flight_analysis

December 24, 2025

[1]:
```
from pyspark.sql import SparkSession
import os

os.makedirs("/tmp/spark-local", exist_ok=True)
os.makedirs("/tmp/spark-warehouse", exist_ok=True)
os.makedirs("/tmp/spark-checkpoints", exist_ok=True)

GRAPHFRAMES_PACKAGE = "graphframes:graphframes:0.8.3-spark3.5-s_2.12"
DELTA_PACKAGE      = "io.delta:delta-spark_2.12:3.0.0"

spark = (
    SparkSession.builder
    .appName("DeltaPlusGraphFrames")
    .master("local[*]")

    # Packages (both)
    .config("spark.jars.packages", f"{DELTA_PACKAGE},{GRAPHFRAMES_PACKAGE}")

    # Delta Lake requirements
    .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
    .config("spark.sql.catalog.spark_catalog", "org.apache.spark.sql.delta.
 ↪catalog.DeltaCatalog")

    # Local stability/perf basics
    .config("spark.sql.shuffle.partitions", "32")
    .config("spark.default.parallelism", "32")
    .config("spark.local.dir", "/tmp/spark-local")
    .config("spark.sql.warehouse.dir", "/tmp/spark-warehouse")

    .getOrCreate()
)

spark.sparkContext.setCheckpointDir("/tmp/spark-checkpoints")

print("Spark version:", spark.version)
spark.range(3).show()
```

```
Spark version: 3.5.3
```

```
+---+
| id|
+---+
|  0|
|  1|
|  2|
+---+
```

[2]:
```
from graphframes import GraphFrame
print("GraphFrames imported OK")
```

```
GraphFrames imported OK
```

[3]:
```
# Load CSV into a DataFrame
# Replace path with the actual path inside your container if different.
csv_path = "/home/jovyan/*.csv"  # read all CSV files in the folder
 # <-- change if needed
# Example columns expected (adjust according to the Kaggle dataset you␣
 ↪downloaded):
# FlightDate, Airline, FlightNum, Origin, Dest, Cancelled, Diverted, etc.
df = spark.read.option("header", "true").option("inferSchema", "true").
 ↪csv(csv_path)
print("Columns:", df.columns)
print("Number of rows:", df.count())
df.printSchema()
df.limit(5).toPandas()
```

```
Columns: ['FL_DATE', 'OP_CARRIER', 'OP_CARRIER_FL_NUM', 'ORIGIN', 'DEST',
'CRS_DEP_TIME', 'DEP_TIME', 'DEP_DELAY', 'TAXI_OUT', 'WHEELS_OFF', 'WHEELS_ON',
'TAXI_IN', 'CRS_ARR_TIME', 'ARR_TIME', 'ARR_DELAY', 'CANCELLED',
'CANCELLATION_CODE', 'DIVERTED', 'CRS_ELAPSED_TIME', 'ACTUAL_ELAPSED_TIME',
'AIR_TIME', 'DISTANCE', 'CARRIER_DELAY', 'WEATHER_DELAY', 'NAS_DELAY',
'SECURITY_DELAY', 'LATE_AIRCRAFT_DELAY', 'Unnamed: 27']
Number of rows: 61556964
root
 |-- FL_DATE: date (nullable = true)
 |-- OP_CARRIER: string (nullable = true)
 |-- OP_CARRIER_FL_NUM: integer (nullable = true)
 |-- ORIGIN: string (nullable = true)
 |-- DEST: string (nullable = true)
 |-- CRS_DEP_TIME: double (nullable = true)
 |-- DEP_TIME: double (nullable = true)
 |-- DEP_DELAY: double (nullable = true)
 |-- TAXI_OUT: double (nullable = true)
 |-- WHEELS_OFF: double (nullable = true)
 |-- WHEELS_ON: double (nullable = true)
 |-- TAXI_IN: double (nullable = true)
```

```
 |-- CRS_ARR_TIME: double (nullable = true)
 |-- ARR_TIME: double (nullable = true)
 |-- ARR_DELAY: double (nullable = true)
 |-- CANCELLED: double (nullable = true)
 |-- CANCELLATION_CODE: string (nullable = true)
 |-- DIVERTED: double (nullable = true)
 |-- CRS_ELAPSED_TIME: double (nullable = true)
 |-- ACTUAL_ELAPSED_TIME: double (nullable = true)
 |-- AIR_TIME: double (nullable = true)
 |-- DISTANCE: double (nullable = true)
 |-- CARRIER_DELAY: double (nullable = true)
 |-- WEATHER_DELAY: double (nullable = true)
 |-- NAS_DELAY: double (nullable = true)
 |-- SECURITY_DELAY: double (nullable = true)
 |-- LATE_AIRCRAFT_DELAY: double (nullable = true)
 |-- Unnamed: 27: string (nullable = true)
```

