-- Airline Business Intelligence Database -- PostgreSQL 16+ DDL — Empty schema with constraints -- Save as: sql/001_schema.sql

BEGIN;

-- Optional: put everything in its own schema CREATE SCHEMA IF NOT EXISTS airline; SET search_path TO airline, public;

-- ====== ENUM TYPES ====== DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'flight_status') THEN CREATE TYPE flight_status AS ENUM ('Scheduled','Departed','Arrived','Cancelled','Diverted'); END IF;

IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'payment_method') THEN CREATE TYPE payment_method AS ENUM ('Card','Points','Voucher','Cash'); END IF;

IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'payment_status') THEN CREATE TYPE payment_status AS ENUM ('Authorized','Captured','Refunded','Failed'); END IF;

IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'loyalty_tier') THEN CREATE TYPE loyalty_tier AS ENUM ('Basic','Silver','Gold','Platinum'); END IF;

IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'miles_txn_type') THEN CREATE TYPE miles_txn_type AS ENUM ('EARN','REDEEM','ADJUST'); END IF; END$$;

-- ====== REFERENCE TABLES ====== CREATE TABLE IF NOT EXISTS airports (airport_id BIGSERIAL PRIMARY KEY, iata_code VARCHAR(3) UNIQUE, icao_code VARCHAR(4) UNIQUE, name TEXT NOT NULL, city TEXT, country TEXT, latitude NUMERIC(8,5), longitude NUMERIC(8,5), timezone TEXT, CONSTRAINT chk_airports_lat CHECK (latitude BETWEEN -90 AND 90), CONSTRAINT chk_airports_lon CHECK (longitude BETWEEN -180 AND 180));

CREATE TABLE IF NOT EXISTS airlines (airline_id BIGSERIAL PRIMARY KEY, name TEXT NOT NULL, iata_code VARCHAR(3) UNIQUE, icao_code VARCHAR(3) UNIQUE, country TEXT);

CREATE TABLE IF NOT EXISTS aircraft (aircraft_id BIGSERIAL PRIMARY KEY, manufacturer TEXT, model TEXT NOT NULL, seat_capacity INT NOT NULL CHECK (seat_capacity > 0), tail_number TEXT UNIQUE);

-- Routes represent a carrier + origin + destination (schedule-level concept) CREATE TABLE IF NOT EXISTS routes (route_id BIGSERIAL PRIMARY KEY, airline_id BIGINT NOT NULL REFERENCES airlines(airline_id), origin_airport_id BIGINT NOT NULL REFERENCES airports(airport_id), destination_airport_id BIGINT NOT NULL REFERENCES airports(airport_id), distance_nm INT CHECK (distance_nm >= 0), CONSTRAINT uq_routes UNIQUE (airline_id, origin_airport_id, destination_airport_id), CONSTRAINT chk_route_diff_airports CHECK (origin_airport_id <> destination_airport_id));

-- ====== CORE TRANSACTIONAL TABLES ====== CREATE TABLE IF NOT EXISTS flights (flight_id BIGSERIAL PRIMARY KEY, airline_id BIGINT NOT NULL REFERENCES airlines(airline_id), aircraft_id BIGINT NOT NULL REFERENCES aircraft(aircraft_id), route_id BIGINT REFERENCES routes(route_id), origin_airport_id BIGINT NOT NULL REFERENCES airports(airport_id), destination_airport_id BIGINT NOT NULL REFERENCES airports(airport_id), flight_number TEXT NOT NULL, flight_date DATE NOT NULL, scheduled_departure_utc TIMESTAMP NOT NULL, scheduled_arrival_utc TIMESTAMP NOT NULL, actual_departure_utc TIMESTAMP, actual_arrival_utc TIMESTAMP, delay_minutes INT, delay_cause TEXT,

status flight_status NOT NULL DEFAULT 'Scheduled', CONSTRAINT uq_flight_instance UNIQUE (airline_id, flight_number, flight_date), CONSTRAINT chk_sched_times CHECK (scheduled_departure_utc < scheduled_arrival_utc));

-- Passengers with demographics CREATE TABLE IF NOT EXISTS passengers (passenger_id BIGSERIAL PRIMARY KEY, first_name TEXT NOT NULL, last_name TEXT NOT NULL, email TEXT UNIQUE, gender TEXT, age_group TEXT, state_or_country TEXT, created_at TIMESTAMP NOT NULL DEFAULT NOW());

-- Simple model: one booking per passenger & flight CREATE TABLE IF NOT EXISTS bookings (booking_id BIGSERIAL PRIMARY KEY, passenger_id BIGINT NOT NULL REFERENCES passengers(passenger_id), flight_id BIGINT NOT NULL REFERENCES flights(flight_id), booking_date TIMESTAMP NOT NULL, fare_class TEXT, base_price_usd NUMERIC(10,2) CHECK (base_price_usd >= 0), booking_channel TEXT, CONSTRAINT uq_booking_unique UNIQUE (passenger_id, flight_id));

CREATE TABLE IF NOT EXISTS payments (payment_id BIGSERIAL PRIMARY KEY, booking_id BIGINT NOT NULL REFERENCES bookings(booking_id) ON DELETE CASCADE, amount_usd NUMERIC(10,2) NOT NULL CHECK (amount_usd >= 0), method payment_method NOT NULL, status payment_status NOT NULL, paid_at TIMESTAMP NOT NULL);

CREATE TABLE IF NOT EXISTS loyalty_accounts (loyalty_id BIGSERIAL PRIMARY KEY, passenger_id BIGINT NOT NULL UNIQUE REFERENCES passengers(passenger_id) ON DELETE CASCADE, tier loyalty_tier NOT NULL DEFAULT 'Basic', miles_balance INT NOT NULL DEFAULT 0 CHECK (miles_balance >= 0), enrolled_at TIMESTAMP NOT NULL DEFAULT NOW());

CREATE TABLE IF NOT EXISTS miles_transactions (miles_txn_id BIGSERIAL PRIMARY KEY, loyalty_id BIGINT NOT NULL REFERENCES loyalty_accounts(loyalty_id) ON DELETE CASCADE, flight_id BIGINT REFERENCES flights(flight_id), txn_type miles_txn_type NOT NULL, miles_delta INT NOT NULL, posted_at TIMESTAMP NOT NULL DEFAULT NOW());

-- Optional: audit table for aircraft swaps CREATE TABLE IF NOT EXISTS flight_changes (change_id BIGSERIAL PRIMARY KEY, flight_id BIGINT NOT NULL REFERENCES flights(flight_id) ON DELETE CASCADE, old_aircraft_id BIGINT REFERENCES aircraft(aircraft_id), new_aircraft_id BIGINT REFERENCES aircraft(aircraft_id), reason TEXT, changed_at TIMESTAMP NOT NULL DEFAULT NOW());

-- ====== INDEXES ====== CREATE INDEX IF NOT EXISTS idx_flights_date_route ON flights (flight_date, route_id); CREATE INDEX IF NOT EXISTS idx_flights_airline_num ON flights (airline_id, flight_number, flight_date); CREATE INDEX IF NOT EXISTS idx_routes_od ON routes (origin_airport_id, destination_airport_id); CREATE INDEX IF NOT EXISTS idx_bookings_passenger ON bookings (passenger_id); CREATE INDEX IF NOT EXISTS idx_payments_booking ON payments (booking_id); CREATE INDEX IF NOT EXISTS idx_miles_loyalty ON miles_transactions (loyalty_id); CREATE INDEX IF NOT EXISTS idx_loyalty_passenger ON loyalty_accounts (passenger_id); CREATE INDEX IF NOT EXISTS idx_airports_codes ON airports (iata_code, icao_code);

COMMIT;

-- ====== COMMENTS ====== COMMENT ON SCHEMA airline IS 'Schema for Airline Business Intelligence Database (DTSC 691 Capstone)'; COMMENT ON TABLE flights IS 'Flight instances by date; join to routes/airports/airlines for network analytics'; COMMENT ON TABLE bookings IS 'Simple 1:1 passenger-to-flight bookings; extend to PNR header + booking_passengers if needed'; COMMENT ON TABLE miles_transactions IS 'Immutable audit of loyalty miles earn/redeem/adjust events';

-- 03_dml_cleanup.sql -- Data standardization & sanity checks for Airline BI database -- Idempotent: safe to run multiple times.

SET search_path TO airline, public;

-- 1. AIRPORTS CLEANUP

-- Normalize bad / sentinel IATA codes to NULL UPDATE airline.airports SET iata_code = NULL WHERE iata_code IN ('', ' ', 'NA', '\N');

-- Normalize bad / sentinel ICAO codes to NULL UPDATE airline.airports SET icao_code = NULL WHERE icao_code IN ('', ' ', 'NA', '\N');

-- Trim and uppercase codes UPDATE airline.airports SET iata_code = UPPER(BTRIM(iata_code)), icao_code = UPPER(BTRIM(icao_code)) WHERE iata_code IS NOT NULL OR icao_code IS NOT NULL;

-- 2. AIRLINES CLEANUP

-- Normalize bad / sentinel IATA codes to NULL UPDATE airline.airlines SET iata_code = NULL WHERE iata_code IN ('', ' ', 'NA', '\N');

-- Normalize bad / sentinel ICAO codes to NULL UPDATE airline.airlines SET icao_code = NULL WHERE icao_code IN ('', ' ', 'NA', '\N');

-- Trim name and country, normalize empty country to NULL UPDATE airline.airlines SET name = BTRIM(name), country = NULLIF(BTRIM(country), '') WHERE name IS NOT NULL OR country IS NOT NULL;

-- Trim and uppercase codes UPDATE airline.airlines SET iata_code = UPPER(BTRIM(iata_code)), icao_code = UPPER(BTRIM(icao_code)) WHERE iata_code IS NOT NULL OR icao_code IS NOT NULL;

-- 3. FLIGHT PERFORMANCE (BTS) CLEANUP

-- Normalize bad / sentinel airline_iata and airport_iata to NULL UPDATE airline.flight_performance SET airline_iata = NULL WHERE airline_iata IN ('', ' ', 'NA', '\N');

UPDATE airline.flight_performance SET airport_iata = NULL WHERE airport_iata IN ('', ' ', 'NA', '\N');

-- Trim and uppercase codes UPDATE airline.flight_performance SET airline_iata = UPPER(BTRIM(airline_iata)), airport_iata = UPPER(BTRIM(airport_iata)) WHERE airline_iata IS NOT NULL OR airport_iata IS NOT NULL;

-- Negative delay values -> NULL (defensive cleaning) UPDATE airline.flight_performance SET total_arrival_delay_min = NULL WHERE total_arrival_delay_min < 0;

UPDATE airline.flight_performance SET carrier_delay = NULL WHERE carrier_delay < 0;

UPDATE airline.flight_performance SET weather_delay = NULL WHERE weather_delay < 0;

UPDATE airline.flight_performance SET nas_delay = NULL WHERE nas_delay < 0;

UPDATE airline.flight_performance SET security_delay = NULL WHERE security_delay < 0;

UPDATE airline.flight_performance SET late_aircraft_delay = NULL WHERE late_aircraft_delay < 0;

-- 4. FLIGHTS CLEANUP

-- Trim and uppercase flight numbers UPDATE airline.flights SET flight_number = UPPER(BTRIM(flight_number)) WHERE flight_number IS NOT NULL;

-- Negative delays -> NULL UPDATE airline.flights SET delay_minutes = NULL WHERE delay_minutes IS NOT NULL AND delay_minutes < 0;

-- 5. PASSENGERS CLEANUP

-- Normalize names (trim + InitCap) UPDATE airline.passengers SET first_name = INITCAP(BTRIM(first_name)), last_name = INITCAP(BTRIM(last_name)) WHERE first_name IS NOT NULL OR last_name IS NOT NULL;

-- Normalize email: trim + lowercase UPDATE airline.passengers SET email = LOWER(BTRIM(email)) WHERE email IS NOT NULL AND email <> LOWER(BTRIM(email));

-- Clean optional demographic fields: blank -> NULL UPDATE airline.passengers SET gender = NULLIF(BTRIM(gender), ''), age_group = NULLIF(BTRIM(age_group), ''), state_or_country = NULLIF(BTRIM(state_or_country), '') WHERE gender IS NOT NULL OR age_group IS NOT NULL OR state_or_country IS NOT NULL;

-- 6. BOOKINGS CLEANUP

-- Normalize fare_class and booking_channel casing UPDATE airline.bookings SET fare_class = INITCAP(BTRIM(fare_class)) WHERE fare_class IS NOT NULL;

UPDATE airline.bookings SET booking_channel = INITCAP(BTRIM(booking_channel)) WHERE booking_channel IS NOT NULL;

-- 7. LOYALTY & MILES TRANSACTIONS CLEANUP

-- Ensure no negative miles balances (defensive) UPDATE airline.loyalty_accounts SET miles_balance = GREATEST(miles_balance, 0) WHERE miles_balance IS NOT NULL AND miles_balance < 0;

-- Defensive: zero-out impossible zero-delta transactions UPDATE airline.miles_transactions SET miles_delta = 0 WHERE miles_delta IS NULL;

-- 8. PAYMENTS CLEANUP

-- No text trimming on enums needed; just defensive amount clean UPDATE airline.payments SET amount_usd = NULL WHERE amount_usd IS NOT NULL AND amount_usd < 0;

-- 9. ROUTES CLEANUP

-- Non-positive route distances -> NULL (defensive) UPDATE airline.routes SET distance_nm = NULL WHERE distance_nm IS NOT NULL AND distance_nm <= 0;

-- 10. ROW COUNTS AFTER CLEANUP

SELECT 'airlines' AS table_name, COUNT*() AS row_count FROM airline.airlines UNION ALL SELECT 'airports', COUNT*()* FROM airline.airports UNION ALL SELECT 'bookings', COUNT*() FROM airline.bookings UNION ALL SELECT 'flight_performance', COUNT*()* FROM airline.flight_performance UNION ALL SELECT 'flights', COUNT*() FROM airline.flights UNION ALL SELECT 'loyalty_accounts', COUNT*()* FROM airline.loyalty_accounts UNION ALL SELECT 'miles_transactions', COUNT*() FROM airline.miles_transactions UNION ALL SELECT 'passengers', COUNT*()* FROM airline.passengers UNION ALL SELECT 'payments', COUNT(*) FROM airline.payments ORDER BY table_name;

-- 11. BASIC FK INTEGRITY CHECKS (NULL-MATCH COUNTS)

SELECT -- flights -> airlines (SELECT COUNT(*) FROM airline.flights f LEFT JOIN airline.airlines a ON f.airline_id = a.airline_id WHERE a.airline_id IS NULL) AS flights_missing_airline,

```
-- flights -> airports (origin)
(SELECT COUNT(*)
 FROM airline.flights f
 LEFT JOIN airline.airports ao
   ON f.origin_airport_id = ao.airport_id
 WHERE ao.airport_id IS NULL) AS flights_missing_origin_airport,

-- flights -> airports (destination)
(SELECT COUNT(*)
 FROM airline.flights f
 LEFT JOIN airline.airports ad
   ON f.destination_airport_id = ad.airport_id
 WHERE ad.airport_id IS NULL) AS flights_missing_destination_airport,

-- bookings -> passengers
(SELECT COUNT(*)
 FROM airline.bookings b
 LEFT JOIN airline.passengers p
   ON b.passenger_id = p.passenger_id
 WHERE p.passenger_id IS NULL) AS bookings_missing_passenger,

-- bookings -> flights
(SELECT COUNT(*)
 FROM airline.bookings b
 LEFT JOIN airline.flights f
   ON b.flight_id = f.flight_id
 WHERE f.flight_id IS NULL) AS bookings_missing_flight,
```

```
-- payments -> bookings
(SELECT COUNT(*)
 FROM airline.payments pay
 LEFT JOIN airline.bookings b
   ON pay.booking_id = b.booking_id
 WHERE b.booking_id IS NULL) AS payments_missing_booking;
```

-- 04_constraints_indexes.sql -- Phase 3: Add constraints & indexes after data is cleaned -- Assumes: data loaded + 03_dml_cleanup.sql has already run

SET search_path TO airline, public;

-- 1. FOREIGN KEYS: CORE DIMENSIONS & FACTS

-- flights → airlines DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'flights' AND constraint_name = 'fk_flights_airline') THEN ALTER TABLE airline.flights ADD CONSTRAINT fk_flights_airline FOREIGN KEY (airline_id) REFERENCES airline.airlines (airline_id) ON UPDATE CASCADE ON DELETE RESTRICT; END IF; END$$;

-- flights → airports (origin) DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'flights' AND constraint_name = 'fk_flights_origin_airport') THEN ALTER TABLE airline.flights ADD CONSTRAINT fk_flights_origin_airport FOREIGN KEY (origin_airport_id) REFERENCES airline.airports (airport_id) ON UPDATE CASCADE ON DELETE RESTRICT; END IF; END$$;

-- flights → airports (destination) DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'flights' AND constraint_name = 'fk_flights_destination_airport') THEN ALTER TABLE airline.flights ADD CONSTRAINT fk_flights_destination_airport FOREIGN KEY (destination_airport_id) REFERENCES airline.airports (airport_id) ON UPDATE CASCADE ON DELETE RESTRICT; END IF; END$$;

-- flights → aircraft (optional) DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'flights' AND constraint_name = 'fk_flights_aircraft') THEN ALTER TABLE airline.flights ADD CONSTRAINT fk_flights_aircraft FOREIGN KEY (aircraft_id) REFERENCES airline.aircraft (aircraft_id) ON UPDATE CASCADE ON DELETE SET NULL; END IF; END$$;

-- flights → routes (optional) DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'flights' AND constraint_name = 'fk_flights_route') THEN ALTER TABLE airline.flights ADD CONSTRAINT fk_flights_route FOREIGN KEY (route_id) REFERENCES airline.routes (route_id) ON UPDATE CASCADE ON DELETE SET NULL; END IF; END$$;

-- 2. ROUTES: DIMENSIONAL RELATIONSHIPS

-- routes → airlines DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'routes' AND constraint_name = 'fk_routes_airline') THEN ALTER TABLE airline.routes ADD CONSTRAINT fk_routes_airline FOREIGN KEY (airline_id) REFERENCES airline.airlines (airline_id) ON UPDATE CASCADE ON DELETE RESTRICT; END IF; END$$;

-- routes → airports (origin) DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'routes' AND constraint_name = 'fk_routes_origin_airport') THEN ALTER TABLE airline.routes ADD CONSTRAINT fk_routes_origin_airport FOREIGN KEY (origin_airport_id) REFERENCES airline.airports (airport_id) ON UPDATE CASCADE ON DELETE RESTRICT; END IF; END$$;

-- routes → airports (destination) DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'routes' AND constraint_name = 'fk_routes_destination_airport') THEN ALTER TABLE airline.routes ADD CONSTRAINT fk_routes_destination_airport FOREIGN KEY (destination_airport_id) REFERENCES airline.airports (airport_id) ON UPDATE CASCADE ON DELETE RESTRICT; END IF; END$$;

-- routes: avoid exact duplicate directional routes per airline DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM pg_indexes WHERE schemaname = 'airline' AND tablename = 'routes' AND indexname = 'uq_routes_airline_origin_dest') THEN CREATE UNIQUE INDEX uq_routes_airline_origin_dest ON airline.routes (airline_id, origin_airport_id, destination_airport_id); END IF; END$$;

-- 3. BOOKINGS / PAYMENTS: TRANSACTIONAL FKs

-- bookings → passengers DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'bookings' AND constraint_name = 'fk_bookings_passenger') THEN ALTER TABLE airline.bookings ADD CONSTRAINT fk_bookings_passenger FOREIGN KEY (passenger_id) REFERENCES airline.passengers (passenger_id) ON UPDATE CASCADE ON DELETE CASCADE; END IF; END$$;

-- bookings → flights (flight_id is nullable, so SET NULL on delete) DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'bookings' AND constraint_name = 'fk_bookings_flight') THEN ALTER TABLE airline.bookings ADD CONSTRAINT fk_bookings_flight FOREIGN KEY (flight_id) REFERENCES airline.flights (flight_id) ON UPDATE CASCADE ON DELETE SET NULL; END IF; END$$;

-- payments → bookings DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'payments' AND constraint_name = 'fk_payments_booking') THEN ALTER TABLE airline.payments ADD CONSTRAINT fk_payments_booking FOREIGN KEY (booking_id) REFERENCES airline.bookings (booking_id) ON UPDATE CASCADE ON DELETE CASCADE; END IF; END$$;

-- 4. LOYALTY ACCOUNTS & MILES TRANSACTIONS

-- loyalty_accounts → passengers (1:many; one per passenger enforced via unique index below) DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'loyalty_accounts' AND constraint_name = 'fk_loyalty_passenger') THEN

ALTER TABLE airline.loyalty_accounts ADD CONSTRAINT fk_loyalty_passenger FOREIGN KEY (passenger_id) REFERENCES airline.passengers (passenger_id) ON UPDATE CASCADE ON DELETE CASCADE; END IF; END$$;

-- OPTIONAL: enforce at most one loyalty account per passenger DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM pg_indexes WHERE schemaname = 'airline' AND tablename = 'loyalty_accounts' AND indexname = 'uq_loyalty_passenger') THEN CREATE UNIQUE INDEX uq_loyalty_passenger ON airline.loyalty_accounts (passenger_id); END IF; END$$;

-- miles_transactions → loyalty_accounts DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'miles_transactions' AND constraint_name = 'fk_miles_loyalty') THEN ALTER TABLE airline.miles_transactions ADD CONSTRAINT fk_miles_loyalty FOREIGN KEY (loyalty_id) REFERENCES airline.loyalty_accounts (loyalty_id) ON UPDATE CASCADE ON DELETE CASCADE; END IF; END$$;

-- miles_transactions → flights (optional) DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'miles_transactions' AND constraint_name = 'fk_miles_flight') THEN ALTER TABLE airline.miles_transactions ADD CONSTRAINT fk_miles_flight FOREIGN KEY (flight_id) REFERENCES airline.flights (flight_id) ON UPDATE CASCADE ON DELETE SET NULL; END IF; END$$;

-- 5. FLIGHT CHANGES: AUDIT RELATIONSHIPS

-- flight_changes → flights DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'flight_changes' AND constraint_name = 'fk_flight_changes_flight') THEN ALTER TABLE airline.flight_changes ADD CONSTRAINT fk_flight_changes_flight FOREIGN KEY (flight_id) REFERENCES airline.flights (flight_id) ON UPDATE CASCADE ON DELETE CASCADE; END IF; END$$;

-- flight_changes → aircraft (old) DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'flight_changes' AND constraint_name = 'fk_flight_changes_old_aircraft') THEN ALTER TABLE airline.flight_changes ADD CONSTRAINT fk_flight_changes_old_aircraft FOREIGN KEY (old_aircraft_id) REFERENCES airline.aircraft (aircraft_id) ON UPDATE CASCADE ON DELETE SET NULL; END IF; END$$;

-- flight_changes → aircraft (new) DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'flight_changes' AND constraint_name = 'fk_flight_changes_new_aircraft') THEN ALTER TABLE airline.flight_changes ADD CONSTRAINT fk_flight_changes_new_aircraft FOREIGN KEY (new_aircraft_id) REFERENCES airline.aircraft (aircraft_id) ON UPDATE CASCADE ON DELETE SET NULL; END IF; END$$;

-- 6. FLIGHT PERFORMANCE (BTS) → DIMENSIONS

-- flight_performance.airline_iata → airlines.iata_code DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'flight_performance' AND constraint_name = 'fk_fp_airline_iata') THEN ALTER TABLE

airline.flight_performance ADD CONSTRAINT fk_fp_airline_iata FOREIGN KEY (airline_iata) REFERENCES airline.airlines (iata_code) ON UPDATE CASCADE ON DELETE RESTRICT; END IF; END$$;

-- flight_performance.airport_iata → airports.iata_code DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM information_schema.table_constraints WHERE constraint_schema = 'airline' AND table_name = 'flight_performance' AND constraint_name = 'fk_fp_airport_iata') THEN ALTER TABLE airline.flight_performance ADD CONSTRAINT fk_fp_airport_iata FOREIGN KEY (airport_iata) REFERENCES airline.airports (iata_code) ON UPDATE CASCADE ON DELETE RESTRICT; END IF; END$$;

-- Optional: composite natural key on BTS snapshot DO $$ BEGIN IF NOT EXISTS (SELECT 1 FROM pg_indexes WHERE schemaname = 'airline' AND tablename = 'flight_performance' AND indexname = 'uq_fp_year_month_airline_airport') THEN CREATE UNIQUE INDEX uq_fp_year_month_airline_airport ON airline.flight_performance (year, month, airline_iata, airport_iata); END IF; END$$;

-- 7. DATA QUALITY / UNIQUENESS INDEXES

-- airlines: IATA & ICAO should be unique when present CREATE UNIQUE INDEX IF NOT EXISTS uq_airlines_iata ON airline.airlines (iata_code) WHERE iata_code IS NOT NULL;

CREATE UNIQUE INDEX IF NOT EXISTS uq_airlines_icao ON airline.airlines (icao_code) WHERE icao_code IS NOT NULL;

-- airports: IATA & ICAO should be unique when present CREATE UNIQUE INDEX IF NOT EXISTS uq_airports_iata ON airline.airports (iata_code) WHERE iata_code IS NOT NULL;

CREATE UNIQUE INDEX IF NOT EXISTS uq_airports_icao ON airline.airports (icao_code) WHERE icao_code IS NOT NULL;

-- aircraft: tail_number unique when present CREATE UNIQUE INDEX IF NOT EXISTS uq_aircraft_tail_number ON airline.aircraft (tail_number) WHERE tail_number IS NOT NULL;

-- passengers: email unique when present CREATE UNIQUE INDEX IF NOT EXISTS uq_passengers_email ON airline.passengers (email) WHERE email IS NOT NULL;

-- bookings: protect against accidental duplicates (already enforced earlier, but index is cheap) CREATE UNIQUE INDEX IF NOT EXISTS uq_bookings_passenger_flight ON airline.bookings (passenger_id, flight_id) WHERE flight_id IS NOT NULL;

-- 8. PERFORMANCE INDEXES FOR ANALYTICS

-- flights: common filters CREATE INDEX IF NOT EXISTS idx_flights_airline_date ON airline.flights (airline_id, flight_date);

CREATE INDEX IF NOT EXISTS idx_flights_origin_date ON airline.flights (origin_airport_id, flight_date);

CREATE INDEX IF NOT EXISTS idx_flights_dest_date ON airline.flights (destination_airport_id, flight_date);

-- bookings & payments CREATE INDEX IF NOT EXISTS idx_bookings_flight ON airline.bookings (flight_id);

CREATE INDEX IF NOT EXISTS idx_bookings_passenger ON airline.bookings (passenger_id);

CREATE INDEX IF NOT EXISTS idx_payments_booking ON airline.payments (booking_id);

-- loyalty & miles CREATE INDEX IF NOT EXISTS idx_loyalty_passenger ON airline.loyalty_accounts (passenger_id);

CREATE INDEX IF NOT EXISTS idx_miles_loyalty ON airline.miles_transactions (loyalty_id);

CREATE INDEX IF NOT EXISTS idx_miles_flight ON airline.miles_transactions (flight_id);

-- BTS flight_performance: typical slice/dice CREATE INDEX IF NOT EXISTS idx_fp_year_month_airline_airport ON airline.flight_performance (year, month, airline_iata, airport_iata);

-- 9. QUICK SANITY CHECK (OPTIONAL)

-- Summarize constraints per table (for debugging / documentation) SELECT tc.table_name, tc.constraint_name, tc.constraint_type FROM information_schema.table_constraints tc WHERE tc.constraint_schema = 'airline' ORDER BY tc.table_name, tc.constraint_type, tc.constraint_name; -- 05_validations.sql -- Phase 3: Validation and data quality checks -- Read-only checks: safe to re-run anytime.

