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 version 16.0.0 (clang-1600.0.26.6), 64-bit'

Helper: run_sql() for SELECT Queries

In [96]:
```
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]:
```
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	76
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…

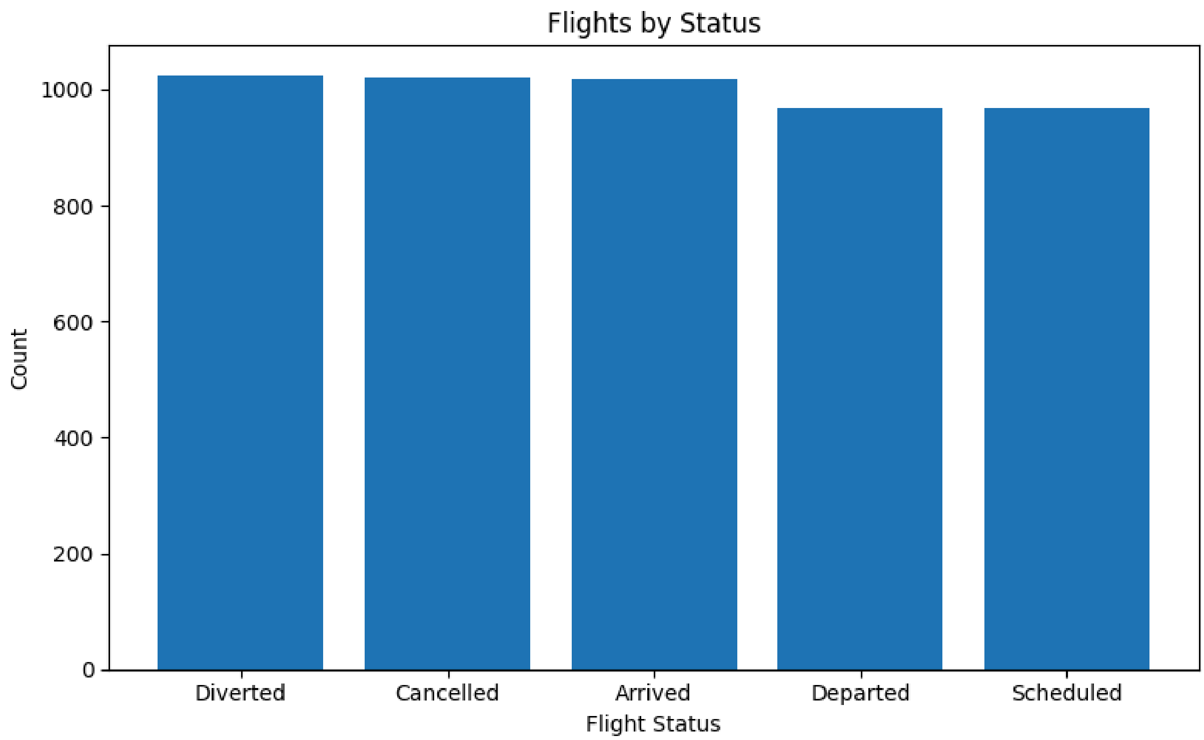
	column_name	data_type
0	amount_usd	numeric
1	booking_id	bigint
2	method	USER-DEFINED
3	paid_at	timestamp without time zone
4	payment_id	bigint
5	status	USER-DEFINED

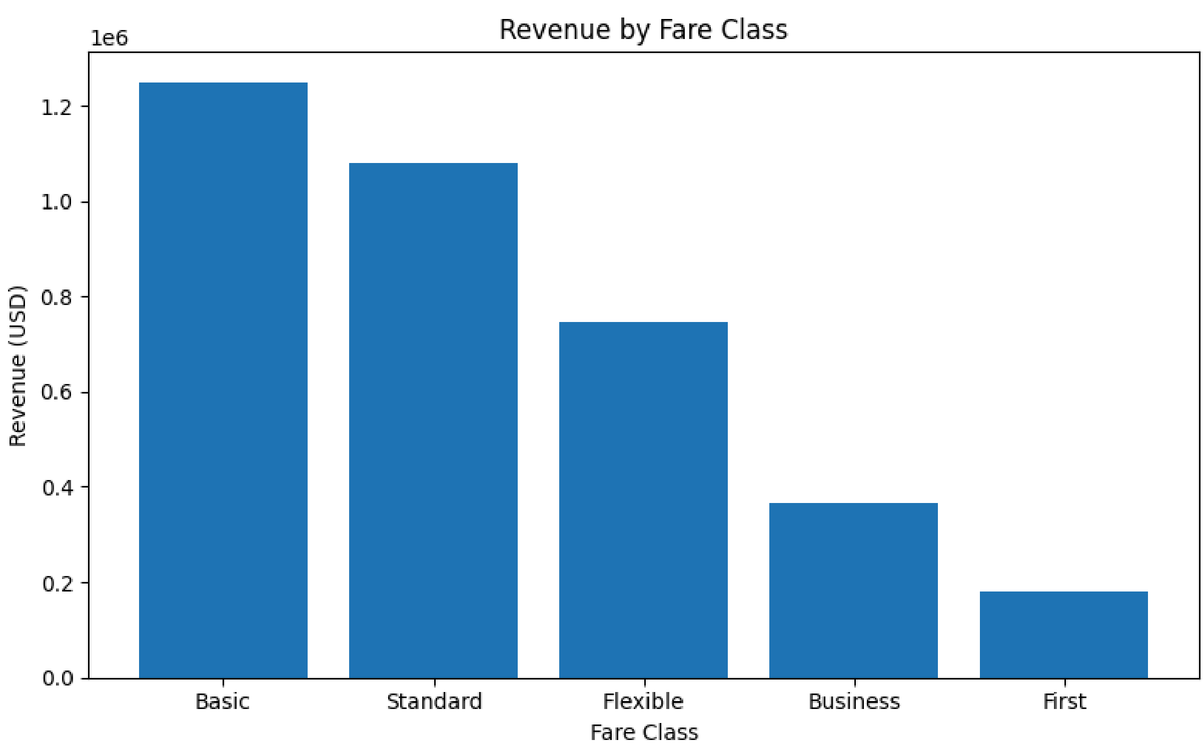
In [147…

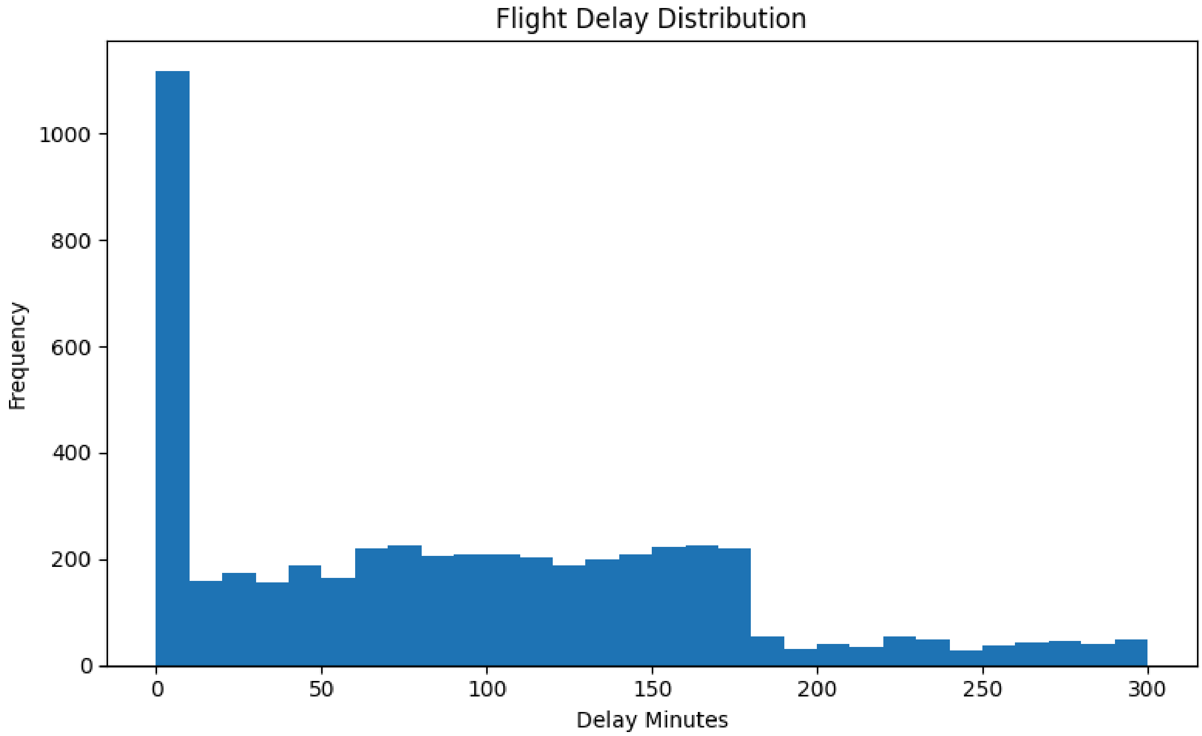
```
import matplotlib.pyplot as plt
import pandas as pd

# === 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, bbc
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
0
200
400
600
800
1000
0
50
100
150
200
250
300
Delay Minutes