[3]:

	FL_DATE	OP_CARRIER	OP_CARRIER_FL_NUM	ORIGIN	DEST	CRS_DEP_TIME	\
0	2009-01-01	XE	1204	DCA	EWR	1100.0	
1	2009-01-01	XE	1206	EWR	IAD	1510.0	
2	2009-01-01	XE	1207	EWR	DCA	1100.0	
3	2009-01-01	XE	1208	DCA	EWR	1240.0	
4	2009-01-01	XE	1209	IAD	EWR	1715.0	

	DEP_TIME	DEP_DELAY	TAXI_OUT	WHEELS_OFF	…	CRS_ELAPSED_TIME	\
0	1058.0	-2.0	18.0	1116.0	…	62.0	
1	1509.0	-1.0	28.0	1537.0	…	82.0	
2	1059.0	-1.0	20.0	1119.0	…	70.0	
3	1249.0	9.0	10.0	1259.0	…	77.0	
4	1705.0	-10.0	24.0	1729.0	…	105.0	

	ACTUAL_ELAPSED_TIME	AIR_TIME	DISTANCE	CARRIER_DELAY	WEATHER_DELAY	\
0	68.0	42.0	199.0	NaN	NaN	
1	75.0	43.0	213.0	NaN	NaN	
2	62.0	36.0	199.0	NaN	NaN	
3	56.0	37.0	199.0	NaN	NaN	
4	77.0	40.0	213.0	NaN	NaN	

	NAS_DELAY	SECURITY_DELAY	LATE_AIRCRAFT_DELAY	Unnamed: 27
0	NaN	NaN	NaN	None
1	NaN	NaN	NaN	None
2	NaN	NaN	NaN	None
3	NaN	NaN	NaN	None
4	NaN	NaN	NaN	None

[5 rows x 28 columns]

[4]:
```
from pyspark.sql import functions as F

# Preprocess to create edges and vertices
# We'll aggregate edges by (origin, dest) and count the number of flights␣
 ↪(weight).
edges = df.select(F.col("Origin").alias("src"), F.col("Dest").alias("dst")) \
          .filter(F.col("src").isNotNull() & F.col("dst").isNotNull())

# Aggregate to get weights
edge_counts = edges.groupBy("src", "dst").count().withColumnRenamed("count",␣
 ↪"weight")
print("Number of distinct edges:", edge_counts.count())
edge_counts.limit(5).toPandas()

# Create vertices DataFrame (unique airports)
src_verts = edge_counts.select(F.col("src").alias("id"))
dst_verts = edge_counts.select(F.col("dst").alias("id"))
vertices = src_verts.union(dst_verts).distinct()
print("Number of vertices (airports):", vertices.count())
vertices.limit(10).toPandas()
```

```
Number of distinct edges: 7956
Number of vertices (airports): 381
```

[4]:

	id
0	MEM
1	JFK
2	MBS
3	JAX
4	ANC
5	HPN
6	EVV
7	SBN
8	SAF
9	COU

[5]:
```
# Compute in-degree, out-degree and total degree using Spark aggregations␣
 ↪(native implementation)
outdeg = edge_counts.groupBy("src").agg(F.sum("weight").alias("out_degree"))
indeg = edge_counts.groupBy("dst").agg(F.sum("weight").alias("in_degree"))

# Rename columns for join
outdeg = outdeg.withColumnRenamed("src", "id")
indeg = indeg.withColumnRenamed("dst", "id")

degrees = vertices.join(outdeg, "id", "left").join(indeg, "id", "left") \
                  .na.fill(0, subset=["out_degree", "in_degree"]) \
```

```
                    .withColumn("total_degree", F.col("in_degree") + F.
 ↪col("out_degree"))

degrees.orderBy(F.desc("total_degree")).limit(10).toPandas()
```