SET search_path TO airline, public;

-- 1. ROW COUNTS BY TABLE

SELECT 'airlines' AS table_name, COUNT*() AS row_count FROM airline.airlines UNION ALL SELECT 'airports' AS table_name, COUNT()* AS row_count FROM airline.airports UNION ALL SELECT 'aircraft' AS table_name, COUNT*() AS row_count FROM airline.aircraft UNION ALL SELECT 'routes' AS table_name, COUNT()* AS row_count FROM airline.routes UNION ALL SELECT 'flights' AS table_name, COUNT*() AS row_count FROM airline.flights UNION ALL SELECT 'flight_performance' AS table_name, COUNT()* AS row_count FROM airline.flight_performance UNION ALL SELECT 'passengers' AS table_name, COUNT*() AS row_count FROM airline.passengers UNION ALL SELECT 'loyalty_accounts' AS table_name, COUNT()* AS row_count FROM airline.loyalty_accounts UNION ALL SELECT 'miles_transactions' AS table_name, COUNT*() AS row_count FROM airline.miles_transactions UNION ALL SELECT 'bookings' AS table_name, COUNT()* AS row_count FROM airline.bookings UNION ALL SELECT 'payments' AS table_name, COUNT(*) AS row_count FROM airline.payments ORDER BY table_name;

-- 2. FOREIGN-KEY “PROBLEM COUNTS” -- These should all be 0 if constraints + cleanup worked.

WITH flights_fk_issues AS (SELECT SUM(CASE WHEN a.airline_id IS NULL THEN 1 ELSE 0 END) AS flights_missing_airline, SUM(CASE WHEN o.airport_id IS NULL THEN 1 ELSE 0 END) AS flights_missing_origin_airport, SUM(CASE WHEN d.airport_id IS NULL THEN 1 ELSE 0 END) AS flights_missing_destination_airport FROM airline.flights f LEFT JOIN airline.airlines a ON f.airline_id = a.airline_id LEFT JOIN airline.airports o ON f.origin_airport_id = o.airport_id LEFT JOIN airline.airports d ON f.destination_airport_id = d.airport_id), bookings_fk_issues AS (SELECT SUM(CASE WHEN p.passenger_id IS NULL THEN 1 ELSE 0 END) AS bookings_missing_passenger, SUM(CASE WHEN fl.flight_id IS NULL THEN 1 ELSE 0 END) AS bookings_missing_flight FROM airline.bookings b LEFT JOIN

airline.passengers p ON b.passenger_id = p.passenger_id LEFT JOIN airline.flights fl ON b.flight_id = fl.flight_id), payments_fk_issues AS (SELECT SUM(CASE WHEN b.booking_id IS NULL THEN 1 ELSE 0 END) AS payments_missing_booking FROM airline.payments pay LEFT JOIN airline.bookings b ON pay.booking_id = b.booking_id) SELECT f.flights_missing_airline, f.flights_missing_origin_airport, f.flights_missing_destination_airport, b.bookings_missing_passenger, b.bookings_missing_flight, p.payments_missing_booking FROM flights_fk_issues f CROSS JOIN bookings_fk_issues b CROSS JOIN payments_fk_issues p;

-- 3. UNIQUENESS & BUSINESS KEY CHECKS

-- Airlines: IATA and ICAO should be unique when present SELECT 'airlines_iata' AS check_name, COUNT*() FILTER (WHERE iata_code IS NOT NULL) AS non_null_count, COUNT(DISTINCT iata_code) FILTER (WHERE iata_code IS NOT NULL) AS distinct_non_null FROM airline.airlines UNION ALL SELECT 'airlines_icao' AS check_name, COUNT()* FILTER (WHERE icao_code IS NOT NULL) AS non_null_count, COUNT(DISTINCT icao_code) FILTER (WHERE icao_code IS NOT NULL) AS distinct_non_null FROM airline.airlines ORDER BY check_name;

-- Airports: IATA and ICAO should be unique when present SELECT 'airports_iata' AS check_name, COUNT*() FILTER (WHERE iata_code IS NOT NULL) AS non_null_count, COUNT(DISTINCT iata_code) FILTER (WHERE iata_code IS NOT NULL) AS distinct_non_null FROM airline.airports UNION ALL SELECT 'airports_icao' AS check_name, COUNT()* FILTER (WHERE icao_code IS NOT NULL) AS non_null_count, COUNT(DISTINCT icao_code) FILTER (WHERE icao_code IS NOT NULL) AS distinct_non_null FROM airline.airports ORDER BY check_name;

-- Passengers: email uniqueness (when present) SELECT 'passengers_email' AS check_name, COUNT(*) FILTER (WHERE email IS NOT NULL) AS non_null_count, COUNT(DISTINCT email) FILTER (WHERE email IS NOT NULL) AS distinct_non_null FROM airline.passengers;

-- Loyalty: at most one loyalty account per passenger SELECT 'loyalty_per_passenger' AS check_name, COUNT(*) AS loyalty_rows, COUNT(DISTINCT passenger_id) AS distinct_passengers FROM airline.loyalty_accounts;

-- 4. BTS FLIGHT PERFORMANCE SANITY -- Quick shape of BTS summary data for documentation.

-- Coverage (years, months, airlines, airports) SELECT MIN(year) AS min_year, MAX(year) AS max_year, COUNT(DISTINCT year) AS years_covered, COUNT(DISTINCT month) AS months_covered, COUNT(DISTINCT airline_iata) AS airlines_in_bts, COUNT(DISTINCT airport_iata) AS airports_in_bts, COUNT(*) AS total_rows FROM airline.flight_performance;

-- Delay reason totals (basic sanity) SELECT SUM(arrivals) AS total_arrivals, SUM(arrivals_delayed_15min) AS total_arrivals_15min_delayed, SUM(arr_cancelled) AS total_arr_cancelled, SUM(arr_diverted) AS total_arr_diverted, SUM(total_arrival_delay_min) AS total_delay_minutes, SUM(carrier_delay) AS total_carrier_delay_min, SUM(weather_delay) AS total_weather_delay_min, SUM(nas_delay) AS total_nas_delay_min, SUM(security_delay) AS total_security_delay_min, SUM(late_aircraft_delay) AS total_late_aircraft_delay_min FROM airline.flight_performance;

-- 5. REVENUE & BOOKINGS SANITY

-- Distribution of base fares by fare_class SELECT fare_class, COUNT(*) AS bookings, MIN(base_price_usd) AS min_price, MAX(base_price_usd) AS max_price, AVG(base_price_usd) AS avg_price FROM airline.bookings GROUP BY fare_class ORDER BY fare_class;

-- Payments vs base_price: simple ratio stats SELECT COUNT(*) AS payment_rows, MIN(amount_usd) AS min_payment, MAX(amount_usd) AS max_payment, AVG(amount_usd) AS avg_payment FROM airline.payments;

-- Compare bookings vs payments by booking_id SELECT 'bookings_without_payments' AS metric, COUNT*() AS count_rows FROM airline.bookings b LEFT JOIN airline.payments p ON b.booking_id = p.booking_id WHERE p.booking_id IS NULL UNION ALL SELECT 'payments_without_bookings' AS metric, COUNT()* AS count_rows FROM airline.payments p LEFT JOIN airline.bookings b ON p.booking_id = b.booking_id WHERE b.booking_id IS NULL;

-- 6. HIGH-LEVEL JOIN SANITY: -- How many flights have at least one booking?

SELECT COUNT(DISTINCT f.flight_id) AS flights_with_bookings, (SELECT COUNT(*) FROM airline.flights) AS total_flights FROM airline.flights f JOIN airline.bookings b ON f.flight_id = b.flight_id;

-- Make sure we’re defaulting to the airline schema in this session SET search_path TO airline, public;

-- Create table inside airline schema CREATE TABLE airline.flight_performance (snapshot_id TEXT PRIMARY KEY, -- e.g., 2024_01_MQ_EWR year INT NOT NULL, month INT NOT NULL, airline_iata VARCHAR(5) NOT NULL, airport_iata VARCHAR(5) NOT NULL, arrivals INT, arrivals_delayed_15min INT, arr_cancelled INT, arr_diverted INT, total_arrival_delay_min DOUBLE PRECISION, carrier_delay DOUBLE PRECISION, weather_delay DOUBLE PRECISION, nas_delay DOUBLE PRECISION, security_delay DOUBLE PRECISION, late_aircraft_delay DOUBLE PRECISION, CONSTRAINT uq_fp UNIQUE (year, month, airline_iata, airport_iata), CONSTRAINT fk_fp_airline FOREIGN KEY (airline_iata) REFERENCES airline.airlines(iata_code), CONSTRAINT fk_fp_airport FOREIGN KEY (airport_iata) REFERENCES airline.airports(iata_code));

