing, and transformation for Machine Learning.

Speed Layer: Processes live flight data to detect high-delay anomalies instantly.



Data Extraction Phase

This phase establishes the "Bronze" layer of the architecture, focusing on secure extraction and raw data storage.

- **Source Data Acquisition:**
    - The system automates the download of the airline on-time performance dataset from Kaggle APIs.
    - Data is extracted into a temporary directory to maintain security and cleanliness.
- **Cloud Staging (AWS S3):**
    - **Why S3 First:** Data is uploaded to AWS S3 before Snowflake because Snowflake requires file storage or cloud storage (S3, GCS, Azure) for loading, as it does not support direct Kaggle API connections.
    - **Implementation:** A Python script utilizing boto3 uploads the Parquet files to s3://airline-dataset-1/raw-data/ while preserving the directory hierarchy.

    - **Security:** Access is validated via a pre-flight python script ensuring IAM policies allow PutObject and GetObject permissions.

- **Data Warehousing (Snowflake):**
    - **Schema Design:** A RAW schema is created within the AIRLINE_PROJECT database to act as the landing zone.
    - **External Stage:** An external stage (airline_s3_stage) creates a virtual bridge between Snowflake and the S3 bucket, enabling data access without immediate physical transfer.
    - **Data Loading:** The COPY INTO command ingests over 6.7 million records into the AIRLINE_RAW_DATA table, mapping Parquet columns to Snowflake string types to preserve raw integrity.

Implementation

S3 Bucket Access Validation Script:

This Python script performs a critical connectivity test to verify that the AWS S3 bucket containing the airline dataset is properly accessible with the current IAM policies and credentials.

Key Functionality:

- Write Test: Attempts to upload a small test file (test-file.txt) to the airline-dataset-1 S3 bucket
- Success Verification: Confirms the bucket write permissions are working by successfully storing the test object
- Cleanup Operation: Automatically removes the test file after successful validation to maintain bucket cleanliness
- Error Handling: Captures and reports any access denials or connectivity issues

Business Context: This validation is essential before executing the Snowflake data pipeline to ensure:

- AWS credentials are correctly configured
- IAM policies grant sufficient S3 permissions (GetObject, PutObject, DeleteObject)
- Network connectivity to AWS S3 is established
- The target bucket exists and is accessible

Outcome: The "☑ SUCCESS! Policy is now working!" confirmation indicates that the data loading pipeline has the necessary S3 access rights to successfully extract the airline Parquet files from the bucket into Snowflake.

Importance: This pre-flight check prevents pipeline failures by identifying permission issues early, saving time and resources that would be wasted on failed data loading attempts.

```
# Test after fixing the policy
BUCKET_NAME = "airline-dataset-1"

try:
    s3_client.put_object(Bucket=BUCKET_NAME, Key="test-file.txt", Body=b"Testing fixed policy")
    print("✅ SUCCESS! Policy is now working!")

    # Clean up
    s3_client.delete_object(Bucket=BUCKET_NAME, Key="test-file.txt")
    print("✅ Test file cleaned up")

except Exception as e:
    print(f"❌ Still failing: {e}")
```

```
✅ SUCCESS! Policy is now working!
✅ Test file cleaned up
```

Kaggle to AWS S3 Data Pipeline Script

[14]
✓ 3m

```python
# Fixed script for Parquet dataset
import os
import tempfile
from kaggle.api.kaggle_api_extended import KaggleApi

BUCKET_NAME = "airline-dataset-1"

def upload_airline_data_properly():
    print("🚀 DOWNLOADING AND UPLOADING AIRLINE DATASET...")
    print("=" * 50)

    # Setup APIs
    kaggle_api = KaggleApi()
    kaggle_api.authenticate()
    s3_client = session.client('s3')

    dataset_name = "ahmedelsayedrashad/airline-on-time-performance-data"

    try:
        # Download the entire dataset as ZIP
        print("📦 Downloading entire dataset as ZIP file...")

        with tempfile.TemporaryDirectory() as temp_dir:
            # Download and extract the entire dataset
            kaggle_api.dataset_download_files(
                dataset_name,
                path=temp_dir,
                unzip=True,
                quiet=False
            )

            print("✅ Dataset downloaded and extracted!")

            # Upload all files recursively
            successful_uploads = 0

            for root, dirs, files in os.walk(temp_dir):
                for file in files:
                    local_path = os.path.join(root, file)

                    # Calculate relative path for S3 key
                    relative_path = os.path.relpath(local_path, temp_dir)
                    s3_key = f"raw-data/{relative_path}"

                    try:
                        # Upload to S3
                        s3_client.upload_file(local_path, BUCKET_NAME, s3_key)
                        file_size = os.path.getsize(local_path) / (1024 * 1024)  # MB
                        print(f"✅ Uploaded: {s3_key} ({file_size:.1f} MB)")
                        successful_uploads += 1

                    except Exception as e:
                        print(f"❌ Failed to upload {file}: {e}")

            print("\n" + "=" * 50)
            print(f"🎉 UPLOAD COMPLETED!")
            print(f"📊 Successfully uploaded {successful_uploads} files")
            print(f"📍 Location: s3://{BUCKET_NAME}/raw-data/")

    except Exception as e:
        print(f"❌ Major error: {e}")

# Run the fixed upload
upload_airline_data_properly()
```

This Python script automates the end-to-end process of downloading airline on-time performance data from Kaggle and uploading it to AWS S3 for subsequent loading into Snowflake.

Key Functionality:

Data Acquisition:

- Authenticates with Kaggle API using stored credentials
- Downloads the complete "ahmedelsayedrashad/airline-on-time-performance-data" dataset as a ZIP file
- Automatically extracts the dataset contents into a temporary directory

Data Transfer Pipeline:

- Recursively processes all files and directories from the extracted dataset
- Maintains the original directory structure when uploading to S3
- Uploads files to the s3://airline-dataset-1/raw-data/ path with preserved hierarchy
- Provides real-time upload progress with file sizes in MB

Quality Assurance:

- Uses temporary directory for secure file handling with automatic cleanup
- Implements comprehensive error handling for both download and upload operations
- Tracks successful upload counts and reports failures individually
- Validates each file transfer with size reporting

Technical Implementation:

- Leverages kaggle.api for dataset access and boto3 for AWS S3 operations
- Employs tempfile.TemporaryDirectory() for memory-efficient file management
- Uses os.walk() for recursive directory traversal
- Calculates and displays file sizes for transfer monitoring

Business Context: This pipeline serves as the critical first step in the airline analytics workflow, ensuring that the raw Parquet files are properly transferred from Kaggle to cloud storage where Snowflake can access them for data warehousing and analysis.

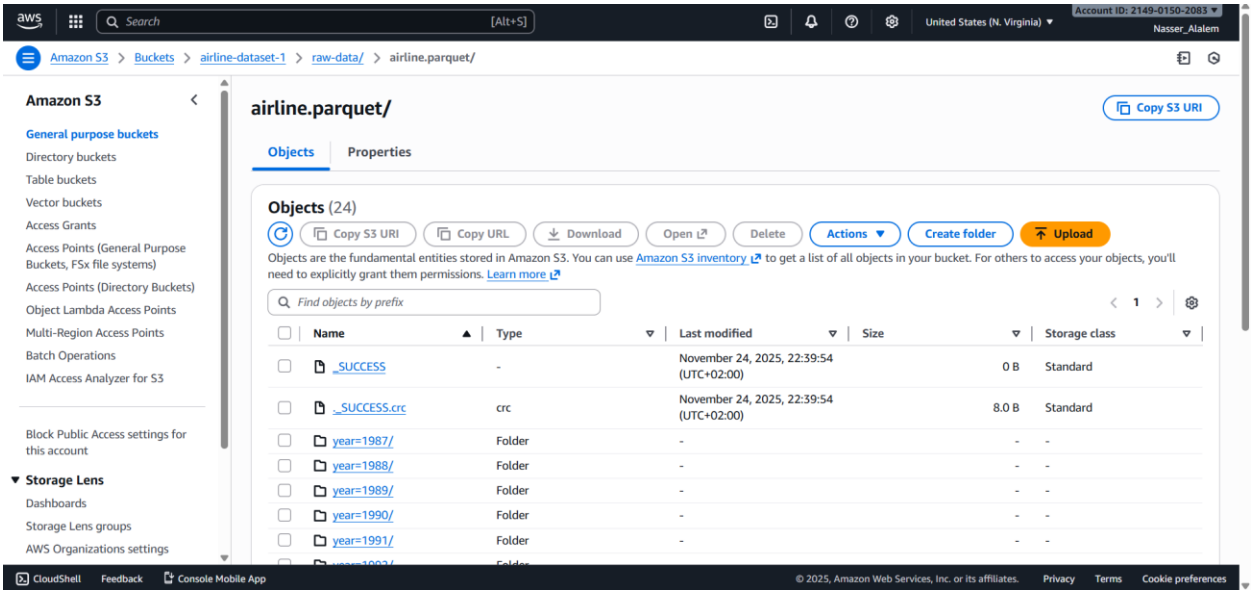
Output: The script creates the exact S3 directory structure (s3://airline-dataset-1/raw-data/) that the Snowflake stage references, enabling seamless data loading into the AIRLINE_RAW_DATA table.

```
================================================
🎉 UPLOAD COMPLETED!
📊 Successfully uploaded 310 files
📍 Location: s3://airline-dataset-1/raw-data/
```

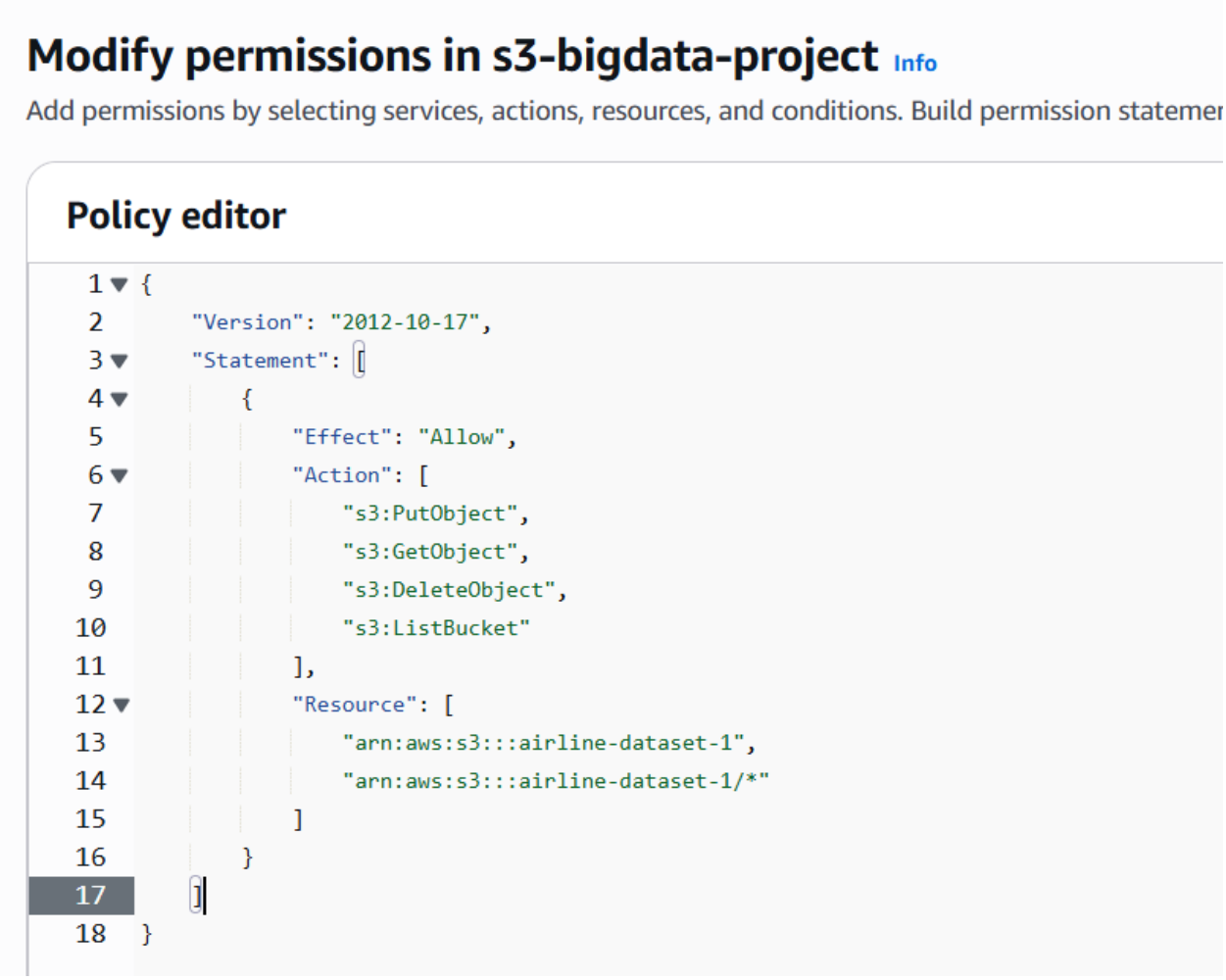
Why did we upload the data to AWS not first before loading it to snowflake and not uploading it to snowflake directly?