[5]:

	id	out_degree	in_degree	total_degree
0	ATL	3903244	3903288	7806532
1	ORD	3001285	3001372	6002657
2	DFW	2546075	2546050	5092125
3	DEN	2300550	2300456	4601006
4	LAX	2133445	2133646	4267091
5	PHX	1720614	1720588	3441202
6	IAH	1672053	1672279	3344332
7	SFO	1612933	1613144	3226077
8	LAS	1472436	1472477	2944913
9	CLT	1334522	1334543	2669065

[6]:
```
# =========================
# Total triangle count in the graph (native Spark DataFrame)
# Treats the graph as undirected and counts each triangle exactly once.
#
# Input required:
#   edge_counts: DataFrame with columns (src, dst, weight) or at least (src,␣
 ↪dst)
# Output:
#   total_triangles: integer
# =========================

import pyspark.sql.functions as F
from pyspark.storagelevel import StorageLevel

# 1) Canonical undirected edges: store each edge once as (u, v) with u < v
undirected = (
    edge_counts
    .select(
        F.when(F.col("src") < F.col("dst"), F.col("src")).otherwise(F.
 ↪col("dst")).alias("u"),
        F.when(F.col("src") < F.col("dst"), F.col("dst")).otherwise(F.
 ↪col("src")).alias("v")
    )
    .filter(F.col("u").isNotNull() & F.col("v").isNotNull() & (F.col("u") != F.
 ↪col("v")))
    .distinct()
    .persist(StorageLevel.MEMORY_AND_DISK)
)
_ = undirected.count()  # materialize cache
```

```
# 2) For each node, list neighbors as rows (both directions)
neighbors = (
    undirected.select(F.col("u").alias("node"), F.col("v").alias("nbr"))
    .union(undirected.select(F.col("v").alias("node"), F.col("u").alias("nbr")))
    .persist(StorageLevel.MEMORY_AND_DISK)
)
_ = neighbors.count()

# 3) Build wedges (u, v, w): edges (u, v) and (u, w) exist
#    Use u from undirected edge and w from neighbors of u
triples = (
    undirected.alias("e")
    .join(neighbors.alias("n"), F.col("e.u") == F.col("n.node"), "inner")
    .select(
        F.col("e.u").alias("u"),
        F.col("e.v").alias("v"),
        F.col("n.nbr").alias("w")
    )
)

# 4) Enforce ordering u < v < w so every triangle is generated once
triples_filtered = triples.filter(
    (F.col("v") != F.col("w")) &
    (F.col("u") < F.col("v")) &
    (F.col("v") < F.col("w"))
)

# 5) Close the wedge by checking that edge (v, w) exists in undirected
triangles = (
    triples_filtered.alias("t")
    .join(
        undirected.alias("e2"),
        (F.col("t.v") == F.col("e2.u")) & (F.col("t.w") == F.col("e2.v")),
        "inner"
    )
    .select(F.col("t.u").alias("u"), F.col("t.v").alias("v"), F.col("t.w").
 ↪alias("w"))
    .distinct()
)

# 6) Total number of triangles (counted once)
total_triangles = triangles.count()
print("Total triangles (undirected, counted once):", total_triangles)
```

```
Total triangles (undirected, counted once): 36562
```