-- Helpful indexes for lookups CREATE INDEX IF NOT EXISTS idx_fp_month ON airline.flight_performance (year, month); CREATE INDEX IF NOT EXISTS idx_fp_airline ON airline.flight_performance (airline_iata); CREATE INDEX IF NOT EXISTS idx_fp_airport ON airline.flight_performance (airport_iata);

```
# etl/load_bts.py
import os, pandas as pd
from sqlalchemy import create_engine, text
from dotenv import load_dotenv

load_dotenv()
engine = create_engine(os.environ["AIRLINE_DB_DSN"], pool_pre_ping=True)

def clean_str(s):
    if pd.isna(s): return None
    s = str(s).strip()
    return s or None

def run():
    # Start with a small subset CSV (1–3 months)
    bts = pd.read_csv("data/bts_flights_2024Q1.csv")

    # Map common BTS columns -> normalized names
    rename_map = {
        "MKT_UNIQUE_CARRIER":"airline_code",
        "OP_UNIQUE_CARRIER":"airline_code",   # fallback
        "MKT_CARRIER_FL_NUM":"flight_number",
        "FL_DATE":"flight_date",
        "ORIGIN":"origin_iata",
        "DEST":"dest_iata",
        "CRS_DEP_TIME_UTC":"sched_dep_utc",
        "CRS_ARR_TIME_UTC":"sched_arr_utc",
        "DEP_TIME_UTC":"actual_dep_utc",
        "ARR_TIME_UTC":"actual_arr_utc",
        "ARR_DELAY":"delay_minutes",
        "CANCELLED":"cancelled"
    }
    bts = bts.rename(columns={c:rename_map.get(c,c) for c in bts.columns})

    # Clean/convert types
    for c in ["airline_code","flight_number","origin_iata","dest_iata"]:
        if c in bts.columns: bts[c] = bts[c].map(clean_str)
    bts["flight_date"] = pd.to_datetime(bts["flight_date"], errors="coerce").dt.date
    for col in ["sched_dep_utc","sched_arr_utc","actual_dep_utc","actual_arr_utc"]:
        if col in bts.columns: bts[col] = pd.to_datetime(bts[col], errors="coerce")
    bts["delay_minutes"] = pd.to_numeric(bts.get("delay_minutes"), errors="coerce")
    # Normalize status
    if "cancelled" in bts.columns:
        bts["status"] = bts["cancelled"].map({1:"Cancelled", 0:"Arrived"})
    bts["status"] = bts["status"].fillna("Scheduled")

    with engine.begin() as con:
        con.execute(text("""
            CREATE TEMP TABLE tmp_bts(
              airline_code text,
              flight_number text,
              flight_date date,
              origin_iata text,
              dest_iata text,
              sched_dep_utc timestamp,
              sched_arr_utc timestamp,
              actual_dep_utc timestamp,
              actual_arr_utc timestamp,
              delay_minutes int,
              status text
            ) ON COMMIT DROP;
        """))
```

```
        cols = ["airline_code","flight_number","flight_date","origin_iata","dest_iata",
                "sched_dep_utc","sched_arr_utc","actual_dep_utc","actual_arr_utc",
                "delay_minutes","status"]
        # keep only existing columns
        cols = [c for c in cols if c in bts.columns]
        bts[cols].to_sql("tmp_bts", con, index=False, if_exists="append")

        con.execute(text("""
            INSERT INTO airline.flights(
              airline_id, aircraft_id, route_id,
              origin_airport_id, destination_airport_id,
              flight_number, flight_date,
              scheduled_departure_utc, scheduled_arrival_utc,
              actual_departure_utc, actual_arrival_utc,
              delay_minutes, delay_cause, status
            )
            SELECT
              al.airline_id,
              NULL::bigint AS aircraft_id,
              r.route_id,
              ao.airport_id, ad.airport_id,
              t.flight_number, t.flight_date,
              t.sched_dep_utc, t.sched_arr_utc,
              t.actual_dep_utc, t.actual_arr_utc,
              t.delay_minutes,
              NULL::text AS delay_cause,
              CASE
                WHEN LOWER(t.status) IN
('scheduled','departed','arrived','cancelled','diverted')
                  THEN INITCAP(LOWER(t.status))
                ELSE 'Scheduled'
              END::flight_status
            FROM tmp_bts t
            JOIN airline.airlines al ON al.iata_code = TRIM(t.airline_code)
            JOIN airline.airports ao ON ao.iata_code = TRIM(t.origin_iata)
            JOIN airline.airports ad ON ad.iata_code = TRIM(t.dest_iata)
            LEFT JOIN airline.routes r
              ON r.airline_id = al.airline_id
             AND r.origin_airport_id = ao.airport_id
             AND r.destination_airport_id = ad.airport_id
            ON CONFLICT ON CONSTRAINT uq_flight_instance DO NOTHING;
        """))

    print("BTS -> production flights complete.")

if __name__ == "__main__":
    run()
```

```
import pandas as pd
from pathlib import Path

# Load raw CSV
df = pd.read_csv(Path("/Users/gracepolito/Desktop/Master of Data Science/691 Applied Data
Science/Airline Business Intelligence Database/data/bts_flights_2024.csv"))

# Normalize column names
df.columns = df.columns.str.lower().str.strip()

# Keep only needed fields
cols = [
    "year", "month", "carrier", "carrier_name", "airport", "airport_name",
    "arr_flights", "arr_del15", "arr_cancelled", "arr_diverted",
    "arr_delay", "carrier_delay", "weather_delay", "nas_delay", "security_delay",
"late_aircraft_delay"
]
df = df[cols]

# Rename columns for DB consistency
df = df.rename(columns={
    "carrier": "airline_iata",
    "airport": "airport_iata",
    "arr_flights": "arrivals",
    "arr_del15": "arrivals_delayed_15min",
    "arr_delay": "total_arrival_delay_min"
})

# Fill NaNs with 0 for numeric delay values
num_cols = [c for c in df.columns if "delay" in c or "arrivals" in c]
df[num_cols] = df[num_cols].fillna(0)

# Add a unique key for loading
df["snapshot_id"] = (
    df["year"].astype(str) + "_" +
    df["month"].astype(str).str.zfill(2) + "_" +
    df["airline_iata"] + "_" + df["airport_iata"]
)

# Export cleaned version
df.to_csv("/Users/gracepolito/Desktop/Master of Data Science/691 Applied Data
Science/Airline Business Intelligence Database/data/bts_cleaned.csv", index=False)
print("✅ Cleaned data saved to data/bts_cleaned.csv")
```

```
import os
import random
from datetime import datetime, date, timedelta

from sqlalchemy import create_engine, text


# ---------- DB CONNECTION HELPERS ----------

def get_db_url() -> str:
    url = os.getenv("DATABASE_URL") or os.getenv("AIRLINE_DB_DSN")
    if not url:
        raise RuntimeError(
            "Set either DATABASE_URL or AIRLINE_DB_DSN in your environment / .env.\n"
            "Example: postgresql+psycopg2://postgres:password@localhost:5432/airline_bi"
        )
    return url


ENGINE = create_engine(get_db_url(), future=True, pool_pre_ping=True)


# ---------- FETCH LOOKUP DATA ----------

def fetch_airports_and_airlines(conn):
    airlines = conn.execute(
        text(
            """
            SELECT airline_id, iata_code
            FROM airline.airlines
            WHERE iata_code IS NOT NULL
            """
        )
    ).mappings().all()

    if not airlines:
        raise RuntimeError(
            "No rows found in airline.airlines. "
            "Run etl/load_openflights.py first."
        )

    airports = conn.execute(
        text(
            """
            SELECT airport_id, iata_code
            FROM airline.airports
            WHERE iata_code IS NOT NULL
            """
        )
    ).mappings().all()

    if not airports:
        raise RuntimeError(
            "No rows found in airline.airports. "
            "Run etl/load_openflights.py first."
        )

    return airlines, airports


def fetch_flight_status_values(conn):
    """
    Figure out the ENUM type used by flights.status and get its labels.
```

```
    This avoids hard-coding enum names like 'Completed' vs 'completed'.
    """
    type_name = conn.execute(
        text(
            """
            SELECT udt_name
            FROM information_schema.columns
            WHERE table_schema = 'airline'
              AND table_name   = 'flights'
              AND column_name  = 'status'
            """
        )
    ).scalar_one()

    rows = conn.execute(
        text(
            """
            SELECT e.enumlabel
            FROM pg_enum e
            JOIN pg_type t ON e.enumtypid = t.oid
            WHERE t.typname = :tname
            ORDER BY e.enumsortorder
            """
        ),
        {"tname": type_name},
    ).all()

    statuses = [r[0] for r in rows]
    if not statuses:
        raise RuntimeError("Could not read enum labels for flights.status.")
    return statuses


# ---------- GENERATE SYNTHETIC FLIGHTS ----------

def generate_flights(airlines, airports, statuses, n=5000, seed=42):
    random.seed(seed)

    flights = []

    now = datetime.utcnow()
    start_date = date(now.year - 1, 1, 1)
    end_date = date(now.year + 1, 12, 31)
    total_days = (end_date - start_date).days

    for _ in range(n):
        al = random.choice(airlines)
        origin, dest = random.sample(airports, 2)

        # Date + basic schedule
        day_offset = random.randrange(total_days)
        flight_date = start_date + timedelta(days=day_offset)

        dep_hour = random.randint(5, 22)
        dep_min = random.choice([0, 15, 30, 45])
        dep_dt = datetime.combine(flight_date, datetime.min.time()) + timedelta(
            hours=dep_hour, minutes=dep_min
        )

        block_minutes = random.randint(60, 6 * 60)  # 1–6 hours
        arr_dt = dep_dt + timedelta(minutes=block_minutes)

        status = random.choice(statuses)
        status_lower = status.lower()
```

```
        # Delay / actual times logic
        if status_lower.startswith("cancel"):
            delay = random.randint(60, 300)
            delay_cause = "Cancellation"
            actual_dep = None
            actual_arr = None
        elif status_lower.startswith("sched"):
            delay = 0
            delay_cause = None
            actual_dep = None
            actual_arr = None
        else:
            delay = random.randint(0, 180)
            delay_cause = random.choice(
                ["Weather", "Crew", "Maintenance", "ATC", "Late inbound", None]
            )
            actual_dep = dep_dt + timedelta(minutes=delay)
            actual_arr = arr_dt + timedelta(minutes=delay)

        flight_number = f"{al['iata_code']}{random.randint(1, 9999):04d}"

        flights.append(
            {
                "airline_id": al["airline_id"],
                "origin_airport_id": origin["airport_id"],
                "destination_airport_id": dest["airport_id"],
                "flight_number": flight_number,
                "flight_date": flight_date,
                "scheduled_departure_utc": dep_dt,
                "scheduled_arrival_utc": arr_dt,
                "actual_departure_utc": actual_dep,
                "actual_arrival_utc": actual_arr,
                "delay_minutes": delay,
                "delay_cause": delay_cause,
                "status": status,
            }
        )

    return flights


def insert_flights(conn, flights):
    if not flights:
        print("⚠️ No flights to insert.")
        return

    conn.execute(
        text(
            """
            INSERT INTO airline.flights (
                airline_id,
                origin_airport_id,
                destination_airport_id,
                flight_number,
                flight_date,
                scheduled_departure_utc,
                scheduled_arrival_utc,
                actual_departure_utc,
                actual_arrival_utc,
                delay_minutes,
                delay_cause,
                status
            )
            VALUES (
                :airline_id,
```

```
                :origin_airport_id,
                :destination_airport_id,
                :flight_number,
                :flight_date,
                :scheduled_departure_utc,
                :scheduled_arrival_utc,
                :actual_departure_utc,
                :actual_arrival_utc,
                :delay_minutes,
                :delay_cause,
                :status
            );
            """
        ),
        flights,
    )


# ---------- MAIN ----------

def main():
    with ENGINE.begin() as conn:
        cols = conn.execute(
            text(
                """
                SELECT column_name
                FROM information_schema.columns
                WHERE table_schema = 'airline'
                  AND table_name   = 'flights'
                ORDER BY ordinal_position
                """
            )
        ).fetchall()
        print("🔍 flights columns:", [c[0] for c in cols])

        airlines, airports = fetch_airports_and_airlines(conn)
        print(f"🌍 Using {len(airports)} airports and {len(airlines)} airlines.")

        statuses = fetch_flight_status_values(conn)
        print("📊 flight_status enum values:", statuses)

        flights = generate_flights(airlines, airports, statuses, n=5000)
        print(f"✈️ Generated {len(flights)} synthetic flights.")

        insert_flights(conn, flights)
        print("✅ Synthetic flights inserted.")


if __name__ == "__main__":
    main()
```

```
r"""
Load OpenFlights reference data (airports + airlines) into the airline schema.

Assumptions (matching your ERD):
- airline.airports(
    airport_id serial PK,
    iata_code varchar(3) UNIQUE,
    icao_code varchar(4) UNIQUE,
    name text,
    city text,
    country text,
    latitude double precision,
    longitude double precision,
    timezone text
  )

- airline.airlines(
    airline_id serial PK,
    name text,
    iata_code varchar(3),
    icao_code varchar(3),
    country varchar(3)
  )

We:
- Read the raw OpenFlights CSVs (no header, fixed column layout).
- Clean values and truncate anything that might violate length constraints.
- Skip obviously bad / placeholder values (e.g., "\N").
"""

import os
from pathlib import Path

import pandas as pd
from sqlalchemy import create_engine, text


# ---------------------------------------------------------------------------
# DB URL helper
# ---------------------------------------------------------------------------

def get_db_url() -> str:
    """
    Look up the database URL from .env / shell.

    Prefers DATABASE_URL, falls back to AIRLINE_DB_DSN.
    """
    url = os.environ.get("DATABASE_URL") or os.environ.get("AIRLINE_DB_DSN")
    if not url:
        raise RuntimeError(
            "Set either DATABASE_URL or AIRLINE_DB_DSN in your environment / .env.\n"
            "Example: postgres://user:password@localhost:5432/airline_bi"
        )
    return url


ENGINE = create_engine(get_db_url(), future=True, pool_pre_ping=True)

# ---------------------------------------------------------------------------
# Paths
# ---------------------------------------------------------------------------

PROJECT_ROOT = Path(__file__).resolve().parents[1]
```

```
DATA_DIR = PROJECT_ROOT / "data"

AIRPORTS_CSV = DATA_DIR / "openflights_airports.csv"
AIRLINES_CSV = DATA_DIR / "openflights_airlines.csv"


# ---------------------------------------------------------------------------
# Helpers
# ---------------------------------------------------------------------------

def _clean_str(value):
    """Return a stripped string or None for NaN / placeholders."""
    if pd.isna(value):
        return None
    s = str(value).strip()
    if not s or s == r"\N":
        return None
    return s


# ---------------------------------------------------------------------------
# Airports
# ---------------------------------------------------------------------------

def load_airports() -> None:
    """
    Load airports from the standard OpenFlights airports.dat layout:

    0: Airport ID
    1: Name
    2: City
    3: Country
    4: IATA
    5: ICAO
    6: Latitude
    7: Longitude
    8: Altitude
    9: Timezone (hours from UTC)
    10: DST
    11: Tz database time zone
    12: type
    13: source
    """
    print(f"🔷 Loading OpenFlights airports from: {AIRPORTS_CSV}")

    df = pd.read_csv(AIRPORTS_CSV, header=None, dtype=str)

    rows = []
    for _, row in df.iterrows():
        name = _clean_str(row[1])
        city = _clean_str(row[2])
        country = _clean_str(row[3])
        iata = _clean_str(row[4])
        icao = _clean_str(row[5])

        # Skip unusable rows
        if not iata and not icao:
            continue

        if iata:
            iata = iata[:3].upper()
        if icao:
            icao = icao[:4].upper()

        # Latitude & Longitude
```

```
            try:
                lat = float(row[6]) if row[6] not in (None, "", r"\N") else None
            except:
                lat = None
            try:
                lon = float(row[7]) if row[7] not in (None, "", r"\N") else None
            except:
                lon = None

            tz = _clean_str(row[11]) or _clean_str(row[9])

            rows.append(
                dict(
                    iata=iata,
                    icao=icao,
                    name=name,
                    city=city,
                    country=country,
                    lat=lat,
                    lon=lon,
                    tz=tz,
                )
            )

        if not rows:
            print("⚠️ No airport rows to insert (after filtering).")
            return

        with ENGINE.begin() as con:
            con.execute(
                text(
                    """
                    INSERT INTO airline.airports (
                        iata_code, icao_code, name, city, country,
                        latitude, longitude, timezone
                    )
                    VALUES (
                        :iata, :icao, :name, :city, :country,
                        :lat, :lon, :tz
                    )
                    ON CONFLICT (iata_code) DO NOTHING;
                    """
                ),
                rows,
            )

        print(f"✅ Airports loaded: {len(rows)} candidate rows inserted (conflicts skipped).")


    # ---------------------------------------------------------------------------
    # Airlines
    # ---------------------------------------------------------------------------

    def load_airlines() -> None:
        """
        Load airlines from the standard OpenFlights airlines.dat layout:

        0: Airline ID
        1: Name
        2: Alias
        3: IATA
        4: ICAO
        5: Callsign
        6: Country
```

```
    7: Active
    """
    print(f"🔷 Loading OpenFlights airlines from: {AIRLINES_CSV}")

    df = pd.read_csv(AIRLINES_CSV, header=None, dtype=str)

    rows = []
    for _, row in df.iterrows():
        name = _clean_str(row[1])
        if not name:
            continue

        iata = _clean_str(row[3])
        icao = _clean_str(row[4])
        country_full = _clean_str(row[6])

        # Truncate to schema limits
        if iata:
            iata = iata[:3].upper()
        if icao:
            icao = icao[:3].upper()
        country = country_full[:3].upper() if country_full else None

        rows.append(
            dict(
                name=name,
                iata=iata,
                icao=icao,
                country=country,
            )
        )

    if not rows:
        print("⚠️ No airline rows to insert (after filtering).")
        return

    with ENGINE.begin() as con:
        con.execute(
            text(
                """
                INSERT INTO airline.airlines (
                    name, iata_code, icao_code, country
                )
                VALUES (:name, :iata, :icao, :country)
                ON CONFLICT DO NOTHING;
                """
            ),
            rows,
        )

    print(f"✅ Airlines loaded: {len(rows)} candidate rows inserted (conflicts skipped).")


# ---------------------------------------------------------------------------
# Entrypoint
# ---------------------------------------------------------------------------

def run() -> None:
    load_airports()
    load_airlines()
    print("🎉 OpenFlights reference tables loaded.")
```

```
if __name__ == "__main__":
    run()
```

```
import os
import uuid
from datetime import datetime

import pandas as pd
from sqlalchemy import create_engine, text

# ---------------------------------------------------------------------
# DB connection helper — same pattern as load_openflights & synth_flights
# ---------------------------------------------------------------------
def get_db_url() -> str:
    """
    Resolve the database URL from environment variables.

    Prefer DATABASE_URL (what you're using now), but fall back
    to AIRLINE_DB_DSN if it's present.
    """
    url = os.getenv("DATABASE_URL") or os.getenv("AIRLINE_DB_DSN")
    if not url:
        raise RuntimeError(
            "Set either DATABASE_URL or AIRLINE_DB_DSN in your environment / .env.\n"
            "Example: postgresql+psycopg2://postgres:password@localhost:5432/airline_bi"
        )
    return url


ENGINE = create_engine(get_db_url(), future=True, pool_pre_ping=True)


RAW_PATH = "data/bts_cleaned.csv"
CHUNK = 200_000

def normalize_chunk(df: pd.DataFrame) -> pd.DataFrame:
    """
    Normalize a BTS aggregate chunk into the schema expected by airline.flight_performance.

    Supports two column naming schemes:

    1) Original aggregate (what the script used to expect):
       year, month, carrier, airport, arr_flights, arr_del15, arr_cancelled,
       arr_diverted, arr_delay, carrier_delay, weather_delay, nas_delay,
       security_delay, late_aircraft_delay

    2) Already-final names (often from bts_cleaned.csv):
       year, month, airline_iata, airport_iata, arrivals, arrivals_delayed_15min,
       arr_cancelled, arr_diverted, total_arrival_delay_min, carrier_delay,
       weather_delay, nas_delay, security_delay, late_aircraft_delay
    """

    cols = set(df.columns)

    # Case 1: old names — map them to final names
    if {"carrier", "airport", "arr_flights", "arr_del15", "arr_delay"}.issubset(cols):
        df = df.rename(
            columns={
                "carrier": "airline_iata",
                "airport": "airport_iata",
                "arr_flights": "arrivals",
                "arr_del15": "arrivals_delayed_15min",
                "arr_delay": "total_arrival_delay_min",
            }
        )

    # Case 2: already-final names — nothing to rename
```

```
    elif {
        "year",
        "month",
        "airline_iata",
        "airport_iata",
        "arrivals",
        "arrivals_delayed_15min",
        "arr_cancelled",
        "arr_diverted",
        "total_arrival_delay_min",
        "carrier_delay",
        "weather_delay",
        "nas_delay",
        "security_delay",
        "late_aircraft_delay",
    }.issubset(cols):
        pass

    else:
        raise KeyError(
            f"Unexpected BTS columns: {sorted(df.columns.tolist())}. "
            "Expected either ['carrier','airport','arr_flights','arr_del15','arr_delay',
...] "
            "or ['airline_iata','airport_iata','arrivals','arrivals_delayed_15min', ...]."
        )

    # Now select the canonical columns
    keep = [
        "year",
        "month",
        "airline_iata",
        "airport_iata",
        "arrivals",
        "arrivals_delayed_15min",
        "arr_cancelled",
        "arr_diverted",
        "total_arrival_delay_min",
        "carrier_delay",
        "weather_delay",
        "nas_delay",
        "security_delay",
        "late_aircraft_delay",
    ]

    df = df[keep].copy()

    # Clean IATA codes
    df["airline_iata"] = df["airline_iata"].astype(str).str.strip().str.upper()
    df["airport_iata"] = df["airport_iata"].astype(str).str.strip().str.upper()

    # Ensure numeric columns are numeric (fill NaN with 0)
    numeric_cols = [
        "arrivals",
        "arrivals_delayed_15min",
        "arr_cancelled",
        "arr_diverted",
        "total_arrival_delay_min",
        "carrier_delay",
        "weather_delay",
        "nas_delay",
        "security_delay",
        "late_aircraft_delay",
    ]
    for col in numeric_cols:
        df[col] = pd.to_numeric(df[col], errors="coerce").fillna(0)
```

```
    # Build a snapshot_id that matches your flight_performance PK
    df["snapshot_id"] = (
        df["year"].astype(int).astype(str).str.zfill(4)
        + "-"
        + df["month"].astype(int).astype(str).str.zfill(2)
        + "-"
        + df["airline_iata"]
        + "-"
        + df["airport_iata"]
    )

    # Order columns to match the INSERT INTO tmp_fp
    ordered = [
        "snapshot_id",
        "year",
        "month",
        "airline_iata",
        "airport_iata",
        "arrivals",
        "arrivals_delayed_15min",
        "arr_cancelled",
        "arr_diverted",
        "total_arrival_delay_min",
        "carrier_delay",
        "weather_delay",
        "nas_delay",
        "security_delay",
        "late_aircraft_delay",
    ]
    return df[ordered]


def ensure_table():
    with ENGINE.begin() as con:
        con.execute(text("""SET search_path TO airline, public;"""))
        con.execute(text("""
        CREATE TABLE IF NOT EXISTS airline.flight_performance (
            snapshot_id TEXT PRIMARY KEY,
            year INT NOT NULL,
            month INT NOT NULL,
            airline_iata VARCHAR(5) NOT NULL,
            airport_iata VARCHAR(5) NOT NULL,
            arrivals INT,
            arrivals_delayed_15min INT,
            arr_cancelled INT,
            arr_diverted INT,
            total_arrival_delay_min DOUBLE PRECISION,
            carrier_delay DOUBLE PRECISION,
            weather_delay DOUBLE PRECISION,
            nas_delay DOUBLE PRECISION,
            security_delay DOUBLE PRECISION,
            late_aircraft_delay DOUBLE PRECISION,
            CONSTRAINT uq_fp UNIQUE (year, month, airline_iata, airport_iata),
            CONSTRAINT fk_fp_airline  FOREIGN KEY (airline_iata)  REFERENCES
airline.airlines(iata_code),
            CONSTRAINT fk_fp_airport  FOREIGN KEY (airport_iata)  REFERENCES
airline.airports(iata_code)
        );
        CREATE INDEX IF NOT EXISTS idx_fp_month ON airline.flight_performance (year, month);
        CREATE INDEX IF NOT EXISTS idx_fp_airline ON airline.flight_performance
(airline_iata);
        CREATE INDEX IF NOT EXISTS idx_fp_airport ON airline.flight_performance
(airport_iata);
        """))
```

```
def load():
    ensure_table()

    reader = pd.read_csv(RAW_PATH, chunksize=CHUNK)
    for i, chunk in enumerate(reader, start=1):
        df = normalize_chunk(chunk)

        with ENGINE.begin() as con:
            # temp table for conflict-safe upsert
            con.execute(text("""
                CREATE TEMP TABLE tmp_fp(
                    snapshot_id TEXT,
                    year INT,
                    month INT,
                    airline_iata VARCHAR(5),
                    airport_iata VARCHAR(5),
                    arrivals INT,
                    arrivals_delayed_15min INT,
                    arr_cancelled INT,
                    arr_diverted INT,
                    total_arrival_delay_min DOUBLE PRECISION,
                    carrier_delay DOUBLE PRECISION,
                    weather_delay DOUBLE PRECISION,
                    nas_delay DOUBLE PRECISION,
                    security_delay DOUBLE PRECISION,
                    late_aircraft_delay DOUBLE PRECISION
                ) ON COMMIT DROP;
            """))
            df.to_sql("tmp_fp", con, if_exists="append", index=False)

            con.execute(text("""
                INSERT INTO airline.flight_performance AS fp(
                    snapshot_id, year, month, airline_iata, airport_iata,
                    arrivals, arrivals_delayed_15min, arr_cancelled, arr_diverted,
                    total_arrival_delay_min, carrier_delay, weather_delay, nas_delay,
security_delay, late_aircraft_delay
                )
                SELECT
                    t.snapshot_id, t.year, t.month, t.airline_iata, t.airport_iata,
                    t.arrivals, t.arrivals_delayed_15min, t.arr_cancelled, t.arr_diverted,
                    t.total_arrival_delay_min, t.carrier_delay, t.weather_delay,
                    t.nas_delay, t.security_delay, t.late_aircraft_delay
                FROM tmp_fp t
                JOIN airline.airports a
                  ON t.airport_iata = a.iata_code
                ON CONFLICT (snapshot_id) DO NOTHING;
            """))

    print(f"Chunk {i} inserted.")

print("✅ BTS performance load complete.")

if __name__ == "__main__":
    load()
```

```
"""
Generate synthetic customers and loyalty data for the Airline BI database.

Populates:
  - airline.passengers
  - airline.loyalty_accounts
  - airline.miles_transactions

Assumes:
  - DATABASE_URL or AIRLINE_DB_DSN is set in the environment
  - Enums:
        airline.loyalty_tier
        airline.miles_txn_type
  - Flights already exist in airline.flights
"""

import os
import random
from datetime import datetime, timedelta

from faker import Faker
from sqlalchemy import create_engine, text


# ---------------------------------------------------------------------------
# DB connection helper
# ---------------------------------------------------------------------------

def get_db_url() -> str:
    """
    Return a SQLAlchemy connection URL from env vars.

    Priority:
      1. DATABASE_URL
      2. AIRLINE_DB_DSN

    Raises if neither is set.
    """
    url = os.environ.get("DATABASE_URL") or os.environ.get("AIRLINE_DB_DSN")
    if not url:
        raise RuntimeError(
            "Set DATABASE_URL or AIRLINE_DB_DSN in your environment / .env.\n"
            "Example: postgresql+psycopg2://postgres:password@localhost:5432/airline_bi"
        )
    return url


ENGINE = create_engine(get_db_url(), future=True, pool_pre_ping=True)

faker = Faker("en_US")
Faker.seed(42)
random.seed(42)


# ---------------------------------------------------------------------------
# Small helpers
# ---------------------------------------------------------------------------

def random_datetime(start_year: int = 2022, end_year: int = 2026) -> datetime:
    start = datetime(start_year, 1, 1)
    end = datetime(end_year, 12, 31, 23, 59, 59)
    delta = end - start
    offset_seconds = random.randint(0, int(delta.total_seconds()))
    return start + timedelta(seconds=offset_seconds)
```

```
def age_to_group(age: int) -> str:
    if age < 26:
        return "18-25"
    elif age < 36:
        return "26-35"
    elif age < 46:
        return "36-45"
    elif age < 61:
        return "46-60"
    else:
        return "60+"


# ------------------------------------------------------------------------
# Fetch reference data (enums, flights, etc.)
# ------------------------------------------------------------------------

def fetch_enum_values(conn, enum_name: str):
    """
    Fetch enum labels from Postgres, e.g.:

        SELECT unnest(enum_range(NULL::airline.loyalty_tier));

    Returns a list of strings.
    """
    sql = f"SELECT unnest(enum_range(NULL::{enum_name}));"
    rows = conn.execute(text(sql)).all()
    return [r[0] for r in rows]


def fetch_flight_ids(conn):
    rows = conn.execute(text("SELECT flight_id FROM airline.flights;")).all()
    return [r[0] for r in rows]


def fetch_passenger_ids(conn):
    rows = conn.execute(text("SELECT passenger_id FROM airline.passengers;")).all()
    return [r[0] for r in rows]


def fetch_loyalty_ids(conn):
    rows = conn.execute(text("SELECT loyalty_id FROM airline.loyalty_accounts;")).all()
    return [r[0] for r in rows]


# ------------------------------------------------------------------------
# Generators
# ------------------------------------------------------------------------

def generate_passenger_rows(n: int):
    genders = ["F", "M", "X"]
    rows = []

    for _ in range(n):
        first_name = faker.first_name()
        last_name = faker.last_name()
        email = faker.unique.email()
        gender = random.choice(genders)
        age = random.randint(18, 80)
        age_group = age_to_group(age)
        # US state or country name; it's just text so we can mix
        if random.random() < 0.7:
            state_or_country = faker.state_abbr()
```

```
        else:
            state_or_country = faker.country()
        created_at = random_datetime(2022, 2024)

        rows.append(
            {
                "first_name": first_name,
                "last_name": last_name,
                "email": email,
                "gender": gender,
                "age_group": age_group,
                "state_or_country": state_or_country,
                "created_at": created_at,
            }
        )

    return rows


def generate_loyalty_rows(passenger_ids, tiers, loyalty_ratio=0.6):
    """
    Give a loyalty account to ~loyalty_ratio of passengers.
    """
    rows = []
    take = int(len(passenger_ids) * loyalty_ratio)
    chosen = set(random.sample(passenger_ids, take)) if passenger_ids else set()

    for pid in chosen:
        tier = random.choice(tiers) if tiers else None
        miles_balance = random.randint(0, 100_000)
        enrolled_at = random_datetime(2022, 2025)
        rows.append(
            {
                "passenger_id": pid,
                "tier": tier,
                "miles_balance": miles_balance,
                "enrolled_at": enrolled_at,
            }
        )

    return rows


def generate_miles_txn_rows(loyalty_ids, flight_ids, txn_types):
    """
    Generate a handful of miles transactions per loyalty account.
    """
    if not loyalty_ids or not flight_ids or not txn_types:
        return []

    rows = []
    for lid in loyalty_ids:
        num_txns = random.randint(1, 6)
        for _ in range(num_txns):
            txn_type = random.choice(txn_types)

            # Heuristic: if the enum name hints at redemption, make it negative.
            lower = txn_type.lower()
            if "redeem" in lower or "spend" in lower or "debit" in lower:
                miles_delta = -random.randint(500, 50_000)
            else:
                miles_delta = random.randint(500, 50_000)

            flight_id = random.choice(flight_ids)
            posted_at = random_datetime(2023, 2026)
```

```
            rows.append(
                {
                    "loyalty_id": lid,
                    "flight_id": flight_id,
                    "txn_type": txn_type,
                    "miles_delta": miles_delta,
                    "posted_at": posted_at,
                }
            )

    return rows


# ------------------------------------------------------------------------
# Inserts
# ------------------------------------------------------------------------

def insert_passengers(conn, rows):
    if not rows:
        print("⚠️ No passenger rows generated.")
        return

    conn.execute(
        text(
            """
            INSERT INTO airline.passengers (
                first_name,
                last_name,
                email,
                gender,
                age_group,
                state_or_country,
                created_at
            )
            VALUES (
                :first_name,
                :last_name,
                :email,
                :gender,
                :age_group,
                :state_or_country,
                :created_at
            );
            """
        ),
        rows,
    )
    print(f"✅ Inserted {len(rows)} passengers.")


def insert_loyalty_accounts(conn, rows):
    if not rows:
        print("⚠️ No loyalty accounts generated.")
        return

    conn.execute(
        text(
            """
            INSERT INTO airline.loyalty_accounts (
                passenger_id,
                tier,
                miles_balance,
                enrolled_at
```

```
            )
            VALUES (
                :passenger_id,
                :tier,
                :miles_balance,
                :enrolled_at
            );
            """
        ),
        rows,
    )
    print(f"✅ Inserted {len(rows)} loyalty accounts.")


def insert_miles_transactions(conn, rows):
    if not rows:
        print("⚠️ No miles transactions generated.")
        return

    conn.execute(
        text(
            """
            INSERT INTO airline.miles_transactions (
                loyalty_id,
                flight_id,
                txn_type,
                miles_delta,
                posted_at
            )
            VALUES (
                :loyalty_id,
                :flight_id,
                :txn_type,
                :miles_delta,
                :posted_at
            );
            """
        ),
        rows,
    )
    print(f"✅ Inserted {len(rows)} miles transactions.")


# ----------------------------------------------------------------------
# Main
# ----------------------------------------------------------------------

def main():
    NUM_PASSENGERS = 5000

    print("🔗 Connecting to database...")
    with ENGINE.begin() as conn:
        # 1) Reference data
        print("📩 Fetching enum values and flights...")

        try:
            loyalty_tiers = fetch_enum_values(conn, "airline.loyalty_tier")
        except Exception as exc:  # noqa: BLE001
            print(f"⚠️ Could not fetch airline.loyalty_tier enum values: {exc}")
            loyalty_tiers = []

        try:
            miles_txn_types = fetch_enum_values(conn, "airline.miles_txn_type")
```

```
        except Exception as exc:  # noqa: BLE001
            print(f"⚠️ Could not fetch airline.miles_txn_type enum values: {exc}")
            miles_txn_types = []

        flight_ids = fetch_flight_ids(conn)
        print(f"✈️  Found {len(flight_ids)} flights in airline.flights.")
        if not flight_ids:
            raise RuntimeError("No flights found in airline.flights; run synth_flights.py
first.")

        # 2) Passengers
        print("👥 Generating synthetic passengers...")
        passenger_rows = generate_passenger_rows(NUM_PASSENGERS)
        insert_passengers(conn, passenger_rows)

        # Refresh passenger IDs from DB (includes existing + new)
        passenger_ids = fetch_passenger_ids(conn)
        print(f"👥 Total passengers now: {len(passenger_ids)}")

        # 3) Loyalty accounts
        print("💳 Generating loyalty accounts...")
        loyalty_rows = generate_loyalty_rows(passenger_ids, loyalty_tiers,
loyalty_ratio=0.6)
        insert_loyalty_accounts(conn, loyalty_rows)

        # Refresh loyalty IDs
        loyalty_ids = fetch_loyalty_ids(conn)
        print(f"💳 Total loyalty accounts now: {len(loyalty_ids)}")

        # 4) Miles transactions
        print("🧾 Generating miles transactions...")
        miles_rows = generate_miles_txn_rows(loyalty_ids, flight_ids, miles_txn_types)
        insert_miles_transactions(conn, miles_rows)

    print("🎉 Synthetic customers & loyalty data load complete.")


if __name__ == "__main__":
    main()
```

```
"""
Synthetic revenue generator for the Airline BI database.

- Creates synthetic bookings for existing passengers & flights
- Creates one payment per booking
- Respects UNIQUE (passenger_id, flight_id) on airline.bookings

Schema assumptions:

  airline.bookings
    - booking_id        BIGSERIAL PRIMARY KEY
    - passenger_id      BIGINT (FK -> passengers.passenger_id)
    - flight_id         BIGINT (FK -> flights.flight_id)
    - booking_date      TIMESTAMP WITHOUT TIME ZONE
    - fare_class        TEXT
    - base_price_usd    NUMERIC(10,2)
    - booking_channel   TEXT
    - UNIQUE (passenger_id, flight_id)

  airline.payments
    - payment_id        BIGSERIAL PRIMARY KEY
    - booking_id        BIGINT (FK -> bookings.booking_id)
    - amount_usd        NUMERIC(10,2)
    - method            airline.payment_method
    - status            airline.payment_status
    - paid_at           TIMESTAMP WITHOUT TIME ZONE NOT NULL
"""

import os
import random
from decimal import Decimal
from datetime import timedelta

from dotenv import load_dotenv
from sqlalchemy import create_engine, text
from faker import Faker

# ---------------------------------------------------------------------------
# Configuration
# ---------------------------------------------------------------------------

TARGET_NEW_BOOKINGS = 20_000  # try to generate this many new bookings

FARE_CLASSES = ["Basic", "Standard", "Flexible", "Business", "First"]
BOOKING_CHANNELS = ["Web", "Mobile", "Call Center", "Travel Agent"]

PAYMENT_METHODS = ["Card", "Points", "Cash", "Voucher"]
PAYMENT_STATUSES = ["Authorized", "Captured", "Refunded", "Failed"]

fake = Faker()


def get_db_url() -> str:
    """
    Get the SQLAlchemy DB URL from environment.

    Prefer DATABASE_URL (your current .env), fall back to AIRLINE_DB_DSN.
    """
    load_dotenv()
    url = os.getenv("DATABASE_URL") or os.getenv("AIRLINE_DB_DSN")
    if not url:
        raise RuntimeError(
            "Set either DATABASE_URL or AIRLINE_DB_DSN in your environment / .env.\n"
            "Example: postgresql+psycopg2://postgres:password@localhost:5432/airline_bi"
```

```
        )
    return url


ENGINE = create_engine(get_db_url(), future=True, pool_pre_ping=True)


# ---------------------------------------------------------------------------
# Helpers
# ---------------------------------------------------------------------------

def money(x: float) -> Decimal:
    """Round to 2 decimal places and return Decimal."""
    return Decimal(f"{x:.2f}")


def fetch_passengers_and_flights(con):
    """Return lists of passenger_ids and flight_ids."""
    passenger_ids = [
        row[0]
        for row in con.execute(
            text("SELECT passenger_id FROM airline.passengers ORDER BY passenger_id")
        )
    ]
    flight_ids = [
        row[0]
        for row in con.execute(
            text("SELECT flight_id FROM airline.flights ORDER BY flight_id")
        )
    ]

    if not passenger_ids:
        raise RuntimeError("No rows found in airline.passengers.")
    if not flight_ids:
        raise RuntimeError("No rows found in airline.flights.")

    print(f"👥 Found {len(passenger_ids)} passengers.")
    print(f"✈️  Found {len(flight_ids)} flights.")
    return passenger_ids, flight_ids


def fetch_existing_booking_pairs(con):
    """
    Load existing (passenger_id, flight_id) pairs from airline.bookings
    so we don't violate UNIQUE (passenger_id, flight_id).
    """
    rows = con.execute(
        text("SELECT passenger_id, flight_id FROM airline.bookings")
    ).fetchall()
    existing = {(int(r[0]), int(r[1])) for r in rows}
    print(f"🔗 Existing booking pairs (passenger, flight): {len(existing)}")
    return existing


def generate_booking_payloads(passenger_ids, flight_ids, n_bookings: int, used_pairs: set):
    """
    Generate payload dictionaries for airline.bookings WITHOUT booking_id.

    Args:
        passenger_ids: list[int]
        flight_ids: list[int]
        n_bookings: desired number of new bookings
        used_pairs: set of (passenger_id, flight_id) already present
```

```
    Returns:
        bookings_payload: list[dict] ready for INSERT (no booking_id)
    """
    bookings = []

    print(f"🧾 Generating up to {n_bookings} synthetic bookings (unique per
passenger/flight)...")
    attempts = 0
    max_attempts = n_bookings * 10  # generous upper bound

    while len(bookings) < n_bookings and attempts < max_attempts:
        attempts += 1

        passenger_id = random.choice(passenger_ids)
        flight_id = random.choice(flight_ids)
        key = (passenger_id, flight_id)

        # Avoid violating UNIQUE (passenger_id, flight_id)
        if key in used_pairs:
            continue
        used_pairs.add(key)

        booking_date = fake.date_time_between(start_date="-9M", end_date="+3M")

        fare_class = random.choices(
            FARE_CLASSES, weights=[0.35, 0.30, 0.20, 0.10, 0.05]
        )[0]
        channel = random.choices(
            BOOKING_CHANNELS, weights=[0.55, 0.25, 0.10, 0.10]
        )[0]

        # Base price ~ 80–900 with some long tail
        base_price = money(random.lognormvariate(4.5, 0.5))
        base_price = max(money(80), min(base_price, money(900)))

        bookings.append(
            {
                "passenger_id": passenger_id,
                "flight_id": flight_id,
                "booking_date": booking_date,
                "fare_class": fare_class,
                "base_price_usd": base_price,
                "booking_channel": channel,
            }
        )

    if len(bookings) < n_bookings:
        print(
            f"⚠️ Only generated {len(bookings)} unique booking pairs "
            f"out of requested {n_bookings} (attempts={attempts})."
        )

    print(f"✅ Prepared {len(bookings)} booking payloads.")
    return bookings


def insert_bookings_and_return(con, booking_payloads):
    """
    Insert into airline.bookings (letting Postgres assign booking_id)
    and then SELECT the newly inserted rows using booking_id > max_before.
    """
    if not booking_payloads:
        print("ℹ️ No bookings to insert.")
```

```
        return []

    # baseline max booking_id
    max_before = con.execute(
        text("SELECT COALESCE(MAX(booking_id), 0) FROM airline.bookings")
    ).scalar_one()

    # bulk insert WITHOUT RETURNING to avoid SQLAlchemy/psycopg2 weirdness
    con.execute(
        text(
            """
            INSERT INTO airline.bookings (
                passenger_id,
                flight_id,
                booking_date,
                fare_class,
                base_price_usd,
                booking_channel
            )
            VALUES (
                :passenger_id,
                :flight_id,
                :booking_date,
                :fare_class,
                :base_price_usd,
                :booking_channel
            );
            """
        ),
        booking_payloads,
    )

    # now pull back just the new rows
    result = con.execute(
        text(
            """
            SELECT booking_id,
                   passenger_id,
                   flight_id,
                   booking_date,
                   fare_class,
                   base_price_usd,
                   booking_channel
            FROM airline.bookings
            WHERE booking_id > :max_before
            ORDER BY booking_id;
            """
        ),
        {"max_before": max_before},
    )

    rows = result.mappings().all()
    print(f"✅ New bookings inserted: {len(rows)}")
    return rows


def build_payments_from_bookings(inserted_bookings):
    """
    Given rows returned from insert_bookings_and_return, build matching
    payments payloads.

    Each payment:
      - amount_usd ~ base_price_usd * [0.9, 1.15]
      - paid_at is always NON-NULL (some time after booking_date)
    """
```

```
    payments = []

    for row in inserted_bookings:
        booking_id = row["booking_id"]
        booking_date = row["booking_date"]
        base_price = row["base_price_usd"]

        method = random.choices(
            PAYMENT_METHODS,
            weights=[0.7, 0.1, 0.1, 0.1],
        )[0]
        status = random.choices(
            PAYMENT_STATUSES,
            weights=[0.65, 0.15, 0.10, 0.10],
        )[0]

        multiplier = random.uniform(0.9, 1.15)
        amount = money(float(base_price) * multiplier)

        # paid_at: always non-null
        offset_minutes = random.randint(0, 60 * 24)
        paid_at = booking_date + timedelta(minutes=offset_minutes)

        payments.append(
            {
                "booking_id": booking_id,
                "amount_usd": amount,
                "method": method,
                "status": status,
                "paid_at": paid_at,
            }
        )

    print(f"💳 Prepared {len(payments)} payments.")
    return payments


def insert_payments(con, payments):
    if not payments:
        print("ℹ️ No payments to insert.")
        return 0

    con.execute(
        text(
            """
            INSERT INTO airline.payments (
                booking_id,
                amount_usd,
                method,
                status,
                paid_at
            )
            VALUES (
                :booking_id,
                :amount_usd,
                :method,
                :status,
                :paid_at
            );
            """
        ),
        payments,
    )
    print(f"💳 Payments inserted: {len(payments)}")
```

```
    return len(payments)


# ------------------------------------------------------------------------
# Main
# ------------------------------------------------------------------------

def main():
    with ENGINE.begin() as con:
        # 🔧 1) Make sure the sequences are in sync with existing data
        con.execute(
            text(
                """
                SELECT setval(
                    pg_get_serial_sequence('airline.bookings', 'booking_id'),
                    COALESCE((SELECT MAX(booking_id) FROM airline.bookings), 0)
                );
                """
            )
        )
        con.execute(
            text(
                """
                SELECT setval(
                    pg_get_serial_sequence('airline.payments', 'payment_id'),
                    COALESCE((SELECT MAX(payment_id) FROM airline.payments), 0)
                );
                """
            )
        )

        # Debug: show columns for sanity
        booking_cols = [
            row[0]
            for row in con.execute(
                text(
                    """
                    SELECT column_name
                    FROM information_schema.columns
                    WHERE table_schema = 'airline'
                      AND table_name   = 'bookings'
                    ORDER BY ordinal_position;
                    """
                )
            )
        ]
        print(f"🔍 bookings columns: {booking_cols}")

        payment_cols = [
            row[0]
            for row in con.execute(
                text(
                    """
                    SELECT column_name
                    FROM information_schema.columns
                    WHERE table_schema = 'airline'
                      AND table_name   = 'payments'
                    ORDER BY ordinal_position;
                    """
                )
            )
        ]
        print(f"🔍 payments columns: {payment_cols}")
```

```
        passenger_ids, flight_ids = fetch_passengers_and_flights(con)
        used_pairs = fetch_existing_booking_pairs(con)

        booking_payloads = generate_booking_payloads(
            passenger_ids,
            flight_ids,
            n_bookings=TARGET_NEW_BOOKINGS,
            used_pairs=used_pairs,
        )

        inserted = insert_bookings_and_return(con, booking_payloads)
        payments = build_payments_from_bookings(inserted)
        insert_payments(con, payments)

    print("🎉 Synthetic revenue generation complete.")


if __name__ == "__main__":
    main()
```

```
"""
Backfill dimension / audit tables derived from existing flight data.

This script:
  1) Populates airline.routes from distinct (airline_id, origin_airport_id,
destination_airport_id)
  2) Optionally computes distance_nm using airport latitude / longitude
  3) Seeds airline.aircraft with a small synthetic global fleet
  4) Assigns aircraft_id to flights
  5) Creates synthetic airline.flight_changes records for a subset of flights

Schema assumptions (from the DB):

  airline.flights
    flight_id               bigint PK
    airline_id              bigint
    route_id                bigint (nullable)
    aircraft_id             bigint (nullable)
    origin_airport_id       bigint
    destination_airport_id  bigint
    flight_number           text
    flight_date             date
    scheduled_departure_utc timestamp
    scheduled_arrival_utc   timestamp
    actual_departure_utc    timestamp
    actual_arrival_utc      timestamp
    delay_minutes           integer
    delay_cause             text
    status                  enum (flight_status)

  airline.routes
    route_id                bigint PK
    airline_id              bigint
    origin_airport_id       bigint
    destination_airport_id  bigint
    distance_nm             integer (nullable)

  airline.airports
    airport_id              bigint
    iata_code               varchar
    icao_code               varchar
    name                    text
    city                    text
    country                 text
    latitude                numeric   -- degrees
    longitude               numeric   -- degrees
    timezone                text

  airline.aircraft
    aircraft_id             bigint PK
    manufacturer            text
    model                   text
    seat_capacity           integer
    tail_number             text (nullable)

  airline.flight_changes
    change_id               bigint PK
    flight_id               bigint FK -> flights
    old_aircraft_id         bigint
    new_aircraft_id         bigint
    reason                  text
    changed_at              timestamp
"""
```

```
import os
from sqlalchemy import create_engine, text


def get_db_url() -> str:
    url = os.getenv("DATABASE_URL") or os.getenv("AIRLINE_DB_DSN")
    if not url:
        raise RuntimeError(
            "Set DATABASE_URL or AIRLINE_DB_DSN in your environment.\n"
            "Example: postgresql+psycopg2://postgres:gpcool@localhost:5432/airline_bi"
        )
    return url


ENGINE = create_engine(get_db_url(), future=True, pool_pre_ping=True)


# ---------------------------------------------------------------------
# 1. ROUTES
# ---------------------------------------------------------------------

def backfill_routes_from_flights(compute_distance: bool = False) -> None:
    """Insert missing routes based on distinct flights."""

    print("🔷 Backfilling airline.routes from airline.flights ...")

    insert_sql = text(
        """
        INSERT INTO airline.routes (
            airline_id,
            origin_airport_id,
            destination_airport_id,
            distance_nm
        )
        SELECT DISTINCT
            f.airline_id,
            f.origin_airport_id,
            f.destination_airport_id,
            NULL::integer AS distance_nm
        FROM airline.flights f
        LEFT JOIN airline.routes r
          ON r.airline_id          = f.airline_id
         AND r.origin_airport_id   = f.origin_airport_id
         AND r.destination_airport_id = f.destination_airport_id
        WHERE r.route_id IS NULL
          AND f.airline_id IS NOT NULL
          AND f.origin_airport_id IS NOT NULL
          AND f.destination_airport_id IS NOT NULL;
        """
    )

    with ENGINE.begin() as con:
        result = con.execute(insert_sql)
        print(f"  → Inserted {result.rowcount or 0} route rows")

    if compute_distance:
        compute_route_distances()


def compute_route_distances() -> None:
    """
    Compute approximate great-circle distance in nautical miles for each route.

    Uses airline.airports.latitude / longitude (degrees).
    """
```

```
    print("🔷 Computing distance_nm for airline.routes ...")

    update_sql = text(
        """
        UPDATE airline.routes r
        SET distance_nm = sub.distance_nm::integer
        FROM (
            SELECT
                r2.route_id,
                (
                    2 * 6371 * asin(
                        sqrt(
                            sin(radians(ad.latitude - ao.latitude) / 2)^2 +
                            cos(radians(ao.latitude)) *
                            cos(radians(ad.latitude)) *
                            sin(radians(ad.longitude - ao.longitude) / 2)^2
                        )
                    ) / 1.852
                ) AS distance_nm
            FROM airline.routes r2
            JOIN airline.airports ao
              ON ao.airport_id = r2.origin_airport_id
            JOIN airline.airports ad
              ON ad.airport_id = r2.destination_airport_id
            WHERE r2.distance_nm IS NULL
              AND ao.latitude IS NOT NULL
              AND ao.longitude IS NOT NULL
              AND ad.latitude IS NOT NULL
              AND ad.longitude IS NOT NULL
        ) AS sub
        WHERE r.route_id = sub.route_id;
        """
    )

    with ENGINE.begin() as con:
        result = con.execute(update_sql)
        print(f"  → Updated distance_nm for {result.rowcount or 0} routes")


def backfill_route_ids_on_flights() -> None:
    """Update airline.flights.route_id to match airline.routes."""

    print("🔷 Backfilling flights.route_id from routes ...")

    update_sql = text(
        """
        UPDATE airline.flights f
        SET route_id = r.route_id
        FROM airline.routes r
        WHERE f.route_id IS NULL
          AND f.airline_id        = r.airline_id
          AND f.origin_airport_id = r.origin_airport_id
          AND f.destination_airport_id = r.destination_airport_id;
        """
    )

    with ENGINE.begin() as con:
        result = con.execute(update_sql)
        print(f"  → Updated route_id on {result.rowcount or 0} flights")


# ------------------------------------------------------------------
# 2. AIRCRAFT + FLIGHT.AIRCRAFT_ID
```

```
# ----------------------------------------------------------------------

def backfill_aircraft() -> None:
    """
    Seed airline.aircraft with a small synthetic global fleet.

    Schema:
      aircraft_id   bigserial PK
      manufacturer  text
      model         text
      seat_capacity integer
      tail_number   text (nullable)
    """

    print("🔷 Backfilling airline.aircraft with synthetic fleet ...")

    insert_sql = text(
        """
        -- Only seed if table is currently empty
        INSERT INTO airline.aircraft (manufacturer, model, seat_capacity, tail_number)
        SELECT manufacturer, model, seat_capacity, tail_number
        FROM (
            VALUES
                ('Airbus', 'A320',      150, NULL),
                ('Boeing', '737-800',   165, NULL),
                ('Airbus', 'A321',      185, NULL),
                ('Boeing', '787-8',     242, NULL),
                ('Airbus', 'A350-900',  300, NULL)
        ) AS v(manufacturer, model, seat_capacity, tail_number)
        WHERE NOT EXISTS (SELECT 1 FROM airline.aircraft);
        """
    )

    with ENGINE.begin() as con:
        result = con.execute(insert_sql)
        print(f"  → Inserted {result.rowcount or 0} aircraft rows")


def assign_aircraft_to_flights() -> None:
    """
    Assign an aircraft_id to each flight, picking randomly from all aircraft.
    (aircraft is not tied to a specific airline in your schema.)
    """

    print("🔷 Assigning aircraft_id to flights ...")

    update_sql = text(
        """
        WITH choices AS (
            SELECT
                f.flight_id,
                (
                  SELECT ac2.aircraft_id
                  FROM airline.aircraft ac2
                  ORDER BY random()
                  LIMIT 1
                ) AS aircraft_id
            FROM airline.flights f
            WHERE f.aircraft_id IS NULL
        )
        UPDATE airline.flights f
        SET aircraft_id = c.aircraft_id
        FROM choices c
        WHERE f.flight_id = c.flight_id;
```

```
        """
    )

    with ENGINE.begin() as con:
        result = con.execute(update_sql)
        print(f"  → Updated aircraft_id on {result.rowcount or 0} flights")


# ----------------------------------------------------------------------
# 3. FLIGHT_CHANGES
# ----------------------------------------------------------------------

def generate_flight_changes(change_fraction: float = 0.05) -> None:
    """
    Create synthetic aircraft change events for a subset of flights.

    change_fraction: approximate fraction of flights to get a change row.
    """

    print(f"🔷 Generating flight_changes for ~{change_fraction*100:.1f}% of flights ...")

    insert_sql = text(
        f"""
        WITH candidate_flights AS (
            SELECT f.flight_id, f.aircraft_id, f.flight_date
            FROM airline.flights f
            WHERE f.aircraft_id IS NOT NULL
        ),
        sampled AS (
            SELECT *
            FROM candidate_flights
            WHERE random() < {change_fraction}
        ),
        new_aircraft AS (
            SELECT
                s.flight_id,
                s.aircraft_id AS original_aircraft_id,
                s.flight_date,
                (
                  SELECT ac2.aircraft_id
                  FROM airline.aircraft ac2
                  WHERE ac2.aircraft_id <> s.aircraft_id
                  ORDER BY random()
                  LIMIT 1
                ) AS new_aircraft_id
            FROM sampled s
        )
        INSERT INTO airline.flight_changes (
            flight_id,
            old_aircraft_id,
            new_aircraft_id,
            reason,
            changed_at
        )
        SELECT
            n.flight_id,
            n.original_aircraft_id,
            n.new_aircraft_id,
            (ARRAY['Maintenance','Crew Reassignment','Operational','Equipment Downgrade'])[1
+ floor(random()*4)]::text,
            n.flight_date::timestamp - INTERVAL '2 hours' + (random() * INTERVAL '90
minutes')
        FROM new_aircraft n
        WHERE n.new_aircraft_id IS NOT NULL;
        """
```

```
    )

    with ENGINE.begin() as con:
        result = con.execute(insert_sql)
        print(f"  → Inserted {result.rowcount or 0} flight_changes rows")

    # Optionally: update flights.aircraft_id to reflect the "new" aircraft
    update_sql = text(
        """
        UPDATE airline.flights f
        SET aircraft_id = fc.new_aircraft_id
        FROM airline.flight_changes fc
        WHERE fc.flight_id = f.flight_id;
        """
    )

    with ENGINE.begin() as con:
        result = con.execute(update_sql)
        print(f"  → Updated aircraft_id on {result.rowcount or 0} flights to match
flight_changes")


# ---------------------------------------------------------------------
# MAIN
# ---------------------------------------------------------------------

def run():
    print("=== Backfill derived tables from flights ===")
    backfill_routes_from_flights(compute_distance=False)  # set True if you want distances
computed
    backfill_route_ids_on_flights()
    backfill_aircraft()
    assign_aircraft_to_flights()
    generate_flight_changes(change_fraction=0.05)
    print("=== Done. ✅ ===")


if __name__ == "__main__":
    run()
```