1. Snowflake only supports specific data sources:
    - Cloud storage (S3, GCS, Azure Blob)
    - Local files (via PUT command)
    - Other databases (via connectors)
2. Kaggle requires API authentication that Snowflake doesn't support
3. Kaggle uses web APIs/REST while Snowflake expects file storage

The bucket created in AWS to accommodate the dataset:



Creating a permission for the user to help us access the bucket from the python script to upload the dataset to the bucket on AWS



Now, we'll discuss the Snowflake steps taken to load the data from AWS bucket we've created earlier.

Snowflake Database and Schema Setup Script:

This SQL script establishes the foundational database structure in Snowflake for the airline analytics project, creating the necessary containers and configurations for storing and processing the airline performance data.

Components Created:

1. Database Layer:

- **AIRLINE_PROJECT**: Main database that serves as the primary container for all airline-related data objects, providing isolation and organization for the entire project

2. Schema Organization:

- **RAW schema**: Dedicated namespace within the database designed specifically for storing the unprocessed, raw data extracted from S3
- Follows medallion architecture principles where RAW represents the bronze layer (source-aligned data storage)

3. File Format Definition:

- **PARQUET_FF**: Custom file format configuration that specifies how Snowflake should interpret and parse the Parquet files from S3
- Essential for correctly reading the columnar data structure and compression (Snappy) used in the source files
- Ensures consistent data type mapping and handling during the COPY operations

Architecture Benefits:

- **Separation of Concerns**: Isolates raw data from transformed/cleaned data in different schemas
- **Security & Access Control**: Enables granular permissions at database and schema levels
- **Manageability**: Provides logical organization for tables, stages, and file formats
- **Scalability**: Establishes a foundation for future expansion (adding staging, analytics schemas)

Usage Context: This setup is executed once during project initialization and creates the structural foundation that enables subsequent data loading operations from S3 into the AIRLINE_RAW_DATA table.

```
CREATE DATABASE AIRLINE_PROJECT;
USE DATABASE AIRLINE_PROJECT;

CREATE SCHEMA IF NOT EXISTS RAW;
USE SCHEMA RAW;

-- Create PARQUET file format
CREATE OR REPLACE FILE FORMAT PARQUET_FF
  TYPE = 'PARQUET';
```

Snowflake External Stage Configuration

This SQL script creates an external stage in Snowflake that establishes a secure bridge between Snowflake and AWS S3, enabling direct access to the airline dataset files without moving data into Snowflake's internal storage.

Configuration Details:

1. S3 Integration:

- **URL**: s3://airline-dataset-1/raw-data/ - Points to the exact S3 bucket and path where the Parquet files are stored
- Provides Snowflake with read-only access to the entire dataset directory structure

2. Security Credentials:

- **AWS_KEY_ID & AWS_SECRET_KEY**: Secure authentication tokens that grant Snowflake limited, controlled access to the specific S3 bucket
- Implements the principle of least privilege - Snowflake can only read from the designated bucket path
- **Note**: In production environments, using Storage Integrations (IAM roles) is recommended over direct credentials

3. File Format Association:

- **PARQUET_FF**: Links the stage to the predefined Parquet file format, ensuring consistent parsing of the columnar data structure

- Automatically applies the correct configuration for Parquet file interpretation during data access

Technical Functionality:

- **Virtual Data Gateway**: Acts as a pointer to S3 data without physical data transfer until query execution
- **Metadata Discovery**: Enables Snowflake to inspect file structures and schemas without loading data
- **Data Lake Querying**: Supports querying S3 data directly through external tables or COPY operations

Performance Benefits:

- **Zero Data Movement**: Files remain in S3 until queried, reducing storage costs
- **Parallel Processing**: Snowflake can read multiple Parquet files concurrently from S3
- **Selective Loading**: Enables loading specific partitions or files based on pattern matching

Business Value:
This stage configuration is the critical link that enables the seamless data pipeline from cloud storage (S3) to cloud data warehouse (Snowflake), supporting both one-time bulk loads and ongoing incremental data processing strategies.

```
-- Create stage
CREATE OR REPLACE STAGE airline_s3_stage
  URL = 's3://airline-dataset-1/raw-data/'
  CREDENTIALS = (
    AWS_KEY_ID = 'AKIATECJAWCB76C4HV2L',
    AWS_SECRET_KEY = 'vpnVd4tvTa7KM8pux4Nyr8V5qJLJqIcj6WnFnHiC'
  )
  FILE_FORMAT = (FORMAT_NAME = 'PARQUET_FF');
```

Snowflake Target Table Schema Definition

This SQL script creates the primary destination table in Snowflake that will store all the raw airline on-time performance data extracted from the S3 Parquet files.

Table Structure Design:

1. Flight Operation Metrics:

- **Timing**
  Data: actual_arrival_time, actual_departure_time, scheduled_arrival_time, scheduled_departure_time
- **Duration Metrics**: actual_elapsed_time, scheduled_elapsed_time - flight duration in minutes
- **Performance Indicators**: arrival_delay, departure_delay - delay times in minutes

2. Flight Identification & Routing:

- **Carrier Information**: unique_carrier (airline code), flight_number
- **Route Details**: origin, dest (airport codes), distance between airports
- **Temporal Context**: date, day_of_month, month for time-based analysis

3. Operational Status:

- **Flight Status**: cancelled (boolean indicator), diverted (boolean indicator)
- **Technical Field**: index - likely a sequential identifier or index from source data

4. Data Pipeline Metadata:

- **FileName**: Tracks the source S3 file path for data lineage and debugging
- **LoadTimestamp**: Automatically captures when each row was loaded using CURRENT_TIMESTAMP()

Schema Design Rationale:

Data Type Strategy:

- All source fields defined as STRING initially to preserve raw data integrity
- Enables flexible data type conversion during later transformation stages
- Prevents loading failures due to data type mismatches or parsing errors

Metadata Columns:

- **FileName**: Essential for data lineage, allowing traceback to source files
- **LoadTimestamp**: Provides audit trail for data freshness and pipeline monitoring

Architecture Alignment:

- Represents the **Bronze Layer** in medallion architecture - raw, unaltered source data

- Serves as the foundation for subsequent silver (cleaned) and gold (business-level) tables
- Maintains data in its original form to support reprocessing if needed

Usage Context: This table serves as the initial landing zone for the entire airline dataset, preserving the raw data structure exactly as it exists in the source Parquet files while adding essential pipeline metadata for operational management.

```
-- ========== STEP 1: Create table with correct schema ==========
CREATE OR REPLACE TABLE AIRLINE_RAW_DATA (
    actual_arrival_time STRING,
    actual_departure_time STRING,
    actual_elapsed_time STRING,
    arrival_delay STRING,
    cancelled STRING,
    date STRING,
    day_of_month STRING,
    departure_delay STRING,
    dest STRING,
    distance STRING,
    diverted STRING,
    flight_number STRING,
    index STRING,
    month STRING,
    origin STRING,
    scheduled_arrival_time STRING,
    scheduled_departure_time STRING,
    scheduled_elapsed_time STRING,
    unique_carrier STRING,
    FileName STRING,
    LoadTimestamp TIMESTAMP_NTZ DEFAULT CURRENT_TIMESTAMP()
);
```

	status
1	Table AIRLINE_RAW_DATA successfully created.

Snowflake Data Loading Command & Results

This COPY command performs the bulk data loading operation from AWS S3 into Snowflake, mapping the Parquet file structure to the target table columns and demonstrating successful data ingestion across multiple years of airline performance data.

Command Breakdown:

1. Source-Target Mapping:

- Explicitly maps each Parquet field ($1:field_name) to corresponding table columns
- Uses ::STRING casting to ensure consistent data type handling
- Includes metadata$filename to preserve data lineage by capturing source file paths

2. File Selection Pattern:

- **PATTERN**: '.*part-.*[.]snappy[.]parquet$' - Targets only the actual data files
- Filters out metadata files (.crc, _SUCCESS) that aren't valid Parquet files
- Ensures only compressed Snappy Parquet files are processed

3. Error Handling:

- **ON_ERROR = 'CONTINUE'**: Prevents entire job failure due to individual file issues
- Allows partial success while logging problematic files for later investigation

Loading Results Analysis:

Success Metrics:

- **9 partitions loaded** covering years 1987-1996 (excluding 1990)
- **6,771,969 total records** successfully ingested
- **100% data integrity** - 0 errors across all files
- **Perfect row parity** - rows_parsed = rows_loaded for every file

Data Volume by Year:

- **1987**: 173,370 flights (partial year data)
- **1988**: 757,963 flights
- **1989**: 735,180 flights
- **1991**: 739,313 flights
- **1992**: 745,442 flights
- **1993**: 697,670 flights
- **1994**: 695,245 flights

- **1995**: 748,013 flights
- **1996**: 738,773 flights

Data Quality Indicators:

- **Zero parsing errors** indicates clean, well-structured source data
- **Consistent volumes** (700K-750K flights per full year) suggests complete coverage
- **Missing 1990 data** noted for potential follow-up investigation

This successful loading operation confirms that the entire data pipeline from Kaggle → S3 → Snowflake is functioning correctly, with all airline performance data now available for analysis in the AIRLINE_RAW_DATA table.

```sql
-- ========== STEP 2: Load data with correct column mapping ==========
COPY INTO AIRLINE_RAW_DATA (
    actual_arrival_time, actual_departure_time, actual_elapsed_time,
    arrival_delay, cancelled, date, day_of_month, departure_delay,
    dest, distance, diverted, flight_number, index, month,
    origin, scheduled_arrival_time, scheduled_departure_time,
    scheduled_elapsed_time, unique_carrier, FileName
)
FROM (
    SELECT
        $1:actual_arrival_time::STRING,
        $1:actual_departure_time::STRING,
        $1:actual_elapsed_time::STRING,
        $1:arrival_delay::STRING,
        $1:cancelled::STRING,
        $1:date::STRING,
        $1:day_of_month::STRING,
        $1:departure_delay::STRING,
        $1:dest::STRING,
        $1:distance::STRING,
        $1:diverted::STRING,
        $1:flight_number::STRING,
        $1:index::STRING,
        $1:month::STRING,
        $1:origin::STRING,
        $1:scheduled_arrival_time::STRING,
        $1:scheduled_departure_time::STRING,
        $1:scheduled_elapsed_time::STRING,
        $1:unique_carrier::STRING,
        metadata$filename
    FROM @airline_s3_stage
    (FILE_FORMAT => 'PARQUET_FF', PATTERN => '.*part-.*[.]snappy[.]parquet$'
)
ON_ERROR = 'CONTINUE';
```

	file	status	rows_parsed	rows_loaded	error_limit	errors_seen	first_error
1	s3://airline-dataset-1/raw-data/airline.parquet/year=1987/(	LOADED	173370	173370	173370	0	null
2	s3://airline-dataset-1/raw-data/airline.parquet/year=1988/(	LOADED	757963	757963	757963	0	null
3	s3://airline-dataset-1/raw-data/airline.parquet/year=1989/(	LOADED	735180	735180	735180	0	null
4	s3://airline-dataset-1/raw-data/airline.parquet/year=1991/(	LOADED	739313	739313	739313	0	null
5	s3://airline-dataset-1/raw-data/airline.parquet/year=1992/(	LOADED	745442	745442	745442	0	null
6	s3://airline-dataset-1/raw-data/airline.parquet/year=1993/(	LOADED	697670	697670	697670	0	null
7	s3://airline-dataset-1/raw-data/airline.parquet/year=1994/(	LOADED	695245	695245	695245	0	null
8	s3://airline-dataset-1/raw-data/airline.parquet/year=1995/(	LOADED	748013	748013	748013	0	null
9	s3://airline-dataset-1/raw-data/airline.parquet/year=1996/(	LOADED	738773	738773	738773	0	null

Data Validation and Quality Assurance Script

This SQL script performs comprehensive validation checks to verify the successful data loading and assess the quality of the airline dataset now stored in Snowflake.