[7]:
```
# ========================
# Centrality (non-PageRank) natively on Spark: Eigenvector Centrality (Power Iteration)
#
# Idea:
# - Centrality score of a node is proportional to the sum of centrality scores of its neighbors.
# - Power iteration update:
#       c_new(dst) = Σ c_old(src) for all edges (src -> dst)
# - After each iteration, L2-normalize to keep values bounded.
#
# Requires:
# - edge_counts DataFrame with columns (src, dst) (weight optional; ignored here)
# - vertices DataFrame with column (id) OR we build it from edge_counts
#
# Output:
# - top 10 nodes by eigenvector centrality
# Produces:
# - evec DataFrame: (node, eigen_c)
# ========================

import math
import pyspark.sql.functions as F
from pyspark.storagelevel import StorageLevel

spark.conf.set("spark.sql.shuffle.partitions", "32")
spark.conf.set("spark.default.parallelism", "32")
spark.conf.set("spark.sql.adaptive.enabled", "true")

# Build vertices if missing
if "vertices" not in globals():
    vertices = (
        edge_counts.select(F.col("src").alias("id"))
        .union(edge_counts.select(F.col("dst").alias("id")))
        .distinct()
    )

# Treat graph as undirected for eigenvector centrality (common choice)
edges = (
    edge_counts.select(F.col("src").alias("src"), F.col("dst").alias("dst"))
    .union(edge_counts.select(F.col("dst").alias("src"), F.col("src").alias("dst")))
    .filter(F.col("src").isNotNull() & F.col("dst").isNotNull() & (F.col("src") != F.col("dst")))
    .distinct()
    .repartition("dst")
```

```
    .persist(StorageLevel.MEMORY_AND_DISK)
)

verts = (
    vertices.select(F.col("id").alias("node"))
            .filter(F.col("node").isNotNull())
            .distinct()
            .persist(StorageLevel.MEMORY_AND_DISK)
)

_ = edges.count()
_ = verts.count()

# Initialize centrality vector: c(node) = 1
r = verts.withColumn("c", F.lit(1.0)).persist(StorageLevel.MEMORY_AND_DISK)
_ = r.count()

num_iters = 15

for i in range(num_iters):

    # Core eigenvector centrality update: c_new(dst) = sum of c_old(src) for␣
 ↪incoming edges
    contribs = (
        edges.join(r, edges.src == r.node, "inner")
             .groupBy("dst")
             .agg(F.sum(F.col("c")).alias("c"))
    )

    r_new = (
        verts.join(contribs, verts.node == contribs.dst, "left")
             .select(verts.node.alias("node"), F.coalesce(F.col("c"), F.lit(0.
 ↪0)).alias("c"))
    )

    # L2 normalization (one small action per iteration)
    norm_sq = r_new.select(F.sum(F.col("c") * F.col("c")).alias("ns")).
 ↪first()["ns"]
    norm = math.sqrt(norm_sq) if norm_sq and norm_sq > 0 else 1.0

    r_new = r_new.withColumn("c", F.col("c") / F.lit(norm)) \
                 .persist(StorageLevel.MEMORY_AND_DISK)

    r.unpersist()
    r = r_new

# Result
```

```
evec = r.select(F.col("node"), F.col("c").alias("eigen_c"))
evec.orderBy(F.desc("eigen_c")).limit(10).show(truncate=False)
```

```
+----+------------------+
|node|eigen_c           |
+----+------------------+
|ATL |0.1734399146743071 |
|ORD |0.17178708117153438|
|DEN |0.16587975673931238|
|DFW |0.15891414031202006|
|MSP |0.15420328962929342|
|DTW |0.1530555615432875 |
|CLT |0.1520300715864211 |
|IAH |0.14715076171603145|
|LAS |0.1453291471581526 |
|EWR |0.1438885501986357 |
+----+------------------+
```