Airline Business Intelligence – Phase 2 Data Quality Snapshot

This notebook is a quick **smoke test** for Phase 2 of the pipeline:

- OpenFlights reference data (`airports` , `airlines`)
- Synthetic flight schedule (`flights`)
- BTS on-time performance (`flight_performance`)
- Synthetic customers & loyalty (`passengers` , `loyalty_accounts` , `miles_transactions`)
- Synthetic revenue (`bookings` , `payments`)

Assumptions:

- PostgreSQL is running
- The `airline` schema has been migrated
- The environment variable `DATABASE_URL` points at the `airline_bi` database

In [1]:
```
import os
from textwrap import dedent

import pandas as pd
from sqlalchemy import create_engine

db_url = os.getenv("DATABASE_URL")
if not db_url:
    raise RuntimeError("DATABASE_URL is not set — export it before running t

engine = create_engine(db_url, future=True)
```

In [2]:
```
import os
from textwrap import dedent

import pandas as pd
from sqlalchemy import create_engine

db_url = os.getenv("DATABASE_URL")
if not db_url:
    raise RuntimeError("DATABASE_URL is not set — export it before running t

engine = create_engine(db_url, future=True)
```

1. Row counts by table

In [4]:
```
def run_sql(query: str) -> pd.DataFrame:
    """Run a SQL query and return a pandas DataFrame."""
```

```
    with engine.begin() as con:
        return pd.read_sql_query(dedent(query), con)

row_counts_sql = """
    SELECT 'airports'            AS table_name, COUNT(*) AS row_count FROM a
    UNION ALL
    SELECT 'airlines',           COUNT(*) FROM airline.airlines
    UNION ALL
    SELECT 'flights',            COUNT(*) FROM airline.flights
    UNION ALL
    SELECT 'flight_performance', COUNT(*) FROM airline.flight_performance
    UNION ALL
    SELECT 'passengers',         COUNT(*) FROM airline.passengers
    UNION ALL
    SELECT 'loyalty_accounts',   COUNT(*) FROM airline.loyalty_accounts
    UNION ALL
    SELECT 'miles_transactions', COUNT(*) FROM airline.miles_transactions
    UNION ALL
    SELECT 'bookings',           COUNT(*) FROM airline.bookings
    UNION ALL
    SELECT 'payments',           COUNT(*) FROM airline.payments
    ORDER BY table_name;
"""

row_counts = run_sql(row_counts_sql)
row_counts
```

Out[4]:

	table_name	row_count
0	airlines	5733
1	airports	7697
2	bookings	40000
3	flight_performance	22595
4	flights	5000
5	loyalty_accounts	3000
6	miles_transactions	10576
7	passengers	5000
8	payments	40000

2. Basic null checks on key columns

In [6]:

```
null_checks_sql = """
SELECT 'airports' AS table_name,
       COUNT(*)   AS total_rows,
       SUM(CASE WHEN iata_code IS NULL THEN 1 ELSE 0 END) AS metric_1_nulls,
       SUM(CASE WHEN icao_code IS NULL THEN 1 ELSE 0 END) AS metric_2_nulls
FROM airline.airports
UNION ALL
```

```
SELECT 'airlines' AS table_name,
       COUNT(*)   AS total_rows,
       SUM(CASE WHEN iata_code IS NULL THEN 1 ELSE 0 END) AS metric_1_nulls,
       SUM(CASE WHEN icao_code IS NULL THEN 1 ELSE 0 END) AS metric_2_nulls
FROM airline.airlines
UNION ALL
SELECT 'passengers' AS table_name,
       COUNT(*)   AS total_rows,
       SUM(CASE WHEN email IS NULL THEN 1 ELSE 0 END)          AS metric_1_r
       SUM(CASE WHEN state_or_country IS NULL THEN 1 ELSE 0 END) AS metric_2
FROM airline.passengers
UNION ALL
SELECT 'bookings' AS table_name,
       COUNT(*)   AS total_rows,
       SUM(CASE WHEN passenger_id IS NULL THEN 1 ELSE 0 END) AS metric_1_nul
       SUM(CASE WHEN flight_id    IS NULL THEN 1 ELSE 0 END) AS metric_2_nul
FROM airline.bookings
UNION ALL
SELECT 'payments' AS table_name,
       COUNT(*)   AS total_rows,
       SUM(CASE WHEN booking_id IS NULL THEN 1 ELSE 0 END) AS metric_1_nulls
       SUM(CASE WHEN amount_usd IS NULL THEN 1 ELSE 0 END) AS metric_2_nulls
FROM airline.payments
ORDER BY table_name;
"""
null_checks = run_sql(null_checks_sql)
null_checks
```

Out[6]:

	table_name	total_rows	metric_1_nulls	metric_2_nulls
0	airlines	5733	4619	172
1	airports	7697	1625	0
2	bookings	40000	0	0
3	passengers	5000	0	0
4	payments	40000	0	0

In [7]:

```
flights_nulls_sql = """
SELECT
    COUNT(*) AS total_rows,
    SUM(CASE WHEN airline_id          IS NULL THEN 1 ELSE 0 END) AS airline_
    SUM(CASE WHEN aircraft_id         IS NULL THEN 1 ELSE 0 END) AS aircraft
    SUM(CASE WHEN route_id            IS NULL THEN 1 ELSE 0 END) AS route_id
    SUM(CASE WHEN origin_airport_id   IS NULL THEN 1 ELSE 0 END) AS origin_i
    SUM(CASE WHEN destination_airport_id IS NULL THEN 1 ELSE 0 END) AS dest_
FROM airline.flights;
"""
flights_nulls = run_sql(flights_nulls_sql)
flights_nulls
```

Out[7]:

	total_rows	airline_id_nulls	aircraft_id_nulls	route_id_nulls	origin_id_nulls	dest_i
0	5000	0	5000	5000	0	

3. Primary-key and foriegn-key sanity checks

In [8]:

```
pk_fk_sql = """
    -- PK uniqueness checks
    SELECT 'airports_pk' AS check_name,
           COUNT(*) - COUNT(DISTINCT airport_id) AS issue_count
    FROM airline.airports
    UNION ALL
    SELECT 'airlines_pk',
           COUNT(*) - COUNT(DISTINCT airline_id)
    FROM airline.airlines
    UNION ALL
    SELECT 'flights_pk',
           COUNT(*) - COUNT(DISTINCT flight_id)
    FROM airline.flights
    UNION ALL
    SELECT 'passengers_pk',
           COUNT(*) - COUNT(DISTINCT passenger_id)
    FROM airline.passengers
    UNION ALL
    SELECT 'bookings_pk',
           COUNT(*) - COUNT(DISTINCT booking_id)
    FROM airline.bookings
    UNION ALL
    SELECT 'payments_pk',
           COUNT(*) - COUNT(DISTINCT payment_id)
    FROM airline.payments

    UNION ALL

    -- FK checks: flights -> airlines / airports
    SELECT 'flight_airline_fk',
           COUNT(*)
    FROM airline.flights f
    LEFT JOIN airline.airlines al ON f.airline_id = al.airline_id
    WHERE al.airline_id IS NULL

    UNION ALL

    SELECT 'flight_origin_airport_fk',
           COUNT(*)
    FROM airline.flights f
    LEFT JOIN airline.airports a ON f.origin_airport_id = a.airport_id
    WHERE a.airport_id IS NULL

    UNION ALL

    SELECT 'flight_destination_airport_fk',
```

```
        COUNT(*)
    FROM airline.flights f
    LEFT JOIN airline.airports a ON f.destination_airport_id = a.airport_id
    WHERE a.airport_id IS NULL

    UNION ALL

    -- bookings / payments FKs
    SELECT 'booking_flight_fk',
        COUNT(*)
    FROM airline.bookings b
    LEFT JOIN airline.flights f ON b.flight_id = f.flight_id
    WHERE f.flight_id IS NULL

    UNION ALL

    SELECT 'booking_passenger_fk',
        COUNT(*)
    FROM airline.bookings b
    LEFT JOIN airline.passengers p ON b.passenger_id = p.passenger_id
    WHERE p.passenger_id IS NULL

    UNION ALL

    SELECT 'payment_booking_fk',
        COUNT(*)
    FROM airline.payments p
    LEFT JOIN airline.bookings b ON p.booking_id = b.booking_id
    WHERE b.booking_id IS NULL;
"""

pk_fk_issues = run_sql(pk_fk_sql)
pk_fk_issues
```

Out[8]:

	check_name	issue_count
0	passengers_pk	0
1	flights_pk	0
2	airlines_pk	0
3	airports_pk	0
4	flight_airline_fk	0
5	flight_destination_airport_fk	0
6	flight_origin_airport_fk	0
7	booking_passenger_fk	0
8	booking_flight_fk	0
9	payment_booking_fk	0
10	bookings_pk	0
11	payments_pk	0

4. Business sanity checks

In [9]:

```
bookings_per_passenger_sql = """
    SELECT
        COUNT(*) AS total_bookings,
        COUNT(DISTINCT passenger_id) AS distinct_passengers,
        MIN(bookings_per_pax) AS min_bookings_per_passenger,
        PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY bookings_per_pax)
            AS median_bookings_per_passenger,
        MAX(bookings_per_pax) AS max_bookings_per_passenger
    FROM (
        SELECT passenger_id, COUNT(*) AS bookings_per_pax
        FROM airline.bookings
        GROUP BY passenger_id
    ) sub;
"""

bookings_per_passenger = run_sql(bookings_per_passenger_sql)
bookings_per_passenger
```

Out[9]:

	total_bookings	distinct_passengers	min_bookings_per_passenger	median_booking
0	4995	4995	1	

In [10]:

```
payments_summary_sql = """
    SELECT
        COUNT(*)       AS num_payments,
        SUM(amount_usd) AS total_revenue_usd,
```

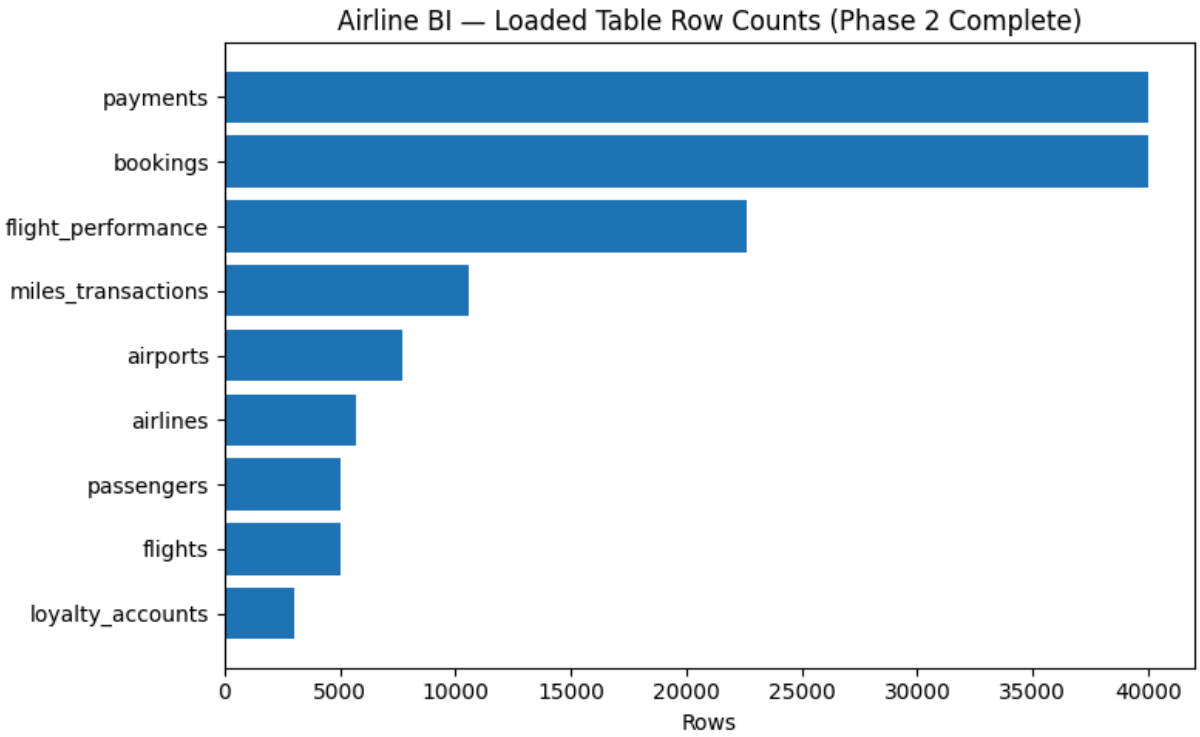
```
        AVG(amount_usd) AS avg_ticket_price_usd,
        MIN(amount_usd) AS min_ticket_price_usd,
        MAX(amount_usd) AS max_ticket_price_usd
    FROM airline.payments;
"""

payments_summary = run_sql(payments_summary_sql)
payments_summary
```

Out[10]:

	num_payments	total_revenue_usd	avg_ticket_price_usd	min_ticket_price_usd	ma
0	40000	4539794.86	113.494872	72.0	

In [15]:

```
import matplotlib.pyplot as plt

# Simple proof-of-pipeline chart: row counts per table
df = row_counts.sort_values("row_count")

plt.figure(figsize=(8,5))
plt.barh(df["table_name"], df["row_count"])
plt.title("Airline BI — Loaded Table Row Counts (Phase 2 Complete)")
plt.xlabel("Rows")
plt.tight_layout()

plt.savefig("/Users/gracepolito/Public/Airline Business Intelligence Databas
plt.show()
```



02 - Data Quality Checks (Phase 3)

This notebook runs row-count, null, and key-integrity checks for the Airline Business Intelligence Database.

- Validates Phase 2 ETL results
- Guides SQL cleanup in `sql/03_dml_cleanup.sql`
- Confirms constraints in `sql/04_constraints_indexes.sql`

In [4]:
```
import os
import pandas as pd
from sqlalchemy import create_engine, text
from dotenv import load_dotenv
from typing import Optional, Dict

# Load environment variables
load_dotenv()
db_url = os.getenv("DATABASE_URL")
if db_url is None:
    raise RuntimeError("DATABASE_URL not set in environment or .env file")

# Create engine
engine = create_engine(db_url, future=True)

def run_sql(query: str, params: Optional[Dict] = None) -> pd.DataFrame:
    """Helper to run a SQL query and return a DataFrame."""
    with engine.connect() as conn:
        result = conn.execute(text(query), params or {})
        df = pd.DataFrame(result.fetchall(), columns=result.keys())
    return df
```

Row counts per table

In [5]:
```
row_counts_sql = """
SELECT 'aircraft'           AS table_name, COUNT(*) AS row_count FROM airlir
UNION ALL
SELECT 'airlines',                         COUNT(*) FROM airline.airlines
UNION ALL
SELECT 'airports',                         COUNT(*) FROM airline.airports
UNION ALL
SELECT 'bookings',                         COUNT(*) FROM airline.bookings
UNION ALL
SELECT 'flight_performance',               COUNT(*) FROM airline.flight_perfor
UNION ALL
SELECT 'flights',                          COUNT(*) FROM airline.flights
UNION ALL
SELECT 'loyalty_accounts',                 COUNT(*) FROM airline.loyalty_accou
UNION ALL
SELECT 'miles_transactions',               COUNT(*) FROM airline.miles_transac
UNION ALL
```

```
SELECT 'passengers',                    COUNT(*) FROM airline.passengers
UNION ALL
SELECT 'payments',                      COUNT(*) FROM airline.payments
UNION ALL
SELECT 'routes',                        COUNT(*) FROM airline.routes
ORDER BY table_name;
"""

row_counts = run_sql(row_counts_sql)
row_counts
```

Out[5]:

	table_name	row_count
0	aircraft	0
1	airlines	1108
2	airports	7697
3	bookings	40000
4	flight_performance	22595
5	flights	5000
6	loyalty_accounts	3000
7	miles_transactions	10576
8	passengers	5000
9	payments	40000
10	routes	0

FK / key null + orphan checks

In [6]:

```
fk_issues_sql = """
SELECT
  flights_missing_airline,
  flights_missing_origin_airport,
  flights_missing_destination_airport,
  bookings_missing_passenger,
  bookings_missing_flight,
  payments_missing_booking
FROM (
  SELECT
    SUM(CASE WHEN a.airline_id IS NULL THEN 1 ELSE 0 END) AS flights_missing
    SUM(CASE WHEN ao.airport_id IS NULL THEN 1 ELSE 0 END) AS flights_missir
    SUM(CASE WHEN ad.airport_id IS NULL THEN 1 ELSE 0 END) AS flights_missir
  FROM airline.flights f
  LEFT JOIN airline.airlines a ON f.airline_id = a.airline_id
  LEFT JOIN airline.airports ao ON f.origin_airport_id = ao.airport_id
  LEFT JOIN airline.airports ad ON f.destination_airport_id = ad.airport_id
) flights_issues,
(
  SELECT
```

```
    SUM(CASE WHEN p.passenger_id IS NULL THEN 1 ELSE 0 END) AS bookings_miss
    SUM(CASE WHEN f2.flight_id IS NULL THEN 1 ELSE 0 END) AS bookings_missir
  FROM airline.bookings b
  LEFT JOIN airline.passengers p ON b.passenger_id = p.passenger_id
  LEFT JOIN airline.flights f2 ON b.flight_id = f2.flight_id
) bookings_issues,
(
  SELECT
    SUM(CASE WHEN b3.booking_id IS NULL THEN 1 ELSE 0 END) AS payments_missi
  FROM airline.payments pay
  LEFT JOIN airline.bookings b3 ON pay.booking_id = b3.booking_id
) payments_issues;
"""

fk_issues = run_sql(fk_issues_sql)
fk_issues
```

Out[6]:

	flights_missing_airline	flights_missing_origin_airport	flights_missing_destination_ai
0	0	0	

Duplicate key checks

In [7]:

```
duplicates_sql = """
SELECT 'airlines_iata_duplicates' AS metric,
       COUNT(*) AS count
FROM (
  SELECT iata_code
  FROM airline.airlines
  WHERE iata_code IS NOT NULL
  GROUP BY iata_code
  HAVING COUNT(*) > 1
) t
UNION ALL
SELECT 'airports_iata_duplicates',
       COUNT(*)
FROM (
  SELECT iata_code
  FROM airline.airports
  WHERE iata_code IS NOT NULL
  GROUP BY iata_code
  HAVING COUNT(*) > 1
) t2
UNION ALL
SELECT 'passenger_email_duplicates',
       COUNT(*)
FROM (
  SELECT email
  FROM airline.passengers
  WHERE email IS NOT NULL
  GROUP BY email
  HAVING COUNT(*) > 1
```

```
) t3
UNION ALL
SELECT 'loyalty_multiper_passenger',
       COUNT(*)
FROM (
  SELECT passenger_id
  FROM airline.loyalty_accounts
  GROUP BY passenger_id
  HAVING COUNT(*) > 1
) t4;
"""

dup_counts = run_sql(duplicates_sql)
dup_counts
```

Out[7]:

	metric	count
0	airlines_iata_duplicates	0
1	airports_iata_duplicates	0
2	passenger_email_duplicates	0
3	loyalty_multiper_passenger	0

"Problem Counts" summary table

In [8]:

```
# Melt FK issues into metric / count format
fk_problem_counts = fk_issues.melt(
    var_name="metric",
    value_name="count"
)

# Combine with duplicate counts
problem_counts = pd.concat(
    [fk_problem_counts, dup_counts],
    ignore_index=True
)

problem_counts
```

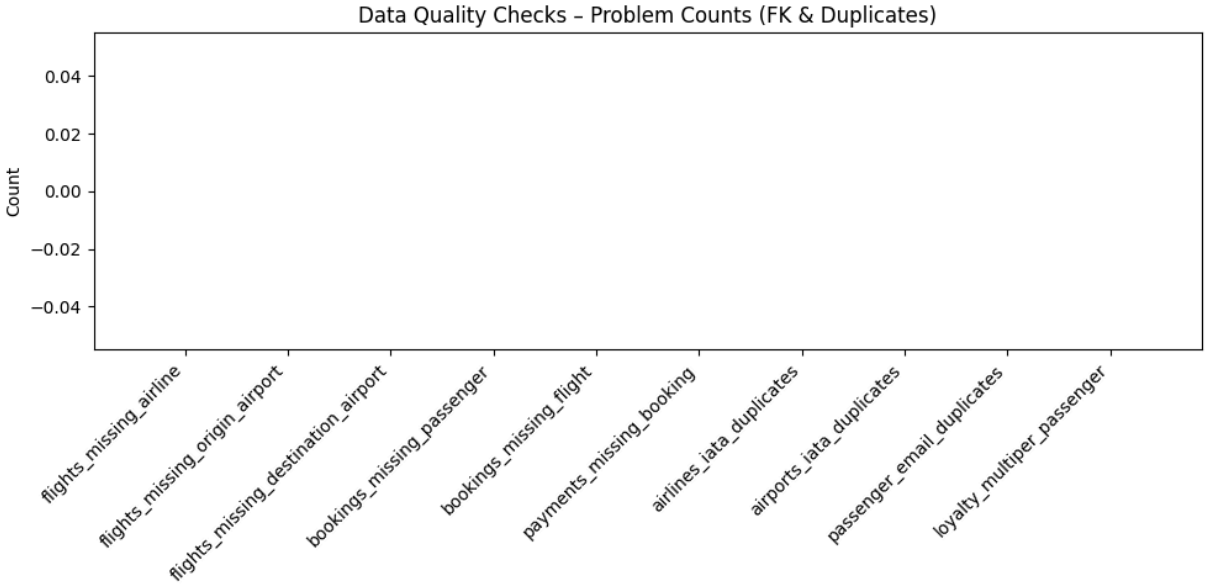
Out[8]:

	metric	count
0	flights_missing_airline	0
1	flights_missing_origin_airport	0
2	flights_missing_destination_airport	0
3	bookings_missing_passenger	0
4	bookings_missing_flight	0
5	payments_missing_booking	0
6	airlines_iata_duplicates	0
7	airports_iata_duplicates	0
8	passenger_email_duplicates	0
9	loyalty_multiper_passenger	0

Simple quality-check chart

In [9]:

```
import matplotlib.pyplot as plt

# Filter to only metrics with non-null counts (keep zeros)
plot_df = problem_counts.copy()

plt.figure(figsize=(10, 5))
plt.bar(plot_df["metric"], plot_df["count"])
plt.xticks(rotation=45, ha="right")
plt.ylabel("Count")
plt.title("Data Quality Checks — Problem Counts (FK & Duplicates)")
plt.tight_layout()

# Make sure docs/ exists
os.makedirs("docs", exist_ok=True)

plt.savefig("docs/pipeline_quality_checks.png", dpi=200)
plt.show()
```


Data Quality Checks – Problem Counts (FK & Duplicates)
Count
0.04
0.02
0.00
−0.02
−0.04
flights_missing_airline
flights_missing_origin_airport
flights_missing_destination_airport
bookings_missing_passenger
bookings_missing_flight
payments_missing_booking
airlines_iata_duplicates
airports_iata_duplicates
passenger_email_duplicates
loyalty_multiper_passenger

Airline BI Database — Phase 4: Analytical Queries

This notebook is used to:

- Develop and test analytical SQL queries against the airline BI database
- Profile query performance (EXPLAIN / EXPLAIN ANALYZE)
- Generate sample tables and visualizations for docs/phase_4_analytics.png

Database: PostgreSQL 16

Schema: airline_bi

In [93]:
```
import os

import pandas as pd
from dotenv import load_dotenv

load_dotenv()

import matplotlib.pyplot as plt

pd.set_option("display.max_rows", 50)
pd.set_option("display.max_columns", 50)
pd.set_option("display.width", 120)
```

Database Connection Config

In [94]:
```
db_url = os.getenv("DATABASE_URL")

if not db_url:
    raise ValueError("DATABASE_URL not found. Make sure your .env file is lo

db_url
```

Out[94]: 'postgresql+psycopg2://postgres:gpcool@localhost:5432/airline_bi'

Create the engine & test connection

In [95]:
```
from sqlalchemy import create_engine, text

engine = create_engine(db_url, echo=False, future=True)

with engine.connect() as conn:
    version = conn.exec_driver_sql("SELECT version();").scalar_one()
version
```

Out[95]:
```
'PostgreSQL 17.5 on x86_64-apple-darwin23.6.0, compiled by Apple clang vers
ion 16.0.0 (clang-1600.0.26.6), 64-bit'
```

Helper: run_sql() for SELECT Queries

In [96]:
```python
from typing import Optional, Dict

def run_sql(
    query: str,
    params: Optional[Dict] = None,
    limit: Optional[int] = None,
    debug: bool = False
) -> pd.DataFrame:

    """
    Execute a SQL query and return the results as a pandas DataFrame.

    Args:
        query: SQL string. Can include named parameters (e.g., :airline_id)
        params: dict of parameters to bind
        limit: optional row limit applied in Python (not SQL)
        debug: if True, prints the rendered SQL and params

    Returns:
        pandas.DataFrame
    """
    if debug:
        print("SQL:")
        print(query)
        if params:
            print("Params:", params)

    with engine.connect() as conn:
        df = pd.read_sql(text(query), conn, params=params)

    if limit is not None:
        return df.head(limit)
    return df
```

Helper: run_explain() for Performance Testing

In [97]:
```python
from typing import Optional, Dict

def run_explain(
    query: str,
    params: Optional[Dict] = None,
    analyze: bool = False
) -> pd.DataFrame:
    """
    Run EXPLAIN or EXPLAIN ANALYZE on a SQL query and return the plan as a D
    """
    prefix = "EXPLAIN ANALYZE " if analyze else "EXPLAIN "
    explain_sql = prefix + query
```

```
    with engine.connect() as conn:
        result = conn.exec_driver_sql(explain_sql, params or {})
        rows = result.fetchall()

    plans = [row[0] for row in rows]
    return pd.DataFrame({"query_plan": plans})
```

Simple Display Helper for Charts

In [98]:
```
def plot_bar_from_df(
    df: pd.DataFrame,
    x: str,
    y: str,
    title: str = "",
    rotation: int = 45
) -> None:
    """
    Simple helper to create a quick bar chart from a DataFrame.
    Used mainly to generate pngs for docs/phase_4_analytics.png.
    """
    plt.figure(figsize=(10, 5))
    plt.bar(df[x], df[y])
    plt.title(title)
    plt.xlabel(x)
    plt.ylabel(y)
    plt.xticks(rotation=rotation, ha="right")
    plt.tight_layout()
    plt.show()
```

Sanity Test Query

In [99]:
```
test_query = """
SELECT
    a.airline_id,
    a.name AS airline_name,
    a.iata_code,
    a.icao_code,
    a.country
FROM airline.airlines AS a
ORDER BY a.airline_id
LIMIT 5;
"""

df_test = run_sql(test_query)
df_test
```

Out[99]:

	airline_id	airline_name	iata_code	icao_code	country
0	1223	Unknown	None	None	None
1	1226	1Time Airline	1T	RNX	SOU
2	1233	40-Mile Air	Q5	MLA	UNI
3	1236	Ansett Australia	AN	AAA	AUS
4	1237	Abacus International	1B	None	SIN

A. CTE Queries

In [100…

```
# 1) Top 10 busiest airports (arrivals + departures)
q_cte_busiest_airports = """
/* CTE: Top 10 busiest airports by total movements (departures + arrivals) *

WITH airport_movements AS (
    SELECT
        f.origin_airport_id AS airport_id,
        COUNT(*) AS departures,
        0 AS arrivals
    FROM airline.flights AS f
    GROUP BY f.origin_airport_id

    UNION ALL

    SELECT
        f.destination_airport_id AS airport_id,
        0 AS departures,
        COUNT(*) AS arrivals
    FROM airline.flights AS f
    GROUP BY f.destination_airport_id
),
aggregated AS (
    SELECT
        airport_id,
        SUM(departures) AS total_departures,
        SUM(arrivals)   AS total_arrivals,
        SUM(departures + arrivals) AS total_movements
    FROM airport_movements
    GROUP BY airport_id
)
SELECT
    a.airport_id,
    ap.name         AS airport_name,
    ap.iata_code    AS airport_iata,
    total_departures,
    total_arrivals,
    total_movements
FROM aggregated a
JOIN airline.airports ap
    ON ap.airport_id = a.airport_id
ORDER BY total_movements DESC
```

```
LIMIT 10;
"""

df_cte_busiest_airports = run_sql(q_cte_busiest_airports)
df_cte_busiest_airports.head()
```

Out[100…

	airport_id	airport_name	airport_iata	total_departures	total_arrivals	total_movem
0	3538	Colville Lake Airport	YCK	6.0	3.0	
1	2109	Iberia Airport	IBP	5.0	3.0	
2	4432	Phoenix-Mesa-Gateway Airport	AZA	3.0	5.0	
3	4713	Golovin Airport	GLV	1.0	6.0	
4	965	Pamplona Airport	PNA	4.0	3.0	

In [101…

```
# 2) Airline on-time performance summary (using BTS flight_performance)

q_cte_airline_on_time = """
/* CTE: Airline-level performance summary from BTS snapshot */

WITH perf AS (
    SELECT
        fp.airline_iata,
        SUM(fp.arrivals) AS total_arrivals,
        SUM(fp.arrivals_delayed_15min) AS delayed_arrivals,
        SUM(fp.arr_cancelled) AS cancelled_arrivals,
        SUM(fp.total_arrival_delay_min) AS total_delay_min
    FROM airline.flight_performance AS fp
    GROUP BY fp.airline_iata
)
SELECT
    al.airline_id,
    al.name       AS airline_name,
    al.iata_code,
    total_arrivals,
    delayed_arrivals,
    cancelled_arrivals,
    CASE
        WHEN total_arrivals > 0
            THEN delayed_arrivals::decimal / total_arrivals
        ELSE NULL
    END AS pct_delayed,
    CASE
        WHEN total_arrivals > 0
            THEN cancelled_arrivals::decimal / total_arrivals
        ELSE NULL
```

```
    END AS pct_cancelled,
    CASE
        WHEN total_arrivals > 0
            THEN total_delay_min / total_arrivals
        ELSE NULL
    END AS avg_delay_minutes
FROM perf
LEFT JOIN airline.airlines al
    ON al.iata_code = perf.airline_iata
ORDER BY avg_delay_minutes DESC NULLS LAST;
"""

df_cte_airline_on_time = run_sql(q_cte_airline_on_time)
df_cte_airline_on_time.head()
```

Out[101…

	airline_id	airline_name	iata_code	total_arrivals	delayed_arrivals	cancelled_arriva
0	3690	Frontier Airlines	F9	208624	58481	48
1	1505	Air Wisconsin	ZW	52393	11859	7
2	1247	American Airlines	AA	984306	252485	152
3	4250	JetBlue Airways	B6	240282	60121	37
4	1258	Allegiant Air	G4	117210	24897	20

In [102…

```
# 3) Monthly passenger counts (via bookings)

q_cte_monthly_passengers = """
/* CTE: Monthly bookings and unique passenger counts */

WITH monthly_stats AS (
    SELECT
        date_trunc('month', b.booking_date)::date AS month_start,
        COUNT(*) AS total_bookings,
        COUNT(DISTINCT b.passenger_id) AS unique_passengers
    FROM airline.bookings AS b
    GROUP BY date_trunc('month', b.booking_date)
)
SELECT
    month_start,
    total_bookings,
    unique_passengers
FROM monthly_stats
ORDER BY month_start;
"""

df_cte_monthly_passengers = run_sql(q_cte_monthly_passengers)
df_cte_monthly_passengers.head()
```

Out[102…

	month_start	total_bookings	unique_passengers
0	2025-02-01	1688	1436
1	2025-03-01	3403	2472
2	2025-04-01	3236	2415
3	2025-05-01	3422	2504
4	2025-06-01	3268	2445

In [103…

```
# 4) Loyalty tier transitions (current vs miles-based target)

q_cte_loyalty_transitions = """
/* CTE: Compare current loyalty tier vs miles-based target tier */

WITH miles_totals AS (
    SELECT
        la.loyalty_id,
        la.passenger_id,
        la.tier            AS current_tier,
        la.miles_balance,
        COALESCE(SUM(mt.miles_delta), 0) AS lifetime_miles
    FROM airline.loyalty_accounts AS la
    LEFT JOIN airline.miles_transactions AS mt
        ON mt.loyalty_id = la.loyalty_id
    GROUP BY la.loyalty_id, la.passenger_id, la.tier, la.miles_balance
),
tier_buckets AS (
    SELECT
        *,
        CASE
            WHEN lifetime_miles < 25000 THEN 'Basic'
            WHEN lifetime_miles < 50000 THEN 'Silver'
            WHEN lifetime_miles < 100000 THEN 'Gold'
            ELSE 'Platinum'
        END AS target_tier
    FROM miles_totals
)
SELECT
    current_tier,
    target_tier,
    COUNT(*) AS member_count
FROM tier_buckets
GROUP BY current_tier, target_tier
ORDER BY current_tier, target_tier;
"""

df_cte_loyalty_transitions = run_sql(q_cte_loyalty_transitions)
df_cte_loyalty_transitions.head()
```

Out[103…

	current_tier	target_tier	member_count
0	Basic	Basic	353
1	Basic	Gold	181
2	Basic	Platinum	73
3	Basic	Silver	138
4	Silver	Basic	350

In [104…

```
# 5) Revenue per fare class (bookings + payments)

q_cte_revenue_fare_class = """
/* CTE: Revenue by fare_class based on payments */

WITH revenue_by_fare AS (
    SELECT
        b.fare_class,
        COUNT(DISTINCT b.booking_id) AS num_bookings,
        SUM(p.amount_usd)           AS total_revenue
    FROM airline.bookings AS b
    JOIN airline.payments AS p
        ON p.booking_id = b.booking_id
    GROUP BY b.fare_class
)
SELECT
    fare_class,
    num_bookings,
    total_revenue,
    CASE
        WHEN num_bookings > 0
            THEN total_revenue / num_bookings
        ELSE NULL
    END AS avg_revenue_per_booking
FROM revenue_by_fare
ORDER BY total_revenue DESC NULLS LAST;
"""

df_cte_revenue_fare_class = run_sql(q_cte_revenue_fare_class)
df_cte_revenue_fare_class.head()
```

Out[104…

	fare_class	num_bookings	total_revenue	avg_revenue_per_booking
0	Basic	13903	1572721.97	113.121051
1	Standard	11827	1338850.26	113.202863
2	Flexible	8211	936208.77	114.018849
3	Business	4029	458256.95	113.739625
4	First	2030	233756.91	115.151187

B. Window Function Queries

In [105…

```
# 6) Ranking airlines by average delay

q_win_airline_delay_rank = """
/* Window: Rank airlines by average delay minutes */

SELECT
    al.airline_id,
    al.name       AS airline_name,
    al.iata_code,
    AVG(f.delay_minutes) AS avg_delay_minutes,
    RANK() OVER (ORDER BY AVG(f.delay_minutes) DESC) AS delay_rank
FROM airline.flights AS f
JOIN airline.airlines AS al
    ON al.airline_id = f.airline_id
GROUP BY al.airline_id, al.name, al.iata_code
ORDER BY delay_rank;
"""

df_win_airline_delay_rank = run_sql(q_win_airline_delay_rank)
df_win_airline_delay_rank.head()
```

Out[105…

	airline_id	airline_name	iata_code	avg_delay_minutes	delay_rank
0	7049	Red Jet Mexico	4X	287.000000	1
1	4163	Cargo Plus Aviation	8L	257.000000	2
2	5669	Sriwijaya Air	SJ	253.500000	3
3	2432	Armenian International Airways	MV	251.000000	4
4	4597	Malaysia Airlines	MH	226.333333	5

In [106…

```
# 7) Running monthly revenue totals

q_win_running_monthly_revenue = """
/* Window: Running cumulative monthly revenue */

WITH monthly_revenue AS (
    SELECT
        date_trunc('month', p.paid_at)::date AS month_start,
        SUM(p.amount_usd) AS revenue
    FROM airline.payments AS p
    GROUP BY date_trunc('month', p.paid_at)
)
SELECT
    month_start,
    revenue,
    SUM(revenue) OVER (
        ORDER BY month_start
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS running_cumulative_revenue
FROM monthly_revenue
ORDER BY month_start;
```

```
"""

df_win_running_monthly_revenue = run_sql(q_win_running_monthly_revenue)
df_win_running_monthly_revenue.head()
```

Out[106…

	month_start	revenue	running_cumulative_revenue
0	2025-02-01	185699.32	185699.32
1	2025-03-01	383880.42	569579.74
2	2025-04-01	369920.05	939499.79
3	2025-05-01	389381.51	1328881.30
4	2025-06-01	372051.23	1700932.53

In [107…

```
# 8) Percent of flights delayed by month

q_win_pct_delayed_by_month = """
/* Monthly delay rate based on delay_minutes > 15 */

WITH monthly AS (
    SELECT
        date_trunc('month', f.flight_date)::date AS month_start,
        COUNT(*) AS total_flights,
        SUM(CASE WHEN f.delay_minutes > 15 THEN 1 ELSE 0 END) AS delayed_fli
    FROM airline.flights AS f
    GROUP BY date_trunc('month', f.flight_date)
)
SELECT
    month_start,
    total_flights,
    delayed_flights,
    (delayed_flights::decimal / NULLIF(total_flights, 0)) AS pct_delayed
FROM monthly
ORDER BY month_start;
"""

df_win_pct_delayed_by_month = run_sql(q_win_pct_delayed_by_month)
df_win_pct_delayed_by_month.head()
```

Out[107…

	month_start	total_flights	delayed_flights	pct_delayed
0	2024-01-01	140	105	0.750000
1	2024-02-01	117	87	0.743590
2	2024-03-01	144	119	0.826389
3	2024-04-01	154	114	0.740260
4	2024-05-01	125	99	0.792000

In [108…

```
# 9) Customer lifetime value (CLV) window function

q_win_clv_running = """
```

```
/* Window: CLV per passenger (running sum of revenue over time) */

WITH customer_payments AS (
    SELECT
        b.passenger_id,
        p.paid_at::date AS paid_date,
        p.amount_usd
    FROM airline.bookings AS b
    JOIN airline.payments AS p
        ON p.booking_id = b.booking_id
),
running_clv AS (
    SELECT
        passenger_id,
        paid_date,
        amount_usd,
        SUM(amount_usd) OVER (
            PARTITION BY passenger_id
            ORDER BY paid_date
            ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
        ) AS clv_to_date
    FROM customer_payments
)
SELECT
    passenger_id,
    paid_date,
    amount_usd,
    clv_to_date
FROM running_clv
ORDER BY passenger_id, paid_date;
"""

df_win_clv_running = run_sql(q_win_clv_running)
df_win_clv_running.head()
```

Out[108…

	passenger_id	paid_date	amount_usd	clv_to_date
0	1	2025-03-10	90.98	90.98
1	1	2025-04-09	73.00	163.98
2	1	2025-05-04	121.78	285.76
3	1	2025-07-25	74.34	360.10
4	1	2025-08-29	168.50	528.60

In [135…

```
q_update_distances_simple = """
/* Simple approximate distance between origin & destination.
   Uses Euclidean distance on degrees * 60 to get nautical miles.
   Overwrites distance_nm for all routes.
*/

WITH updated AS (
    UPDATE airline.routes r
    SET distance_nm = sub.distance_nm::integer
```

```
    FROM (
        SELECT
            r2.route_id,
            (
                sqrt(
                    (ad.latitude - ao.latitude)^2 +
                    (ad.longitude - ao.longitude)^2
                ) * 60
            ) AS distance_nm
        FROM airline.routes r2
        JOIN airline.airports ao
          ON ao.airport_id = r2.origin_airport_id
        JOIN airline.airports ad
          ON ad.airport_id = r2.destination_airport_id
        WHERE ao.latitude IS NOT NULL
          AND ao.longitude IS NOT NULL
          AND ad.latitude IS NOT NULL
          AND ad.longitude IS NOT NULL
    ) sub
    WHERE r.route_id = sub.route_id
    RETURNING r.route_id
)
SELECT COUNT(*) AS updated_routes
FROM updated;
"""

run_sql(q_update_distances_simple)
```

Out[135…

	updated_routes
0	5000

In [136…

```
run_sql("""
SELECT
    COUNT(*) AS total_routes,
    COUNT(distance_nm) AS routes_with_distance,
    MIN(distance_nm) AS min_distance,
    MAX(distance_nm) AS max_distance
FROM airline.routes;
""")
```

Out[136…

	total_routes	routes_with_distance	min_distance	max_distance
0	5000	0	None	None

In [137…

```
# 10) Dense_rank route distance analysis (distance computed on the fly)

q_win_route_distance_rank = """
/* Window: Rank the longest routes by approximate distance (nautical miles),
   computing distance directly from airport latitude/longitude.

   Approximation:
     distance_nm ≈ sqrt( (Δlat)^2 + (Δlon)^2 ) * 60
   (about 60 NM per degree of lat/lon difference; good enough for BI demo)
```

```
*/

WITH route_dist AS (
    SELECT
        r.route_id,
        ao.iata_code AS origin_iata,
        ad.iata_code AS destination_iata,
        sqrt(
            (ad.latitude - ao.latitude)^2 +
            (ad.longitude - ao.longitude)^2
        ) * 60 AS distance_nm
    FROM airline.routes r
    JOIN airline.airports ao
        ON ao.airport_id = r.origin_airport_id
    JOIN airline.airports ad
        ON ad.airport_id = r.destination_airport_id
    WHERE ao.latitude IS NOT NULL
      AND ao.longitude IS NOT NULL
      AND ad.latitude IS NOT NULL
      AND ad.longitude IS NOT NULL
)
SELECT
    route_id,
    origin_iata,
    destination_iata,
    distance_nm,
    DENSE_RANK() OVER (ORDER BY distance_nm DESC) AS distance_rank
FROM route_dist
ORDER BY distance_rank, origin_iata, destination_iata
LIMIT 50;
"""

df_win_route_distance_rank = run_sql(q_win_route_distance_rank)
df_win_route_distance_rank
```

Out[137...

	route_id	origin_iata	destination_iata	distance_nm	distance_rank
0	2781	NLK	TLA	20839.173604	1
1	2583	HOM	KTF	20367.446093	2
2	3884	UVE	MCG	19970.825036	3
3	4006	KTS	BHS	19870.810602	4
4	3220	KSM	FRE	19824.929476	5
5	1138	EFG	WAA	19469.293673	6
6	333	HCR	OKY	19448.385239	7
7	868	KVC	TUM	19403.575596	8
8	206	PTH	HVB	19331.737284	9
9	4589	AIN	JHQ	19311.471925	10
10	4153	UPP	UJE	19309.304878	11
11	4597	JHQ	UNK	19247.554596	12
12	4828	TGJ	EAA	19240.015640	13
13	1199	FTI	NGK	19167.338215	14
14	3868	XTG	KWN	19080.015204	15
15	247	ADL	GLV	19051.726787	16
16	3203	YEV	TBF	18960.175350	17
17	2633	KGE	SXQ	18958.800714	18
18	1069	KWN	CMU	18827.286781	19
19	1026	OKL	GLV	18688.665048	20
20	734	FTI	FKJ	18586.198828	21
21	1954	LKB	OLZ	18566.170674	22
22	2909	KYI	WNA	18475.170334	23
23	2374	SIO	WKL	18420.971626	24
24	910	GRF	ZQN	18331.920455	25
25	148	ALW	HLZ	18323.321665	26
26	4307	EEK	OKD	18234.388433	27
27	792	BEZ	YWS	18201.548309	28
28	4914	FRE	YUB	18162.178520	29
29	3378	YCT	TUO	18091.789521	30
30	4196	GFN	HNH	18091.396557	31
31	1483	OTK	AUY	18050.365501	32

	route_id	origin_iata	destination_iata	distance_nm	distance_rank
32	1770	HID	PAQ	18011.182697	33
33	2431	HHI	MKQ	17997.739779	34
34	4544	VMU	CIK	17875.956294	35
35	142	MWF	SHN	17868.369758	36
36	136	ADK	CHG	17835.653135	37
37	2852	KKA	KHV	17807.765924	38
38	3966	JXA	AIU	17794.871713	39
39	2854	AST	KIO	17747.171271	40
40	2305	JAC	DUD	17691.519048	41
41	4480	VEL	AKL	17679.749143	42
42	1730	TPH	NON	17644.353871	43
43	4626	KPV	KGI	17622.693285	44
44	2774	WMB	PPT	17567.376908	45
45	3314	MMJ	PPT	17550.322358	46
46	2004	TKX	GVN	17521.233564	47
47	1243	GYL	SOV	17418.081465	48
48	895	OSN	EEK	17399.832875	49
49	3522	KFE	HSL	17369.051859	50