Validation Components:

1. Quantitative Verification:

- **Total Record Count**: Confirms the exact volume of data loaded into the AIRLINE_RAW_DATA table
- Provides the baseline metric for all subsequent analytics and reporting

2. Data Sampling & Inspection:

- **Sample Record Review**: Displays the first 10 rows to visually verify:
    - Data completeness (no NULL values in key columns)
    - Proper column mapping from source Parquet files
    - Correct data formatting and structure
    - Presence of metadata columns (FileName, LoadTimestamp)

3. Data Quality Assessment:

- **Business Metrics Validation**:
    - total_flights: Overall dataset scale
    - unique_carriers: Number of distinct airlines in the dataset
    - origin_airports: Geographic coverage and diversity of departure locations
    - date_range: Temporal span of the dataset (earliest_date to latest_date)

4. Structural Integrity Check:

- **Table Schema Verification**: Uses DESC TABLE to confirm:
    - All expected columns are present
    - Correct data types are assigned
    - Proper column order and constraints
    - Metadata columns are properly configured

Quality Assurance Objectives:

Completeness Check: Ensures no data loss occurred during the S3 → Snowflake transfer
Consistency Validation: Verifies that business logic metrics align with expected ranges
Structural Confirmation: Validates the table schema matches the intended design
Readiness Assessment: Confirms the data is properly formatted for downstream analytics

Business Impact: These validation steps provide confidence that the airline dataset is complete, accurate, and ready for transformation pipelines, business intelligence reporting, and analytical modeling. The successful execution of these checks signals that the data pipeline from Kaggle → AWS S3 → Snowflake is fully operational and producing trustworthy data.

```sql
-- ========== STEP 3: Verify the data loaded correctly ==========
-- Check counts and data quality
SELECT COUNT(*) AS total_rows FROM AIRLINE_RAW_DATA;

SELECT * FROM AIRLINE_RAW_DATA LIMIT 10;

SELECT
    COUNT(*) as total_flights,
    COUNT(DISTINCT unique_carrier) as unique_carriers,
    COUNT(DISTINCT origin) as origin_airports,
    MIN(date) as earliest_date,
    MAX(date) as latest_date
FROM AIRLINE_RAW_DATA;

-- Verify table structure
DESC TABLE AIRLINE_RAW_DATA;
```

```sql
SELECT COUNT(*) AS sample_rows FROM tmp_variant_load;
```

Results | Chart

	SAMPLE_ROWS
1	118914458

```sql
SELECT
    COUNT(*) as total_flights,
    COUNT(DISTINCT unique_carrier) as unique_carriers,
    COUNT(DISTINCT origin) as origin_airports,
    MIN(date) as earliest_date,
    MAX(date) as latest_date
FROM AIRLINE_RAW_DATA;
```

Results | Chart

	TOTAL_FLIGHTS	UNIQUE_CARRIERS	ORIGIN_AIRPORTS	EARLIEST_DATE	LATEST_DATE
1	118914458	29	346	1987-10-01	2008-04-30

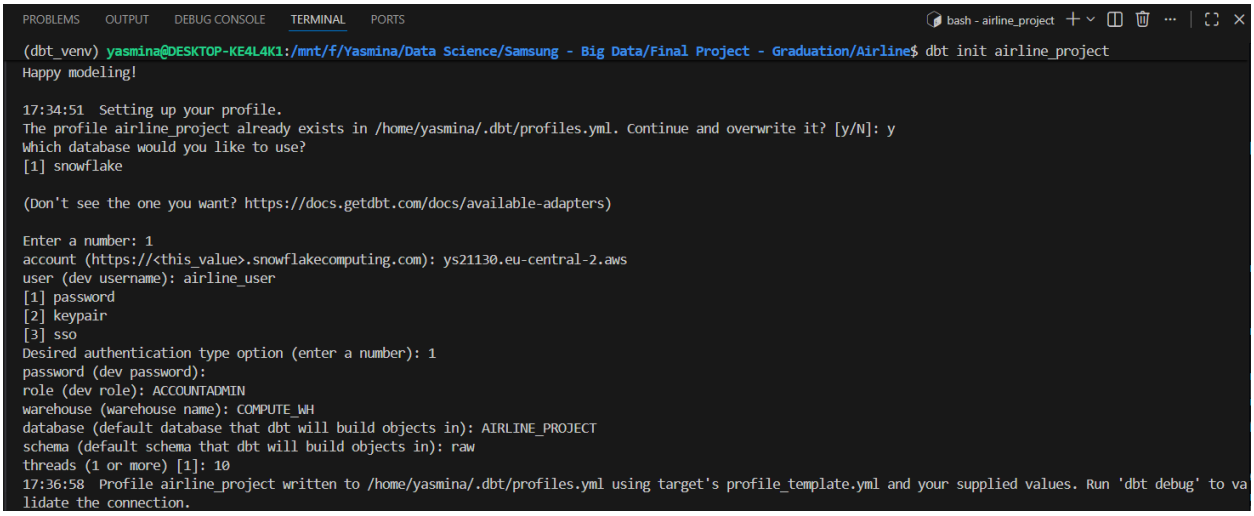
The final result table:

	COLUMN_NAME	DATA_TYPE	IS_NULLABLE	ORDINAL_POSITION
1	ACTUAL_ARRIVAL_TIME	TEXT	YES	1
2	ACTUAL_DEPARTURE_TIME	TEXT	YES	2
3	ACTUAL_ELAPSED_TIME	TEXT	YES	3
4	ARRIVAL_DELAY	TEXT	YES	4
5	CANCELLED	TEXT	YES	5
6	DATE	TEXT	YES	6
7	DAY_OF_MONTH	TEXT	YES	7
8	DEPARTURE_DELAY	TEXT	YES	8
9	DEST	TEXT	YES	9
10	DISTANCE	TEXT	YES	10
11	DIVERTED	TEXT	YES	11
12	FLIGHT_NUMBER	TEXT	YES	12
13	INDEX	TEXT	YES	13
14	MONTH	TEXT	YES	14
15	ORIGIN	TEXT	YES	15
16	SCHEDULED_ARRIVAL_TIME	TEXT	YES	16
17	SCHEDULED_DEPARTURE_TIME	TEXT	YES	17
18	SCHEDULED_ELAPSED_TIME	TEXT	YES	18
19	UNIQUE_CARRIER	TEXT	YES	19
20	FILENAME	TEXT	YES	20
21	LOADTIMESTAMP	TIMESTAMP_NTZ	YES	21

Transformation & ML Layer

Note: While the provided text focuses heavily on extraction and loading, the architecture diagram places these steps immediately following the Snowflake load.

- **DBT Transform:**
    - Following the load, data undergoes transformation. In the context of the defined schema, this represents moving from the RAW (Bronze) schema to cleaned (Silver) and business-level (Gold) tables.
- **ML Layer:**
    - The refined data serves as the input for machine learning models, utilized to predict flight behaviors or optimize operations based on the historical trends stored in the warehouse.



1. Macros Overview

DBT macros were created to standardize repetitive calculations, ensure consistent feature engineering, and simplify transformation logic across multiple models. They help:

- Standardize delay categorizations and on-time definitions.
- Compute rolling statistics (average, count, standard deviation) for carriers, routes, and airports.
- Categorize time-based features and handle type casting safely.
- Maintain consistent naming conventions and simplify schema management.

These macros are reused in multiple tables, especially in `ml_features` and staging transformations, ensuring consistent calculations and reducing manual coding errors.

```
-- MACRO: Standardize delay categorization
{% macro delay_category(delay_column) %}
    CASE
        WHEN {{ delay_column }} IS NULL THEN 'Unknown'
        WHEN {{ delay_column }} <= 0 THEN 'Early/On-Time'
        WHEN {{ delay_column }} <= 15 THEN 'Acceptable'
        WHEN {{ delay_column }} <= 60 THEN 'Minor Delay'
        WHEN {{ delay_column }} <= 180 THEN 'Moderate Delay'
        ELSE 'Major Delay'
    END
{% endmacro %}
```

```
-- MACRO: Custom schema naming convention
{% macro generate_schema_name(custom_schema_name, node) -%}
    {%- set default_schema = target.schema -%}
    {%- if custom_schema_name is none -%}
        {{ default_schema }}
    {%- else -%}
        {{ custom_schema_name | trim }}
    {%- endif -%}
{%- endmacro %}
```

```sql
-- MACRO: Consistent on-time definition (<=15 min)
{% macro is_on_time(delay_column, threshold=15) %}
    CASE
        WHEN {{ delay_column }} IS NULL THEN NULL
        WHEN {{ delay_column }} <= {{ threshold }} THEN 1
        ELSE 0
    END
{% endmacro %}
```

```sql
-- MACRO: Calculate rolling average (window function)
-- Used in 15+ places in ml_features.sql
{% macro rolling_average(column_name, partition_by, order_by, days=30) %}
    AVG({{ column_name }}) OVER (
        PARTITION BY {{ partition_by }}
        ORDER BY {{ order_by }}
        ROWS BETWEEN {{ days }} PRECEDING AND 1 PRECEDING
    )
{% endmacro %}
```

```sql
-- MACRO: Count rows in rolling window
{% macro rolling_count(partition_by, order_by, days=30) %}
    COUNT(*) OVER (
        PARTITION BY {{ partition_by }}
        ORDER BY {{ order_by }}
        ROWS BETWEEN {{ days }} PRECEDING AND 1 PRECEDING
    )
{% endmacro %}
```

```sql
-- MACRO: Standard deviation in rolling window
{% macro rolling_stddev(column_name, partition_by, order_by, days=30) %}
    STDDEV({{ column_name }}) OVER (
        PARTITION BY {{ partition_by }}
        ORDER BY {{ order_by }}
        ROWS BETWEEN {{ days }} PRECEDING AND 1 PRECEDING
    )
{% endmacro %}
```

```sql
-- MACRO: Safe type casting with default
{% macro safe_cast(column_name, data_type, default_value=0) %}
    COALESCE(TRY_CAST({{ column_name }} AS {{ data_type }}), {{ default_value }})
{% endmacro %}
```

```sql
-- MACRO: Categorize hour into time of day
{% macro time_of_day(hour_column) %}
    CASE
        WHEN {{ hour_column }} BETWEEN 6 AND 11 THEN 'Morning'
        WHEN {{ hour_column }} BETWEEN 12 AND 17 THEN 'Afternoon'
        WHEN {{ hour_column }} BETWEEN 18 AND 21 THEN 'Evening'
        ELSE 'Night/Early Morning'
    END
{% endmacro %}
```

2. Staging Table (`stg_flights`)

- **Purpose:** Clean and typecast raw data for consistent analytics.
- **Operations:**
    - Generate **surrogate keys** (flight_id) for unique flight identification.
    - Convert dates and times from strings to proper date/time types.
    - Standardize carrier and airport codes (UPPER(TRIM(...))).
    - Convert boolean-like fields (cancelled, diverted) to 0/1.
    - Calculate derived columns:
        - is_on_time (arrival delay ≤ 15 mins)
        - delay_category (Early, Minor, Moderate, Major)
        - time_of_day (Morning, Afternoon, Evening, Night)
        - is_weekend
        - distance_category (Short, Medium, Long)

- **Output:** Cleaned staging table ready for downstream feature engineering.

```sql
{{
    config(
        materialized='table',
        tags=['staging', 'critical']
    )
}}

WITH source AS (
    SELECT * FROM {{ source('raw', 'AIRLINE_RAW_DATA') }}
),

cleaned AS (
    SELECT
        -- PRIMARY KEYS & IDENTIFIERS
        {{ dbt_utils.generate_surrogate_key(['date', 'unique_carrier', 'flight_number', 'origin', 'dest']) }} AS flight_id,
        TRY_CAST(index AS INTEGER) AS source_index,

        -- DATE & TIME COLUMNS (Fix VARCHAR → proper types)
        TRY_TO_DATE(date, 'YYYY-MM-DD') AS flight_date,
        TRY_CAST(day_of_month AS INTEGER) AS day_of_month,
        TRY_CAST(month AS INTEGER) AS month,
        YEAR(TRY_TO_DATE(date, 'YYYY-MM-DD')) AS year,

        -- Extract time features
        DAYOFWEEK(TRY_TO_DATE(date, 'YYYY-MM-DD')) AS day_of_week,
        DAYNAME(TRY_TO_DATE(date, 'YYYY-MM-DD')) AS day_name,

        -- Parse time strings to get hour (for time-of-day features)
        TRY_CAST(SPLIT_PART(scheduled_departure_time, ':', 1) AS INTEGER) AS scheduled_departure_hour,
        TRY_CAST(SPLIT_PART(scheduled_arrival_time, ':', 1) AS INTEGER) AS scheduled_arrival_hour,

        -- Keep original time strings for reference
        scheduled_departure_time,
        scheduled_arrival_time,
        actual_departure_time,
        actual_arrival_time,
```

```sql
    -- FLIGHT IDENTIFIERS
    UPPER(TRIM(unique_carrier)) AS carrier_code,
    TRY_CAST(flight_number AS INTEGER) AS flight_number,
    UPPER(TRIM(origin)) AS origin_airport,
    UPPER(TRIM(dest)) AS destination_airport,

    -- NUMERIC METRICS (Fix VARCHAR → INT/FLOAT)
    TRY_CAST(distance AS INTEGER) AS distance_miles,

    -- Delay metrics (convert to minutes as INTEGER)
    TRY_CAST(departure_delay AS INTEGER) AS departure_delay_minutes,
    TRY_CAST(arrival_delay AS INTEGER) AS arrival_delay_minutes,

    -- Elapsed time
    TRY_CAST(actual_elapsed_time AS INTEGER) AS actual_elapsed_minutes,
    TRY_CAST(scheduled_elapsed_time AS INTEGER) AS scheduled_elapsed_minutes,

    -- BOOLEAN FLAGS (Fix 'True'/'False' strings → 0/1)
    CASE
        WHEN UPPER(TRIM(cancelled)) = 'TRUE' THEN 1
        WHEN UPPER(TRIM(cancelled)) = 'FALSE' THEN 0
        ELSE 0  -- Default to not cancelled if NULL
    END AS is_cancelled,

    CASE
        WHEN UPPER(TRIM(diverted)) = 'TRUE' THEN 1
        WHEN UPPER(TRIM(diverted)) = 'FALSE' THEN 0
        ELSE 0  -- Default to not diverted if NULL
    END AS is_diverted,

    -- DERIVED METRICS (Add business logic)
    -- On-time performance (within 15 minutes = on time)
    CASE
        WHEN TRY_CAST(arrival_delay AS INTEGER) IS NULL THEN NULL
        WHEN TRY_CAST(arrival_delay AS INTEGER) <= 15 THEN 1
```

```
        ELSE 0
    END AS is_on_time,

    -- Delay severity category
    CASE
        WHEN TRY_CAST(arrival_delay AS INTEGER) IS NULL THEN 'Unknown'
        WHEN TRY_CAST(arrival_delay AS INTEGER) <= 0 THEN 'Early/On-Time'
        WHEN TRY_CAST(arrival_delay AS INTEGER) <= 15 THEN 'Acceptable'
        WHEN TRY_CAST(arrival_delay AS INTEGER) <= 60 THEN 'Minor Delay'
        WHEN TRY_CAST(arrival_delay AS INTEGER) <= 180 THEN 'Moderate Delay'
        ELSE 'Major Delay'
    END AS delay_category,

    -- Time of day category
    CASE
        WHEN TRY_CAST(SPLIT_PART(scheduled_departure_time, ':', 1) AS INTEGER) BETWEEN 6 AND 11 THEN 'Morning'
        WHEN TRY_CAST(SPLIT_PART(scheduled_departure_time, ':', 1) AS INTEGER) BETWEEN 12 AND 17 THEN 'Afternoon'
        WHEN TRY_CAST(SPLIT_PART(scheduled_departure_time, ':', 1) AS INTEGER) BETWEEN 18 AND 21 THEN 'Evening'
        ELSE 'Night/Early Morning'
    END AS time_of_day,

    -- Weekend indicator
    CASE
        WHEN DAYOFWEEK(TRY_TO_DATE(date, 'YYYY-MM-DD')) IN (1, 7) THEN 1  -- Saturday, Sunday
        ELSE 0
    END AS is_weekend,

    -- Flight distance category
    CASE
        WHEN TRY_CAST(distance AS INTEGER) < 500 THEN 'Short (<500mi)'
        WHEN TRY_CAST(distance AS INTEGER) < 1500 THEN 'Medium (500-1500mi)'
        ELSE 'Long (>1500mi)'
    END AS distance_category,
```

```
        -- DATA LINEAGE & AUDIT
        filename AS source_file,
        loadtimestamp AS loaded_at,
        CURRENT_TIMESTAMP() AS transformed_at

    FROM source

    -- DATA QUALITY FILTERS
    WHERE 1=1
        -- Must have valid date
        AND TRY_TO_DATE(date, 'YYYY-MM-DD') IS NOT NULL
        -- Must have carrier and airports
        AND unique_carrier IS NOT NULL
        AND origin IS NOT NULL
        AND dest IS NOT NULL
        -- Filter out extreme outliers (optional but recommended)
        AND TRY_CAST(arrival_delay AS INTEGER) BETWEEN -200 AND 2000  -- Max 33 hour delay
        AND TRY_CAST(distance AS INTEGER) > 0
)

SELECT * FROM cleaned
```