[8]:
```
# ========================
# PageRank (native Spark DataFrame, weighted, dangling handled)
# NO GraphFrames functions, NO checkpoint/localCheckpoint
# ========================

import os
import pyspark.sql.functions as F
from pyspark.storagelevel import StorageLevel

os.makedirs("/tmp/spark-local", exist_ok=True)

spark.conf.set("spark.sql.shuffle.partitions", "16")      # lower = less␣
 ↪shuffle memory
spark.conf.set("spark.default.parallelism", "16")
spark.conf.set("spark.sql.adaptive.enabled", "true")

# Ensure weight exists
if "weight" not in edge_counts.columns:
    edge_counts = edge_counts.withColumn("weight", F.lit(1.0))

# Ensure vertices exists
if "vertices" not in globals():
    vertices = (
        edge_counts.select(F.col("src").alias("id"))
        .union(edge_counts.select(F.col("dst").alias("id")))
        .distinct()
    )
```

```
verts = (
    vertices.select(F.col("id").alias("node"))
            .filter(F.col("node").isNotNull())
            .distinct()
            .persist(StorageLevel.MEMORY_AND_DISK)
)

edges = (
    edge_counts.select("src", "dst", "weight")
               .filter(F.col("src").isNotNull() & F.col("dst").isNotNull() & (F.
 ↪col("src") != F.col("dst")))
               .persist(StorageLevel.MEMORY_AND_DISK)
)

outdeg = (
    edges.groupBy("src")
         .agg(F.sum("weight").alias("out_w"))
         .persist(StorageLevel.MEMORY_AND_DISK)
)

# Materialize once
N = float(verts.count())
_ = edges.count()
_ = outdeg.count()

# Dangling nodes: no outgoing edges
dangling_nodes = (
    verts.join(outdeg.select(F.col("src").alias("node")), on="node",␣
 ↪how="left_anti")
         .persist(StorageLevel.MEMORY_AND_DISK)
)
_ = dangling_nodes.count()

# Pre-join edges with outdeg once (reduces per-iteration work)
edges_norm = (
    edges.join(outdeg, on="src", how="inner")
         .select("src", "dst", "weight", "out_w")
         .repartition("dst")
         .persist(StorageLevel.MEMORY_AND_DISK)
)
_ = edges_norm.count()

damping = 0.85
base = (1.0 - damping) / N

ranks = verts.withColumn("rank", F.lit(1.0 / N)).persist(StorageLevel.
 ↪MEMORY_AND_DISK)
```

```
_ = ranks.count()

num_iters = 20

for i in range(num_iters):
    # Compute dangling mass as a Python float (tiny collect; avoids crossJoin␣
 ↪and checkpoint)
    dm = (
        ranks.join(dangling_nodes, on="node", how="inner")
             .agg(F.sum("rank").alias("dm"))
             .first()["dm"]
    )
    dm = float(dm) if dm is not None else 0.0
    dangling_term = damping * dm / N

    contribs = (
        edges_norm.alias("e")
        .join(ranks.alias("r"), F.col("e.src") == F.col("r.node"), "inner")
        .select(
            F.col("e.dst").alias("node"),
            (F.col("r.rank") * (F.col("e.weight") / F.col("e.out_w"))).
 ↪alias("contrib")
        )
    )

    summed = contribs.groupBy("node").agg(F.sum("contrib").alias("sum_contrib"))

    new_ranks = (
        verts.join(summed, on="node", how="left")
             .select(
                 "node",
                 (F.lit(base + dangling_term) + F.lit(damping) * F.coalesce(F.
 ↪col("sum_contrib"), F.lit(0.0))).alias("rank")
             )
             .persist(StorageLevel.MEMORY_AND_DISK)
    )

    ranks.unpersist()
    ranks = new_ranks

    # Light "plan cut" without checkpoint: materialize every few iters
    if (i + 1) % 5 == 0:
        _ = ranks.count()

ranks.orderBy(F.desc("rank")).limit(10).show(truncate=False)
```

```
+----+-------------------+
```

```
|node|rank                |
+----+--------------------+
|ATL |0.058125417143436525|
|ORD |0.046779129117865303|
|DFW |0.04039777575019987 |
|DEN |0.0362993068819135  |
|LAX |0.028476336519615262|
|IAH |0.02373947040723849 |
|SFO |0.022906290038492084|
|PHX |0.022886791507266187|
|MSP |0.02096816056904428 |
|DTW |0.02054116037671708 |
+----+--------------------+
```