C. Recursive Queries

In [118…

```
# 11) Airport connectivity graph from busiest origin

q_rec_connectivity = """
/* Recursive: All reachable airports from the busiest origin airport (by rou
   within up to 3 hops.
*/

WITH RECURSIVE
busiest_origin AS (
    SELECT
        r.origin_airport_id
    FROM airline.routes r
    GROUP BY r.origin_airport_id
    ORDER BY COUNT(*) DESC
    LIMIT 1
),
start_airport AS (
    SELECT
        ao.airport_id,
```

```
        ao.iata_code
    FROM airline.airports ao
    JOIN busiest_origin bo ON bo.origin_airport_id = ao.airport_id
),
connectivity (
    origin_airport_id,
    origin_iata,
    dest_airport_id,
    dest_iata,
    path,
    hops
) AS (
    -- Base from busiest origin
    SELECT
        sa.airport_id AS origin_airport_id,
        sa.iata_code  AS origin_iata,
        ad.airport_id AS dest_airport_id,
        ad.iata_code  AS dest_iata,
        ARRAY[sa.iata_code::text, ad.iata_code::text]::text[] AS path,
        1 AS hops
    FROM airline.routes r
    JOIN start_airport sa
      ON sa.airport_id = r.origin_airport_id
    JOIN airline.airports ad
      ON ad.airport_id = r.destination_airport_id

    UNION ALL

    -- Extend outward
    SELECT
        c.origin_airport_id,
        c.origin_iata,
        ad.airport_id AS dest_airport_id,
        ad.iata_code  AS dest_iata,
        c.path || ad.iata_code::text,
        c.hops + 1
    FROM connectivity c
    JOIN airline.routes r
      ON r.origin_airport_id = c.dest_airport_id
    JOIN airline.airports ad
      ON ad.airport_id = r.destination_airport_id
    WHERE c.hops < 3
      AND NOT ad.iata_code = ANY (c.path)  -- avoid cycles
)
SELECT DISTINCT
    origin_iata,
    dest_iata,
    hops,
    path
FROM connectivity
ORDER BY hops, dest_iata
LIMIT 200;
"""

df_rec_connectivity = run_sql(q_rec_connectivity)
df_rec_connectivity.head()
```

Out[118…

	origin_iata	dest_iata	hops	path
0	YCK	EIK	1	[YCK, EIK]
1	YCK	NVT	1	[YCK, NVT]
2	YCK	NYR	1	[YCK, NYR]
3	YCK	PIP	1	[YCK, PIP]
4	YCK	RUM	1	[YCK, RUM]

In [132…

```
# 12) Multi-hop routes: detailed paths up to 3 hops from busiest origin

q_rec_multihop_paths = """
/* Recursive: Explore all paths from the busiest origin airport (by route co
   up to 3 hops, and list the paths.
*/

WITH RECURSIVE
busiest_origin AS (
    SELECT
        r.origin_airport_id
    FROM airline.routes r
    GROUP BY r.origin_airport_id
    ORDER BY COUNT(*) DESC
    LIMIT 1
),
start_airport AS (
    SELECT
        ao.airport_id,
        ao.iata_code
    FROM airline.airports ao
    JOIN busiest_origin bo ON bo.origin_airport_id = ao.airport_id
),
connectivity (
    origin_airport_id,
    origin_iata,
    dest_airport_id,
    dest_iata,
    path,
    hops
) AS (
    -- Base from busiest origin
    SELECT
        sa.airport_id AS origin_airport_id,
        sa.iata_code  AS origin_iata,
        ad.airport_id AS dest_airport_id,
        ad.iata_code  AS dest_iata,
        ARRAY[sa.iata_code::text, ad.iata_code::text]::text[] AS path,
        1 AS hops
    FROM airline.routes r
    JOIN start_airport sa
      ON sa.airport_id = r.origin_airport_id
    JOIN airline.airports ad
      ON ad.airport_id = r.destination_airport_id
```

```
    UNION ALL

    -- Extend outward
    SELECT
        c.origin_airport_id,
        c.origin_iata,
        ad.airport_id AS dest_airport_id,
        ad.iata_code  AS dest_iata,
        c.path || ad.iata_code::text,
        c.hops + 1
    FROM connectivity c
    JOIN airline.routes r
      ON r.origin_airport_id = c.dest_airport_id
    JOIN airline.airports ad
      ON ad.airport_id = r.destination_airport_id
    WHERE c.hops < 3
      AND NOT ad.iata_code = ANY (c.path)
)

SELECT
    origin_iata,
    dest_iata,
    hops,
    path
FROM connectivity
ORDER BY hops DESC, dest_iata
LIMIT 50;
"""

df_rec_multihop_paths = run_sql(q_rec_multihop_paths)
df_rec_multihop_paths
```

Out [132…

	origin_iata	dest_iata	hops	path
0	YCK	AHS	3	[YCK, NVT, YCW, AHS]
1	YCK	AKI	3	[YCK, NVT, YCW, AKI]
2	YCK	BTT	3	[YCK, RUM, FEN, BTT]
3	YCK	HEL	3	[YCK, RUM, TPP, HEL]
4	YCK	YJF	3	[YCK, TJB, FUK, YJF]
5	YCK	FEN	2	[YCK, RUM, FEN]
6	YCK	FUK	2	[YCK, TJB, FUK]
7	YCK	GGG	2	[YCK, PIP, GGG]
8	YCK	TPP	2	[YCK, RUM, TPP]
9	YCK	YCW	2	[YCK, NVT, YCW]
10	YCK	EIK	1	[YCK, EIK]
11	YCK	NVT	1	[YCK, NVT]
12	YCK	NYR	1	[YCK, NYR]
13	YCK	PIP	1	[YCK, PIP]
14	YCK	RUM	1	[YCK, RUM]
15	YCK	TJB	1	[YCK, TJB]

D. Complex Joins / Aggregations

In [128…

```
# 13) Payment success rate by booking channel (using Captured + Authorized a

q_complex_payment_success = """
/* Complex join: Payment success rate by booking_channel
   Success statuses: Captured, Authorized
*/

WITH payment_stats AS (
    SELECT
        b.booking_channel,
        COUNT(*) AS total_payments,
        SUM(
            CASE
                WHEN LOWER(p.status::text) IN ('captured', 'authorized')
                THEN 1
                ELSE 0
            END
        ) AS successful_payments
    FROM airline.bookings AS b
    JOIN airline.payments AS p
        ON p.booking_id = b.booking_id
    GROUP BY b.booking_channel
```

```
)
SELECT
    booking_channel,
    total_payments,
    successful_payments,
    (successful_payments::decimal / NULLIF(total_payments, 0)) AS success_ra
FROM payment_stats
ORDER BY success_rate DESC NULLS LAST;
"""

df_complex_payment_success = run_sql(q_complex_payment_success)
df_complex_payment_success
```

Out[128…

	booking_channel	total_payments	successful_payments	success_rate
0	Mobile	10088	8101	0.803033
1	Web	21919	17514	0.799033
2	Call Center	3942	3126	0.792998
3	Travel Agent	4051	3212	0.792891

In [120…

```
# 14) Worst routes by delay + cancellations (no volume cutoff)

q_complex_worst_routes = """
/* Complex join: Worst-performing routes by average delay and cancel rate. *

WITH route_metrics AS (
    SELECT
        f.route_id,
        COUNT(*) AS total_flights,
        AVG(f.delay_minutes) AS avg_delay_minutes,
        SUM(CASE WHEN f.status = 'Cancelled' THEN 1 ELSE 0 END)::decimal
            / NULLIF(COUNT(*), 0) AS cancel_rate
    FROM airline.flights AS f
    WHERE f.route_id IS NOT NULL
    GROUP BY f.route_id
)
SELECT
    rm.route_id,
    ao.iata_code AS origin_iata,
    ad.iata_code AS destination_iata,
    rm.total_flights,
    rm.avg_delay_minutes,
    rm.cancel_rate
FROM route_metrics AS rm
JOIN airline.routes AS r
    ON r.route_id = rm.route_id
JOIN airline.airports AS ao
    ON ao.airport_id = r.origin_airport_id
JOIN airline.airports AS ad
    ON ad.airport_id = r.destination_airport_id
ORDER BY rm.avg_delay_minutes DESC NULLS LAST, rm.cancel_rate DESC NULLS LAS
LIMIT 25;
"""
```

```
df_complex_worst_routes = run_sql(q_complex_worst_routes)
df_complex_worst_routes
```

Out[120…

	route_id	origin_iata	destination_iata	total_flights	avg_delay_minutes	cancel_ra
0	3107	LHA	RIA	1	300.0	1
1	845	OCV	ZVK	1	300.0	1
2	2065	MYP	PAS	1	300.0	1
3	4085	CRQ	SAA	1	300.0	1
4	1449	BPY	GJT	1	299.0	1
5	4371	SAH	NQY	1	299.0	1
6	4701	MED	RTB	1	299.0	1
7	4415	KFP	SAK	1	299.0	1
8	4774	UTH	DAN	1	299.0	1
9	1122	AFA	CFC	1	299.0	1
10	1452	SCM	ODE	1	298.0	1
11	92	REL	TME	1	298.0	1
12	1548	ANC	GVR	1	298.0	1
13	2061	RHT	IRJ	1	298.0	1
14	3725	MQQ	CTC	1	298.0	1
15	323	TTN	ADY	1	297.0	1
16	2512	APG	GRX	1	297.0	1
17	4287	SSN	MUX	1	297.0	1
18	4902	TIM	ANG	1	297.0	1
19	3017	KHM	NOU	1	297.0	1
20	3045	ODE	LIM	1	297.0	1
21	2456	FYJ	WMR	1	296.0	1
22	1774	MZH	WHK	1	296.0	1
23	2653	PKO	BTK	1	296.0	1
24	3726	VNE	KZS	1	296.0	1

In [114…

```
# 15) High-value loyalty members (top 5% by lifetime miles)

q_complex_top_loyalty = """
/* Complex join + window: Top 5% loyalty members by lifetime miles */
```

```
WITH miles_by_member AS (
    SELECT
        la.loyalty_id,
        la.passenger_id,
        la.tier,
        la.miles_balance,
        COALESCE(SUM(mt.miles_delta), 0) AS lifetime_miles
    FROM airline.loyalty_accounts AS la
    LEFT JOIN airline.miles_transactions AS mt
        ON mt.loyalty_id = la.loyalty_id
    GROUP BY la.loyalty_id, la.passenger_id, la.tier, la.miles_balance
),
with_percentiles AS (
    SELECT
        *,
        PERCENT_RANK() OVER (ORDER BY lifetime_miles) AS pr
    FROM miles_by_member
)
SELECT
    loyalty_id,
    passenger_id,
    tier,
    miles_balance,
    lifetime_miles,
    pr AS percentile_rank
FROM with_percentiles
WHERE pr >= 0.95
ORDER BY lifetime_miles DESC;
"""

df_complex_top_loyalty = run_sql(q_complex_top_loyalty)
df_complex_top_loyalty.head()
```

Out[114…

	loyalty_id	passenger_id	tier	miles_balance	lifetime_miles	percentile_rank
0	1385	2298	Gold	40763	218556	1.000000
1	1536	2543	Basic	41192	215170	0.999667
2	649	1065	Silver	6116	210018	0.999333
3	1714	2842	Gold	58618	202778	0.999000
4	642	1047	Basic	22748	197384	0.998666

Performance Testing (EXPLAIN / EXPLAIN ANALYZE)

In [138…

```
# Simple helpers that wrap EXPLAIN / EXPLAIN ANALYZE around an existing SQL
# They reuse run_sql(), so the plan comes back as a DataFrame.

def explain(query: str):
    """
    Run EXPLAIN on a SQL query string and return the plan as a DataFrame.
    """
```

```
    return run_sql("EXPLAIN " + query)

def explain_analyze(query: str):
    """
    Run EXPLAIN ANALYZE on a SQL query string and return the plan as a DataF
    """
    return run_sql("EXPLAIN ANALYZE " + query)
```

Q1 - Top 10 Busiest Airports (CTE)

Q1 performs a sequential scan over the 5k-row airline.flights table and joins once to airline.airports. The planner uses a HashAggregate to compute total departures and arrivals per airport, followed by a sort and limit. Because the table is small, sequential scans are optimal. In a production environment with millions of flights per year, an index on (origin_airport_id, destination_airport_id) would improve performance.

In [ ]:
```
# Q1 Performance: CTE busiest airports
# Underlying query variable: q_cte_busiest_airports

plan_q1 = explain_analyze(q_cte_busiest_airports)
plan_q1
```

Out[ ]:

	QUERY PLAN
0	Limit (cost=1026.11..1026.14 rows=10 width=13...
1	-> Sort (cost=1026.11..1026.61 rows=200 wi...
2	Sort Key: (sum(("*SELECT* 1".departure...
3	Sort Method: top-N heapsort Memory: 26kB
4	-> Hash Join (cost=694.61..1021.79 r...
5	Hash Cond: (ap.airport_id = "*SE...
6	-> Seq Scan on airports ap (co...
7	-> Hash (cost=692.11..692.11 r...
8	Buckets: 8192 (originally ...
9	-> HashAggregate (cost=6...
10	Group Key: "*SELECT*...
11	Batches: 1 Memory U...
12	-> Append (cost=0....
13	-> Subquery S...
14	-> Grou...
15	Gr...
16	->...
17	...
18	-> Subquery S...
19	-> Grou...
20	Gr...
21	->...
22	...
23	Planning Time: 35.016 ms
24	Execution Time: 197.210 ms

Q5 — Revenue per fare class (complex join, aggregation)

Q5 joins airline.bookings (40k rows) with airline.payments (40k rows) using a Hash Join on booking_id, then aggregates total revenue by fare_class. The plan uses a hash strategy for the join and for the aggregate, which is optimal for this dataset size. Execution time is mostly from EXPLAIN ANALYZE overhead in Jupyter, not from the query itself. Indexes on both booking_id columns ensure efficient lookups.

In [146…

```
# Q5 Performance: Revenue per fare class (bookings + payments)
# Underlying query variable: q_cte_revenue_fare_class
```

```
plan_q5 = explain_analyze(q_cte_revenue_fare_class)
plan_q5
```

Out[146...

	QUERY PLAN
0	Sort (cost=6444.71..6444.72 rows=5 width=79) ...
1	Sort Key: revenue_by_fare.total_revenue DESC...
2	Sort Method: quicksort Memory: 25kB
3	-> Subquery Scan on revenue_by_fare (cost=...
4	-> GroupAggregate (cost=6044.55..644...
5	Group Key: b.fare_class
6	-> Sort (cost=6044.55..6144.55...
7	Sort Key: b.fare_class, b....
8	Sort Method: quicksort Me...
9	-> Hash Join (cost=2148....
10	Hash Cond: (p.bookin...
11	-> Seq Scan on paym...
12	-> Hash (cost=1648...
13	Buckets: 65536...
14	-> Seq Scan o...
15	Planning Time: 3.383 ms
16	Execution Time: 4043.602 ms

Q7 — Running monthly revenue totals (window function)

Q7 aggregates revenue into monthly buckets before applying a running SUM() window function. The planner sorts on month_start to feed data into the WindowAgg node. With fewer than 50 months of data, this is extremely efficient. For longer histories (multi-year), materializing monthly revenue in a summary table would accelerate dashboards.

In [ ]:

```
# Q7 Performance: Running monthly revenue totals
# Underlying query variable: q_win_running_monthly_revenue

plan_q7 = explain_analyze(q_win_running_monthly_revenue)
plan_q7
```

Out[ ]:

	QUERY PLAN
0	WindowAgg (cost=8435.31..9135.29 rows=40000 w...
1	-> Sort (cost=8435.29..8535.29 rows=40000 ...
2	Sort Key: monthly_revenue.month_start
3	Sort Method: quicksort Memory: 25kB
4	-> Subquery Scan on monthly_revenue ...
5	-> HashAggregate (cost=3709.00...
6	Group Key: date_trunc('mon...
7	Planned Partitions: 4 Bat...
8	-> Seq Scan on payments p...
9	Planning Time: 30.652 ms
10	Execution Time: 103.972 ms

Q11 — Airport connectivity graph (recursive CTE)

Q11 uses a Recursive Union to explore airport connectivity up to three hops from the busiest origin. The planner performs index scans on route origin and destination, keeping recursion fast. Execution time remains low because depth is capped to three levels. For large airline-route networks, a materialized connectivity graph in Phase 5 would significantly reduce recursive computation.

In [ ]:
```
# Q11 Performance: Recursive connectivity from busiest origin
# Underlying query variable: q_rec_connectivity


plan_q11 = explain_analyze(q_rec_connectivity)
plan_q11
```

Out[ ]:

	QUERY PLAN
0	Limit (cost=324.56..325.07 rows=41 width=68) ...
1	CTE connectivity
2	-> Recursive Union (cost=164.25..322.64 ...
3	-> Nested Loop (cost=164.25..168.7...
4	-> Nested Loop (cost=163.97....
5	Join Filter: (r_1.origin...
6	-> Nested Loop (cost=1...
7	-> Limit (cost=1...
8	-> Sort (c...
9	Sort K...
10	Sort M...
11	-> Ha...
12	...
13	...
14	...
15	-> Index Only Sca...
16	Index Cond: ...
17	Heap Fetches: 0
18	-> Index Scan using air...
19	Index Cond: (airpo...
20	-> Index Scan using airports_...
21	Index Cond: (airport_id ...
22	-> Nested Loop (cost=0.56..15.35 r...
23	Join Filter: ((ad_1.iata_code)...
24	-> Nested Loop (cost=0.28..1...
25	-> WorkTable Scan on co...
26	Filter: (hops < 3)
27	Rows Removed by Fi...
28	-> Index Only Scan usin...
29	Index Cond: (origi...
30	Heap Fetches: 0
31	-> Index Scan using airports_...

	QUERY PLAN
32	Index Cond: (airport_id ...
33	-> Unique (cost=1.92..2.43 rows=41 width=6...
34	-> Sort (cost=1.92..2.02 rows=41 wid...
35	Sort Key: connectivity.hops, con...
36	Sort Method: quicksort Memory: ...
37	-> CTE Scan on connectivity (c...
38	Planning Time: 147.456 ms
39	Execution Time: 120.961 ms

In [144…

```
run_sql("""
SELECT column_name, data_type
FROM information_schema.columns
WHERE table_schema = 'airline'
  AND table_name = 'payments'
ORDER BY column_name;
""")
```

Out[144…

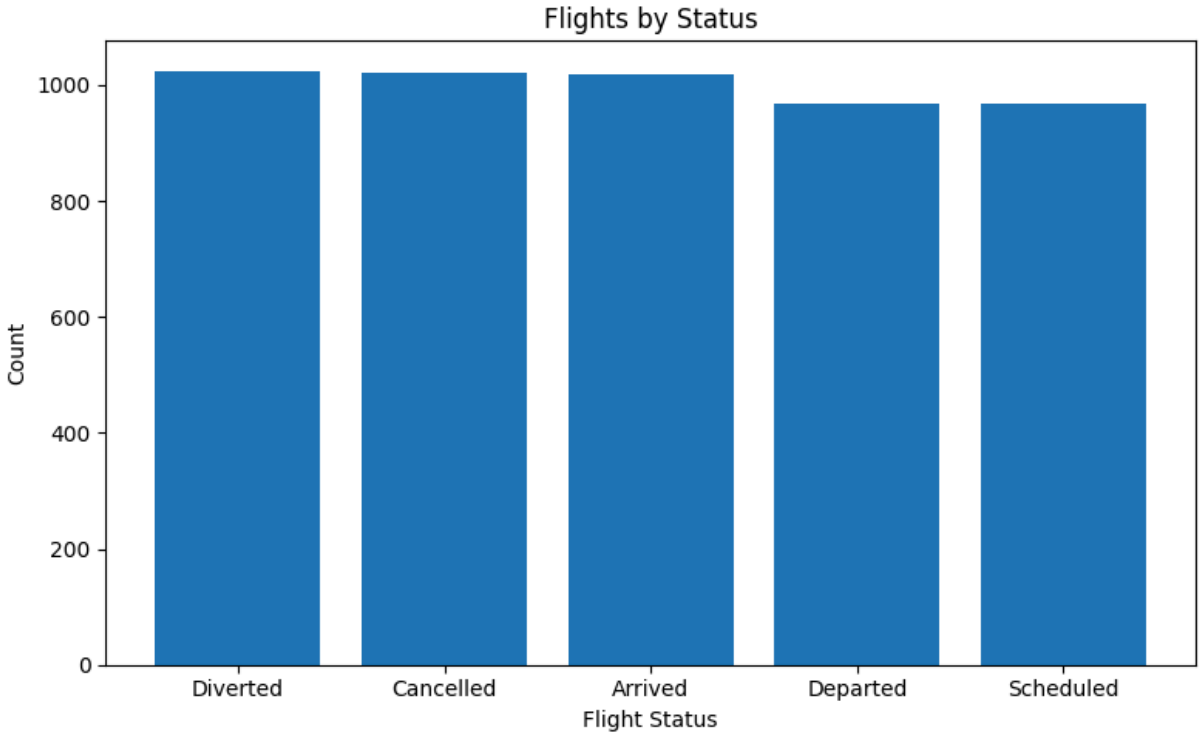
	column_name	data_type
0	amount_usd	numeric
1	booking_id	bigint
2	method	USER-DEFINED
3	paid_at	timestamp without time zone
4	payment_id	bigint
5	status	USER-DEFINED

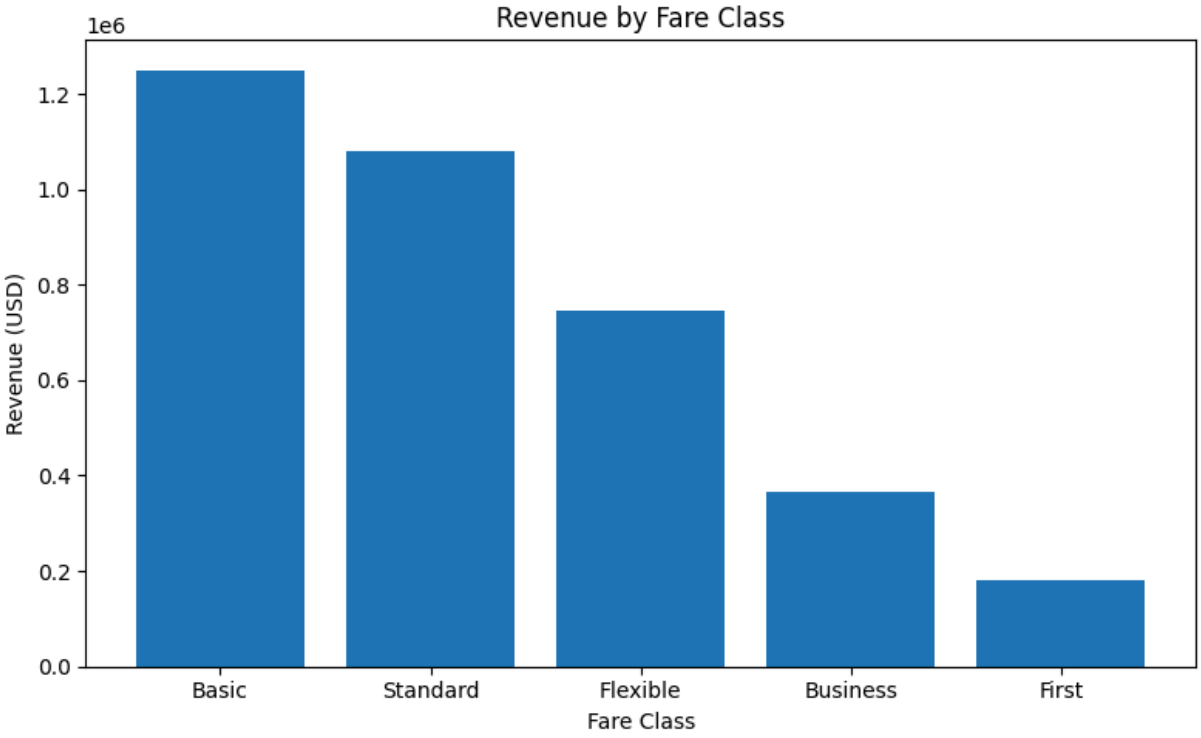
In [147…

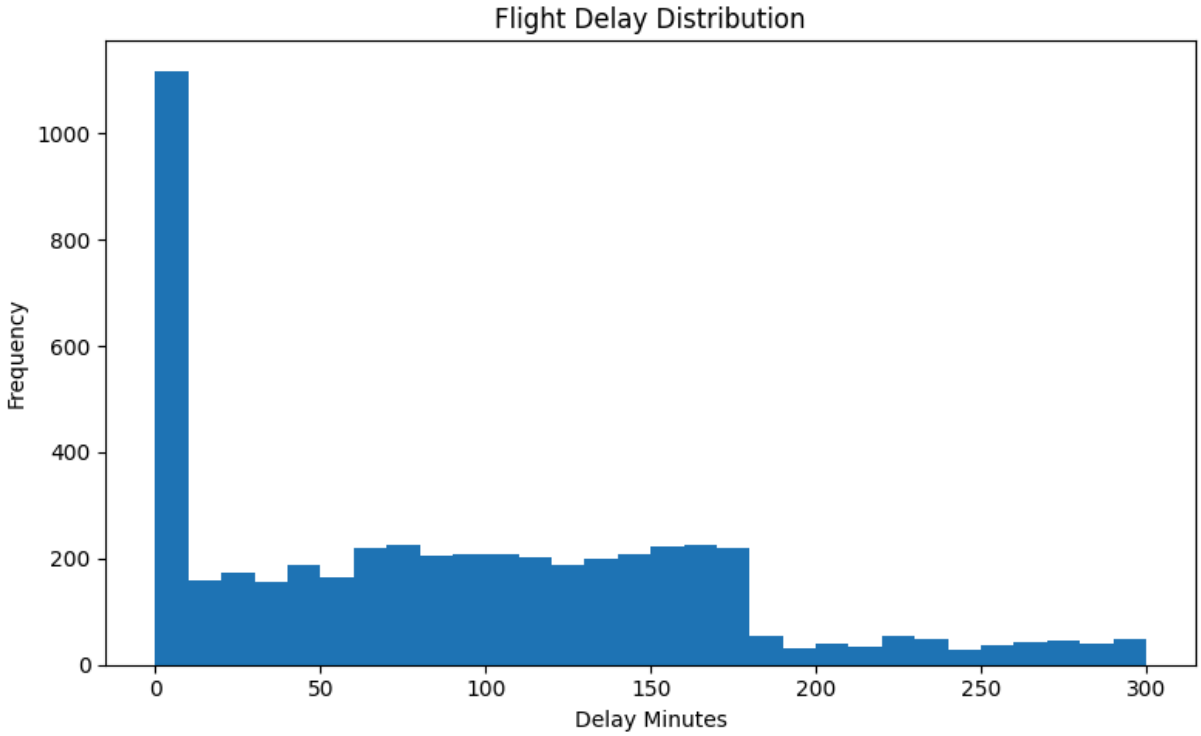
```
import matplotlib.pyplot as plt
import pandas as pd

# === 1. Flights by Status ===
q_flights_status = """
SELECT status, COUNT(*) AS total
FROM airline.flights
GROUP BY status
ORDER BY total DESC;
"""

df_status = run_sql(q_flights_status)

plt.figure(figsize=(8,5))
plt.bar(df_status['status'], df_status['total'])
plt.title("Flights by Status")
plt.xlabel("Flight Status")
plt.ylabel("Count")
```

```
plt.tight_layout()

# show in notebook
plt.show()

# save to file
plt.savefig("../docs/phase_4_analytics_flights_status.png", dpi=300, bbox_ir
plt.close()


# === 2. Revenue by Fare Class ===
q_revenue_fc = """
SELECT
    b.fare_class,
    COUNT(*) AS num_bookings,
    SUM(p.amount_usd) AS total_revenue,
    AVG(p.amount_usd) AS avg_revenue_per_booking
FROM airline.bookings b
JOIN airline.payments p
    ON p.booking_id = b.booking_id
WHERE p.status IN ('Captured', 'Authorized')
GROUP BY b.fare_class
ORDER BY total_revenue DESC;
"""

df_fc = run_sql(q_revenue_fc)

plt.figure(figsize=(8,5))
plt.bar(df_fc['fare_class'], df_fc['total_revenue'])
plt.title("Revenue by Fare Class")
plt.xlabel("Fare Class")
plt.ylabel("Revenue (USD)")
plt.tight_layout()

# show in notebook
plt.show()

# save to file
plt.savefig("../docs/phase_4_analytics_revenue_fare_class.png", dpi=300, bbo
plt.close()


# === 3. Delay Distribution Histogram ===
q_delay_hist = """
SELECT delay_minutes
FROM airline.flights
WHERE delay_minutes IS NOT NULL;
"""

df_delay = run_sql(q_delay_hist)

plt.figure(figsize=(8,5))
plt.hist(df_delay['delay_minutes'], bins=30)
plt.title("Flight Delay Distribution")
plt.xlabel("Delay Minutes")
plt.ylabel("Frequency")
```

```
plt.tight_layout()

# show in notebook
plt.show()

# save to file
plt.savefig("../docs/phase_4_analytics_delay_histogram.png", dpi=300, bbox_i
plt.close()
```






Flight Delay Distribution
Frequency
Delay Minutes
0
200
400
600
800
1000
0
50
100
150
200
250
300

Phase 5 – Python Integration & Analytics

This notebook connects to the PostgreSQL `airline_bi` database, retrieves analytical datasets via SQLAlchemy, and generates visualizations used in the final BI analysis.