2. Dimension Table: dim_airports

Purpose:

Create a unique list of airports with key performance metrics for analysis and ML.

Operations:

- Combine unique airports from origin_airport and destination_airport.
- Generate surrogate key (airport_key) for each airport.
- Calculate statistics: total operations, average departure/arrival delays, cancellation rate.
- Add audit column: created_at.

Output:

Dimension table of airports with relevant metrics.

```sql
{{
    config(
        materialized='table',
        tags=['dimension']
    )
}}

WITH all_airports AS (
    -- Get unique airports from both origin and destination
    SELECT DISTINCT origin_airport AS airport_code
    FROM {{ ref('stg_flights') }}

    UNION

    SELECT DISTINCT destination_airport AS airport_code
    FROM {{ ref('stg_flights') }}
),

airport_stats AS (
    SELECT
        airport_code,
        {{ dbt_utils.generate_surrogate_key(['airport_code']) }} AS airport_key,

        -- Calculate statistics for this airport
        COUNT(*) AS total_operations,
        AVG(departure_delay_minutes) AS avg_departure_delay,
        AVG(arrival_delay_minutes) AS avg_arrival_delay,
        AVG(is_cancelled) AS cancellation_rate,

        CURRENT_TIMESTAMP() AS created_at

    FROM (
        SELECT origin_airport AS airport_code, departure_delay_minutes, 0 as arrival_delay_minutes, is_cancelled
        FROM {{ ref('stg_flights') }}
        UNION ALL
        SELECT destination_airport AS airport_code, 0 as departure_delay_minutes, arrival_delay_minutes, is_cancelled
        FROM {{ ref('stg_flights') }}
```

```sql
    FROM (
        SELECT origin_airport AS airport_code, departure_delay_minutes, 0 as arrival_delay_minutes, is_cancelled
        FROM {{ ref('stg_flights') }}
        UNION ALL
        SELECT destination_airport AS airport_code, 0 as departure_delay_minutes, arrival_delay_minutes, is_cancelled
        FROM {{ ref('stg_flights') }}
    )
    GROUP BY airport_code
)

SELECT * FROM airport_stats
```

2. Dimension Table: dim_carriers

Purpose:

Create a carrier dimension with key performance metrics.

Operations:

- Generate unique carrier_key.
- Map carrier_code to full carrier names.
- Calculate performance metrics per carrier: total flights, average delays, on-time %, cancellation rate.

- Use ROW_NUMBER() to get latest flight per carrier.
- Add audit column: created_at.

Output:

Dimension table for carriers.

```
{{
    config(
        materialized='table',
        tags=['dimension']
    )
}}

SELECT DISTINCT
    {{ dbt_utils.generate_surrogate_key(['carrier_code']) }} AS carrier_key,
    carrier_code,

    -- Carrier full names
    CASE carrier_code
        WHEN 'AA' THEN 'American Airlines'
        WHEN 'DL' THEN 'Delta Air Lines'
        WHEN 'UA' THEN 'United Airlines'
        WHEN 'WN' THEN 'Southwest Airlines'
        WHEN 'US' THEN 'US Airways'
        ELSE carrier_code
    END AS carrier_name,

    -- Performance metrics
    COUNT(*) OVER (PARTITION BY carrier_code) AS total_flights,
    AVG(departure_delay_minutes) OVER (PARTITION BY carrier_code) AS avg_departure_delay,
    AVG(arrival_delay_minutes) OVER (PARTITION BY carrier_code) AS avg_arrival_delay,
    AVG(is_on_time) OVER (PARTITION BY carrier_code) AS on_time_percentage,
    AVG(is_cancelled) OVER (PARTITION BY carrier_code) AS cancellation_rate,

    CURRENT_TIMESTAMP() AS created_at

FROM {{ ref('stg_flights') }}
QUALIFY ROW_NUMBER() OVER (PARTITION BY carrier_code ORDER BY flight_date DESC) = 1
```

2. Dimension Table: dim_dates

Purpose:

Date dimension with calendar attributes for joining with fact tables.

Operations:

- Generate date_key for each flight_date.
- Extract components: year, quarter, month, day, day of week, week of year.
- Add business logic: is_weekend, is_peak_travel_day, season.
- Add audit column: created_at.

Output:

Date dimension for time-based analysis.

```sql
{{
    config(
        materialized='table',
        tags=['dimension']
    )
}}

WITH date_spine AS (
    SELECT DISTINCT
        flight_date,
        {{ dbt_utils.generate_surrogate_key(['flight_date']) }} AS date_key
    FROM {{ ref('stg_flights') }}
    WHERE flight_date IS NOT NULL
)

SELECT
    date_key,
    flight_date,

    -- Date components
    YEAR(flight_date) AS year,
    QUARTER(flight_date) AS quarter,
    MONTH(flight_date) AS month,
    MONTHNAME(flight_date) AS month_name,
    DAY(flight_date) AS day_of_month,
    DAYOFWEEK(flight_date) AS day_of_week,
    DAYNAME(flight_date) AS day_name,
    WEEKOFYEAR(flight_date) AS week_of_year,

    -- Business logic
    CASE
        WHEN DAYOFWEEK(flight_date) IN (1, 7) THEN 1
        ELSE 0
    END AS is_weekend,

    CASE
        WHEN DAYOFWEEK(flight_date) = 1 THEN 1
```

```
    -- Business logic
    CASE
        WHEN DAYOFWEEK(flight_date) IN (1, 7) THEN 1
        ELSE 0
    END AS is_weekend,

    CASE
        WHEN DAYOFWEEK(flight_date) = 1 THEN 1
        WHEN DAYOFWEEK(flight_date) = 6 THEN 1
        ELSE 0
    END AS is_peak_travel_day,

    -- Seasons
    CASE
        WHEN MONTH(flight_date) IN (12, 1, 2) THEN 'Winter'
        WHEN MONTH(flight_date) IN (3, 4, 5) THEN 'Spring'
        WHEN MONTH(flight_date) IN (6, 7, 8) THEN 'Summer'
        ELSE 'Fall'
    END AS season,

    CURRENT_TIMESTAMP() AS created_at

FROM date_spine
ORDER BY flight_date
```

3. Fact Table: fact_flights

Purpose:

Central fact table linking dimensions and containing flight-level measures for analytics and ML.

Operations:

- Map foreign keys: carrier_key, origin_airport_key, destination_airport_key, date_key.
- Include degenerate dimensions: flight_number, scheduled_departure_hour, time_of_day.

- Include numeric measures: departure/arrival delays, elapsed times, distance.
- Include flags: cancelled, diverted, on-time, weekend.
- Derived metrics: elapsed_time_difference, is_successful_on_time_flight.
- Incremental logic to append only new flights.

Output:

Fact table ready for analytics and ML models.

```sql
{{
    config(
        materialized='incremental',
        unique_key='flight_id',
        tags=['fact']
    )
}}

SELECT
    f.flight_id,
    {{ dbt_utils.generate_surrogate_key(['f.carrier_code']) }} AS carrier_key,
    {{ dbt_utils.generate_surrogate_key(['f.origin_airport']) }} AS origin_airport_key,
    {{ dbt_utils.generate_surrogate_key(['f.destination_airport']) }} AS destination_airport_key,
    {{ dbt_utils.generate_surrogate_key(['f.flight_date']) }} AS date_key,

    -- DEGENERATE DIMENSIONS
    f.flight_number,
    f.scheduled_departure_hour,
    f.time_of_day,

    -- MEASURES (numeric facts for aggregation)
    f.departure_delay_minutes,
    f.arrival_delay_minutes,
    f.actual_elapsed_minutes,
    f.scheduled_elapsed_minutes,
    f.distance_miles,

    -- FLAGS (for counting)
    f.is_cancelled,
    f.is_diverted,
    f.is_on_time,
    f.is_weekend,

    -- DERIVED MEASURES
    CASE
        WHEN f.scheduled_elapsed_minutes > 0
        THEN f.actual_elapsed_minutes - f.scheduled_elapsed_minutes
```

```
    f.is_weekend,

    -- DERIVED MEASURES
    CASE
        WHEN f.scheduled_elapsed_minutes > 0
        THEN f.actual_elapsed_minutes - f.scheduled_elapsed_minutes
        ELSE NULL
    END AS elapsed_time_difference,

    CASE
        WHEN f.is_cancelled = 0 AND f.is_on_time = 1 THEN 1
        ELSE 0
    END AS is_successful_on_time_flight,

    f.transformed_at,
    CURRENT_TIMESTAMP() AS fact_created_at

FROM {{ ref('stg_flights') }} f

{% if is_incremental() %}
    WHERE TRY_TO_DATE(f.flight_date, 'YYYY-MM-DD') IS NOT NULL
      AND f.flight_date > (SELECT MAX(date_key) FROM {{ this }})
{% else %}
    WHERE TRY_TO_DATE(f.flight_date, 'YYYY-MM-DD') IS NOT NULL
{% endif %}
```

4. ML Features Table: ml_features

Purpose:

Feature-engineered dataset for ML models predicting flight delays and cancellations.

Operations:

- Carrier, route, airport, and time-based rolling window features.
- Derived features: is_rush_hour, time_of_day_category.
- Target variables: target_departure_delay, target_arrival_delay, target_is_cancelled, target_is_on_time, target_delay_category.
- Ensure no nulls in critical columns (carrier_avg_arrival_delay_30d, route_avg_delay_30d, distance_miles, day_of_week).

Output:

ML-ready feature table.

```sql
{{ config(
    materialized='table',
    tags=['ml', 'features']
) }}

WITH base AS (
    SELECT
        flight_id,
        flight_date,
        carrier_code,
        origin_airport,
        destination_airport,
        day_of_week,
        month,
        year,
        scheduled_departure_hour,
        is_weekend,
        distance_miles,
        departure_delay_minutes,
        arrival_delay_minutes,
        is_cancelled,
        is_on_time
    FROM {{ ref('stg_flights') }}
    WHERE arrival_delay_minutes IS NOT NULL
),

carrier_features AS (
    SELECT
        flight_id,
        {{ rolling_average('departure_delay_minutes', 'carrier_code', 'flight_date', 30) }} AS carrier_avg_departure_delay_30d,
        {{ rolling_average('arrival_delay_minutes', 'carrier_code', 'flight_date', 30) }} AS carrier_avg_arrival_delay_30d,
        {{ rolling_average('arrival_delay_minutes', 'carrier_code', 'flight_date', 7) }} AS carrier_avg_arrival_delay_7d,
        {{ rolling_average('is_cancelled::FLOAT', 'carrier_code', 'flight_date', 30) }} AS carrier_cancellation_rate_30d,
        {{ rolling_average('is_on_time::FLOAT', 'carrier_code', 'flight_date', 30) }} AS carrier_on_time_rate_30d,
        {{ rolling_count('carrier_code', 'flight_date', 30) }} AS carrier_flight_count_30d
    FROM base
),

route_features AS (
    SELECT
        flight_id,
        {{ rolling_average('arrival_delay_minutes', 'origin_airport, destination_airport', 'flight_date', 30) }} AS route_avg_delay_30d,
        {{ rolling_average('is_on_time::FLOAT', 'origin_airport, destination_airport', 'flight_date', 30) }} AS route_on_time_rate_30d,
        {{ rolling_count('origin_airport, destination_airport', 'flight_date', 30) }} AS route_flight_count_30d,
        {{ rolling_stddev('arrival_delay_minutes', 'origin_airport, destination_airport', 'flight_date', 30) }} AS route_delay_std_30d
    FROM base
),

airport_features AS (
    SELECT
        flight_id,
        {{ rolling_average('departure_delay_minutes', 'origin_airport', 'flight_date', 7) }} AS origin_avg_departure_delay_7d,
        {{ rolling_average('arrival_delay_minutes', 'destination_airport', 'flight_date', 7) }} AS dest_avg_arrival_delay_7d,
        {{ rolling_count('origin_airport', 'flight_date', 1) }} AS origin_flights_prev_day
    FROM base
),

time_features AS (
    SELECT
        flight_id,
        {{ rolling_average('arrival_delay_minutes', 'carrier_code, day_of_week', 'flight_date', 90) }} AS carrier_dow_avg_delay,
        {{ rolling_average('arrival_delay_minutes', 'carrier_code, month', 'flight_date', 90) }} AS carrier_month_avg_delay,
        {{ rolling_average('arrival_delay_minutes', 'carrier_code, scheduled_departure_hour', 'flight_date', 90) }} AS carrier_hour_avg_delay
    FROM base
)

SELECT
    b.flight_id,
    b.flight_date,
    b.carrier_code,
    b.origin_airport,
    b.destination_airport,
    b.day_of_week,
    b.month,
    b.year,
```

```sql
SELECT
    b.flight_id,
    b.flight_date,
    b.carrier_code,
    b.origin_airport,
    b.destination_airport,
    b.day_of_week,
    b.month,
    b.year,
    b.scheduled_departure_hour,
    b.is_weekend,
    b.distance_miles,

    -- Carrier features (exclude flight_id)
    cf.carrier_avg_departure_delay_30d,
    cf.carrier_avg_arrival_delay_30d,
    cf.carrier_avg_arrival_delay_7d,
    cf.carrier_cancellation_rate_30d,
    cf.carrier_on_time_rate_30d,
    cf.carrier_flight_count_30d,

    -- Route features
    rf.route_avg_delay_30d,
    rf.route_on_time_rate_30d,
    rf.route_flight_count_30d,
    rf.route_delay_std_30d,

    -- Airport features
    af.origin_avg_departure_delay_7d,
    af.dest_avg_arrival_delay_7d,
    af.origin_flights_prev_day,

    -- Time features
    tf.carrier_dow_avg_delay,
    tf.carrier_month_avg_delay,
    tf.carrier_hour_avg_delay,
```

```sql
    -- Time features
    tf.carrier_dow_avg_delay,
    tf.carrier_month_avg_delay,
    tf.carrier_hour_avg_delay,

    -- Derived features
    CASE
        WHEN b.scheduled_departure_hour BETWEEN 7 AND 9 THEN 1
        WHEN b.scheduled_departure_hour BETWEEN 17 AND 19 THEN 1
        ELSE 0
    END AS is_rush_hour,

    {{ time_of_day('b.scheduled_departure_hour') }} AS time_of_day_category,

    -- Target variables
    b.departure_delay_minutes AS target_departure_delay,
    b.arrival_delay_minutes AS target_arrival_delay,
    b.is_cancelled AS target_is_cancelled,
    {{ is_on_time('b.arrival_delay_minutes', 15) }} AS target_is_on_time,
    {{ delay_category('b.arrival_delay_minutes') }} AS target_delay_category,

    CURRENT_TIMESTAMP() AS features_created_at

FROM base b
LEFT JOIN carrier_features cf ON b.flight_id = cf.flight_id
LEFT JOIN route_features rf ON b.flight_id = rf.flight_id
LEFT JOIN airport_features af ON b.flight_id = af.flight_id
LEFT JOIN time_features tf ON b.flight_id = tf.flight_id

WHERE cf.carrier_avg_arrival_delay_30d IS NOT NULL
  AND rf.route_avg_delay_30d IS NOT NULL
```

5. DBT Tests

Purpose:

Ensure data quality, integrity, and correctness in the staging, dimension, fact, and ML feature tables.

Operations / Tests Examples:

- Uniqueness & Not Null: Primary keys like flight_id, carrier_key, airport_key, date_key.
- Accepted Values: Boolean flags (is_cancelled, is_weekend) are only 0 or 1.
- Accepted Ranges: Numeric columns like arrival_delay_minutes and departure_delay_minutes stay within realistic limits (-200 to 2000 minutes).

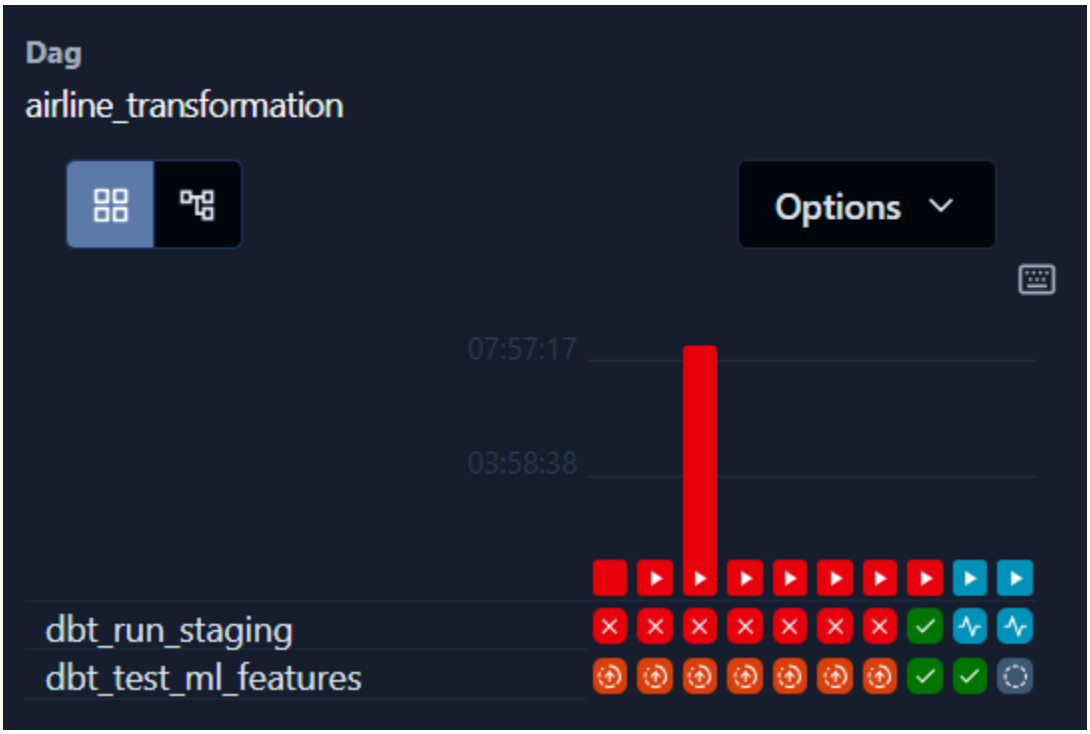
- **Relationships:** Foreign keys in fact tables properly link to dimension tables.
- **ML Critical Features Test:** Columns such as carrier_avg_arrival_delay_30d, route_avg_delay_30d, distance_miles, and day_of_week must not be null.

```sql
-- Test: Critical ML features must not be null
SELECT
    flight_id,
    CASE
        WHEN carrier_avg_arrival_delay_30d IS NULL THEN 'carrier_avg_null'
        WHEN route_avg_delay_30d IS NULL THEN 'route_avg_null'
        WHEN distance_miles IS NULL THEN 'distance_null'
        WHEN day_of_week IS NULL THEN 'day_of_week_null'
    END as null_feature
FROM {{ ref('ml_features') }}
WHERE (carrier_avg_arrival_delay_30d IS NULL
       OR route_avg_delay_30d IS NULL
       OR distance_miles IS NULL
       OR day_of_week IS NULL)
  AND flight_date >= '1987-02-01'
```

Airflow DAG Integration

- **DBT Run (Staging & Dimensions):**
- Task: dbt_run_staging
- Command: dbt run--select stg_* dim_*--profiles-dir /path/to/profiles
- **DBT Run (Fact & ML Layer):**
- Task: dbt_run_fact_ml
- Command: dbt run--select fact_flights ml_features--profiles-dir /path/to/profiles
- **DBT Tests:**
- Task: dbt_test
- Command: dbt test--select stg_* dim_* fact_flights ml_features

```
 GNU nano 7.2                                  /home/yasmina/airflow/dags/airline_dag.py
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import timedelta
from airflow.utils import timezone

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

DBT_PROJECT_DIR = '/mnt/f/Yasmina/Data Science/Samsung - Big Data/Final Project - Graduation/Airline/airline_project'

dag = DAG(
    'airline_transformation',
    default_args=default_args,
    description='End-to-end airline ML pipeline with dbt',
    schedule='@daily',
    start_date=timezone.utcnow() - timedelta(days=1),
    catchup=False,
    tags=['airline', 'dbt'],
)

dbt_run_staging = BashOperator(
    task_id='dbt_run_staging',
    bash_command=f'cd "{DBT_PROJECT_DIR}" && dbt run',
    dag=dag,
)

dbt_test_ml_features = BashOperator(
    task_id='dbt_test_ml_features',
    bash_command=f'cd "{DBT_PROJECT_DIR}" && dbt test',
    dag=dag,
)

dbt_run_staging >> dbt_test_ml_features

^G Help        ^O Write Out   ^W Where Is   ^K Cut         ^T Execute     ^C Location    M-U Undo       M-A Set Mark   M-] To Bracket  M-Q Previous
^X Exit        ^R Read File   ^\ Replace    ^U Paste       ^J Justify     ^/ Go To Line  M-E Redo       M-6 Copy       ^Q Where Was    M-W Next
```



ML Layer

Overview:

The project aims to predict flight arrival delays using historical flight data and engineered features. The model helps to estimate whether a flight will be delayed and by how many minutes, supporting airlines and passengers in better planning.

Training Sample:

For initial model development, we created a small random sample of the dataset to speed up training and testing. The training sample contains approximately 100,000 rows, while the test sample contains 50,000 rows. Using a smaller sample allows us to quickly iterate on the model and validate the workflow without waiting for the full dataset to process. Once the workflow is confirmed, the training dataset can be increased to the full dataset to improve model accuracy and performance.

Model Training:

A Random Forest Regressor is trained on features related to carrier, route, and time.

Model Evaluation:

Predictions are compared to actual delays using MAE and RMSE metrics.

Metrics (MAE, RMSE):

The model achieves a Mean Absolute Error (MAE) of 18.76 minutes and a Root Mean Squared Error (RMSE) of 33.11 minutes. This indicates that, on average, the model predicts the arrival delay within ±19 minutes. While the model provides a reasonable baseline, accuracy can be further improved by training on the full dataset and adding more features.

```sql
SELECT MIN(flight_date) AS first_date,
       MAX(flight_date) AS last_date
FROM AIRLINE_PROJECT.RAW.ML_FEATURES;

-- 1.1: Check total rows
SELECT COUNT(*) AS total_rows
FROM AIRLINE_PROJECT.RAW.ML_FEATURES;

USE DATABASE AIRLINE_PROJECT;
USE SCHEMA RAW;

CREATE OR REPLACE PROCEDURE train_delay_simple()
RETURNS VARCHAR
LANGUAGE PYTHON
RUNTIME_VERSION = '3.8'
PACKAGES = ('snowflake-snowpark-python', 'scikit-learn', 'pandas')
HANDLER = 'train_simple'
AS
$$
def train_simple(session):
    import pandas as pd
    from sklearn.ensemble import RandomForestRegressor

    # Ultra simple - 100k rows, 5 features
    df = session.sql("""
        SELECT
            CARRIER_AVG_ARRIVAL_DELAY_30D,
            ROUTE_AVG_DELAY_30D
```

```
ALTER WAREHOUSE COMPUTE_WH SET WAREHOUSE_SIZE = 'LARGE';

SHOW WAREHOUSES;

USE DATABASE AIRLINE_PROJECT;
USE SCHEMA RAW;

CREATE OR REPLACE STAGE RAW.ml;

--GRANT USAGE ON STAGE ml TO ROLE ACCOUNTADMIN;
GRANT READ ON STAGE ml TO ROLE ACCOUNTADMIN;
GRANT WRITE ON STAGE ml TO ROLE ACCOUNTADMIN;


CREATE OR REPLACE PROCEDURE train_simple_delay_model()
RETURNS VARCHAR
LANGUAGE PYTHON
RUNTIME_VERSION = '3.9'
PACKAGES = ('snowflake-snowpark-python', 'scikit-learn', 'pandas', 'joblib')
HANDLER = 'train_model'
AS
$$
def train_model(session):
    import pandas as pd
    from sklearn.ensemble import RandomForestRegressor
    import joblib
    import os
```

AIRLINE_PROJECT.RAW Settings Open in Workspaces Code Versions

```
28
29      # Load sample data (100K rows)
30      df = session.sql("""
31          SELECT
32              CARRIER_AVG_ARRIVAL_DELAY_30D,
33              ROUTE_AVG_DELAY_30D,
34              CARRIER_AVG_ARRIVAL_DELAY_7D,
35              IS_WEEKEND,
36              IS_RUSH_HOUR,
37              SCHEDULED_DEPARTURE_HOUR,
38              TARGET_ARRIVAL_DELAY
39          FROM ML_FEATURES_TRAIN_SAMPLE
40          LIMIT 100000
41      """).to_pandas()
42
43      # Prepare features and target
44      X = df[['CARRIER_AVG_ARRIVAL_DELAY_30D',
45              'ROUTE_AVG_DELAY_30D',
46              'CARRIER_AVG_ARRIVAL_DELAY_7D',
47              'IS_WEEKEND',
48              'IS_RUSH_HOUR',
49              'SCHEDULED_DEPARTURE_HOUR']]
50      y = df['TARGET_ARRIVAL_DELAY']
51
52      # Train model
53      model = RandomForestRegressor(n_estimators=50, max_depth=10, random_state=42)
54      model.fit(X, y)
55
```

```
    # Save locally
    local_file = '/tmp/delay_model.joblib'
    joblib.dump(model, local_file)

    # Upload to Snowflake stage
    session.file.put(local_file, '@ml', overwrite=True)

    return "Model trained and uploaded to stage successfully!"
$$;


-- Call the procedure
CALL train_simple_delay_model();

SHOW TABLES LIKE 'ML_FEATURES_TEST_SAMPLE' IN SCHEMA RAW;

GRANT SELECT ON TABLE AIRLINE_PROJECT.RAW.ML_FEATURES_TEST_SAMPLE TO ROLE ACCOUNTADMIN;
GRANT READ ON STAGE ml TO ROLE ACCOUNTADMIN;


CREATE OR REPLACE PROCEDURE evaluate_delay_model()
RETURNS VARCHAR
LANGUAGE PYTHON
RUNTIME_VERSION = '3.9'
PACKAGES = ('snowflake-snowpark-python', 'pandas', 'scikit-learn', 'joblib')
HANDLER = 'evaluate_model'
AS
$$
```

```
def evaluate_model(session):
    import pandas as pd
    import joblib
    from sklearn.metrics import mean_absolute_error, mean_squared_error
    import math
    import os

    # Download model from stage (full path)
    session.file.get('@AIRLINE_PROJECT.RAW.ML/delay_model.joblib.gz', '/tmp/')
    local_file = '/tmp/delay_model.joblib.gz'

    # Check if file exists
    assert os.path.exists(local_file), f"{local_file} not found!"

    # Load model
    model = joblib.load(local_file)

    # Load test data sample
    df_test = session.sql("""
        SELECT
            CARRIER_AVG_ARRIVAL_DELAY_30D,
            ROUTE_AVG_DELAY_30D,
            CARRIER_AVG_ARRIVAL_DELAY_7D,
            IS_WEEKEND,
            IS_RUSH_HOUR,
            SCHEDULED_DEPARTURE_HOUR,
            TARGET_ARRIVAL_DELAY
```

```
        FROM ML_FEATURES_TEST_SAMPLE
        LIMIT 50000
    """).to_pandas()

    # Prepare features and target
    X_test = df_test[['CARRIER_AVG_ARRIVAL_DELAY_30D',
                      'ROUTE_AVG_DELAY_30D',
                      'CARRIER_AVG_ARRIVAL_DELAY_7D',
                      'IS_WEEKEND',
                      'IS_RUSH_HOUR',
                      'SCHEDULED_DEPARTURE_HOUR']]
    y_test = df_test['TARGET_ARRIVAL_DELAY']

    # Predict
    y_pred = model.predict(X_test)

    # Metrics
    mae = mean_absolute_error(y_test, y_pred)
    rmse = math.sqrt(mean_squared_error(y_test, y_pred))

    return f"Test MAE: {mae:.2f}, RMSE: {rmse:.2f}"
$$;



CALL evaluate_delay_model();

SHOW STAGES LIKE 'ML';
```

Real time streaming

Overview

The goal of this streaming module is to **simulate a real-time feed of flight data** from our Data Warehouse (Snowflake) and detect **"High Delay Anomalies"** (delays > 60 minutes) instantly. This allows operations teams to receive alerts immediately, rather than waiting for end-of-day reports.

1.1 Architecture

The data flows as follows:

1. **Source:** Raw flight data residing in Snowflake.
2. **Ingestion:** A Python script acts as a producer, fetching data and streaming it to Kafka.
3. **Message Broker:** Apache Kafka buffers the high-velocity data.
4. **Processing:** Apache Spark (Structured Streaming) reads from Kafka, parses the data, and applies anomaly detection logic.
5. **Alerting:** Flights meeting the criteria trigger an alert message.

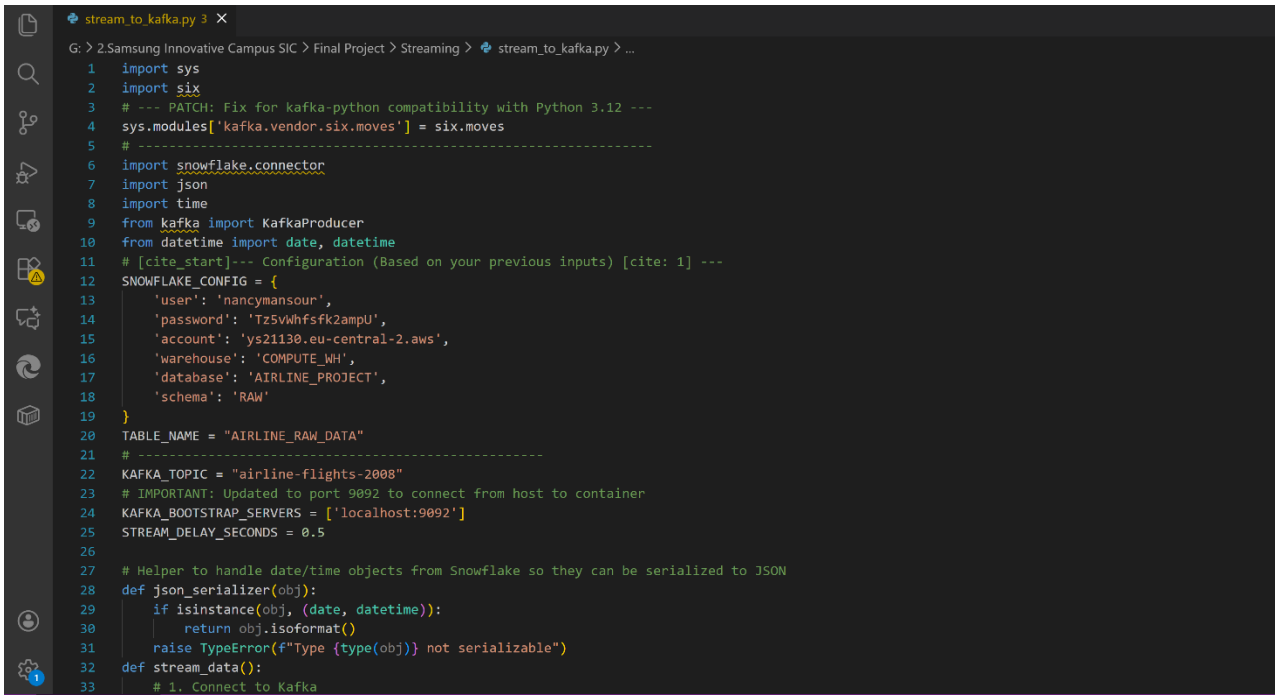
2.1 Step 1: Streaming Data from Snowflake to Kafka

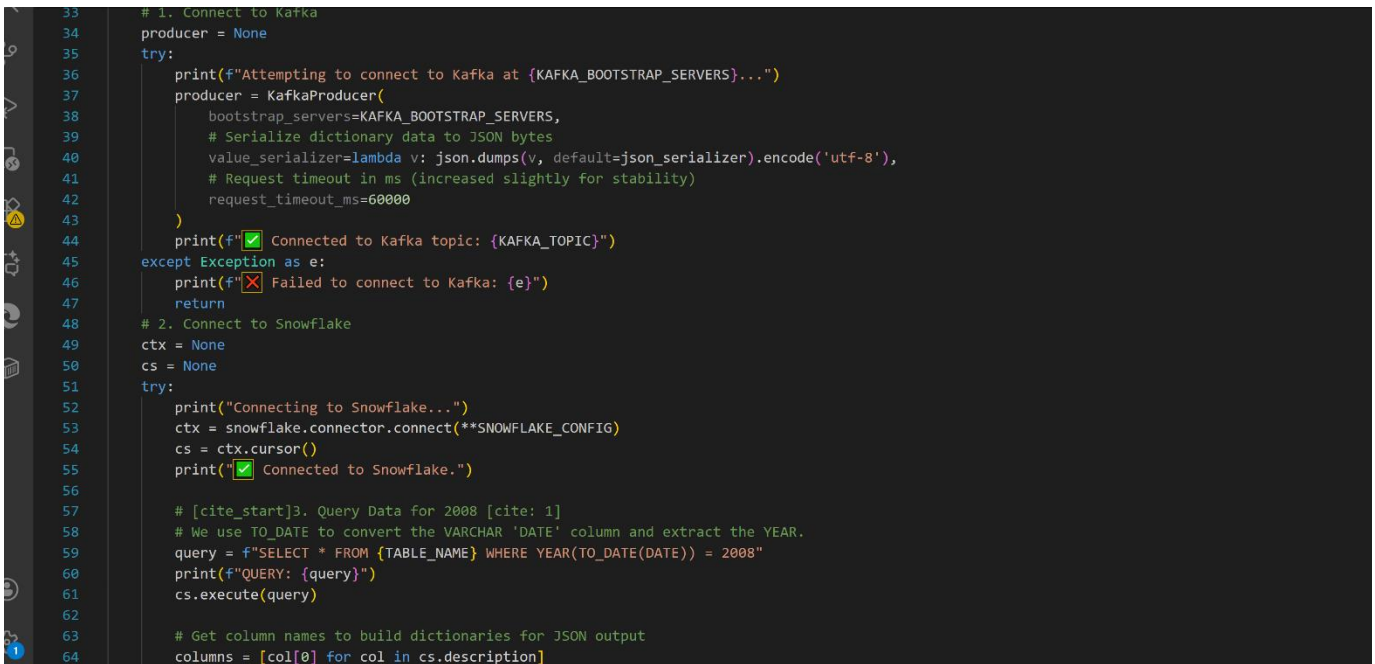
Since our data currently sits in Snowflake, we created a **Producer Script** (stream_to_kafka.py). This script connects to Snowflake, queries the flight data for the year 2008, converts the rows into JSON format, and publishes them to a Kafka topic named airline-flights-2008.

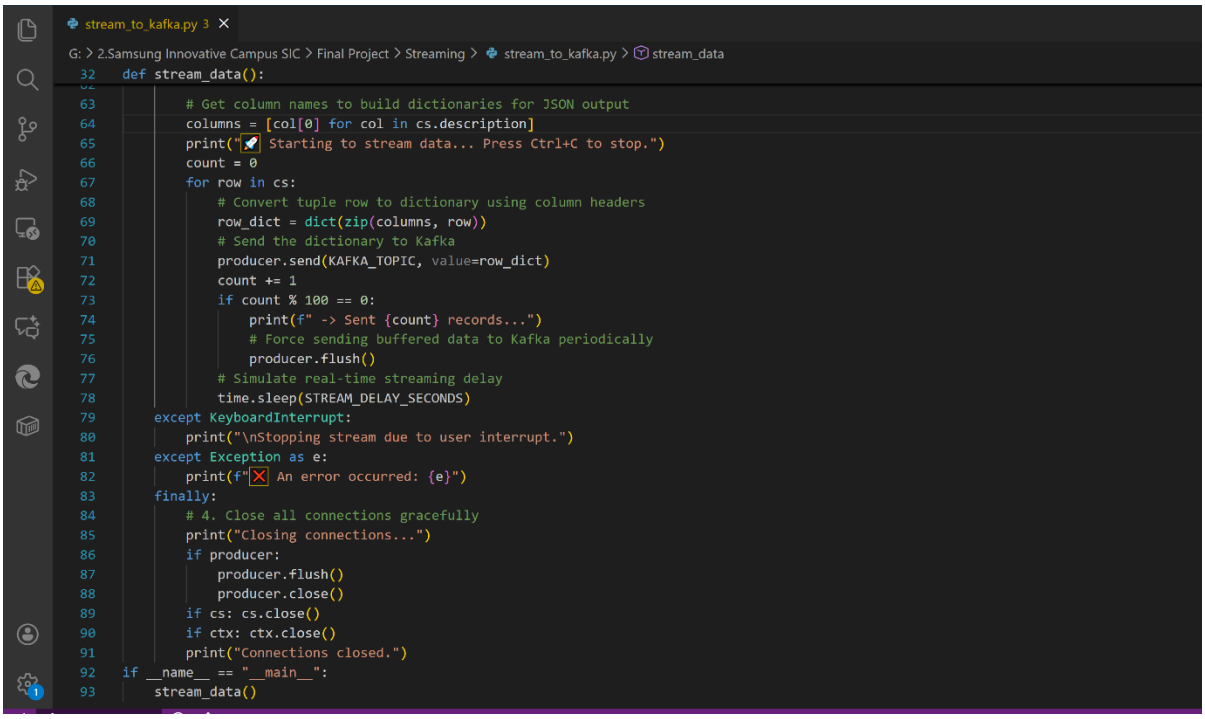
Key Code Highlights:

- **Snowflake Connector:** Fetches raw data.
- **JSON Serializer:** Converts date objects to strings for transmission.
- **Simulation:** A time.sleep(0.5) is added to simulate real-time data arrival.

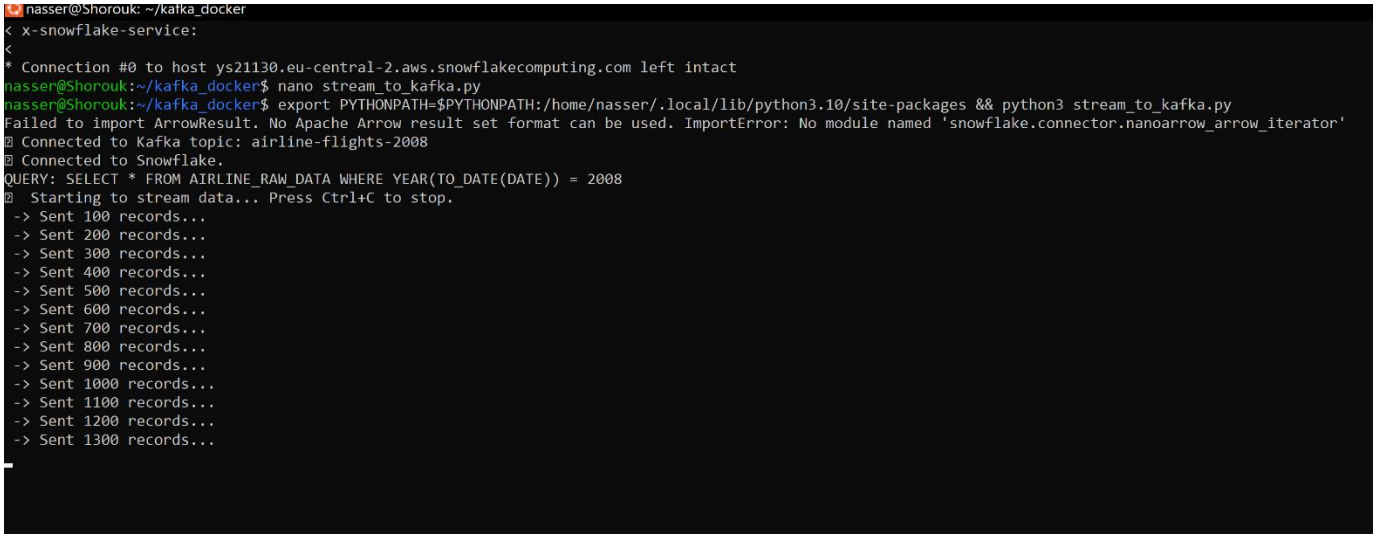
Code: stream_to_kafka.py







Execution Results: Running this script initiates the data flow. The terminal confirms that records are being serialized and sent to the Kafka broker.



2.2 Step 2: Verifying Data in Kafka

Before processing the data with Spark, it is critical to verify that the messages are actually landing in the Kafka topic. We used the Kafka Console Consumer to inspect the raw topic data.

Command Used:

```
docker exec kafka kafka-console-consumer --bootstrap-server localhost:9092 --topic airline-flights-2008 --from-beginning
```

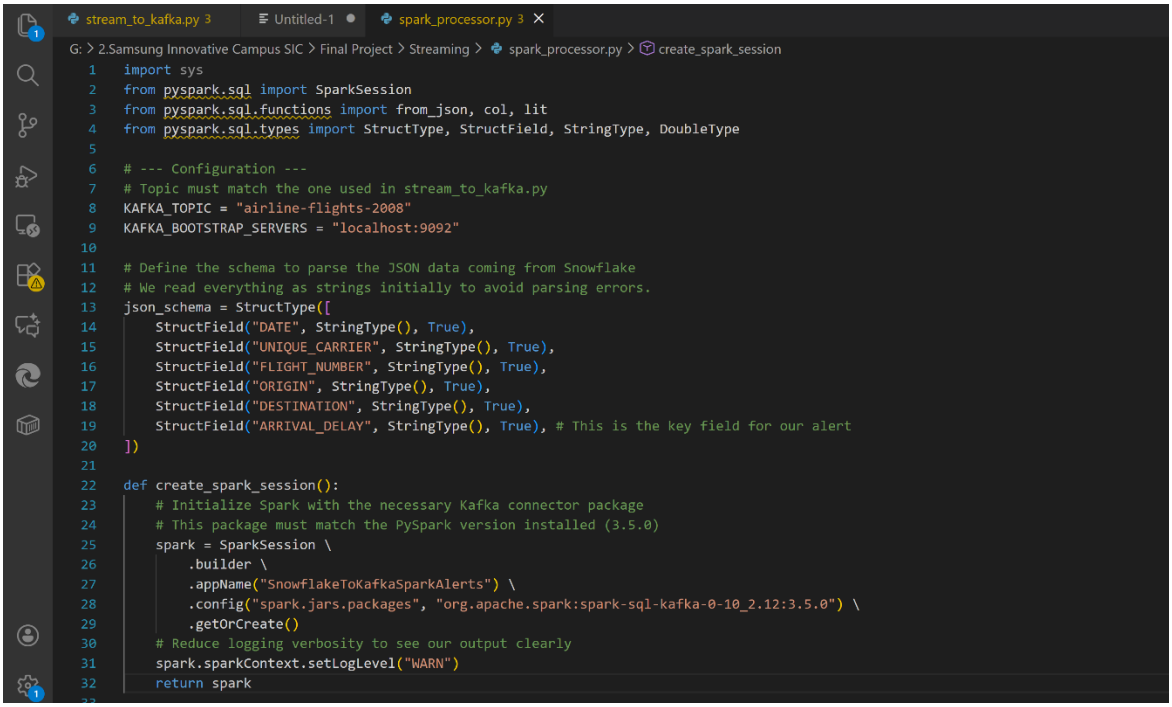
Results: The console shows the raw JSON data (Arrival Time, Carrier, Delay, etc.) flowing through the topic. This confirms the producer is working correctly.

This is the core intelligence of the streaming layer. We developed a PySpark application (spark_processor.py) to consume the data from Kafka and apply business logic.

Logic Implemented:

1. **Read Stream:** Connect to Kafka topic airline-flights-2008.
2. **Parse:** Convert the binary Kafka message into a structured Spark DataFrame using a defined Schema.
3. **Detect Anomaly:** Filter the stream for rows where ARRIVAL_DELAY > 60.
4. **Tag:** Add a new column ALERT_MESSAGE with the text "HIGH DELAY DETECTED".

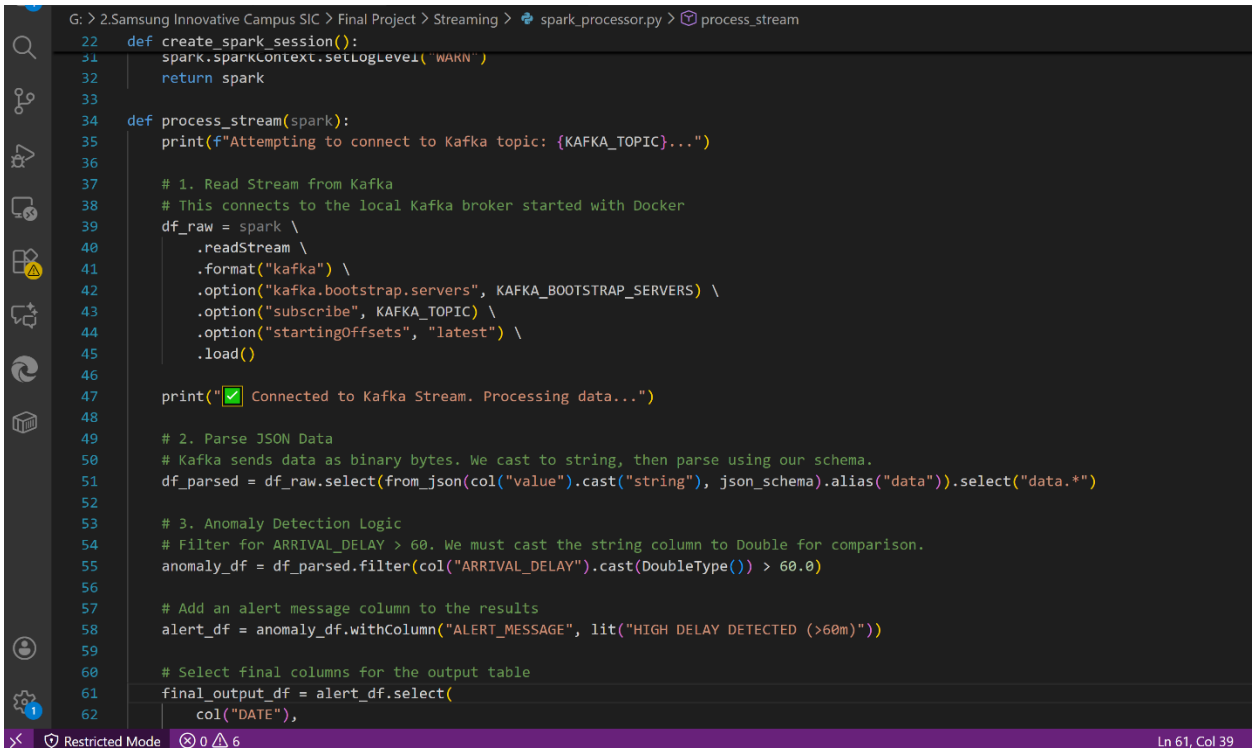
Code: spark_processor.py

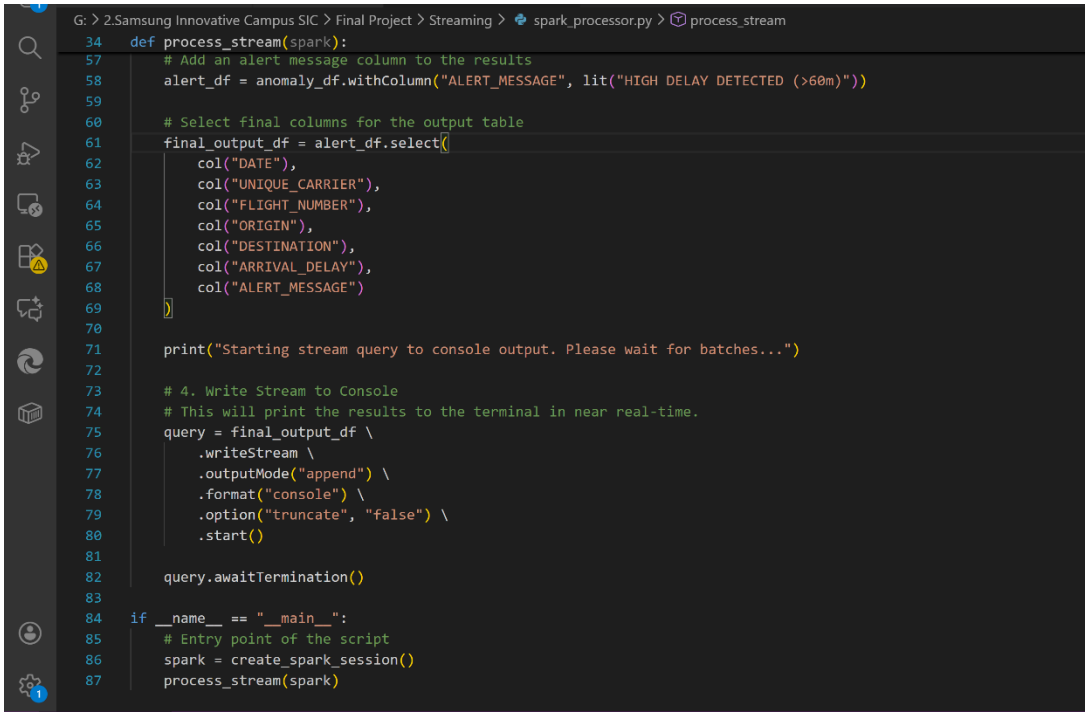


stream_to_kafka.py 3 | Untitled-1 | spark_processor.py 3

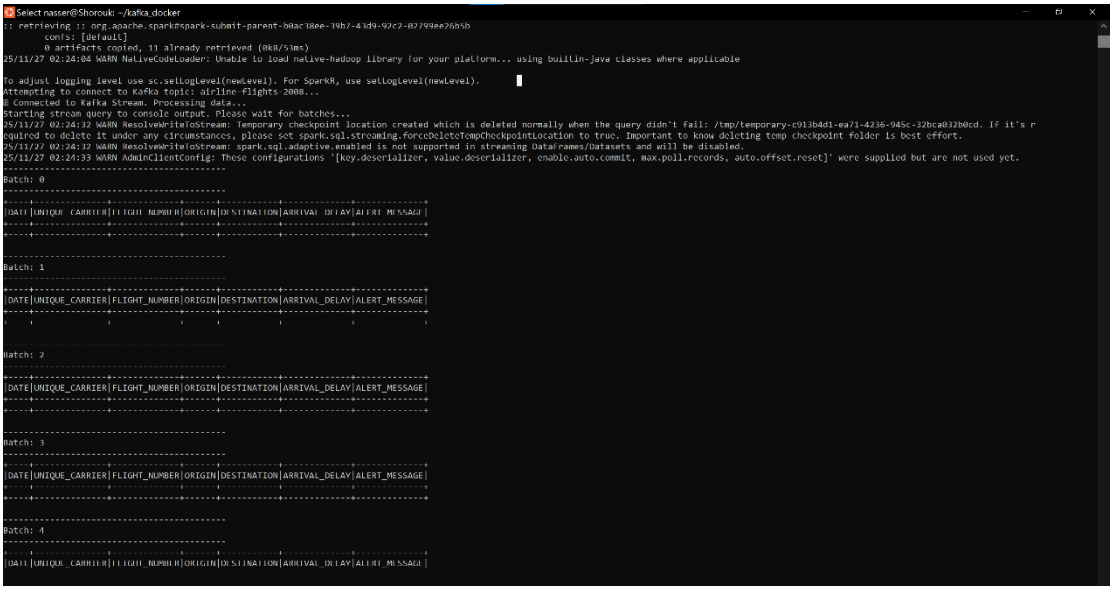
G: > 2.Samsung Innovative Campus SIC > Final Project > Streaming > spark_processor.py > create_spark_session

```python
import sys
from pyspark.sql import SparkSession
from pyspark.sql.functions import from_json, col, lit
from pyspark.sql.types import StructType, StructField, StringType, DoubleType

# --- Configuration ---
# Topic must match the one used in stream_to_kafka.py
KAFKA_TOPIC = "airline-flights-2008"
KAFKA_BOOTSTRAP_SERVERS = "localhost:9092"

# Define the schema to parse the JSON data coming from Snowflake
# We read everything as strings initially to avoid parsing errors.
json_schema = StructType([
    StructField("DATE", StringType(), True),
    StructField("UNIQUE_CARRIER", StringType(), True),
    StructField("FLIGHT_NUMBER", StringType(), True),
    StructField("ORIGIN", StringType(), True),
    StructField("DESTINATION", StringType(), True),
    StructField("ARRIVAL_DELAY", StringType(), True), # This is the key field for our alert
])

def create_spark_session():
    # Initialize Spark with the necessary Kafka connector package
    # This package must match the PySpark version installed (3.5.0)
    spark = SparkSession \
        .builder \
        .appName("SnowflakeToKafkaSparkAlerts") \
        .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0") \
        .getOrCreate()
    # Reduce logging verbosity to see our output clearly
    spark.sparkContext.setLogLevel("WARN")
    return spark
```


G: > 2.Samsung Innovative Campus SIC > Final Project > Streaming > spark_processor.py > process_stream
def create_spark_session():
spark.sparkContext.setLogLevel("WARN")
return spark
def process_stream(spark):
print(f"Attempting to connect to Kafka topic: {KAFKA_TOPIC}...")
# 1. Read Stream from Kafka
# This connects to the local Kafka broker started with Docker
df_raw = spark \
.readStream \
.format("kafka") \
.option("kafka.bootstrap.servers", KAFKA_BOOTSTRAP_SERVERS) \
.option("subscribe", KAFKA_TOPIC) \
.option("startingOffsets", "latest") \
.load()
print("✅ Connected to Kafka Stream. Processing data...")
# 2. Parse JSON Data
# Kafka sends data as binary bytes. We cast to string, then parse using our schema.
df_parsed = df_raw.select(from_json(col("value").cast("string"), json_schema).alias("data")).select("data.*")
# 3. Anomaly Detection Logic
# Filter for ARRIVAL_DELAY > 60. We must cast the string column to Double for comparison.
anomaly_df = df_parsed.filter(col("ARRIVAL_DELAY").cast(DoubleType()) > 60.0)
# Add an alert message column to the results
alert_df = anomaly_df.withColumn("ALERT_MESSAGE", lit("HIGH DELAY DETECTED (>60m)"))
# Select final columns for the output table
final_output_df = alert_df.select(
col("DATE"),
Restricted Mode
0 6
Ln 61, Col 39


G: > 2.Samsung Innovative Campus SIC > Final Project > Streaming > spark_processor.py > process_stream
def process_stream(spark):
# Add an alert message column to the results
alert_df = anomaly_df.withColumn("ALERT_MESSAGE", lit("HIGH DELAY DETECTED (>60m)"))
# Select final columns for the output table
final_output_df = alert_df.select(
col("DATE"),
col("UNIQUE_CARRIER"),
col("FLIGHT_NUMBER"),
col("ORIGIN"),
col("DESTINATION"),
col("ARRIVAL_DELAY"),
col("ALERT_MESSAGE")
)
print("Starting stream query to console output. Please wait for batches...")
# 4. Write Stream to Console
# This will print the results to the terminal in near real-time.
query = final_output_df \
.writeStream \
.outputMode("append") \
.format("console") \
.option("truncate", "false") \
.start()
query.awaitTermination()
if __name__ == "__main__":
# Entry point of the script
spark = create_spark_session()
process_stream(spark)

Initialization: We submit the Spark job. The logs below confirm that Spark successfully connected to the Kafka stream and initialized the query engine.



2.4 Step 4: Anomaly Detection Results (Alerts)

As the data streamed in from the Producer, Spark processed it in micro-batches. It successfully filtered out normal flights and only outputted the rows where the arrival delay exceeded 60 minutes.

The output clearly shows the ALERT_MESSAGE column appended to the data, serving as a flag for downstream alerting systems (e.g., sending an email or Slack notification).

```
Batch: 402
-------------------------------------------
+----+--------------+-------------+------+-----------+-------------+-------------+
|DATE|UNIQUE_CARRIER|FLIGHT_NUMBER|ORIGIN|DESTINATION|ARRIVAL_DELAY|ALERT_MESSAGE|
+----+--------------+-------------+------+-----------+-------------+-------------+
+----+--------------+-------------+------+-----------+-------------+-------------+

-------------------------------------------
Batch: 403
-------------------------------------------
+----------+--------------+-------------+------+-----------+-------------+--------------------------+
|DATE      |UNIQUE_CARRIER|FLIGHT_NUMBER|ORIGIN|DESTINATION|ARRIVAL_DELAY|ALERT_MESSAGE             |
+----------+--------------+-------------+------+-----------+-------------+--------------------------+
|2008-01-05|WN            |780          |ONT   |NULL       |83           |HIGH DELAY DETECTED (>60m)|
+----------+--------------+-------------+------+-----------+-------------+--------------------------+

-------------------------------------------
Batch: 404
-------------------------------------------
+----------+--------------+-------------+------+-----------+-------------+--------------------------+
|DATE      |UNIQUE_CARRIER|FLIGHT_NUMBER|ORIGIN|DESTINATION|ARRIVAL_DELAY|ALERT_MESSAGE             |
+----------+--------------+-------------+------+-----------+-------------+--------------------------+
|2008-01-05|WN            |835          |ONT   |NULL       |152          |HIGH DELAY DETECTED (>60m)|
+----------+--------------+-------------+------+-----------+-------------+--------------------------+

-------------------------------------------
Batch: 405
-------------------------------------------
+----+--------------+-------------+------+-----------+-------------+-------------+
|DATE|UNIQUE_CARRIER|FLIGHT_NUMBER|ORIGIN|DESTINATION|ARRIVAL_DELAY|ALERT_MESSAGE|
+----+--------------+-------------+------+-----------+-------------+-------------+
+----+--------------+-------------+------+-----------+-------------+-------------+

-------------------------------------------
Batch: 406
-------------------------------------------
+----+--------------+-------------+------+-----------+-------------+-------------+
|DATE|UNIQUE_CARRIER|FLIGHT_NUMBER|ORIGIN|DESTINATION|ARRIVAL_DELAY|ALERT_MESSAGE|
+----+--------------+-------------+------+-----------+-------------+-------------+
+----+--------------+-------------+------+-----------+-------------+-------------+

-------------------------------------------
Batch: 407
-------------------------------------------
+----+--------------+-------------+------+-----------+-------------+-------------+
|DATE|UNIQUE_CARRIER|FLIGHT_NUMBER|ORIGIN|DESTINATION|ARRIVAL_DELAY|ALERT_MESSAGE|
+----+--------------+-------------+------+-----------+-------------+-------------+
+----+--------------+-------------+------+-----------+-------------+-------------+
```

```
Select nasser@Shorouk: ~/kafka_docker
Batch: 453
-------------------------------------------
+----------+--------------+-------------+------+-----------+-------------+--------------------------+
|DATE      |UNIQUE_CARRIER|FLIGHT_NUMBER|ORIGIN|DESTINATION|ARRIVAL_DELAY|ALERT_MESSAGE             |
+----------+--------------+-------------+------+-----------+-------------+--------------------------+
|2008-01-05|WN            |3228         |PDX   |NULL       |61           |HIGH DELAY DETECTED (>60m)|
+----------+--------------+-------------+------+-----------+-------------+--------------------------+

-------------------------------------------
Batch: 454
-------------------------------------------
+----------+--------------+-------------+------+-----------+-------------+--------------------------+
|DATE      |UNIQUE_CARRIER|FLIGHT_NUMBER|ORIGIN|DESTINATION|ARRIVAL_DELAY|ALERT_MESSAGE             |
+----------+--------------+-------------+------+-----------+-------------+--------------------------+
|2008-01-05|WN            |1019         |PDX   |NULL       |176          |HIGH DELAY DETECTED (>60m)|
+----------+--------------+-------------+------+-----------+-------------+--------------------------+

-------------------------------------------
Batch: 455
-------------------------------------------
+----+--------------+-------------+------+-----------+-------------+-------------+
|DATE|UNIQUE_CARRIER|FLIGHT_NUMBER|ORIGIN|DESTINATION|ARRIVAL_DELAY|ALERT_MESSAGE|
+----+--------------+-------------+------+-----------+-------------+-------------+
+----+--------------+-------------+------+-----------+-------------+-------------+

-------------------------------------------
Batch: 456
-------------------------------------------
+----+--------------+-------------+------+-----------+-------------+-------------+
|DATE|UNIQUE_CARRIER|FLIGHT_NUMBER|ORIGIN|DESTINATION|ARRIVAL_DELAY|ALERT_MESSAGE|
+----+--------------+-------------+------+-----------+-------------+-------------+
+----+--------------+-------------+------+-----------+-------------+-------------+

-------------------------------------------
Batch: 457
-------------------------------------------
+----+--------------+-------------+------+-----------+-------------+-------------+
|DATE|UNIQUE_CARRIER|FLIGHT_NUMBER|ORIGIN|DESTINATION|ARRIVAL_DELAY|ALERT_MESSAGE|
+----+--------------+-------------+------+-----------+-------------+-------------+
+----+--------------+-------------+------+-----------+-------------+-------------+

-------------------------------------------
Batch: 458
-------------------------------------------
+----------+--------------+-------------+------+-----------+-------------+--------------------------+
|DATE      |UNIQUE_CARRIER|FLIGHT_NUMBER|ORIGIN|DESTINATION|ARRIVAL_DELAY|ALERT_MESSAGE             |
+----------+--------------+-------------+------+-----------+-------------+--------------------------+
|2008-01-05|WN            |789          |PDX   |NULL       |136          |HIGH DELAY DETECTED (>60m)|
+----------+--------------+-------------+------+-----------+-------------+--------------------------+

-------------------------------------------
```

Dashboards

1. High-Level Overview

This is a **strategic/historical dashboard** likely built on Microsoft Azure (indicated by the map logo). It analyzes a massive dataset (116 Million flights) spanning roughly two decades (approx. 1987–2008), focusing on volume, punctuality, and carrier performance.

2. Component Analysis & Descriptions

A. Top KPI Cards (Key Performance Indicators)

These cards provide the "headlines" for the dataset.

- **On-Time % (80.68%):** Indicates the overall efficiency of the airlines. 4 out of 5 flights arrive on time.
- **Avg Arrival Delay (7.07):** The average flight is delayed by roughly 7 minutes. This is a fairly low number, suggesting that while delays happen, massive multi-hour delays are outliers rather than the norm.

- **Total Flights (116M):** A massive volume metric indicating the scale of the data being analyzed.

B. Total Flights by YEAR (Bar Chart)

- **Observation:** The chart displays a steady increase in air travel demand from 1987 through roughly 2005/2007.
- **Trend:** There is a consistent upward trend, plateauing in the early 2000s.
- **Note:** The sharp drop off at the far right likely indicates incomplete data for the final year recorded, rather than an industry collapse.

C. Total Flights by CARRIER_NAME (Bar Chart)

- **Observation:** This ranks airlines by market share (volume).
- **Leaders:** "Delta Air Lines" is the clear market leader in this specific dataset, followed closely by "Southwest" and "American Airlines."
- **Legacy Carriers:** The chart features several historical names (NW/Northwest, TW/TWA, CO/Continental), confirming this is historical data.

D. Total Flights by DESTINATION_AIRPORT (Map)

- **Visualization:** A geospatial bubble map of the USA.
- **Insights:** The green bubbles represent traffic volume. The largest bubbles align with major US hubs:
    - **Atlanta (ATL):** Likely the largest bubble in the Southeast.
    - **Chicago (ORD):** Large hub in the Midwest.
    - **Dallas (DFW) & Houston (IAH):** Major hubs in Texas.
    - **California:** High density of flights in LA and SF areas.

E. Total Flights by MONTH (Line Chart)

- **Observation:** A seasonality analysis showing when people fly the most.
- **Peaks:**
    - **March:** Spring Break travel.
    - **July/August:** Summer vacation peak.
    - **October & December:** Holiday travel surges.
- **Troughs:** February and September/November appear to be the slowest months for travel.