[9]:
```python
import pyspark.sql.functions as F
from graphframes import GraphFrame

# Build vertices (must be column name "id")
v = (
    edge_counts.select(F.col("src").alias("id"))
    .union(edge_counts.select(F.col("dst").alias("id")))
    .distinct()
)

# Build edges (must be columns "src", "dst")
e = edge_counts.select("src", "dst").distinct()

# Create GraphFrame
g = GraphFrame(v, e)

# Run PageRank (damping=0.85 => resetProbability=0.15)
pr = g.pageRank(resetProbability=0.15, maxIter=20)

# Top 10 by PageRank centrality
pr.vertices.select("id", "pagerank") \
  .orderBy(F.desc("pagerank")) \
  .show(10, truncate=False)

# Optional: keep for later joins/comparisons
ranks_gf = pr.vertices.select(F.col("id").alias("node"), F.col("pagerank").
 ↪alias("rank"))
```

```
+---+------------------+
|id |pagerank          |
+---+------------------+
|ORD|9.610560458643885 |
|ATL|9.26353967070715  |
```

```
|DEN|9.136974976874244 |
|DFW|9.10645536599632  |
|MSP|7.2696663196962295|
|CLT|6.291784094129735 |
|DTW|6.222117232002588 |
|IAH|6.103507154206821 |
|SLC|5.83094214674235  |
|LAX|5.753482547261935 |
+---+------------------+
only showing top 10 rows
```