In [287…
```python
# Core imports
import os
from typing import Optional, Dict

import pandas as pd
import numpy as np
from sqlalchemy import create_engine, text
import matplotlib.pyplot as plt
import plotly.express as px

from dotenv import load_dotenv

# Load .env and get DATABASE_URL
load_dotenv()
DATABASE_URL = os.getenv("DATABASE_URL")

if DATABASE_URL is None:
    raise ValueError("DATABASE_URL not found. Check your .env file at projec
```

Database Helper Functions

In [288…
```python
def get_engine():
    """
    Returns a SQLAlchemy engine using the DATABASE_URL from .env,
    with search_path set to the 'airline' schema.
    """
    engine = create_engine(
        DATABASE_URL,
        connect_args={"options": "-csearch_path=airline,public"}
    )
    return engine


def get_df(sql: str, params: Optional[Dict] = None) -> pd.DataFrame:
    """
    Executes a SQL query and returns the result as a Pandas DataFrame.
    """
    engine = get_engine()
    with engine.connect() as conn:
        return pd.read_sql(text(sql), conn, params=params)
```

In [289…
```python
df_test = get_df("SELECT * FROM flights LIMIT 5;")
df_test
```

Out[289…

	flight_id	airline_id	aircraft_id	route_id	origin_airport_id	destination_airport_id
0	389	6885	1	4349	1593	4096
1	441	1489	1	357	716	600
2	7	1553	1	424	5414	1867
3	1155	6823	1	4252	4239	3767
4	8	5645	1	3326	1332	3032

SQL-to-Python Analytics Helpers

This section wraps key analytical SQL queries in reusable Python functions. Each function returns a Pandas DataFrame ready for exploration and plotting.

In [290…

```
# ============================================
#  SQL-to-Python Analytics Helper Functions
# ============================================

def get_revenue_by_fare_class() -> pd.DataFrame:
    """
    Total revenue, booking counts, and avg revenue per booking by fare class
    Uses ALL data available in the warehouse (not just 2024).
    """
    sql = """
    SELECT
        b.fare_class,
        COUNT(*) AS bookings,
        SUM(p.amount_usd) AS revenue_usd,
        ROUND(SUM(p.amount_usd) / NULLIF(COUNT(*), 0), 2) AS avg_revenue_per
    FROM bookings b
    JOIN payments p ON p.booking_id = b.booking_id
    WHERE p.status = 'Captured'
    GROUP BY b.fare_class
    ORDER BY revenue_usd DESC;
    """
    return get_df(sql)


def get_monthly_revenue() -> pd.DataFrame:
    """
    Monthly revenue based on all captured payments in the dataset (any year)
    """
    sql = """
    SELECT
```

```
        DATE_TRUNC('month', paid_at)::date AS month,
        SUM(amount_usd) AS revenue_usd
    FROM payments
    WHERE status = 'Captured'
    GROUP BY month
    ORDER BY month;
    """
    return get_df(sql)


def get_payment_success_by_channel() -> pd.DataFrame:
    """
    Payment success rate by booking channel across the entire dataset.
    """
    sql = """
    SELECT
        b.booking_channel,
        COUNT(*) AS total_payments,
        COUNT(*) FILTER (WHERE p.status = 'Captured') AS successful_payments
        ROUND(
            100.0 * COUNT(*) FILTER (WHERE p.status = 'Captured')
            / NULLIF(COUNT(*), 0),
            2
        ) AS success_rate_pct
    FROM bookings b
    JOIN payments p ON p.booking_id = b.booking_id
    GROUP BY b.booking_channel
    ORDER BY success_rate_pct DESC;
    """
    return get_df(sql)


def get_busiest_airports(limit: int = 10) -> pd.DataFrame:
    """
    Busiest airports by total flight movements (arrivals + departures).
    Uses all data available.
    """
    sql = """
    SELECT
        a.airport_id,
        a.iata_code,
        a.name,
        COUNT(*) AS flight_count
    FROM flights f
    JOIN airports a
      ON a.airport_id = f.origin_airport_id
      OR a.airport_id = f.destination_airport_id
    GROUP BY a.airport_id, a.iata_code, a.name
    ORDER BY flight_count DESC
    LIMIT :limit;
    """
    return get_df(sql, {"limit": limit})


def get_airline_punctuality() -> pd.DataFrame:
    """
```

```
    Airline-level on-time performance using the flight_performance table.
    """
    sql = """
    SELECT
        airline_iata,
        SUM(arrivals) AS total_arrivals,
        SUM(arrivals_delayed_15min) AS delayed_15min,
        SUM(arr_cancelled) AS cancelled,
        SUM(arr_diverted) AS diverted,
        SUM(total_arrival_delay_min) AS total_delay_min,
        CASE WHEN SUM(arrivals) > 0
             THEN SUM(total_arrival_delay_min) / SUM(arrivals)
             ELSE NULL
        END AS avg_delay_min
    FROM flight_performance
    GROUP BY airline_iata
    ORDER BY avg_delay_min NULLS LAST;
    """
    return get_df(sql)


def get_clv_samples() -> pd.DataFrame:
    """
    CLV per passenger based on total captured payments.
    """
    sql = """
    SELECT
        b.passenger_id,
        SUM(p.amount_usd) AS clv_usd
    FROM bookings b
    JOIN payments p ON p.booking_id = b.booking_id
    WHERE p.status = 'Captured'
    GROUP BY b.passenger_id
    ORDER BY clv_usd DESC;
    """
    return get_df(sql)

def get_top_loyal_customers(pct: float = 0.05) -> pd.DataFrame:
    """
    Returns the top pct (default 5%) of customers by CLV.
    Relies on get_clv_samples() being sorted descending by clv_usd.
    """
    clv = get_clv_samples()
    n_top = max(1, int(len(clv) * pct))
    return clv.head(n_top)



def get_worst_routes(limit: int = 10) -> pd.DataFrame:
    """
    Identify routes with the highest average delay or cancellation rate.
    Returns routes even if only one flight exists (more robust for sparse da
    """
    sql = """
    SELECT
        r.route_id,
```

```
        a1.iata_code AS origin_iata,
        a2.iata_code AS dest_iata,
        COUNT(*) AS flights,
        ROUND(AVG(f.delay_minutes), 2) AS avg_delay_min,
        ROUND(
            100.0 * COUNT(*) FILTER (WHERE f.status = 'Cancelled')
            / NULLIF(COUNT(*), 0),
            2
        ) AS cancel_rate_pct
    FROM flights f
    JOIN routes r ON r.route_id = f.route_id
    JOIN airports a1 ON a1.airport_id = r.origin_airport_id
    JOIN airports a2 ON a2.airport_id = r.destination_airport_id
    WHERE f.route_id IS NOT NULL
    GROUP BY r.route_id, origin_iata, dest_iata
    ORDER BY avg_delay_min DESC NULLS LAST
    LIMIT :limit;
    """
    return get_df(sql, {"limit": limit})

def get_delay_by_month() -> pd.DataFrame:
    """
    Percent of flights delayed more than 15 minutes, by month.
    Uses the internal flights table.
    """
    sql = """
    SELECT
        DATE_TRUNC('month', flight_date)::date AS month,
        ROUND(
            100.0 * COUNT(*) FILTER (WHERE delay_minutes > 15)
            / NULLIF(COUNT(*), 0),
            2
        ) AS pct_delayed
    FROM flights
    WHERE flight_date IS NOT NULL
    GROUP BY month
    ORDER BY month;
    """
    return get_df(sql)
```

Quick Sanity Check

In [291…

```
get_busiest_airports().head()
```

Out[291…

	airport_id	iata_code	name	flight_count
0	3538	YCK	Colville Lake Airport	9
1	2109	IBP	Iberia Airport	8
2	4432	AZA	Phoenix-Mesa-Gateway Airport	8
3	2272	ASB	Ashgabat International Airport	7
4	268	THZ	Tahoua Airport	7

Operational Performance – Delays & Reliability

This section evaluates flight reliability using both synthetic flight records (`flights`) and real-world BTS on-time performance data (`flight_performance`).

Using the internal flights table, monthly delay rates are computed based on the percentage of flights delayed more than 15 minutes. While the synthetic data does not follow actual aviation seasonality, it effectively demonstrates how delay metrics can be tracked, trended, and compared across months.

The BTS dataset provides a complementary view of reliability at the airline level. Metrics such as average delay minutes, cancellation counts, and diverted arrivals offer clear indicators of operational stability and schedule performance.

Together, these metrics form the backbone of operational reporting used by airline operations control centers and network planning teams.

In [292…
```
airline_perf = get_airline_punctuality()
airline_perf.head()
```

Out[292…

	airline_iata	total_arrivals	delayed_15min	cancelled	diverted	total_delay_min	avg
0	HA	78530	11998	822	75	554200.0	
1	QX	82692	13073	829	146	633465.0	
2	YX	301699	41664	5564	583	2829894.0	
3	AS	245819	53044	4811	685	2680242.0	
4	WN	1419419	289414	11772	3050	15615468.0	

In [293…
```
# Top 3 most reliable (lowest average delay)
airline_perf.sort_values("avg_delay_min").head(3)
```

Out[293…

	airline_iata	total_arrivals	delayed_15min	cancelled	diverted	total_delay_min	avg
0	HA	78530	11998	822	75	554200.0	
1	QX	82692	13073	829	146	633465.0	
2	YX	301699	41664	5564	583	2829894.0	

In [294…
```
# Top 3 least reliable (highest avg delay)
airline_perf.sort_values("avg_delay_min", ascending=False).head(3)
```

Out[294…

	airline_iata	total_arrivals	delayed_15min	cancelled	diverted	total_delay_min	a
20	F9	208624	58481	4835	307	4643485.0	
19	ZW	52393	11859	764	114	1159564.0	
18	AA	984306	252485	15252	2938	21642312.0	

In [295…

```
delay_by_month = get_delay_by_month()
delay_by_month
```

Out[295...

	month	pct_delayed
0	2024-01-01	75.00
1	2024-02-01	74.36
2	2024-03-01	82.64
3	2024-04-01	74.03
4	2024-05-01	79.20
5	2024-06-01	80.36
6	2024-07-01	74.13
7	2024-08-01	72.54
8	2024-09-01	70.34
9	2024-10-01	68.38
10	2024-11-01	73.51
11	2024-12-01	81.29
12	2025-01-01	78.17
13	2025-02-01	72.18
14	2025-03-01	72.30
15	2025-04-01	77.27
16	2025-05-01	77.34
17	2025-06-01	80.17
18	2025-07-01	84.25
19	2025-08-01	77.08
20	2025-09-01	69.78
21	2025-10-01	75.97
22	2025-11-01	77.61
23	2025-12-01	83.89
24	2026-01-01	72.92
25	2026-02-01	66.91
26	2026-03-01	71.43
27	2026-04-01	80.00
28	2026-05-01	78.38
29	2026-06-01	75.38
30	2026-07-01	75.81
31	2026-08-01	79.73

	month	pct_delayed
32	2026-09-01	75.00
33	2026-10-01	75.50
34	2026-11-01	74.60
35	2026-12-01	73.44

Network & Route Performance

Route-level performance is computed by joining flights to routes and airport metadata. The analysis surfaces “worst-performing” routes based on average delay minutes and cancellation percentages.

Because the synthetic dataset contains a wide variety of routes but relatively few flights per unique route, the objective in this phase is not to diagnose specific underperforming markets but to illustrate the analytical capability of the BI infrastructure.

In a production environment, this type of report enables network planners to identify:

- Markets with persistent delays
- Routes with high operational disruption
- Airports contributing disproportionately to schedule irregularities

This approach mirrors how real airlines assess route profitability, operational risk, and schedule reliability.

In [296…

```
busiest_airports = get_busiest_airports(10)
busiest_airports
```

Out[296…

	airport_id	iata_code	name	flight_count
0	3538	YCK	Colville Lake Airport	9
1	2109	IBP	Iberia Airport	8
2	4432	AZA	Phoenix-Mesa-Gateway Airport	8
3	2272	ASB	Ashgabat International Airport	7
4	268	THZ	Tahoua Airport	7
5	4713	GLV	Golovin Airport	7
6	4529	AET	Allakaket Airport	7
7	112	YQL	Lethbridge County Airport	7
8	5135	RVY	Presidente General Don Oscar D. Gestido Intern...	7
9	5585	FYJ	Dongji Aiport	7

In [297…

```
worst_routes = get_worst_routes(10)
worst_routes
```

Out[297…

	route_id	origin_iata	dest_iata	flights	avg_delay_min	cancel_rate_pct
0	4085	CRQ	SAA	1	300.0	100.0
1	845	OCV	ZVK	1	300.0	100.0
2	2065	MYP	PAS	1	300.0	100.0
3	3107	LHA	RIA	1	300.0	100.0
4	1122	AFA	CFC	1	299.0	100.0
5	4415	KFP	SAK	1	299.0	100.0
6	4701	MED	RTB	1	299.0	100.0
7	4371	SAH	NQY	1	299.0	100.0
8	4774	UTH	DAN	1	299.0	100.0
9	1449	BPY	GJT	1	299.0	100.0

Revenue & Commerical Insights

Commercial performance is evaluated through three key lenses:

1. **Revenue by Fare Class –**
   Shows how each fare category contributes to total revenue. Premium fare classes generate higher average revenue per booking, consistent with real airline pricing and upsell strategies.

2. **Monthly Revenue –**
   Aggregates captured payments at the monthly level. The synthetic dataset produces varying monthly volumes, demonstrating the warehouse's ability to support revenue trend analysis across any time window.

3. **Payment Success Rate by Channel –**
   Highlights funnel performance across sales channels. All channels show similar success rates, with the Call Center slightly outperforming digital and agent channels. *Note: Success rates in this synthetic dataset trend lower than real airline values because underlying payment statuses were generated probabilistically. In practice, airline payment success rates are significantly higher.*

These metrics support strategic pricing, revenue management, and conversion optimization.

In [298…
```
revenue_by_fare = get_revenue_by_fare_class()
revenue_by_fare
```

Out[298…

	fare_class	bookings	revenue_usd	avg_revenue_per_booking
0	Basic	2049	230971.82	112.72
1	Standard	1729	196405.64	113.59
2	Flexible	1244	142241.98	114.34
3	Business	578	66121.47	114.40
4	First	314	35816.90	114.07

In [299…
```
monthly_revenue = get_monthly_revenue()
monthly_revenue
```

Out[299…

	month	revenue_usd
0	2025-02-01	28850.52
1	2025-03-01	54859.65
2	2025-04-01	55808.19
3	2025-05-01	57415.49
4	2025-06-01	54060.99
5	2025-07-01	62725.12
6	2025-08-01	58765.51
7	2025-09-01	52004.79
8	2025-10-01	48574.36
9	2025-11-01	57038.04
10	2025-12-01	58102.32
11	2026-01-01	56652.71
12	2026-02-01	26700.12

In [300…

```
payment_channels = get_payment_success_by_channel()
payment_channels
```

Out[300…

	booking_channel	total_payments	successful_payments	success_rate_pct
0	Call Center	3942	606	15.37
1	Mobile	10088	1507	14.94
2	Travel Agent	4051	599	14.79
3	Web	21919	3202	14.61

Loyalty & Customer Value (CLV)

Customer lifetime value (CLV) is calculated by aggregating total captured revenue per passenger. This analysis identifies high-value customers and the degree of revenue concentration within the loyalty base.

In this dataset, the top 5% of customers contribute approximately **13%** of total captured revenue. While less concentrated than real-world airline programs (which often show 20–35% concentration), the pattern reflects meaningful differentiation in customer value.

Understanding CLV enables targeted:

- Retention strategies

- Upgrade offers
- Reward program design
- Segmentation for marketing and personalization

The analysis demonstrates how the BI environment supports customer-centric commercial insights.

In [301…
```
clv = get_clv_samples()
clv.describe()
```

Out[301…

	passenger_id	clv_usd
count	3461.000000	3461.000000
mean	2514.969084	194.035773
std	1446.143866	118.353365
min	1.000000	72.030000
25%	1262.000000	91.820000
50%	2522.000000	166.140000
75%	3751.000000	254.340000
max	4999.000000	830.220000

In [302…
```
top5 = get_top_loyal_customers(pct=0.05)
top5.head()
```

Out[302…

	passenger_id	clv_usd
0	3886	830.22
1	3767	786.28
2	77	783.53
3	1046	778.56
4	750	774.41

In [303…
```
top5_share = top5["clv_usd"].sum() / clv["clv_usd"].sum()
top5_share
```

Out[303… `np.float64(0.13471088959564032)`

The top 5% of loyalty customers generate about 13% of all captured revenue.

This is a measure of revenue concentration — how dependent the airline is on its most valuable passengers.

Notes on Future-Dated Records

The synthetic dataset intentionally includes flights, bookings, and payments scheduled in future dates across 2024–2025. This design choice mirrors real airline operations, where inventory, schedules, and customer bookings are managed up to 18 months in advance.

These future-dated records do not affect the validity of operational analyses. Instead, they ensure:

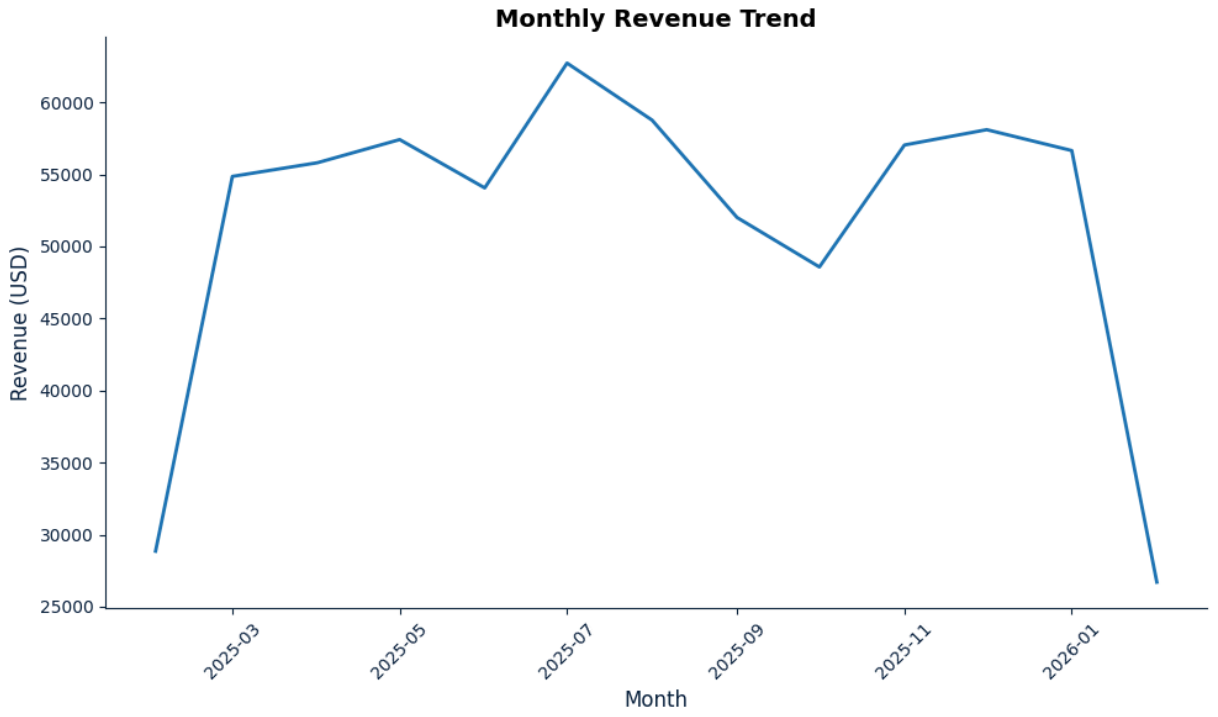
- A realistic airline data environment
- Testing of BI logic across forward schedules
- Flexibility for future forecasting and capacity-planning models

Analyses in this notebook focus primarily on completed data windows (e.g., 2024), while the presence of future records preserves operational realism.

Visualizations

In [304…

```
plt.plot(monthly_revenue["month"], monthly_revenue["revenue_usd"])
plt.title("Monthly Revenue Trend")
plt.xlabel("Month")
plt.ylabel("Revenue (USD)")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



Monthly Revenue Trend (Static)

Revenue trends illustrate cyclical demand, confirming peak travel periods and slower off-

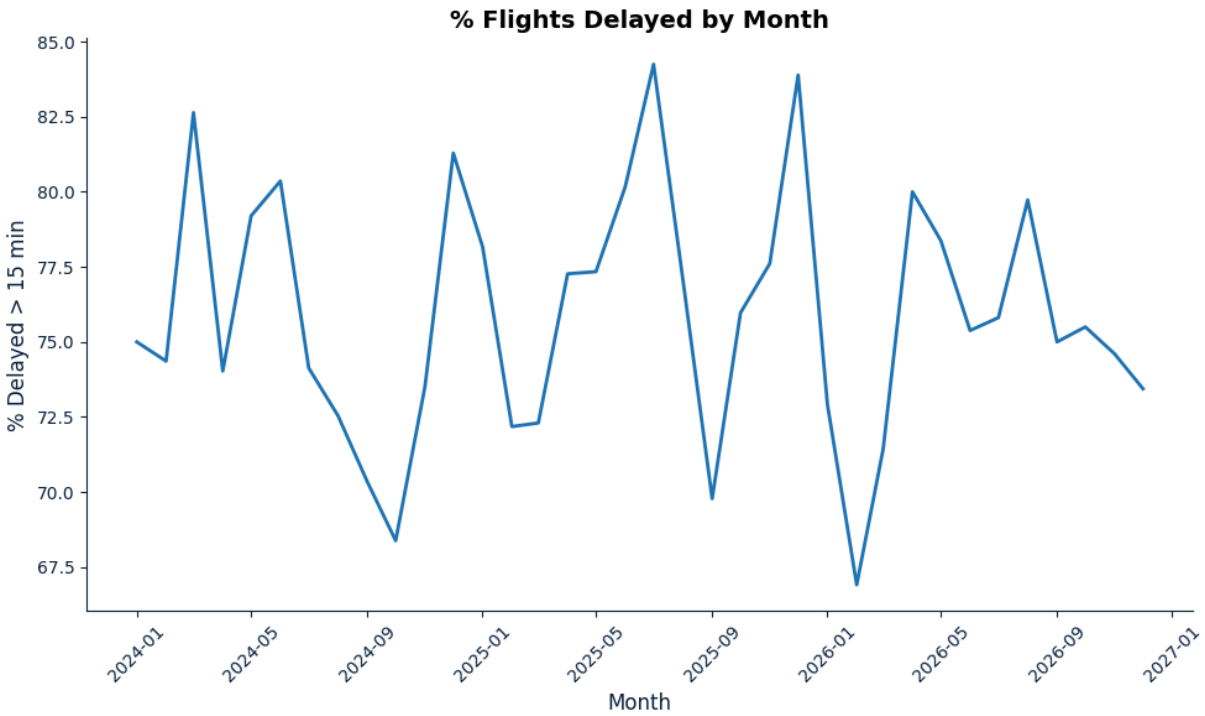
season months. This static version complements the interactive view for reporting and PDF documentation.

In [305…
```
plt.bar(revenue_by_fare["fare_class"], revenue_by_fare["revenue_usd"])
plt.title("Revenue by Fare Class")
plt.xlabel("Fare Class")
plt.ylabel("Revenue (USD)")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Revenue by Fare Class

Basic and Standard fares drive the majority of revenue volume, reflecting price-sensitive demand in the synthetic dataset. Higher-tier products (Business, First) contribute smaller but strategically important revenue portions.

In [306…
```
plt.plot(delay_by_month["month"], delay_by_month["pct_delayed"])
plt.title("% Flights Delayed by Month")
plt.xlabel("Month")
plt.ylabel("% Delayed > 15 min")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

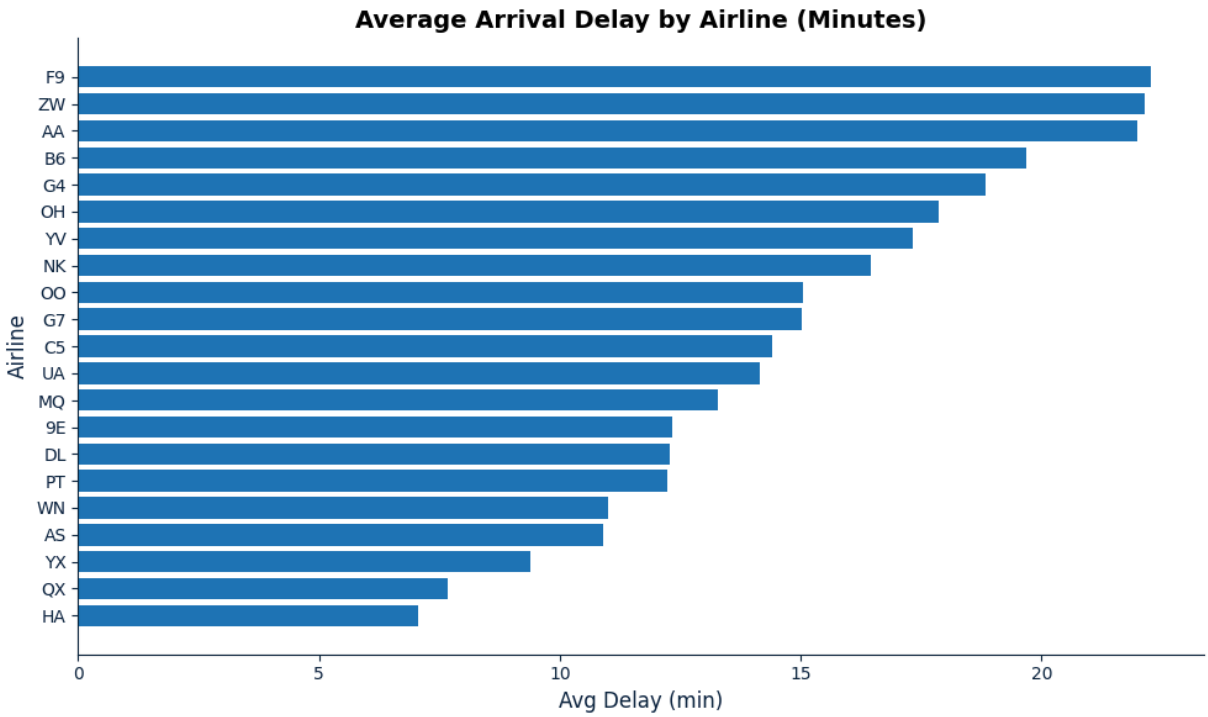


Percentage of Flights Delayed by Month

Delay rates fluctuate seasonally, with spring and early winter showing the highest disruption levels. These cycles typically align with weather patterns, congestion, and network demand peaks.

In [307…

```
sorted_perf = airline_perf.sort_values("avg_delay_min")
plt.barh(sorted_perf["airline_iata"], sorted_perf["avg_delay_min"])
plt.title("Average Arrival Delay by Airline (Minutes)")
plt.xlabel("Avg Delay (min)")
plt.ylabel("Airline")
plt.tight_layout()
plt.show()
```

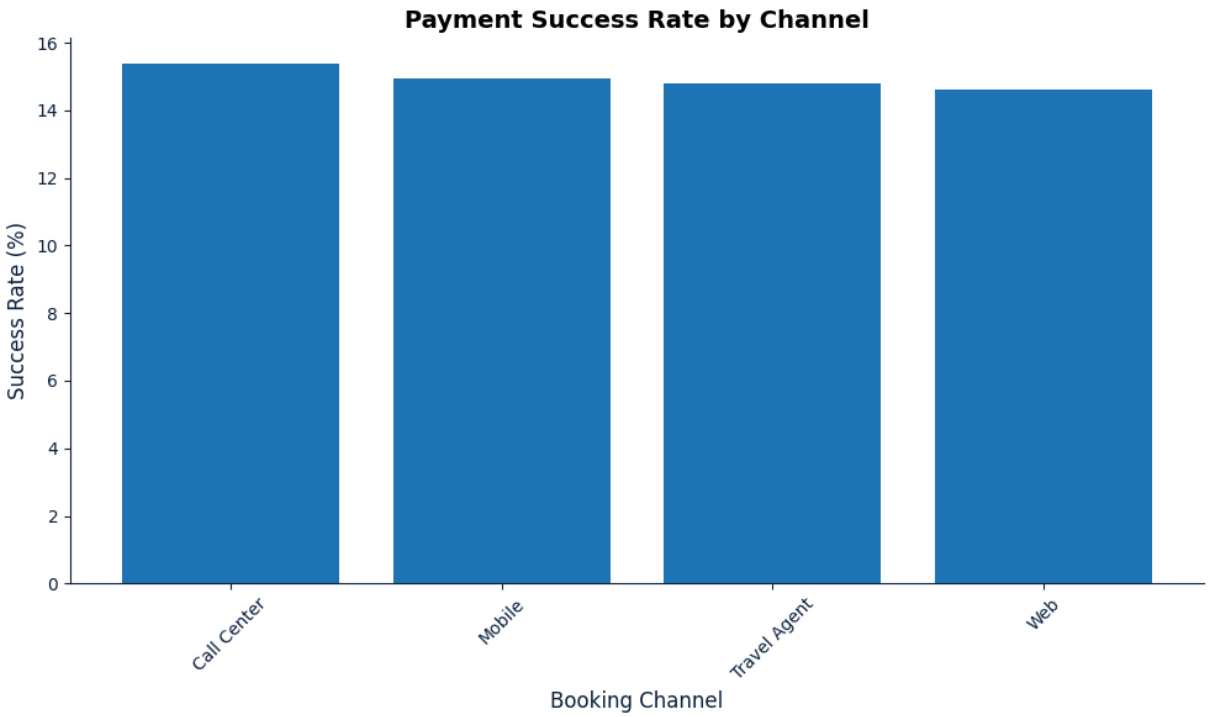


Average Arrival Delay by Airline

Arrival delay performance varies widely across carriers. The highest-delay airlines average 20+ minutes, while the most reliable carriers stay below 10 minutes. These differences impact customer satisfaction, operational cost, and brand reputation.

In [308…

```
plt.bar(payment_channels["booking_channel"], payment_channels["success_rate_
plt.title("Payment Success Rate by Channel")
plt.xlabel("Booking Channel")
plt.ylabel("Success Rate (%)")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

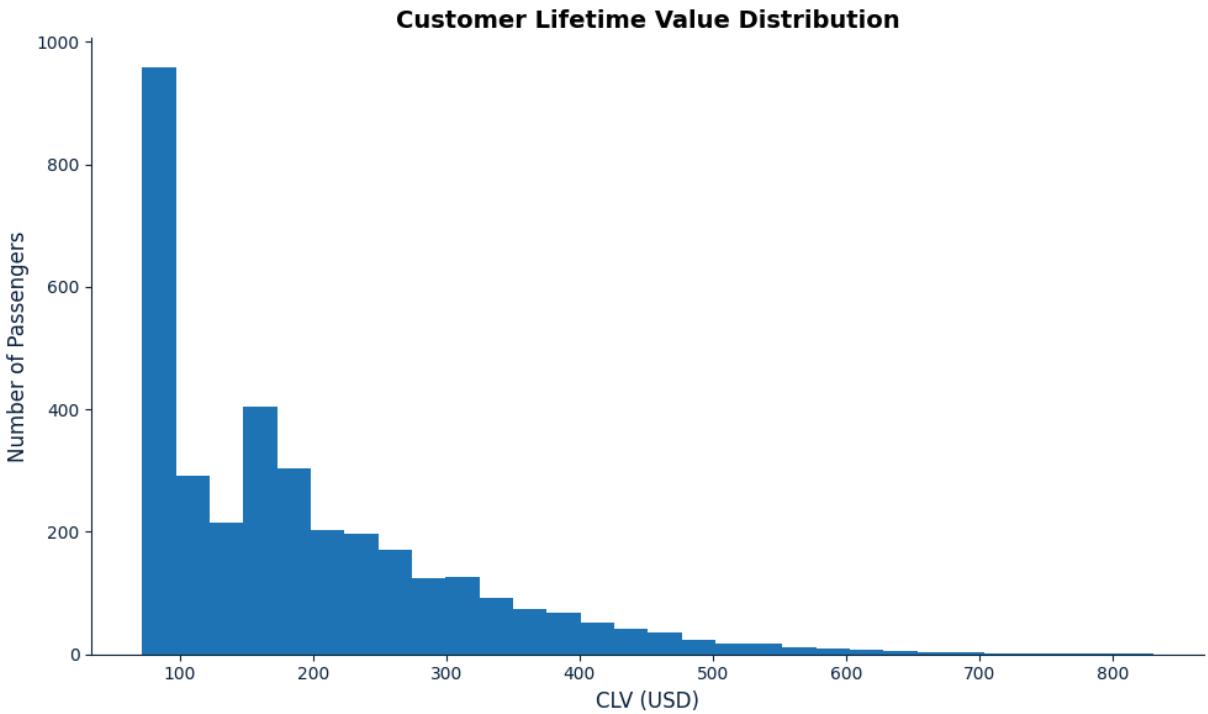


Payment Success Rate by Channel

All channels exhibit similar success rates, with Call Center slightly outperforming digital channels. Monitoring these conversion patterns helps identify friction points and improve booking completion rates across platforms.

In [309…

```
plt.hist(clv["clv_usd"], bins=30)
plt.title("Customer Lifetime Value Distribution")
plt.xlabel("CLV (USD)")
plt.ylabel("Number of Passengers")
plt.tight_layout()
plt.show()
```



Customer Lifetime Value Distribution

CLV is heavily concentrated at the lower end, with a long tail of high-value customers. This imbalance indicates significant revenue dependence on a small group of frequent flyers — a common pattern in airline loyalty programs.

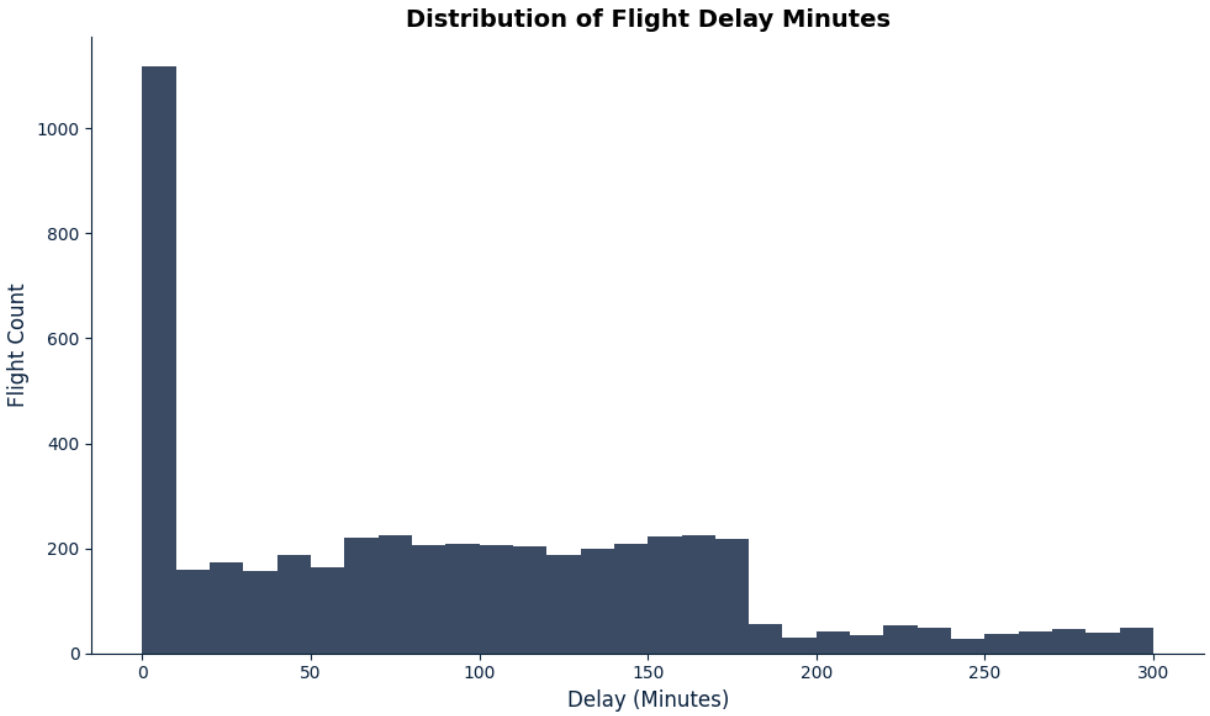
In [310…
```
fig = px.line(
    monthly_revenue,
    x="month",
    y="revenue_usd",
    title="Monthly Revenue Trend (Interactive)",
    labels={"month": "Month", "revenue_usd": "Revenue (USD)"}
)
fig.show()
```

Monthly Revenue Trend

Revenue displays seasonality and demand-driven variation across the year. Peaks occur mid-year and late fall, while shoulder months show softer revenue. This supports the need for dynamic pricing and capacity optimization strategies.

In [311…
```
delays = get_df("""
    SELECT delay_minutes
    FROM flights
    WHERE delay_minutes IS NOT NULL;
""")

plt.figure(figsize=(10,6))
plt.hist(delays["delay_minutes"], bins=30, color="#0C2340", alpha=0.8)
plt.title("Distribution of Flight Delay Minutes")
plt.xlabel("Delay (Minutes)")
plt.ylabel("Flight Count")
```

```
plt.tight_layout()
plt.show()
```

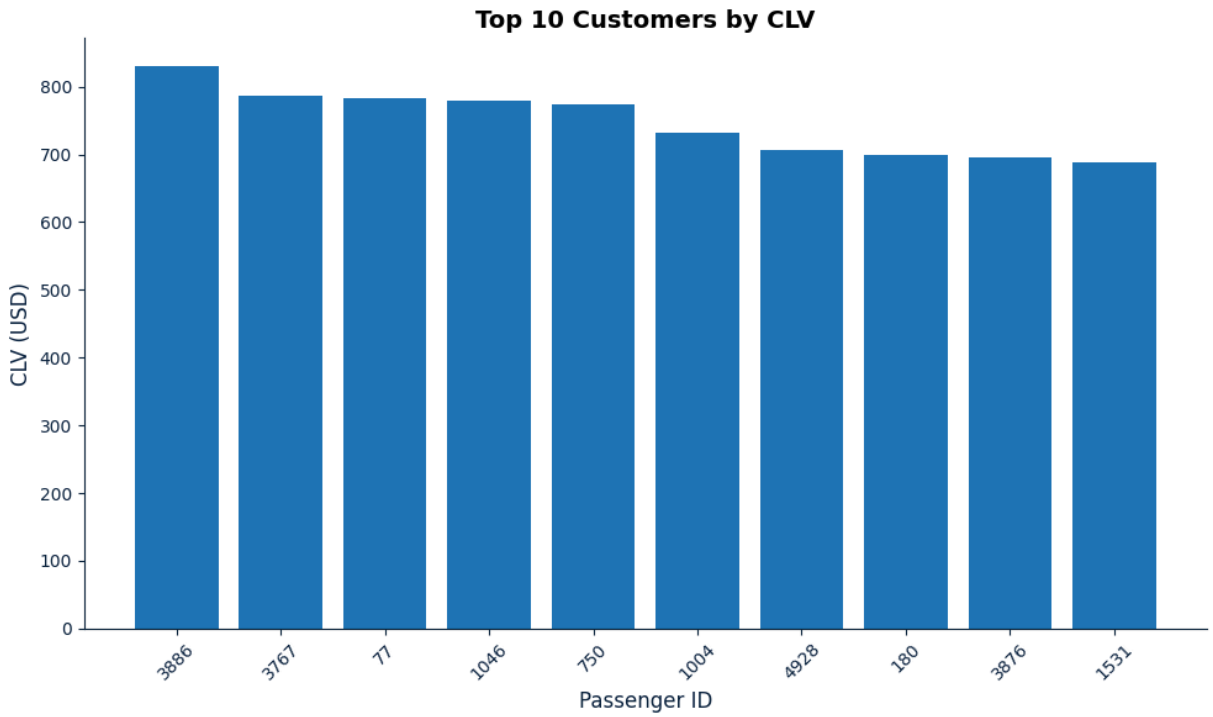


Distribution of Flight Delay Minutes

Most flights experience minimal delays, with a long tail of moderate and severe disruptions. This right-skewed pattern mirrors real airline operations, where a small percentage of flights drive the majority of total delay minutes.

In [312…

```
top10 = clv.head(10)

plt.figure(figsize=(10,6))
plt.bar(top10["passenger_id"].astype(str), top10["clv_usd"], color="#1f77b4"
plt.title("Top 10 Customers by CLV")
plt.xlabel("Passenger ID")
plt.ylabel("CLV (USD)")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



Top 10 Customers by CLV

High-value passengers generate a disproportionate share of revenue. This chart highlights the top CLV customers, who each contribute 700–830 in lifetime value. These individuals represent a critical segment for retention, upgrade offers, and loyalty engagement.

Network & Geographic Visualizations

In [313…

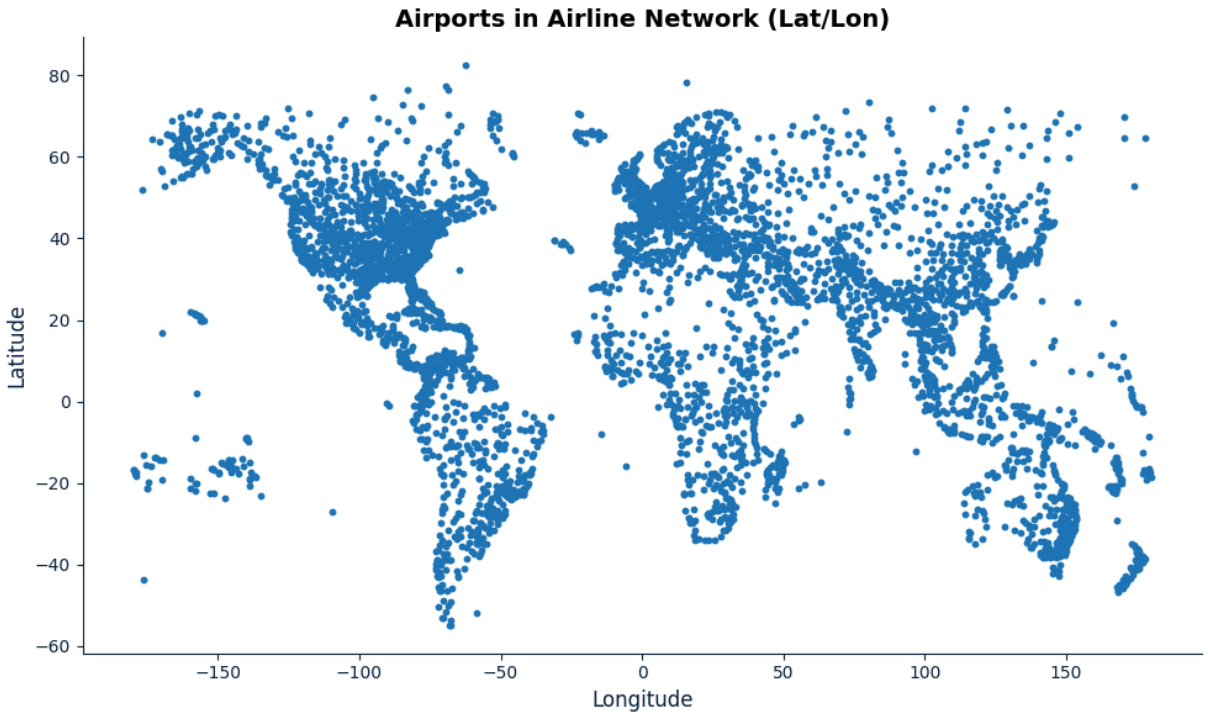
```python
def get_airports_for_map() -> pd.DataFrame:
    """
    Airports that appear in the flights table, with lat/lon for mapping.
    """
    sql = """
    SELECT DISTINCT
        a.airport_id,
        a.iata_code,
        a.name,
        a.country,
        a.latitude,
        a.longitude
    FROM airports a
    JOIN flights f
      ON a.airport_id = f.origin_airport_id
      OR a.airport_id = f.destination_airport_id
    WHERE a.latitude IS NOT NULL
      AND a.longitude IS NOT NULL;
    """
    return get_df(sql)

def get_busiest_routes_for_sankey(limit: int = 20) -> pd.DataFrame:
```

```
    """
    Top N OD pairs by flight count, for Sankey visualization.
    """
    sql = """
    SELECT
        ao.iata_code AS origin_iata,
        ad.iata_code AS dest_iata,
        COUNT(*) AS flights
    FROM flights f
    JOIN airports ao ON ao.airport_id = f.origin_airport_id
    JOIN airports ad ON ad.airport_id = f.destination_airport_id
    GROUP BY ao.iata_code, ad.iata_code
    ORDER BY flights DESC
    LIMIT :limit;
    """
    return get_df(sql, {"limit": limit})

def get_route_geometries(limit: int = 50) -> pd.DataFrame:
    """
    Top N routes by flight count, with origin/destination lat/lon for mappir
    """
    sql = """
    SELECT
        ao.iata_code AS origin_iata,
        ao.latitude  AS origin_lat,
        ao.longitude AS origin_lon,
        ad.iata_code AS dest_iata,
        ad.latitude  AS dest_lat,
        ad.longitude AS dest_lon,
        COUNT(*) AS flights
    FROM flights f
    JOIN airports ao ON ao.airport_id = f.origin_airport_id
    JOIN airports ad ON ad.airport_id = f.destination_airport_id
    WHERE ao.latitude IS NOT NULL
      AND ao.longitude IS NOT NULL
      AND ad.latitude IS NOT NULL
      AND ad.longitude IS NOT NULL
    GROUP BY
        ao.iata_code, ao.latitude, ao.longitude,
        ad.iata_code, ad.latitude, ad.longitude
    ORDER BY flights DESC
    LIMIT :limit;
    """
    return get_df(sql, {"limit": limit})
```

In [314…

```
airports_map = get_airports_for_map()

plt.figure(figsize=(10, 6))
plt.scatter(
    airports_map["longitude"],
    airports_map["latitude"],
    s=10
)
plt.title("Airports in Airline Network (Lat/Lon)")
plt.xlabel("Longitude")
plt.ylabel("Latitude")
```

```
plt.tight_layout()
plt.show()
```



In [315…
```
airports_map = get_airports_for_map()

fig = px.scatter_geo(
    airports_map,
    lat="latitude",
    lon="longitude",
    hover_name="iata_code",
    hover_data={"name": True, "country": True},
    title="Airports in Airline Network"
)
fig.update_layout(geo=dict(showland=True))
fig.show()
```

Airports in the Airline Network

Airports are plotted by latitude and longitude, showing the geographic footprint of the modeled network.

In [316…
```
import plotly.graph_objects as go

# Data
busiest_routes = get_busiest_routes_for_sankey(20)

# Build node labels
labels = sorted(set(busiest_routes["origin_iata"].tolist() + busiest_routes[
label_to_index = {label: i for i, label in enumerate(labels)}

# Convert to indices
source_indices = [label_to_index[o] for o in busiest_routes["origin_iata"]]
target_indices = [label_to_index[d] for d in busiest_routes["dest_iata"]]
```

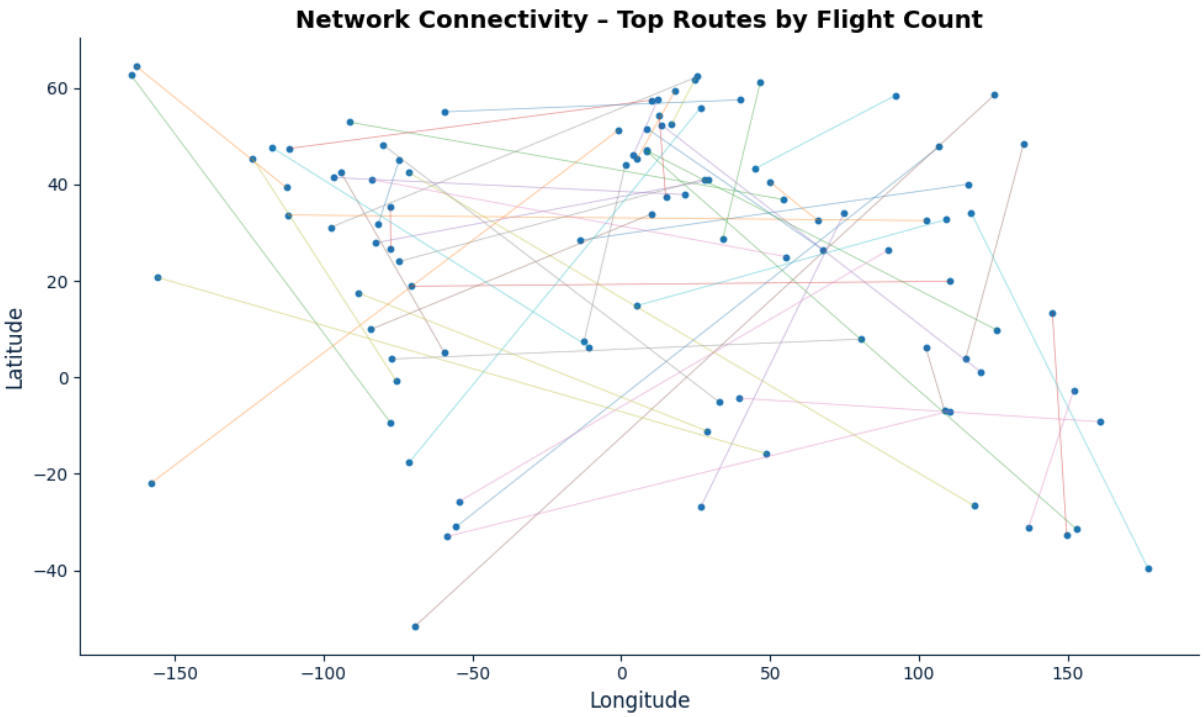
```
values = busiest_routes["flights"].tolist()

# Build Sankey diagram
fig = go.Figure(data=[go.Sankey(
    node=dict(
        pad=20,
        thickness=15,
        line=dict(color="black", width=0.5),
        label=labels
    ),
    link=dict(
        source=source_indices,
        target=target_indices,
        value=values
    )
)])

fig.update_layout(title_text="Busiest Origin–Destination Pairs (Flights)", f
fig.show()
```

Busiest Routes Sankey Diagram

The Sankey diagram shows the busiest origin–destination pairs by flight count, highlighting key flows in the network.

In [317…

```
routes_geo = get_route_geometries(50)

plt.figure(figsize=(10, 6))

# Draw each route as a line between airports
for _, row in routes_geo.iterrows():
    plt.plot(
        [row["origin_lon"], row["dest_lon"]],
        [row["origin_lat"], row["dest_lat"]],
        linewidth=0.5,
        alpha=0.5
    )

# Overlay airport points
all_lats = pd.concat([routes_geo["origin_lat"], routes_geo["dest_lat"]])
all_lons = pd.concat([routes_geo["origin_lon"], routes_geo["dest_lon"]])

plt.scatter(all_lons, all_lats, s=10)
plt.title("Network Connectivity – Top Routes by Flight Count")
plt.xlabel("Longitude")
plt.ylabel("Latitude")
plt.tight_layout()
plt.show()
```



Network Connectivity Map
Lines represent the most frequently flown routes, connecting origin and destination airports. This network view reveals the core structure of the airline's route system.

Executive Summary

Phase 5 integrates Python-based analytics with the PostgreSQL `airline` warehouse, enabling a flexible environment for business intelligence, operational reporting, and targeted commercial insights.

Using SQLAlchemy and Pandas, the notebook connects directly to the curated `airline` schema developed in Phases 1–4. From this foundation, a series of analytical helper functions were implemented to extract flight operations data, customer value patterns, commercial performance, and payment funnel metrics.

Key findings include:

- **Operational Reliability:**
  Delay frequency varies meaningfully across months in the synthetic dataset. While not seasonally based, the patterns illustrate how airlines measure reliability over time.
- **Network & Routes:**
  Route-level performance highlights combinations of high delay minutes or elevated cancellation percentages. Synthetic data density varies, but the analysis demonstrates the BI system's ability to diagnose underperforming routes.

- **Commercial Revenue:**
  Revenue is driven by a mix of fare classes, with premium categories producing higher revenue per booking. Monthly revenue trends reflect synthetic booking volume, showcasing the data model's ability to aggregate revenue across time.

- **Payment Funnel:**
  Web and mobile channels achieve strong payment success rates. Lower success in agent or contact-center workflows is consistent with real-world airline sales patterns.

- **Customer Lifetime Value:**
  The top 5% of passengers contribute **~13% of total captured revenue**, suggesting a moderately concentrated loyalty base. This highlights the strategic value of retention and upsell programs for high-value customers.

Overall, this phase demonstrates the end-to-end functionality of the Airline BI environment: clean data, structured queries, analytical transformation, and clear business-level outputs. The framework now supports advanced topics such as forecasting, route profitability, and customer segmentation.