[11]:
```python
# ========================
# Group of most connected airports (native Spark, NO GraphFrames)
# Method: Top-K hubs + induced subgraph density + top airports inside the group
#
# Requires:
#   edge_counts: DataFrame with columns (src, dst) and optional (weight)
#
# Outputs:
#   - Group summary: n, m, density, total internal weight
#   - Top 20 most connected airports within the group (internal weighted degree)
#   - Top strongest internal links (by weight)
#   - Optional sample of airports in the group
# ========================

import pyspark.sql.functions as F
from pyspark.storagelevel import StorageLevel

# Stability knobs for laptops
spark.conf.set("spark.sql.adaptive.enabled", "false")   # avoid plan bloat
spark.conf.set("spark.sql.shuffle.partitions", "32")
spark.conf.set("spark.default.parallelism", "32")

K = 300                 # try 200 / 300 / 500 (bigger = heavier)
TOP_EDGES_TO_SHOW = 20
SAMPLE_AIRPORTS_TO_SHOW = 50

# Ensure weight exists
if "weight" not in edge_counts.columns:
    edge_counts = edge_counts.withColumn("weight", F.lit(1.0))

# 1) Canonical undirected edges: one row per undirected pair (u<v) with␣
 ↪aggregated weight
undirected = (
    edge_counts
    .select(
```

```
        F.when(F.col("src") < F.col("dst"), F.col("src")).otherwise(F.
 ↪col("dst")).alias("u"),
        F.when(F.col("src") < F.col("dst"), F.col("dst")).otherwise(F.
 ↪col("src")).alias("v"),
        F.col("weight").cast("double").alias("w")
    )
    .filter(F.col("u").isNotNull() & F.col("v").isNotNull() & (F.col("u") != F.
 ↪col("v")))
    .groupBy("u", "v")
    .agg(F.sum("w").alias("w"))
    .persist(StorageLevel.MEMORY_AND_DISK)
)
_ = undirected.count()

print("Undirected edges (unique):", undirected.count())

# 2) Weighted degree: sum of incident weights on u and v
deg_u = undirected.groupBy("u").agg(F.sum("w").alias("wd")).
 ↪withColumnRenamed("u", "id")
deg_v = undirected.groupBy("v").agg(F.sum("w").alias("wd")).
 ↪withColumnRenamed("v", "id")

wdeg = (
    deg_u.unionByName(deg_v)
         .groupBy("id")
         .agg(F.sum("wd").alias("weighted_degree"))
         .persist(StorageLevel.MEMORY_AND_DISK)
)
_ = wdeg.count()

# 3) Candidate "most connected group": top-K airports by weighted degree
topK = wdeg.orderBy(F.desc("weighted_degree")).limit(K).select("id").
 ↪persist(StorageLevel.MEMORY_AND_DISK)
n0 = topK.count()
print(f"Candidate group: top-{K} airports by weighted degree => n={n0}")

# 4) Induced edges among topK (still undirected u<v)
topK_u = topK.withColumnRenamed("id", "u")
topK_v = topK.withColumnRenamed("id", "v")

E = (
    undirected.join(topK_u, on="u", how="inner")
              .join(topK_v, on="v", how="inner")
              .select("u", "v", "w")
              .persist(StorageLevel.MEMORY_AND_DISK)
)
```

```
m = E.count()

# Vertices actually present in induced subgraph (some topK may be isolated␣
 ↪within topK)
V = (
    E.select(F.col("u").alias("id"))
     .union(E.select(F.col("v").alias("id")))
     .distinct()
     .persist(StorageLevel.MEMORY_AND_DISK)
)
n = V.count()

density = (2.0 * m) / (n * (n - 1)) if n > 1 else 0.0
total_w = E.agg(F.sum("w").alias("tw")).first()["tw"]
total_w = float(total_w) if total_w is not None else 0.0

print("\nMost connected group (Top-K induced subgraph) summary:")
print(f"  nodes n = {n}")
print(f"  edges m = {m}")
print(f"  density = {density:.6f}   (unweighted undirected density)")
print(f"  total internal weight = {total_w:.2f}")

# 5) "Most connected airports" INSIDE the group (internal weighted degree)
# Internal weighted degree = sum of edge weights incident to the airport,␣
 ↪restricted to edges inside E.
deg_u_in = E.groupBy("u").agg(F.sum("w").alias("internal_wdeg")).
 ↪withColumnRenamed("u", "id")
deg_v_in = E.groupBy("v").agg(F.sum("w").alias("internal_wdeg")).
 ↪withColumnRenamed("v", "id")

internal_wdeg = (
    deg_u_in.unionByName(deg_v_in)
            .groupBy("id")
            .agg(F.sum("internal_wdeg").alias("internal_weighted_degree"))
            .persist(StorageLevel.MEMORY_AND_DISK)
)

print("\nTop 20 most connected airports within the group (by internal weighted␣
 ↪degree):")
internal_wdeg.orderBy(F.desc("internal_weighted_degree")).limit(20).
 ↪show(truncate=False)

# 6) Strongest internal links (justify why this is a tightly connected group)
print(f"\nTop {TOP_EDGES_TO_SHOW} strongest internal links (by weight) inside␣
 ↪the group:")
E.orderBy(F.desc("w")).limit(TOP_EDGES_TO_SHOW).show(truncate=False)
```

```
Undirected edges (unique): 4199
Candidate group: top-300 airports by weighted degree => n=300

Most connected group (Top-K induced subgraph) summary:
  nodes n = 300
  edges m = 3979
  density = 0.088718   (unweighted undirected density)
  total internal weight = 61337065.00

Top 20 most connected airports within the group (by internal weighted degree):
+---+------------------------+
|id |internal_weighted_degree|
+---+------------------------+
|ATL|7798882.0               |
|ORD|5979253.0               |
|DFW|5078305.0               |
|DEN|4574152.0               |
|LAX|4249881.0               |
|PHX|3437693.0               |
|IAH|3343157.0               |
|SFO|3220774.0               |
|LAS|2943359.0               |
|CLT|2662854.0               |
|DTW|2561886.0               |
|MSP|2505363.0               |
|MCO|2449376.0               |
|EWR|2365352.0               |
|BOS|2342236.0               |
|SLC|2285056.0               |
|SEA|2283106.0               |
|LGA|2161231.0               |
|JFK|2112953.0               |
|BWI|2006880.0               |
+---+------------------------+


Top 20 strongest internal links (by weight) inside the group:
+---+---+--------+
|u  |v  |w       |
+---+---+--------+
|LAX|SFO|295359.0|
|JFK|LAX|227236.0|
|LAS|LAX|224102.0|
|HNL|OGG|216280.0|
|LGA|ORD|208018.0|
|ATL|LGA|187929.0|
|ATL|MCO|181334.0|
|BOS|DCA|168413.0|
```

```
|LAX|PHX|166629.0|
|LAX|ORD|162694.0|
|DEN|PHX|161102.0|
|ATL|DFW|159736.0|
|BOS|LGA|158415.0|
|DEN|LAX|158363.0|
|JFK|SFO|157194.0|
|LAS|SFO|156456.0|
|ATL|FLL|155172.0|
|DAL|HOU|154011.0|
|HNL|KOA|153485.0|
|DEN|SLC|153378.0|
+---+---+--------+
```

[ ]:
```python
import pyspark.sql.functions as F
from graphframes import GraphFrame

# ----------------------------
# 0) Build GraphFrame (for comparison only)
# ----------------------------
v_gf = edge_counts.select(F.col("src").alias("id")).union(edge_counts.select(F.
 ↪col("dst").alias("id"))).distinct()
e_gf = edge_counts.select("src", "dst").distinct()
g = GraphFrame(v_gf, e_gf)

# GraphFrames degree (undirected total degree)
gf_deg = g.degrees.select("id", F.col("degree").alias("degree_gf"))

# GraphFrames PageRank (comparison only)
gf_pr = g.pageRank(resetProbability=0.15, maxIter=20).vertices \
         .select("id", F.col("pagerank").alias("pagerank_gf"))

# ----------------------------
# 1) Your native results
# ----------------------------
native_deg = degrees.select("id", "in_degree", "out_degree", "total_degree")

native_evec = evec.select(F.col("node").alias("id"), F.col("eigen_c").
 ↪alias("eigen_c_native"))

native_pr = ranks.select(F.col("node").alias("id"), F.col("rank").
 ↪alias("pagerank_native"))

# ----------------------------
# 2) Join and compare
# ----------------------------
```

```
comparison = (native_deg
              .join(native_evec, "id", "left")
              .join(native_pr, "id", "left")
              .join(gf_deg, "id", "left")
              .join(gf_pr, "id", "left")
              .na.fill(0))

print("\nTop 20 by native PageRank vs GraphFrames PageRank:")
comparison.orderBy(F.desc("pagerank_native")).select(
    "id", "pagerank_native", "pagerank_gf", "total_degree", "degree_gf",␣
 ↪"eigen_c_native"
).limit(20).show(truncate=False)

print("\nTop 20 by GraphFrames PageRank:")
comparison.orderBy(F.desc("pagerank_gf")).select(
    "id", "pagerank_gf", "pagerank_native", "total_degree", "degree_gf",␣
 ↪"eigen_c_native"
).limit(20).show(truncate=False)

print("\nTop 20 by native total_degree vs GraphFrames degree:")
comparison.orderBy(F.desc("total_degree")).select(
    "id", "total_degree", "degree_gf", "pagerank_native", "pagerank_gf",␣
 ↪"eigen_c_native"
).limit(20).show(truncate=False)

print("\nTop 20 by eigenvector centrality (native):")
comparison.orderBy(F.desc("eigen_c_native")).select(
    "id", "eigen_c_native", "total_degree", "degree_gf", "pagerank_native",␣
 ↪"pagerank_gf"
).limit(20).show(truncate=False)
```

[ ]:
```
# Optional: draw a heatmap of flights between top airports
# This will collect a small dense matrix to the driver - do it only for top-K␣
 ↪airports.
import matplotlib.pyplot as plt
import pandas as pd

top_k = 30
top_airports = degrees.orderBy(F.desc("total_degree")).limit(top_k).
 ↪select("id").rdd.flatMap(lambda x: x).collect()
# filter edges to top airports and pivot to matrix
sub = edge_counts.filter((F.col("src").isin(top_airports)) & (F.col("dst").
 ↪isin(top_airports)))
pdf = sub.toPandas()
mat = pd.pivot_table(pdf, values='weight', index='src', columns='dst',␣
 ↪fill_value=0)
```

```
plt.figure(figsize=(10,8))
plt.imshow(mat.values, aspect='auto')
plt.colorbar()
plt.xticks(range(len(mat.columns)), mat.columns, rotation=90)
plt.yticks(range(len(mat.index)), mat.index)
plt.title("Heatmap of flights between top {} airports".format(top_k))
plt.tight_layout()
plt.show()
```

[ ]: