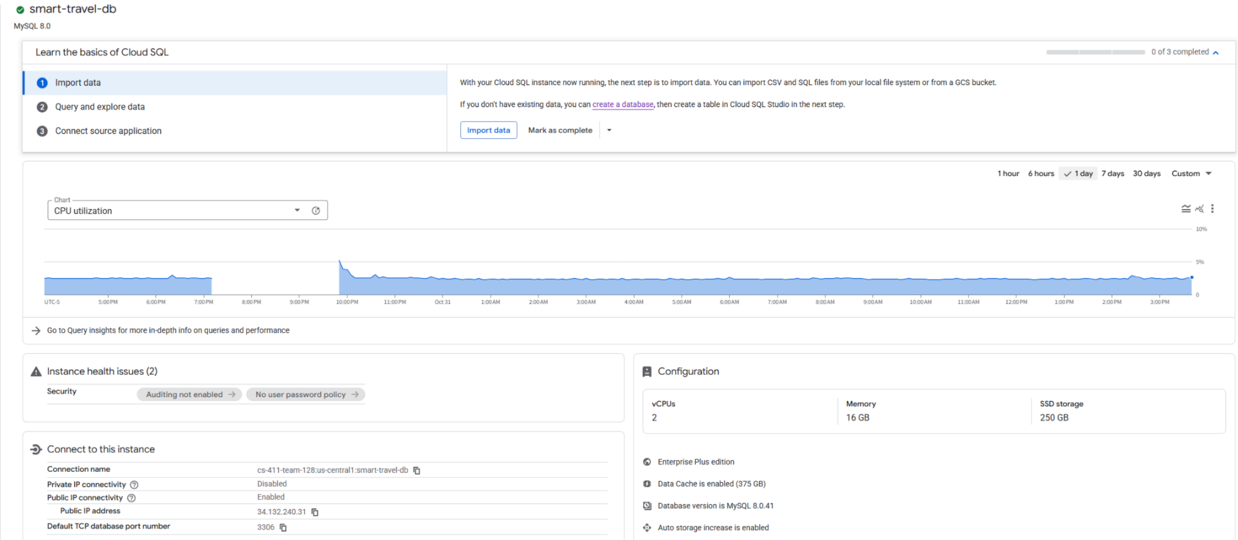
Smart Travel Planner: Part 3

Google GCP

For part 3, we use CloudSQL on Google Cloud Platform to host our database.



smart-travel-db
MySQL 8.0

Learn the basics of Cloud SQL

0 of 3 completed

1. Import data
2. Query and explore data
3. Connect source application

With your Cloud SQL instance now running, the next step is to import data. You can import CSV and SQL files from your local file system or from a GCS bucket.

If you don't have existing data, you can create a database, then create a table in Cloud SQL Studio in the next step.

Go to Query insights for more in-depth info on queries and performance

Instance health issues (2)

Security: Auditing not enabled → No user password policy →

Connect to this instance

Connection name	cs-411-team-128:us-central1:smart-travel-db
Private IP connectivity	Disabled
Public IP connectivity	Enabled
Public IP address	34.132.240.31
Default TCP database port number	3306

Configuration

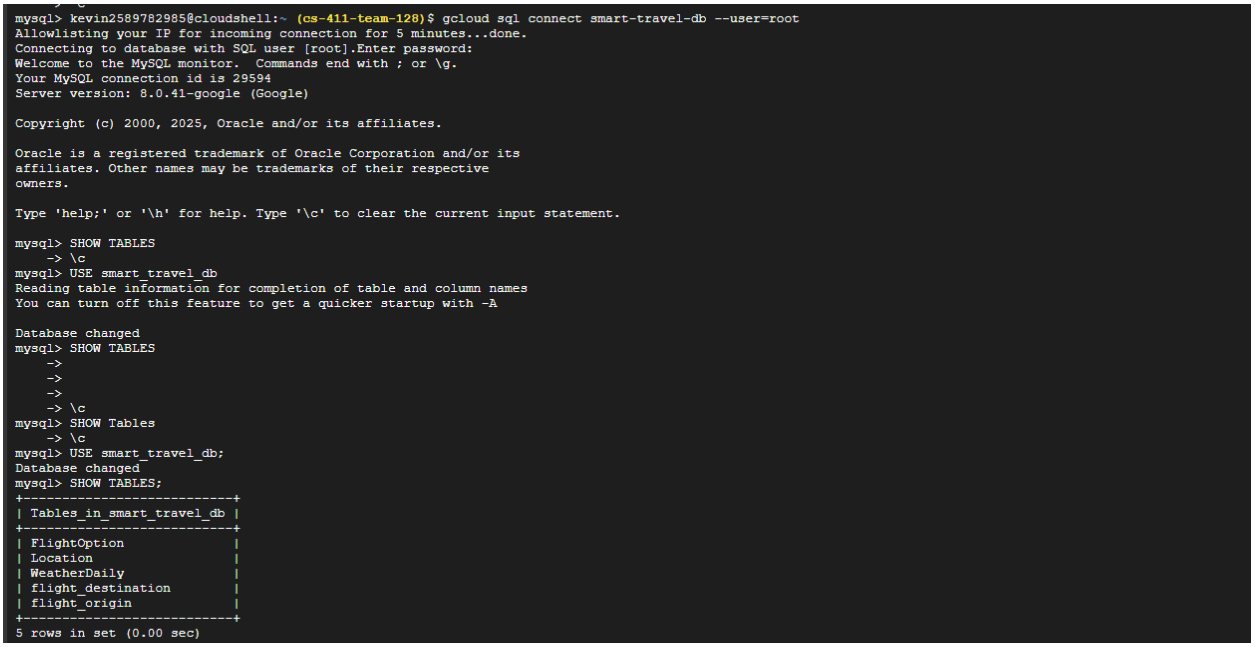
vCPUs	Memory	SSD storage
2	16 GB	250 GB

Enterprise Plus edition

Data Cache is enabled (375 GB)

Database version is MySQL 8.0.41

Auto storage increase is enabled



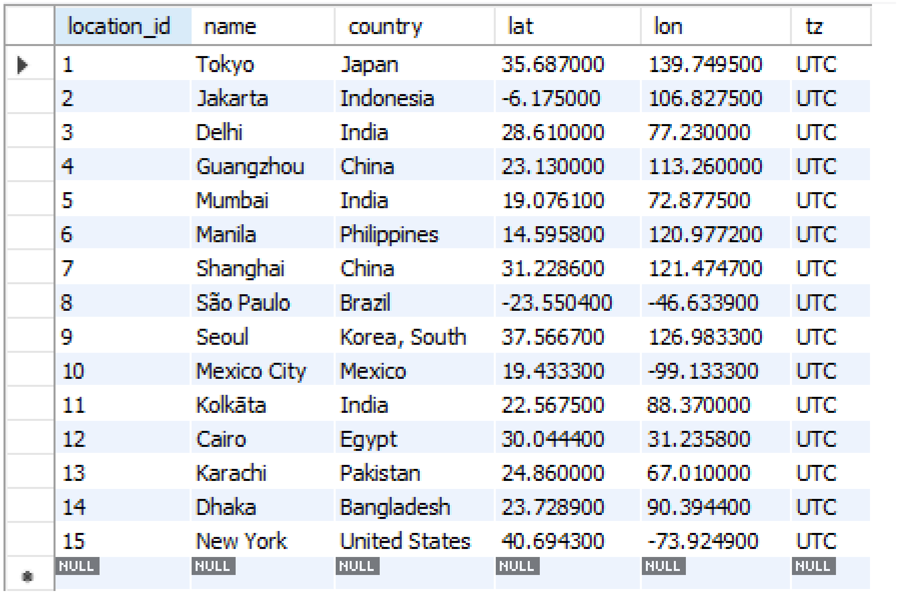
```
mysql> kevin2589782985@cloudshell:~ (cs-411-team-128)$ gcloud sql connect smart-travel-db --user=root
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 29594
Server version: 8.0.41-google (Google)

Copyright (c) 2000, 2025, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> SHOW TABLES
    -> \c
mysql> USE smart_travel_db
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SHOW TABLES
    ->
    ->
    ->
    -> \c
mysql> SHOW Tables
    -> \c
mysql> USE smart_travel_db;
Database changed
mysql> SHOW TABLES;
+---------------------------+
| Tables_in_smart_travel_db |
+---------------------------+
| FlightOption              |
| Location                  |
| WeatherDaily              |
| flight_destination        |
| flight_origin             |
+---------------------------+
5 rows in set (0.00 sec)
```

Table Syntax and Results:

CREATE DATABASE smart_travel_db;
USE smart_travel_db;

Location:

```
CREATE TABLE Location (
  location_id BIGINT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(160) NOT NULL,
  country VARCHAR(80) NOT NULL,
  lat DECIMAL(9,6) NOT NULL,
  lon DECIMAL(9,6) NOT NULL,
  tz VARCHAR(40) NOT NULL
);
```

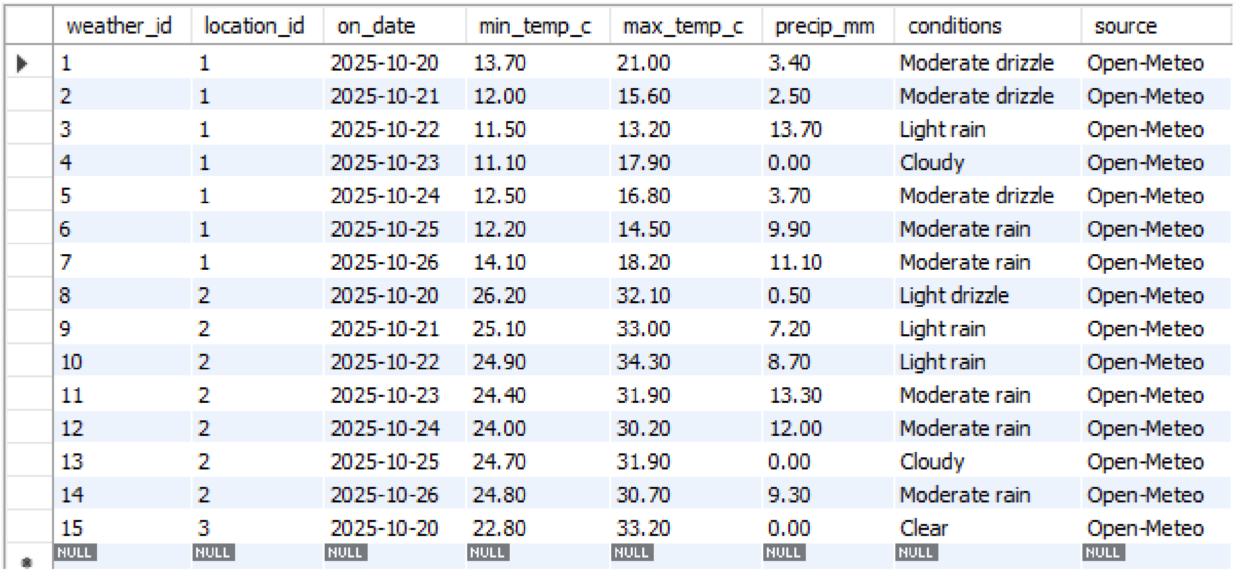


location_id	name	country	lat	lon	tz
1	Tokyo	Japan	35.687000	139.749500	UTC
2	Jakarta	Indonesia	-6.175000	106.827500	UTC
3	Delhi	India	28.610000	77.230000	UTC
4	Guangzhou	China	23.130000	113.260000	UTC
5	Mumbai	India	19.076100	72.877500	UTC
6	Manila	Philippines	14.595800	120.977200	UTC
7	Shanghai	China	31.228600	121.474700	UTC
8	São Paulo	Brazil	-23.550400	-46.633900	UTC
9	Seoul	Korea, South	37.566700	126.983300	UTC
10	Mexico City	Mexico	19.433300	-99.133300	UTC
11	Kolkāta	India	22.567500	88.370000	UTC
12	Cairo	Egypt	30.044400	31.235800	UTC
13	Karachi	Pakistan	24.860000	67.010000	UTC
14	Dhaka	Bangladesh	23.728900	90.394400	UTC
15	New York	United States	40.694300	-73.924900	UTC
NULL	NULL	NULL	NULL	NULL	NULL

The timezone column is temporarily set to UTC because we were unable to find a comprehensive datasheet that includes time zones. We inserted 500 location entries into the Location table for now.

WeatherDaily

```
CREATE TABLE WeatherDaily (
  weather_id BIGINT PRIMARY KEY AUTO_INCREMENT,
  location_id BIGINT NOT NULL,
  on_date DATE NOT NULL,
  min_temp_c DECIMAL(5,2),
  max_temp_c DECIMAL(5,2),
  precip_mm DECIMAL(6,2),
  conditions VARCHAR(64),
  source VARCHAR(40),
  CONSTRAINT weather_loc
    FOREIGN KEY (location_id) REFERENCES Location(location_id)
      ON DELETE CASCADE
);
```

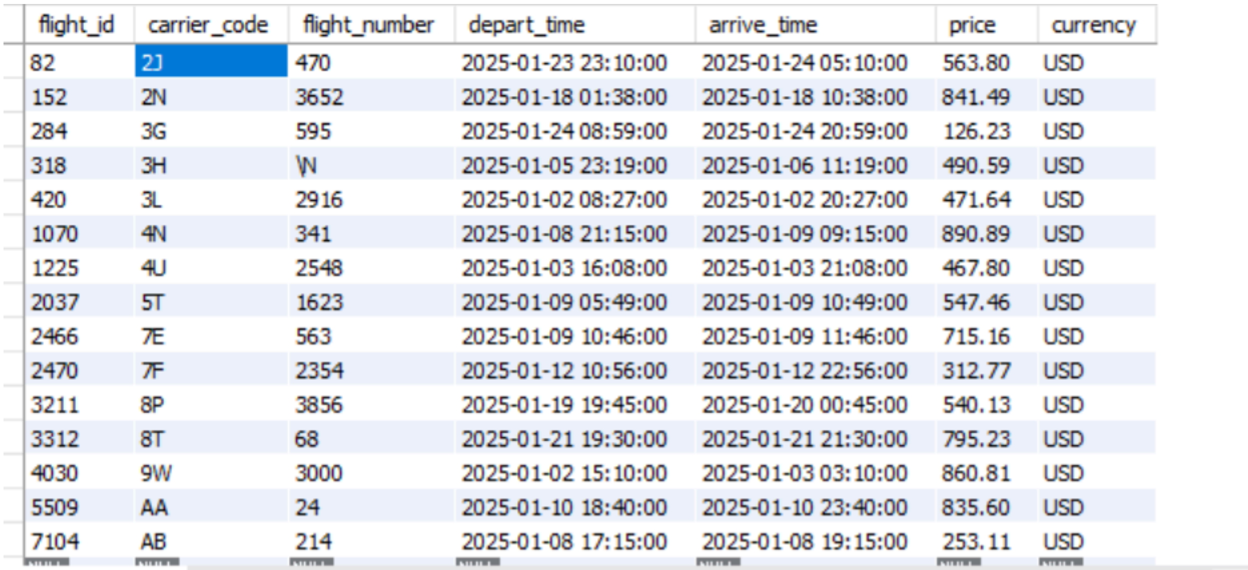


weather_id	location_id	on_date	min_temp_c	max_temp_c	precip_mm	conditions	source
1	1	2025-10-20	13.70	21.00	3.40	Moderate drizzle	Open-Meteo
2	1	2025-10-21	12.00	15.60	2.50	Moderate drizzle	Open-Meteo
3	1	2025-10-22	11.50	13.20	13.70	Light rain	Open-Meteo
4	1	2025-10-23	11.10	17.90	0.00	Cloudy	Open-Meteo
5	1	2025-10-24	12.50	16.80	3.70	Moderate drizzle	Open-Meteo
6	1	2025-10-25	12.20	14.50	9.90	Moderate rain	Open-Meteo
7	1	2025-10-26	14.10	18.20	11.10	Moderate rain	Open-Meteo
8	2	2025-10-20	26.20	32.10	0.50	Light drizzle	Open-Meteo
9	2	2025-10-21	25.10	33.00	7.20	Light rain	Open-Meteo
10	2	2025-10-22	24.90	34.30	8.70	Light rain	Open-Meteo
11	2	2025-10-23	24.40	31.90	13.30	Moderate rain	Open-Meteo
12	2	2025-10-24	24.00	30.20	12.00	Moderate rain	Open-Meteo
13	2	2025-10-25	24.70	31.90	0.00	Cloudy	Open-Meteo
14	2	2025-10-26	24.80	30.70	9.30	Moderate rain	Open-Meteo
15	3	2025-10-20	22.80	33.20	0.00	Clear	Open-Meteo
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

For now, we imported last week's weather information corresponding to each location_id. So, a total of 3500 entries in the WeatherDaily table.

FlightOptions:

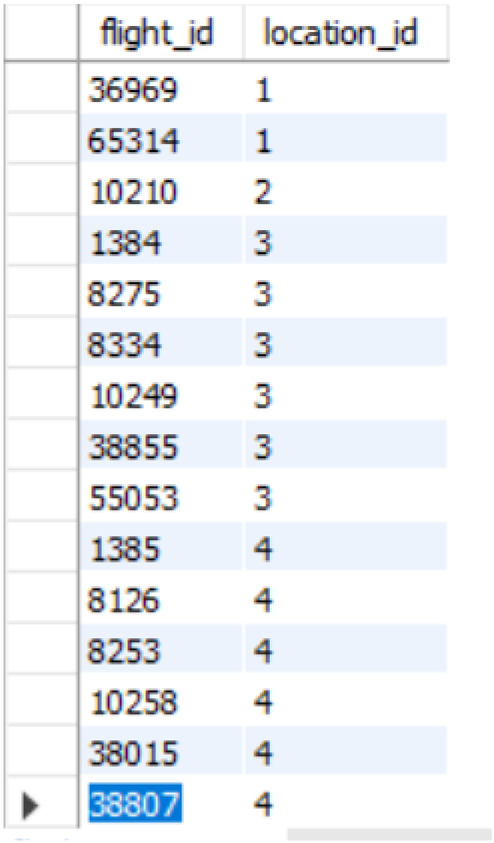
```
CREATE TABLE IF NOT EXISTS FlightOption (
  flight_id     INT PRIMARY KEY,
  carrier_code  VARCHAR(12)  NOT NULL,
  flight_number VARCHAR(12)  NOT NULL,
  depart_time   TIMESTAMP    NOT NULL,
  arrive_time   TIMESTAMP    NOT NULL,
  price         DECIMAL(10,2) NOT NULL CHECK (price >= 0),
  currency      CHAR(3)      NOT NULL
 );
```



flight_id	carrier_code	flight_number	depart_time	arrive_time	price	currency
82	2J	470	2025-01-23 23:10:00	2025-01-24 05:10:00	563.80	USD
152	2N	3652	2025-01-18 01:38:00	2025-01-18 10:38:00	841.49	USD
284	3G	595	2025-01-24 08:59:00	2025-01-24 20:59:00	126.23	USD
318	3H	\N	2025-01-05 23:19:00	2025-01-06 11:19:00	490.59	USD
420	3L	2916	2025-01-02 08:27:00	2025-01-02 20:27:00	471.64	USD
1070	4N	341	2025-01-08 21:15:00	2025-01-09 09:15:00	890.89	USD
1225	4U	2548	2025-01-03 16:08:00	2025-01-03 21:08:00	467.80	USD
2037	5T	1623	2025-01-09 05:49:00	2025-01-09 10:49:00	547.46	USD
2466	7E	563	2025-01-09 10:46:00	2025-01-09 11:46:00	715.16	USD
2470	7F	2354	2025-01-12 10:56:00	2025-01-12 22:56:00	312.77	USD
3211	8P	3856	2025-01-19 19:45:00	2025-01-20 00:45:00	540.13	USD
3312	8T	68	2025-01-21 19:30:00	2025-01-21 21:30:00	795.23	USD
4030	9W	3000	2025-01-02 15:10:00	2025-01-03 03:10:00	860.81	USD
5509	AA	24	2025-01-10 18:40:00	2025-01-10 23:40:00	835.60	USD
7104	AB	214	2025-01-08 17:15:00	2025-01-08 19:15:00	253.11	USD

FlightOrigin:

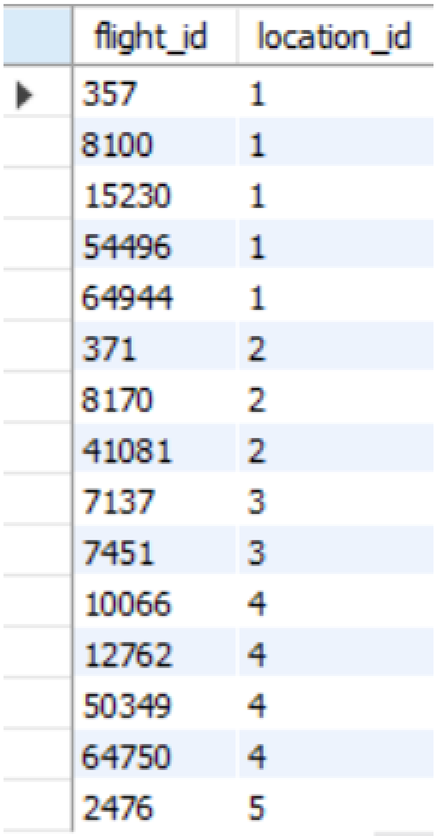
```
CREATE TABLE IF NOT EXISTS flight_origin (
  flight_id   INT NOT NULL,
  location_id BIGINT NOT NULL,
  PRIMARY KEY (flight_id),
  FOREIGN KEY (flight_id)
  REFERENCES FlightOption(flight_id)
  ON DELETE CASCADE ON UPDATE CASCADE,
  FOREIGN KEY (location_id)
  REFERENCES Location(location_id)
 ON DELETE CASCADE ON UPDATE CASCADE
);
```



flight_id	location_id
36969	1
65314	1
10210	2
1384	3
8275	3
8334	3
10249	3
38855	3
55053	3
1385	4
8126	4
8253	4
10258	4
38015	4
38807	4

FlightDestination:

```
CREATE TABLE IF NOT EXISTS flight_destination (
 flight_id   INT NOT NULL,
 location_id BIGINT NOT NULL,
 PRIMARY KEY (flight_id),
 FOREIGN KEY (flight_id)
 REFERENCES FlightOption(flight_id)
 ON DELETE CASCADE ON UPDATE CASCADE,
  FOREIGN KEY (location_id)
   REFERENCES Location(location_id)
   ON DELETE CASCADE ON UPDATE CASCADE
);
```

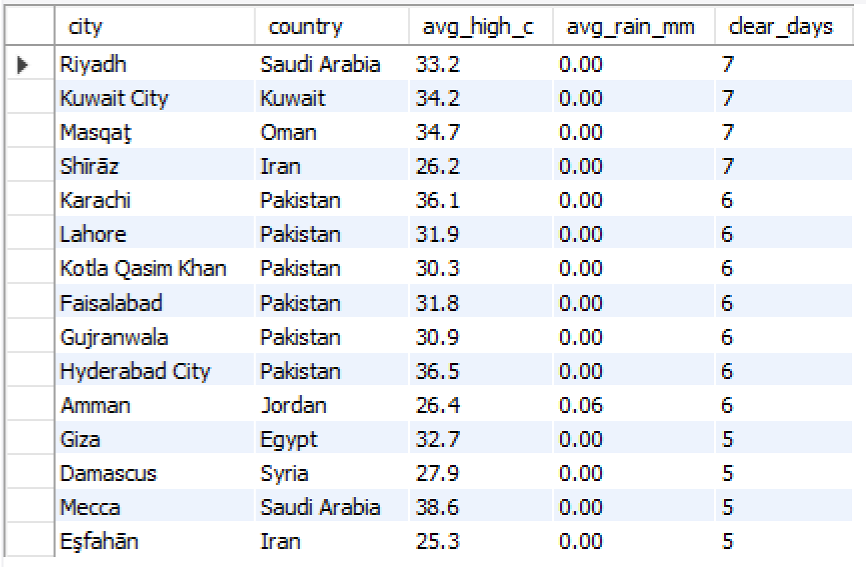


flight_id	location_id
357	1
8100	1
15230	1
54496	1
64944	1
371	2
8170	2
41081	2
7137	3
7451	3
10066	4
12762	4
50349	4
64750	4
2476	5

Advanced SQL Queries and Results:

1. SELECT l.name AS city, l.country, ROUND(AVG(w.max_temp_c), 1) AS avg_high_c, ROUND(AVG(w.precip_mm), 2) AS avg_rain_mm, SUM(CASE WHEN w.conditions = 'Clear' THEN 1 ELSE 0 END) AS clear_days
   FROM Location AS l JOIN WeatherDaily AS w ON l.location_id = w.location_id
   WHERE w.on_date >= '2025-10-20' AND w.on_date <= '2025-10-26'
   GROUP BY l.location_id, l.name, l.country
   HAVING COUNT(*) >= 3
   ORDER BY clear_days DESC, avg_rain_mm ASC
   LIMIT 15;

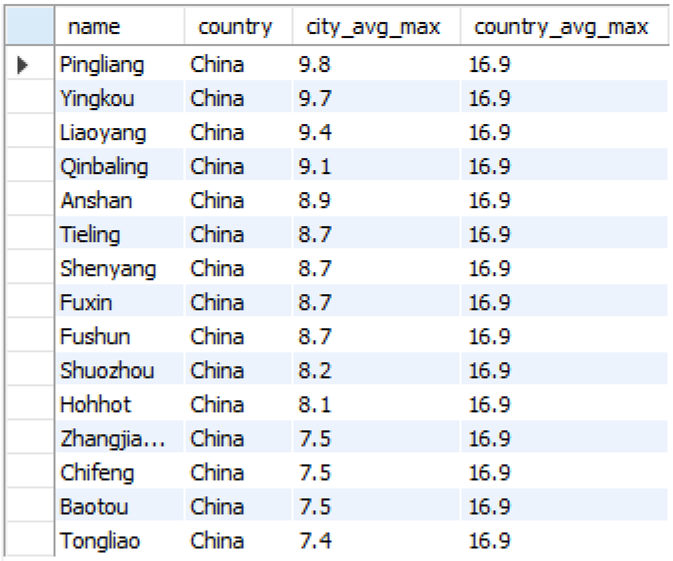
   This query finds the top 15 cities with the maximum number of sunny days and the lowest precipitation in the last week, 10/20 - 10/26. This query uses JOIN between WeatherDaily and Location, GROUP BY for aggregation functions, and HAVING to limit the outputs.



	city	country	avg_high_c	avg_rain_mm	clear_days
▶	Riyadh	Saudi Arabia	33.2	0.00	7
	Kuwait City	Kuwait	34.2	0.00	7
	Masqaţ	Oman	34.7	0.00	7
	Shīrāz	Iran	26.2	0.00	7
	Karachi	Pakistan	36.1	0.00	6
	Lahore	Pakistan	31.9	0.00	6
	Kotla Qasim Khan	Pakistan	30.3	0.00	6
	Faisalabad	Pakistan	31.8	0.00	6
	Gujranwala	Pakistan	30.9	0.00	6
	Hyderabad City	Pakistan	36.5	0.00	6
	Amman	Jordan	26.4	0.06	6
	Giza	Egypt	32.7	0.00	5
	Damascus	Syria	27.9	0.00	5
	Mecca	Saudi Arabia	38.6	0.00	5
	Eşfahān	Iran	25.3	0.00	5

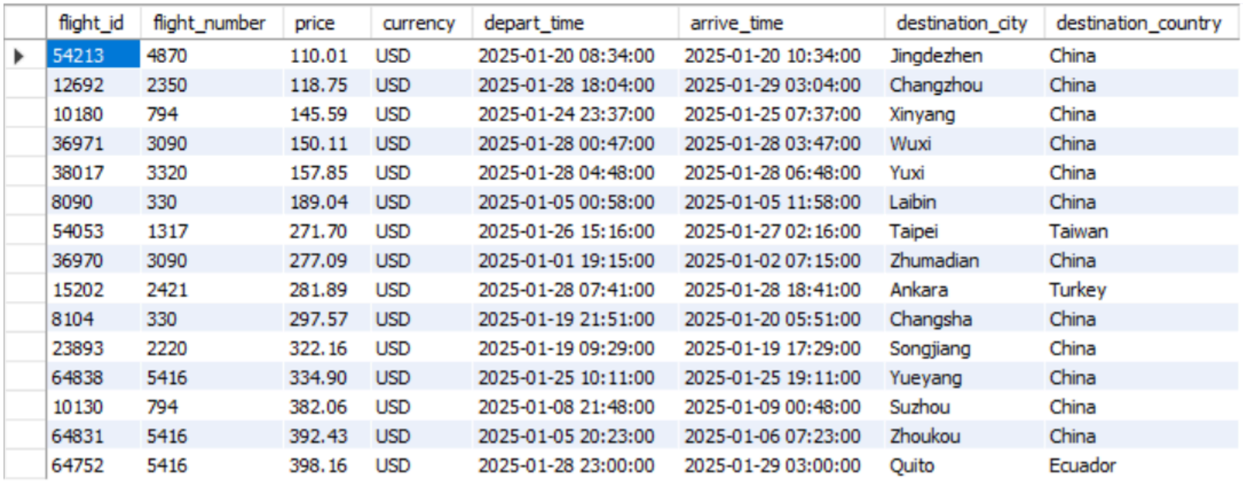
2.
   ```
   SELECT l.name,l.country,ROUND(AVG(w.max_temp_c), 1) AS
   city_avg_max,ROUND
   ((SELECT AVG(w2.max_temp_c) FROM WeatherDaily AS w2 JOIN Location
   AS l2 ON w2.location_id = l2.location_id WHERE l2.country = l.country), 1) AS
   country_avg_max
   FROM Location AS l JOIN WeatherDaily AS w ON l.location_id = w.location_id
   GROUP BY l.location_id, l.name, l.country
   HAVING (country_avg_max - city_avg_max) >= 7
   ORDER BY l.country, city_avg_max DESC
   LIMIT 15;
   ```

   This query finds the top 15 cities with colder temperatures than their country's average temperatures. This can be very useful for users who are looking for places to ski in their countries. It uses JOIN and a Subquery for the countries' average.



name	country	city_avg_max	country_avg_max
Pingliang	China	9.8	16.9
Yingkou	China	9.7	16.9
Liaoyang	China	9.4	16.9
Qinbaling	China	9.1	16.9
Anshan	China	8.9	16.9
Tieling	China	8.7	16.9
Shenyang	China	8.7	16.9
Fuxin	China	8.7	16.9
Fushun	China	8.7	16.9
Shuozhou	China	8.2	16.9
Hohhot	China	8.1	16.9
Zhangjia...	China	7.5	16.9
Chifeng	China	7.5	16.9
Baotou	China	7.5	16.9
Tongliao	China	7.4	16.9

```
3.SELECT
  FlightOption.flight_id,
  FlightOption.flight_number,
  FlightOption.price,
  FlightOption.currency,
  FlightOption.depart_time,
  FlightOption.arrive_time,
  Location.name    AS destination_city,
  Location.country AS destination_country
FROM FlightOption
JOIN flight_origin  ON   flight_origin.flight_id = FlightOption.flight_id
JOIN flight_destination  ON flight_destination.flight_id = FlightOption.flight_id
JOIN Location  ON Location.location_id = flight_destination.location_id
JOIN
(SELECT l.name,
                        l.location_id,
                        l.country,
        ROUND(AVG(w.max_temp_c), 1) AS city_avg_max
FROM Location AS l
JOIN WeatherDaily AS w ON l.location_id = w.location_id
GROUP BY  l.name,l.location_id ,l.country
HAVING city_avg_max BETWEEN 15 AND 25) AS good_temp_places
ON good_temp_places.location_id = flight_destination.location_id
WHERE flight_origin.location_id IN (
  SELECT location_id FROM Location WHERE country='United States'
)ORDER BY FlightOption.price ASC
LIMIT 15;
```

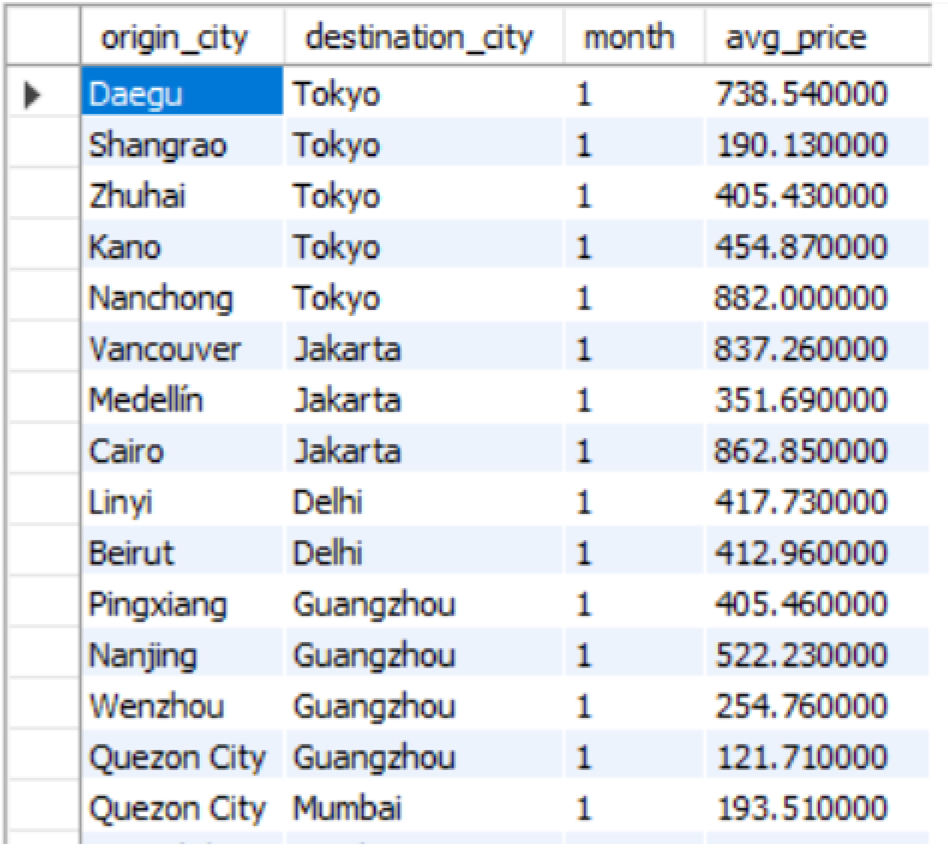


	flight_id	flight_number	price	currency	depart_time	arrive_time	destination_city	destination_country
▶	54213	4870	110.01	USD	2025-01-20 08:34:00	2025-01-20 10:34:00	Jingdezhen	China
	12692	2350	118.75	USD	2025-01-28 18:04:00	2025-01-29 03:04:00	Changzhou	China
	10180	794	145.59	USD	2025-01-24 23:37:00	2025-01-25 07:37:00	Xinyang	China
	36971	3090	150.11	USD	2025-01-28 00:47:00	2025-01-28 03:47:00	Wuxi	China
	38017	3320	157.85	USD	2025-01-28 04:48:00	2025-01-28 06:48:00	Yuxi	China
	8090	330	189.04	USD	2025-01-05 00:58:00	2025-01-05 11:58:00	Laibin	China
	54053	1317	271.70	USD	2025-01-26 15:16:00	2025-01-27 02:16:00	Taipei	Taiwan
	36970	3090	277.09	USD	2025-01-01 19:15:00	2025-01-02 07:15:00	Zhumadian	China
	15202	2421	281.89	USD	2025-01-28 07:41:00	2025-01-28 18:41:00	Ankara	Turkey
	8104	330	297.57	USD	2025-01-19 21:51:00	2025-01-20 05:51:00	Changsha	China
	23893	2220	322.16	USD	2025-01-19 09:29:00	2025-01-19 17:29:00	Songjiang	China
	64838	5416	334.90	USD	2025-01-25 10:11:00	2025-01-25 19:11:00	Yueyang	China
	10130	794	382.06	USD	2025-01-08 21:48:00	2025-01-09 00:48:00	Suzhou	China
	64831	5416	392.43	USD	2025-01-05 20:23:00	2025-01-06 07:23:00	Zhoukou	China
	64752	5416	398.16	USD	2025-01-28 23:00:00	2025-01-29 03:00:00	Quito	Ecuador

This query finds the 15 cheapest flights from the origin(Chicago in this case) to destinations with pleasant temperatures (15–25°C). This can be useful if the user wants to search for destinations based on the flight prices and the weather.

4.

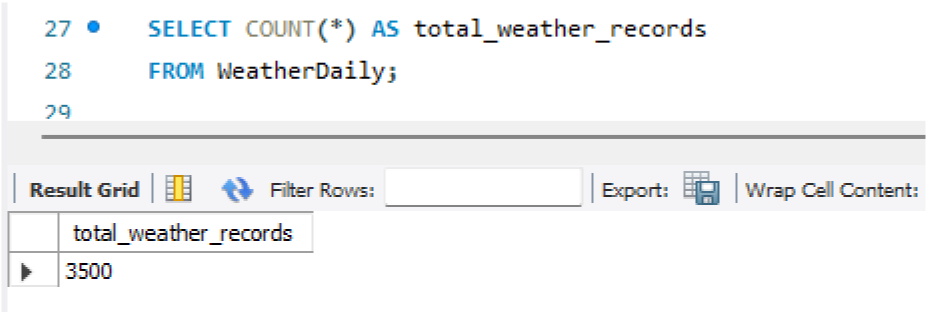
```
SELECT
  a.origin_city,
  a.destination_city,
  a.month,
  AVG(a.price) AS avg_price
FROM (
  SELECT
    FlightOption.price,
    MONTH(FlightOption.depart_time) AS month,
    l1.name AS origin_city,
    l2.name AS destination_city
  FROM FlightOption
  JOIN flight_origin  ON flight_origin.flight_id = FlightOption.flight_id
  JOIN Location AS l1  ON l1.location_id = flight_origin.location_id
  JOIN flight_destination  ON flight_destination.flight_id = FlightOption.flight_id
  JOIN Location AS l2 ON l2.location_id = flight_destination.location_id
) AS a
GROUP BY a.origin_city, a.destination_city, a.month;
```



origin_city	destination_city	month	avg_price
Daegu	Tokyo	1	738.540000
Shangrao	Tokyo	1	190.130000
Zhuhai	Tokyo	1	405.430000
Kano	Tokyo	1	454.870000
Nanchong	Tokyo	1	882.000000
Vancouver	Jakarta	1	837.260000
Medellín	Jakarta	1	351.690000
Cairo	Jakarta	1	862.850000
Linyi	Delhi	1	417.730000
Beirut	Delhi	1	412.960000
Pingxiang	Guangzhou	1	405.460000
Nanjing	Guangzhou	1	522.230000
Wenzhou	Guangzhou	1	254.760000
Quezon City	Guangzhou	1	121.710000
Quezon City	Mumbai	1	193.510000

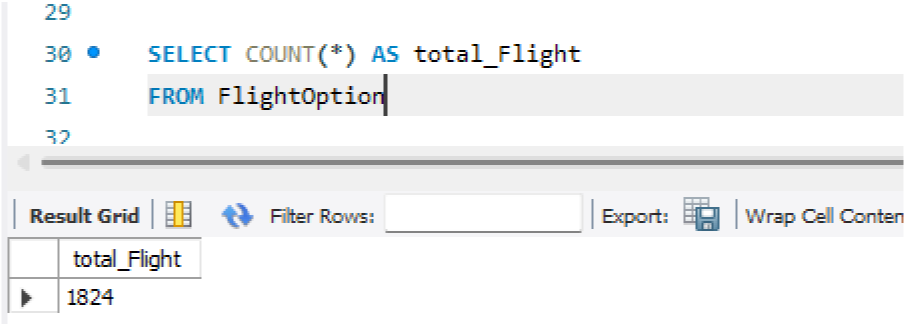
This query gives us the monthly average price for every flight route that is available. This way, the users could pick their origin and destination and check the monthly average trend of the flight prices.

Database entries count:



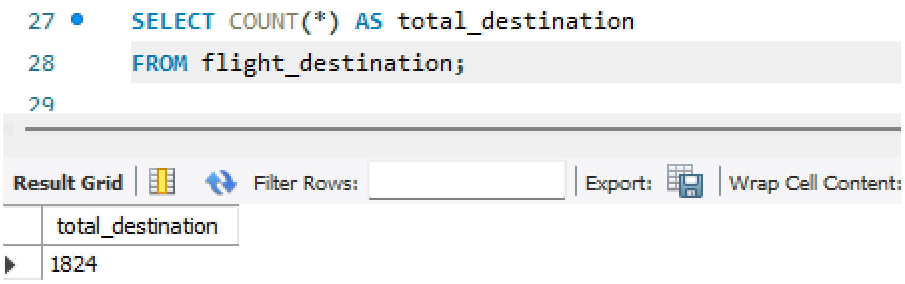
```sql
SELECT COUNT(*) AS total_weather_records
FROM WeatherDaily;
```

total_weather_records
3500



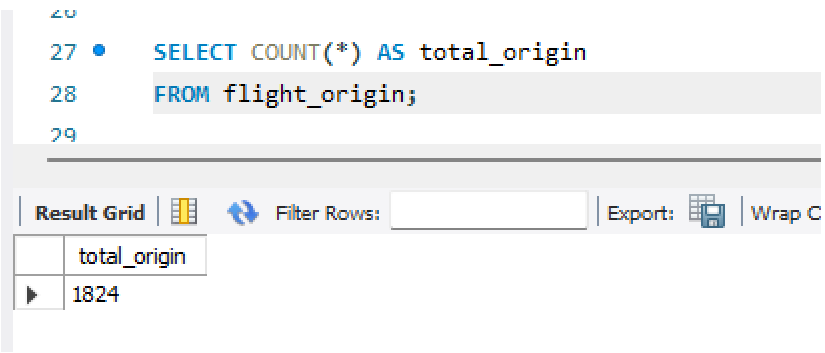
```sql
SELECT COUNT(*) AS total_Flight
FROM FlightOption
```

total_Flight
1824



```sql
SELECT COUNT(*) AS total_destination
FROM flight_destination;
```

total_destination
1824



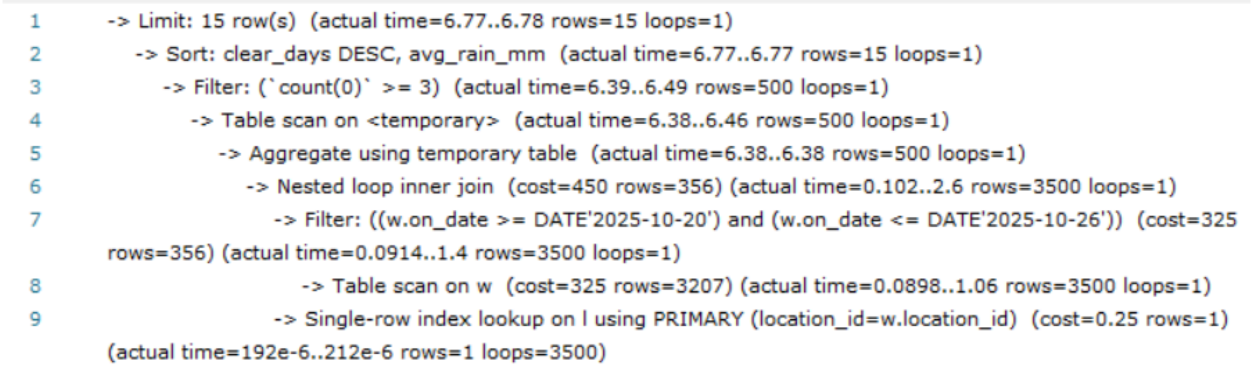
```sql
SELECT COUNT(*) AS total_origin
FROM flight_origin;
```

total_origin
1824

PART 2 : Indexing

Advanced Query #1:

Before adding any additional indexes, the cost of the query using the EXPLAIN ANALYZE command was 450, with the nested loop inner join with Table scan on w (weatherdaily) costing 325. This is with the existing index of idx_location_id in the Location Schema and idx_weatherdaily_location in the WeatherDaily Schema.

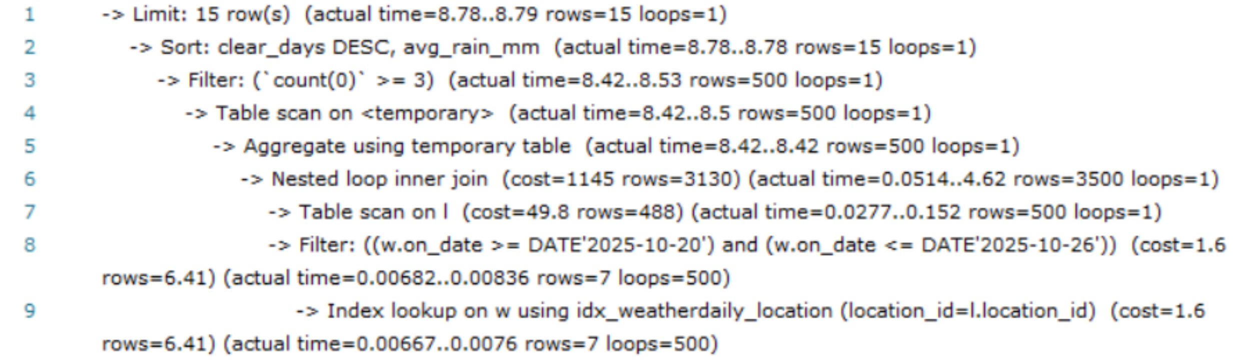


```
-> Limit: 15 row(s)  (actual time=6.77..6.78 rows=15 loops=1)
    -> Sort: clear_days DESC, avg_rain_mm  (actual time=6.77..6.77 rows=15 loops=1)
        -> Filter: (`count(0)` >= 3)  (actual time=6.39..6.49 rows=500 loops=1)
            -> Table scan on <temporary>  (actual time=6.38..6.46 rows=500 loops=1)
                -> Aggregate using temporary table  (actual time=6.38..6.38 rows=500 loops=1)
                    -> Nested loop inner join  (cost=450 rows=356) (actual time=0.102..2.6 rows=3500 loops=1)
                        -> Filter: ((w.on_date >= DATE'2025-10-20') and (w.on_date <= DATE'2025-10-26'))  (cost=325
rows=356) (actual time=0.0914..1.4 rows=3500 loops=1)
                            -> Table scan on w  (cost=325 rows=3207) (actual time=0.0898..1.06 rows=3500 loops=1)
                        -> Single-row index lookup on l using PRIMARY (location_id=w.location_id)  (cost=0.25 rows=1)
(actual time=192e-6..212e-6 rows=1 loops=3500)
```

First Index Test

CREATE INDEX idx_weatherdaily_date ON WeatherDaily(on_date);

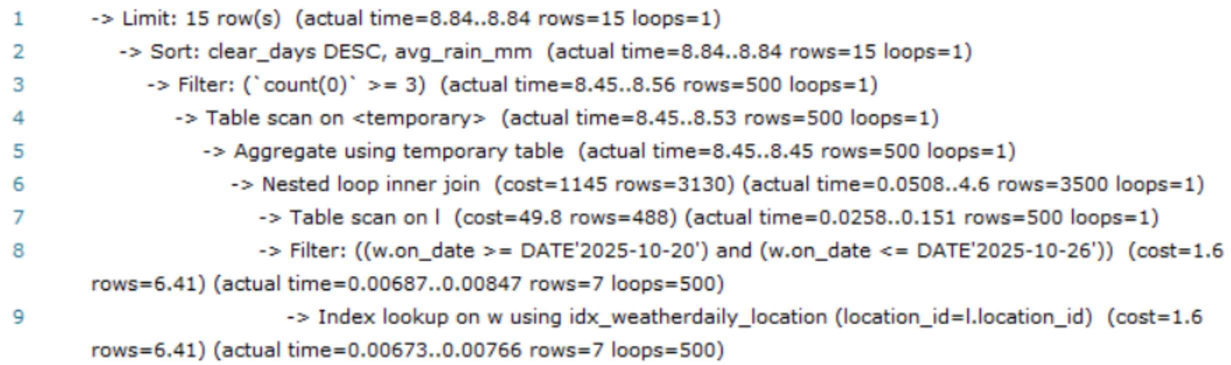
I first tried to create an index for the date, as it was in one of the WHERE statements. This however, increased the cost of the Nested loop inner join to 1145. This might be because the query chose a worse join order due to the fact that the index doesn't match the joining operation. Table scan went down to 49.5 rather than 325, although it was on a different table, meaning it was more of a hindrance as it increased the cost of the full query due to more index scans on a different table.



```
-> Limit: 15 row(s)  (actual time=8.78..8.79 rows=15 loops=1)
    -> Sort: clear_days DESC, avg_rain_mm  (actual time=8.78..8.78 rows=15 loops=1)
        -> Filter: (`count(0)` >= 3)  (actual time=8.42..8.53 rows=500 loops=1)
            -> Table scan on <temporary>  (actual time=8.42..8.5 rows=500 loops=1)
                -> Aggregate using temporary table  (actual time=8.42..8.42 rows=500 loops=1)
                    -> Nested loop inner join  (cost=1145 rows=3130) (actual time=0.0514..4.62 rows=3500 loops=1)
                        -> Table scan on l  (cost=49.8 rows=488) (actual time=0.0277..0.152 rows=500 loops=1)
                        -> Filter: ((w.on_date >= DATE'2025-10-20') and (w.on_date <= DATE'2025-10-26'))  (cost=1.6
rows=6.41) (actual time=0.00682..0.00836 rows=7 loops=500)
                            -> Index lookup on w using idx_weatherdaily_location (location_id=l.location_id)  (cost=1.6
rows=6.41) (actual time=0.00667..0.0076 rows=7 loops=500)
```

Second Index Test

CREATE INDEX idx_weatherdaily_date_loc ON WeatherDaily(on_date, location_id);

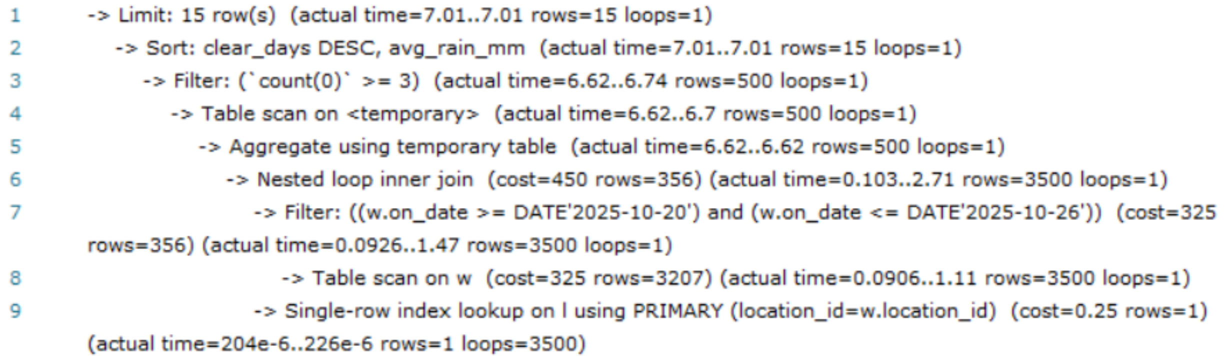


```
-> Limit: 15 row(s)  (actual time=8.84..8.84 rows=15 loops=1)
    -> Sort: clear_days DESC, avg_rain_mm  (actual time=8.84..8.84 rows=15 loops=1)
        -> Filter: (`count(0)` >= 3)  (actual time=8.45..8.56 rows=500 loops=1)
            -> Table scan on <temporary>  (actual time=8.45..8.53 rows=500 loops=1)
                -> Aggregate using temporary table  (actual time=8.45..8.45 rows=500 loops=1)
                    -> Nested loop inner join  (cost=1145 rows=3130) (actual time=0.0508..4.6 rows=3500 loops=1)
                        -> Table scan on l  (cost=49.8 rows=488) (actual time=0.0258..0.151 rows=500 loops=1)
                        -> Filter: ((w.on_date >= DATE'2025-10-20') and (w.on_date <= DATE'2025-10-26'))  (cost=1.6
rows=6.41) (actual time=0.00687..0.00847 rows=7 loops=500)
                            -> Index lookup on w using idx_weatherdaily_location (location_id=l.location_id)  (cost=1.6
rows=6.41) (actual time=0.00673..0.00766 rows=7 loops=500)
```

This indexing strategy also increased the cost to 1145. This indexing strategy is similar to the one I did above, where the date is indexed. Because the tables aren't joined by the date, the indexing might hinder performance, which is what is seen here. Similar to the first index test, the table scan went down to 49.5 rather than 325, although it was on a different table, meaning it was more of a hindrance as it increased the cost of the full query due to more index scans on a different table.

Third Index Test

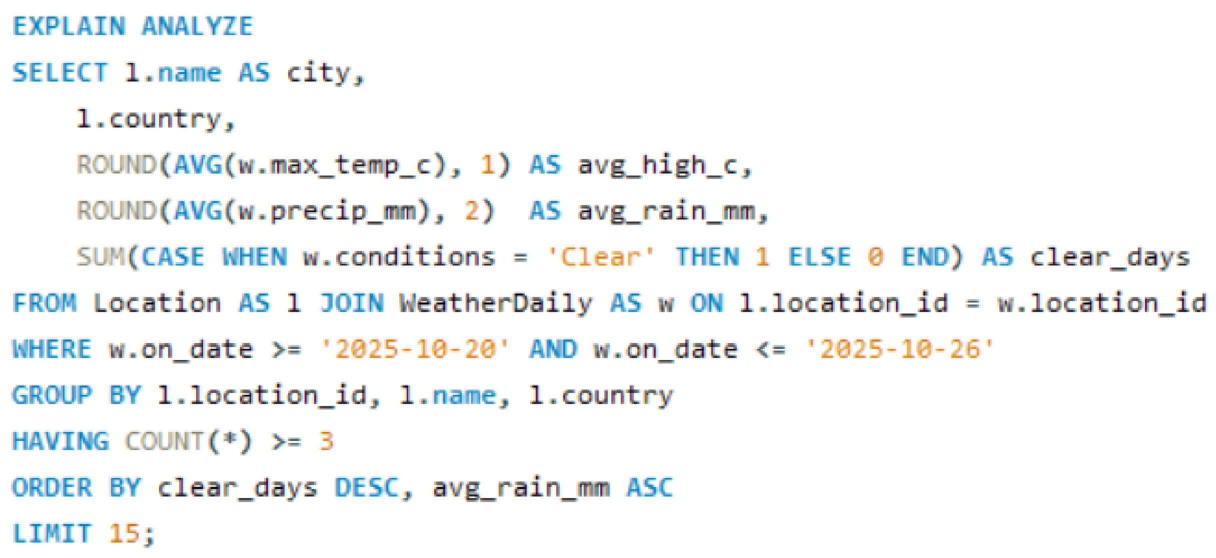
CREATE INDEX idx_weatherdaily_loc_date ON WeatherDaily(location_id, on_date);



```
-> Limit: 15 row(s)  (actual time=7.01..7.01 rows=15 loops=1)
    -> Sort: clear_days DESC, avg_rain_mm  (actual time=7.01..7.01 rows=15 loops=1)
        -> Filter: (`count(0)` >= 3)  (actual time=6.62..6.74 rows=500 loops=1)
            -> Table scan on <temporary>  (actual time=6.62..6.7 rows=500 loops=1)
                -> Aggregate using temporary table  (actual time=6.62..6.62 rows=500 loops=1)
                    -> Nested loop inner join  (cost=450 rows=356) (actual time=0.103..2.71 rows=3500 loops=1)
                        -> Filter: ((w.on_date >= DATE'2025-10-20') and (w.on_date <= DATE'2025-10-26'))  (cost=325
rows=356) (actual time=0.0926..1.47 rows=3500 loops=1)
                            -> Table scan on w  (cost=325 rows=3207) (actual time=0.0906..1.11 rows=3500 loops=1)
                        -> Single-row index lookup on l using PRIMARY (location_id=w.location_id)  (cost=0.25 rows=1)
(actual time=204e-6..226e-6 rows=1 loops=3500)
```

This variant of the index is similar to the second variant, but it leads with the join key and puts the date second. The indexing strategy has the same cost as the baseline, indicating that it isn't really better than what we are already doing (450). This shows that indexing on the joining column/variable can be much more beneficial than other indexes, as we didn't end up changing the joining strategy to one that was worse.

Overall, adding additional indexes to help with this query did not help with the performance cost of it. The existing baseline of indexes, which are idx_location_id in the Location Schema and idx_weatherdaily_location in the WeatherDaily schema, seems to be the best way of indexing for this query. I didn't index l.name or l.country because they are already dependent on the location id.



```
EXPLAIN ANALYZE
SELECT l.name AS city,
    l.country,
    ROUND(AVG(w.max_temp_c), 1) AS avg_high_c,
    ROUND(AVG(w.precip_mm), 2)  AS avg_rain_mm,
    SUM(CASE WHEN w.conditions = 'Clear' THEN 1 ELSE 0 END) AS clear_days
FROM Location AS l JOIN WeatherDaily AS w ON l.location_id = w.location_id
WHERE w.on_date >= '2025-10-20' AND w.on_date <= '2025-10-26'
GROUP BY l.location_id, l.name, l.country
HAVING COUNT(*) >= 3
ORDER BY clear_days DESC, avg_rain_mm ASC
LIMIT 15;
```

Advanced Query #2:

Before adding any indexes, we ran the query using the EXPLAIN ANALYZE command to measure its performance. The output showed that the main join between the Location and WeatherDaily tables had a cost of 1243, while the correlated subquery calculating each country's average temperature had a cost of 159. The execution plan also indicated a full table scan on the Location table (aliased as l2) within the subquery, which is repeated for each outer row, suggesting that indexing on Location(country) could significantly reduce the overall query cost.

First Index Test: CREATE INDEX idx_location_country ON Location(country);

After creating an index on Location(country), we reran EXPLAIN ANALYZE on the same query. The correlated subquery's cost dropped from 159 to 18.7, because MySQL was now able to use idx_location_country to look up rows by country instead of doing a full table scan on Location for each outer row. The main join part of the plan stayed at cost = 1243, so the improvement came specifically from indexing the attribute used in the subquery's WHERE l2.country = l.country condition.

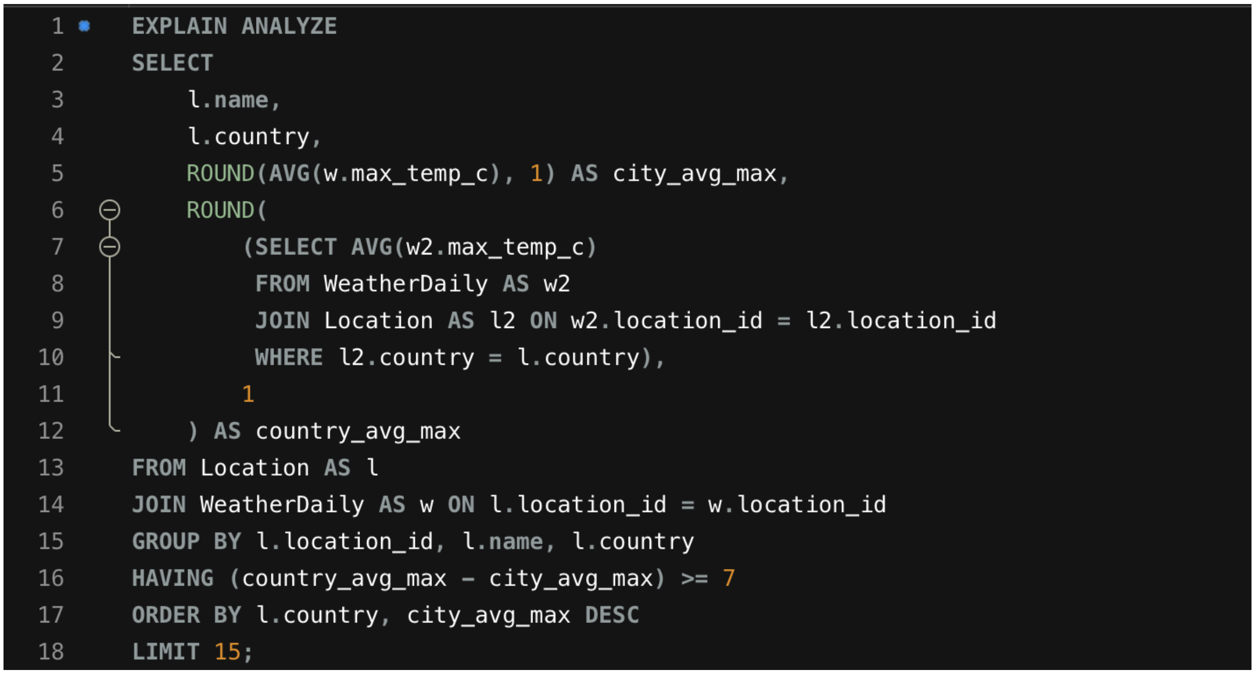
Second Index Test: CREATE INDEX idx_weatherdaily_location ON WeatherDaily(location_id);

Next, we created an index on WeatherDaily(location_id) to improve the performance of the join between Location and WeatherDaily. After rerunning EXPLAIN ANALYZE, the main join cost decreased slightly from 1243 to 1145, indicating a modest efficiency gain. The plan also showed that MySQL was now explicitly using idx_weatherdaily_location for both the main query and the subquery joins, replacing the previous generic lookup. This confirms that indexing join attributes can still provide measurable improvements, though the performance boost was smaller compared to the earlier Location(country) index

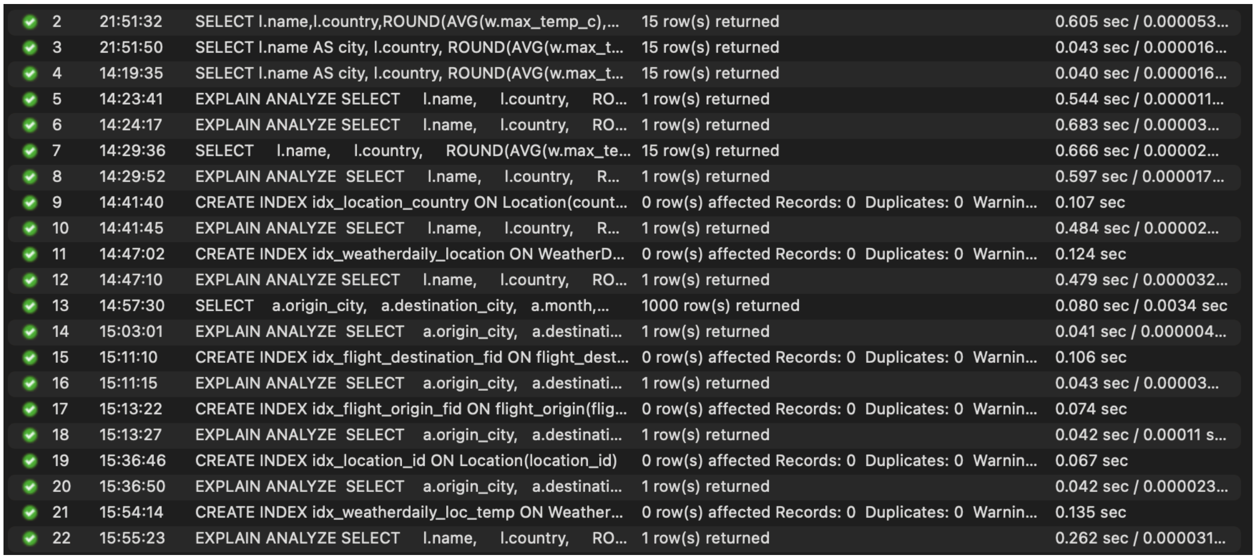
Third Index Test: CREATE INDEX idx_weatherdaily_loc_temp ON WeatherDaily(location_id, max_temp_c);

After creating a composite index on WeatherDaily(location_id, max_temp_c), we reran EXPLAIN ANALYZE on the same query. The main join cost dropped further from 1145 to 795, and the subquery cost decreased from 17.2 to 13.2, indicating a clear performance improvement. The plan now showed covering index lookups on both w and w2 using idx_weatherdaily_loc_temp, meaning MySQL could retrieve both location_id and max_temp_c directly from the index without referring back to the table. This reduced the number of lookups and improved aggregation efficiency.

Overall, indexing had a strong impact on query performance. The Location(country) index produced the largest improvement, reducing the subquery cost from 159 to 18.7, while WeatherDaily(location_id) and WeatherDaily(location_id, max_temp_c) provided additional efficiency gains, lowering the main join cost from 1243 to 795.



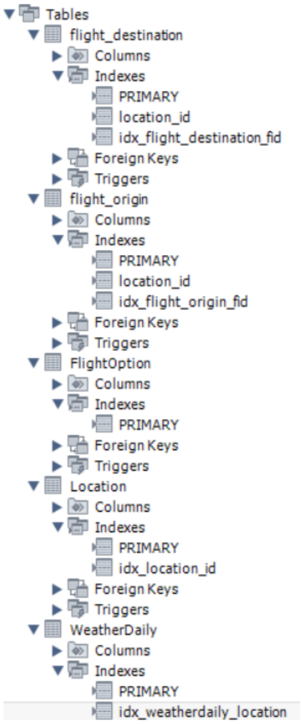
```sql
EXPLAIN ANALYZE
SELECT
    l.name,
    l.country,
    ROUND(AVG(w.max_temp_c), 1) AS city_avg_max,
    ROUND(
        (SELECT AVG(w2.max_temp_c)
         FROM WeatherDaily AS w2
         JOIN Location AS l2 ON w2.location_id = l2.location_id
         WHERE l2.country = l.country),
        1
    ) AS country_avg_max
FROM Location AS l
JOIN WeatherDaily AS w ON l.location_id = w.location_id
GROUP BY l.location_id, l.name, l.country
HAVING (country_avg_max - city_avg_max) >= 7
ORDER BY l.country, city_avg_max DESC
LIMIT 15;
```



```
✓ 2   21:51:32  SELECT l.name,l.country,ROUND(AVG(w.max_temp_c),...        15 row(s) returned                                           0.605 sec / 0.000053...
✓ 3   21:51:50  SELECT l.name AS city, l.country, ROUND(AVG(w.max_t...     15 row(s) returned                                           0.043 sec / 0.000016...
✓ 4   14:19:35  SELECT l.name AS city, l.country, ROUND(AVG(w.max_t...     15 row(s) returned                                           0.040 sec / 0.000016...
✓ 5   14:23:41  EXPLAIN ANALYZE SELECT    l.name,    l.country,    RO...   1 row(s) returned                                            0.544 sec / 0.000011...
✓ 6   14:24:17  EXPLAIN ANALYZE SELECT    l.name,    l.country,    RO...   1 row(s) returned                                            0.683 sec / 0.00003...
✓ 7   14:29:36  SELECT    l.name,    l.country,    ROUND(AVG(w.max_te...   15 row(s) returned                                           0.666 sec / 0.00002...
✓ 8   14:29:52  EXPLAIN ANALYZE  SELECT    l.name,    l.country,    R...   1 row(s) returned                                            0.597 sec / 0.000017...
✓ 9   14:41:40  CREATE INDEX idx_location_country ON Location(count...     0 row(s) affected Records: 0  Duplicates: 0  Warnin...       0.107 sec
✓ 10  14:41:45  EXPLAIN ANALYZE  SELECT    l.name,    l.country,    R...   1 row(s) returned                                            0.484 sec / 0.00002...
✓ 11  14:47:02  CREATE INDEX idx_weatherdaily_location ON WeatherD...      0 row(s) affected Records: 0  Duplicates: 0  Warnin...       0.124 sec
✓ 12  14:47:10  EXPLAIN ANALYZE SELECT    l.name,    l.country,    RO...   1 row(s) returned                                            0.479 sec / 0.000032...
✓ 13  14:57:30  SELECT   a.origin_city,   a.destination_city,   a.month,...  1000 row(s) returned                                       0.080 sec / 0.0034 sec
✓ 14  15:03:01  EXPLAIN ANALYZE  SELECT   a.origin_city,   a.destinati...  1 row(s) returned                                            0.041 sec / 0.000004...
✓ 15  15:11:10  CREATE INDEX idx_flight_destination_fid ON flight_dest...  0 row(s) affected Records: 0  Duplicates: 0  Warnin...       0.106 sec
✓ 16  15:11:15  EXPLAIN ANALYZE  SELECT   a.origin_city,   a.destinati...  1 row(s) returned                                            0.043 sec / 0.00003...
✓ 17  15:13:22  CREATE INDEX idx_flight_origin_fid ON flight_origin(flig...  0 row(s) affected Records: 0  Duplicates: 0  Warnin...     0.074 sec
✓ 18  15:13:27  EXPLAIN ANALYZE  SELECT   a.origin_city,   a.destinati...  1 row(s) returned                                            0.042 sec / 0.00011 s...
✓ 19  15:36:46  CREATE INDEX idx_location_id ON Location(location_id)      0 row(s) affected Records: 0  Duplicates: 0  Warnin...       0.067 sec
✓ 20  15:36:50  EXPLAIN ANALYZE  SELECT   a.origin_city,   a.destinati...  1 row(s) returned                                            0.042 sec / 0.000023...
✓ 21  15:54:14  CREATE INDEX idx_weatherdaily_loc_temp ON Weather...       0 row(s) affected Records: 0  Duplicates: 0  Warnin...       0.135 sec
✓ 22  15:55:23  EXPLAIN ANALYZE SELECT    l.name,    l.country,    RO...   1 row(s) returned                                            0.262 sec / 0.000031...
```

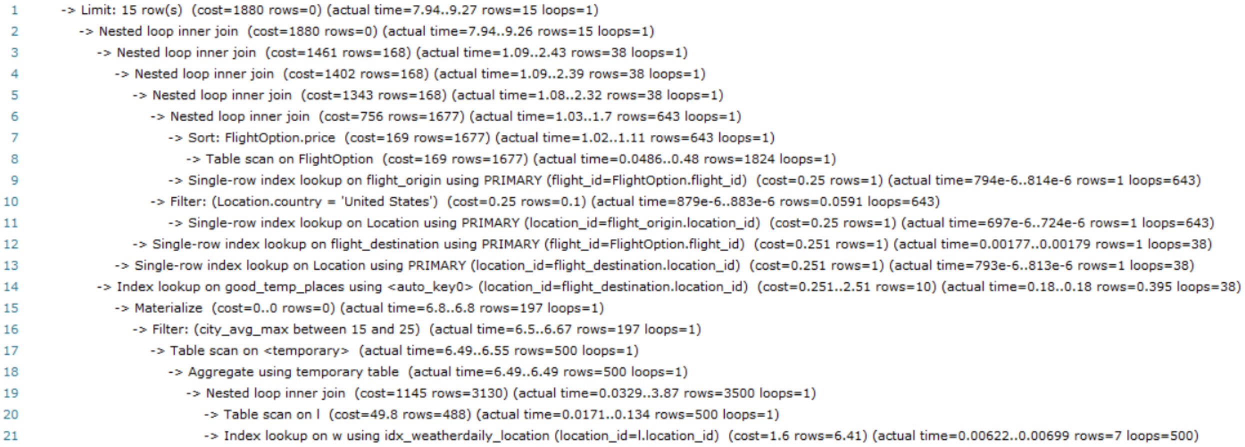
Advanced Query #3:

Existing indexes:



```
Tables
    flight_destination
        Columns
        Indexes
            PRIMARY
            location_id
            idx_flight_destination_fid
        Foreign Keys
        Triggers
    flight_origin
        Columns
        Indexes
            PRIMARY
            location_id
            idx_flight_origin_fid
        Foreign Keys
        Triggers
    FlightOption
        Columns
        Indexes
            PRIMARY
        Foreign Keys
        Triggers
    Location
        Columns
        Indexes
            PRIMARY
            idx_location_id
        Foreign Keys
        Triggers
    WeatherDaily
        Columns
        Indexes
            PRIMARY
            idx_weatherdaily_location
```

Baseline:



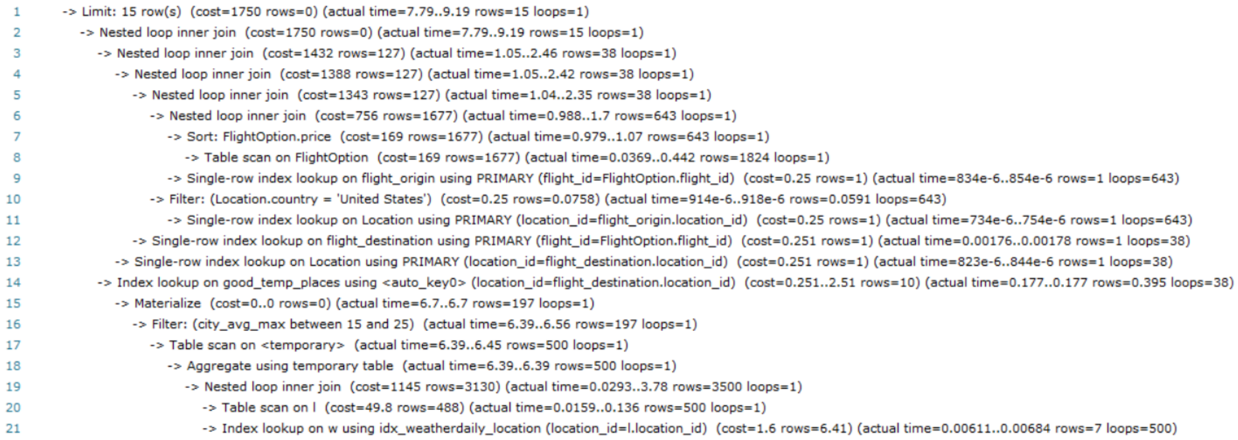
```
-> Limit: 15 row(s)  (cost=1880 rows=0) (actual time=7.94..9.27 rows=15 loops=1)
    -> Nested loop inner join  (cost=1880 rows=0) (actual time=7.94..9.26 rows=15 loops=1)
        -> Nested loop inner join  (cost=1461 rows=168) (actual time=1.09..2.43 rows=38 loops=1)
            -> Nested loop inner join  (cost=1402 rows=168) (actual time=1.09..2.39 rows=38 loops=1)
                -> Nested loop inner join  (cost=1343 rows=168) (actual time=1.08..2.32 rows=38 loops=1)
                    -> Nested loop inner join  (cost=756 rows=1677) (actual time=1.03..1.7 rows=643 loops=1)
                        -> Sort: FlightOption.price  (cost=169 rows=1677) (actual time=1.02..1.11 rows=643 loops=1)
                            -> Table scan on FlightOption  (cost=169 rows=1677) (actual time=0.0486..0.48 rows=1824 loops=1)
                        -> Single-row index lookup on flight_origin using PRIMARY (flight_id=FlightOption.flight_id)  (cost=0.25 rows=1) (actual time=794e-6..814e-6 rows=1 loops=643)
                    -> Filter: (Location.country = 'United States')  (cost=0.25 rows=0.1) (actual time=879e-6..883e-6 rows=0.0591 loops=643)
                        -> Single-row index lookup on Location using PRIMARY (location_id=flight_origin.location_id)  (cost=0.25 rows=1) (actual time=697e-6..724e-6 rows=1 loops=643)
                -> Single-row index lookup on flight_destination using PRIMARY (flight_id=FlightOption.flight_id)  (cost=0.251 rows=1) (actual time=0.00177..0.00179 rows=1 loops=38)
            -> Single-row index lookup on Location using PRIMARY (location_id=flight_destination.location_id)  (cost=0.251 rows=1) (actual time=793e-6..813e-6 rows=1 loops=38)
        -> Index lookup on good_temp_places using <auto_key0> (location_id=flight_destination.location_id)  (cost=0.251..2.51 rows=10) (actual time=0.18..0.18 rows=0.395 loops=38)
            -> Materialize  (cost=0..0 rows=0) (actual time=6.8..6.8 rows=197 loops=1)
                -> Filter: (city_avg_max between 15 and 25)  (actual time=6.5..6.67 rows=197 loops=1)
                    -> Table scan on <temporary>  (actual time=6.49..6.55 rows=500 loops=1)
                        -> Aggregate using temporary table  (actual time=6.49..6.49 rows=500 loops=1)
                            -> Nested loop inner join  (cost=1145 rows=3130) (actual time=0.0329..3.87 rows=3500 loops=1)
                                -> Table scan on l  (cost=49.8 rows=488) (actual time=0.0171..0.134 rows=500 loops=1)
                                -> Index lookup on w using idx_weatherdaily_location (location_id=l.location_id)  (cost=1.6 rows=6.41) (actual time=0.00622..0.00699 rows=7 loops=500)
```

Cost = 1880

*index tests are cumulative

First Index Test

CREATE INDEX idx_location_country ON Location(country);

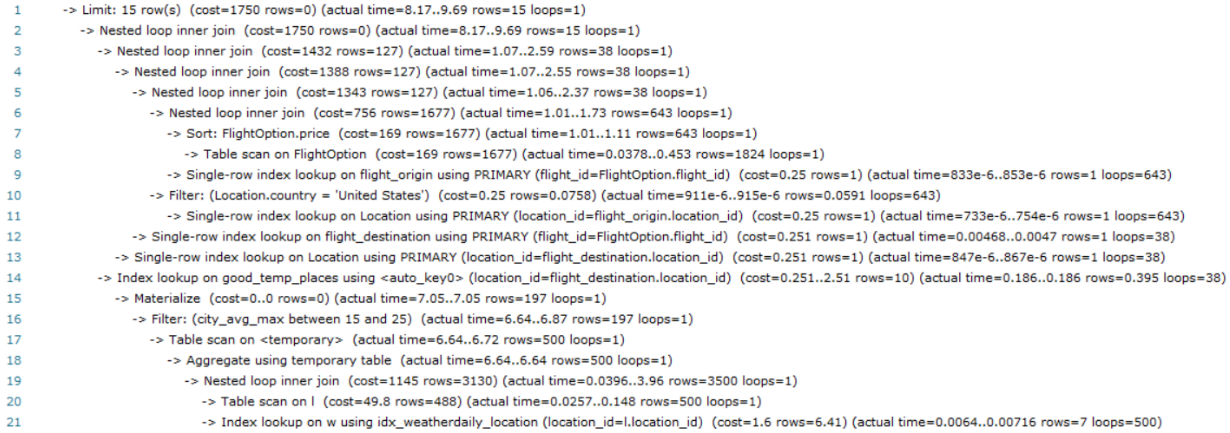


```
-> Limit: 15 row(s)  (cost=1750 rows=0) (actual time=7.79..9.19 rows=15 loops=1)
    -> Nested loop inner join  (cost=1750 rows=0) (actual time=7.79..9.19 rows=15 loops=1)
        -> Nested loop inner join  (cost=1432 rows=127) (actual time=1.05..2.46 rows=38 loops=1)
            -> Nested loop inner join  (cost=1388 rows=127) (actual time=1.05..2.42 rows=38 loops=1)
                -> Nested loop inner join  (cost=1343 rows=127) (actual time=1.04..2.35 rows=38 loops=1)
                    -> Nested loop inner join  (cost=756 rows=1677) (actual time=0.988..1.7 rows=643 loops=1)
                        -> Sort: FlightOption.price  (cost=169 rows=1677) (actual time=0.979..1.07 rows=643 loops=1)
                            -> Table scan on FlightOption  (cost=169 rows=1677) (actual time=0.0369..0.442 rows=1824 loops=1)
                        -> Single-row index lookup on flight_origin using PRIMARY (flight_id=FlightOption.flight_id)  (cost=0.25 rows=1) (actual time=834e-6..854e-6 rows=1 loops=643)
                    -> Filter: (Location.country = 'United States')  (cost=0.25 rows=0.0758) (actual time=914e-6..918e-6 rows=0.0591 loops=643)
                        -> Single-row index lookup on Location using PRIMARY (location_id=flight_origin.location_id)  (cost=0.25 rows=1) (actual time=734e-6..754e-6 rows=1 loops=643)
                -> Single-row index lookup on flight_destination using PRIMARY (flight_id=FlightOption.flight_id)  (cost=0.251 rows=1) (actual time=0.00176..0.00178 rows=1 loops=38)
            -> Single-row index lookup on Location using PRIMARY (location_id=flight_destination.location_id)  (cost=0.251 rows=1) (actual time=823e-6..844e-6 rows=1 loops=38)
        -> Index lookup on good_temp_places using <auto_key0> (location_id=flight_destination.location_id)  (cost=0.251..2.51 rows=10) (actual time=0.177..0.177 rows=0.395 loops=38)
            -> Materialize  (cost=0..0 rows=0) (actual time=6.7..6.7 rows=197 loops=1)
                -> Filter: (city_avg_max between 15 and 25)  (actual time=6.39..6.56 rows=197 loops=1)
                    -> Table scan on <temporary>  (actual time=6.39..6.45 rows=500 loops=1)
                        -> Aggregate using temporary table  (actual time=6.39..6.39 rows=500 loops=1)
                            -> Nested loop inner join  (cost=1145 rows=3130) (actual time=0.0293..3.78 rows=3500 loops=1)
                                -> Table scan on l  (cost=49.8 rows=488) (actual time=0.0159..0.136 rows=500 loops=1)
                                -> Index lookup on w using idx_weatherdaily_location (location_id=l.location_id)  (cost=1.6 rows=6.41) (actual time=0.00611..0.00684 rows=7 loops=500)
```

The cost goes down to 1750 when compared to the baseline of 1880. This can be caused by the subquery reading the Location by the country index, making it so fewer rows have to be examined. This can reduce the number of unnecessary joins to the flight origin. Cutting down origin locations shrinks the downstream join operation and slightly reduces the size of the candidate set that is sorted by price.

Second Index Test

CREATE INDEX idx_flightoption_price ON FlightOption(price);

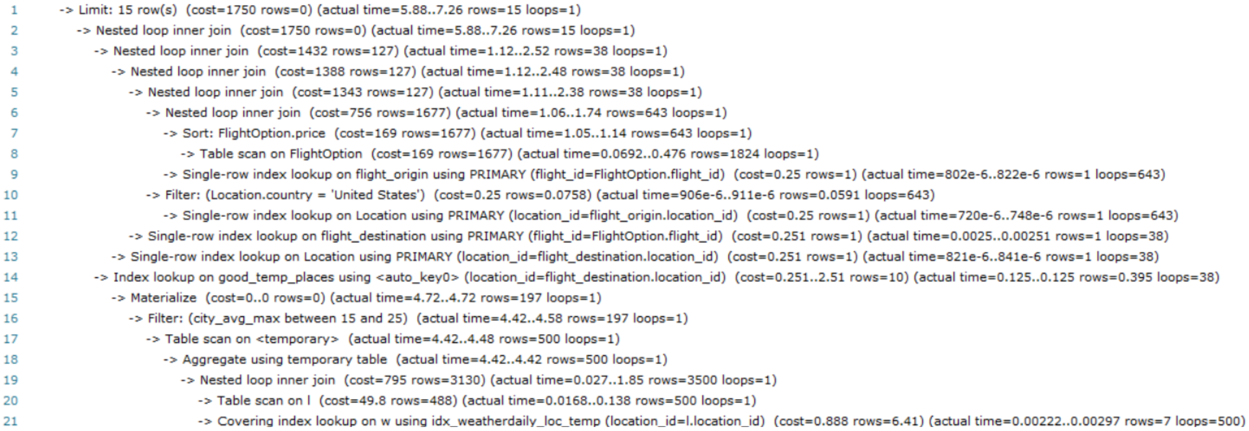


```
-> Limit: 15 row(s)  (cost=1750 rows=0) (actual time=8.17..9.69 rows=15 loops=1)
    -> Nested loop inner join  (cost=1750 rows=0) (actual time=8.17..9.69 rows=15 loops=1)
        -> Nested loop inner join  (cost=1432 rows=127) (actual time=1.07..2.59 rows=38 loops=1)
            -> Nested loop inner join  (cost=1388 rows=127) (actual time=1.07..2.55 rows=38 loops=1)
                -> Nested loop inner join  (cost=1343 rows=127) (actual time=1.06..2.37 rows=38 loops=1)
                    -> Nested loop inner join  (cost=756 rows=1677) (actual time=1.01..1.73 rows=643 loops=1)
                        -> Sort: FlightOption.price  (cost=169 rows=1677) (actual time=1.01..1.11 rows=643 loops=1)
                            -> Table scan on FlightOption  (cost=169 rows=1677) (actual time=0.0378..0.453 rows=1824 loops=1)
                        -> Single-row index lookup on flight_origin using PRIMARY (flight_id=FlightOption.flight_id)  (cost=0.25 rows=1) (actual time=833e-6..853e-6 rows=1 loops=643)
                    -> Filter: (Location.country = 'United States')  (cost=0.25 rows=0.0758) (actual time=911e-6..915e-6 rows=0.0591 loops=643)
                        -> Single-row index lookup on Location using PRIMARY (location_id=flight_origin.location_id)  (cost=0.25 rows=1) (actual time=733e-6..754e-6 rows=1 loops=643)
                -> Single-row index lookup on flight_destination using PRIMARY (flight_id=FlightOption.flight_id)  (cost=0.251 rows=1) (actual time=0.00468..0.0047 rows=1 loops=38)
            -> Single-row index lookup on Location using PRIMARY (location_id=flight_destination.location_id)  (cost=0.251 rows=1) (actual time=847e-6..867e-6 rows=1 loops=38)
        -> Index lookup on good_temp_places using <auto_key0> (location_id=flight_destination.location_id)  (cost=0.251..2.51 rows=10) (actual time=0.186..0.186 rows=0.395 loops=38)
            -> Materialize  (cost=0..0 rows=0) (actual time=7.05..7.05 rows=197 loops=1)
                -> Filter: (city_avg_max between 15 and 25)  (actual time=6.64..6.87 rows=197 loops=1)
                    -> Table scan on <temporary>  (actual time=6.64..6.72 rows=500 loops=1)
                        -> Aggregate using temporary table  (actual time=6.64..6.64 rows=500 loops=1)
                            -> Nested loop inner join  (cost=1145 rows=3130) (actual time=0.0396..3.96 rows=3500 loops=1)
                                -> Table scan on l  (cost=49.8 rows=488) (actual time=0.0257..0.148 rows=500 loops=1)
                                -> Index lookup on w using idx_weatherdaily_location (location_id=l.location_id)  (cost=1.6 rows=6.41) (actual time=0.0064..0.00716 rows=7 loops=500)
```

This index seems to show no improvement over the last index + the baseline. The cost stays at 1750. While the index might help with the filter query and getting the flights with th e lowest prices, it doesn’t remove any of the steps needed, such as the joins and filters on the other tables. The filtering steps come after the joins and the derived subquery. When the qualifying set is determined only after those steps, the index order cannot be used to remove entries to make it quicker. It still needs to build (or partially build) the candidate set before picking the cheapest 15.

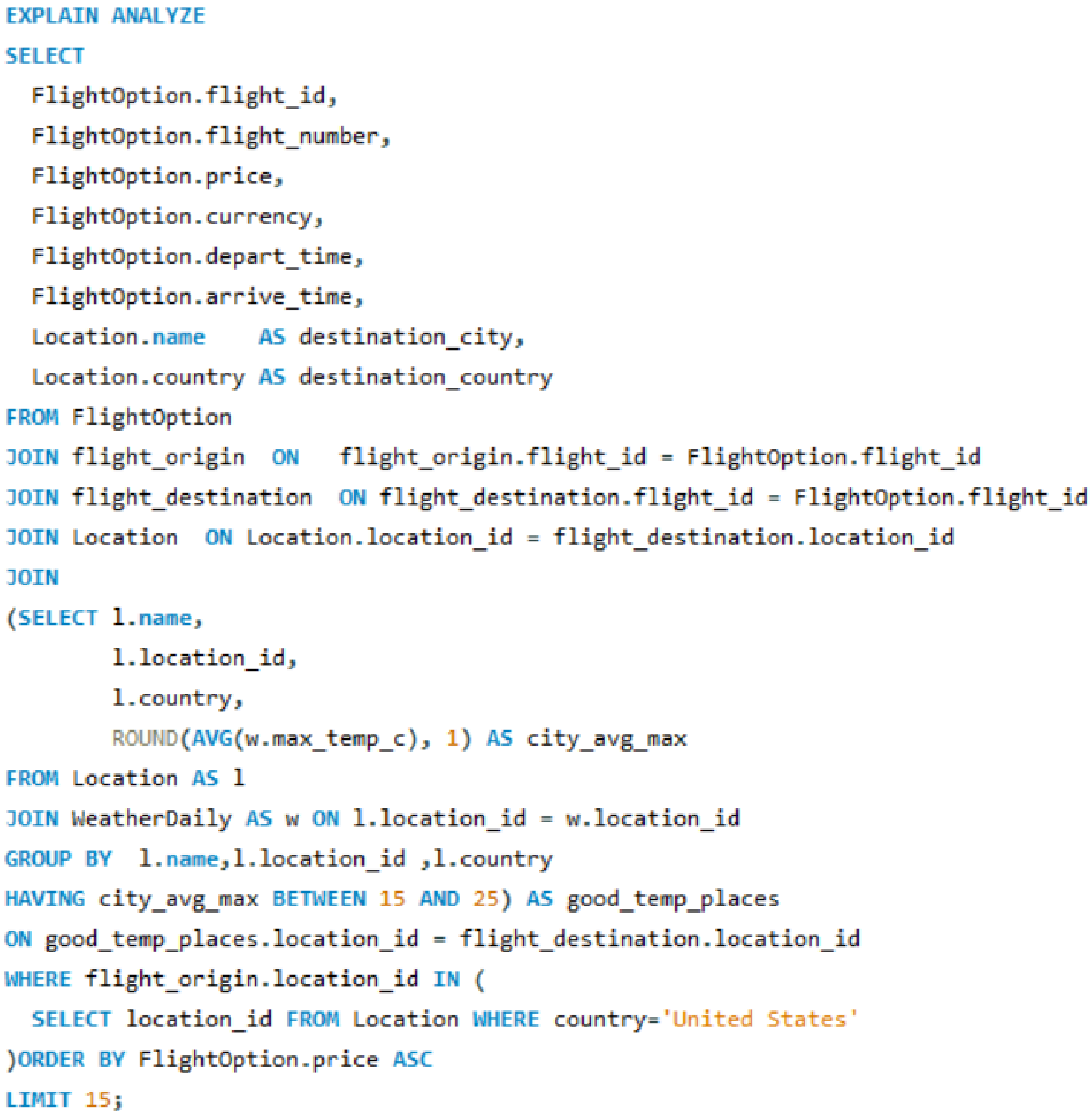
Third Index Test

CREATE INDEX idx_weatherdaily_loc_temp ON WeatherDaily(location_id, max_temp_c);



```
-> Limit: 15 row(s)  (cost=1750 rows=0) (actual time=5.88..7.26 rows=15 loops=1)
    -> Nested loop inner join  (cost=1750 rows=0) (actual time=5.88..7.26 rows=15 loops=1)
        -> Nested loop inner join  (cost=1432 rows=127) (actual time=1.12..2.52 rows=38 loops=1)
            -> Nested loop inner join  (cost=1388 rows=127) (actual time=1.12..2.48 rows=38 loops=1)
                -> Nested loop inner join  (cost=1343 rows=127) (actual time=1.11..2.38 rows=38 loops=1)
                    -> Nested loop inner join  (cost=756 rows=1677) (actual time=1.06..1.74 rows=643 loops=1)
                        -> Sort: FlightOption.price  (cost=169 rows=1677) (actual time=1.05..1.14 rows=643 loops=1)
                            -> Table scan on FlightOption  (cost=169 rows=1677) (actual time=0.0692..0.476 rows=1824 loops=1)
                        -> Single-row index lookup on flight_origin using PRIMARY (flight_id=FlightOption.flight_id)  (cost=0.25 rows=1) (actual time=802e-6..822e-6 rows=1 loops=643)
                    -> Filter: (Location.country = 'United States')  (cost=0.25 rows=0.0758) (actual time=906e-6..911e-6 rows=0.0591 loops=643)
                        -> Single-row index lookup on Location using PRIMARY (location_id=flight_origin.location_id)  (cost=0.25 rows=1) (actual time=720e-6..748e-6 rows=1 loops=643)
                -> Single-row index lookup on flight_destination using PRIMARY (flight_id=FlightOption.flight_id)  (cost=0.251 rows=1) (actual time=0.0025..0.00251 rows=1 loops=38)
            -> Single-row index lookup on Location using PRIMARY (location_id=flight_destination.location_id)  (cost=0.251 rows=1) (actual time=821e-6..841e-6 rows=1 loops=38)
        -> Index lookup on good_temp_places using <auto_key0> (location_id=flight_destination.location_id)  (cost=0.251..2.51 rows=10) (actual time=0.125..0.125 rows=0.395 loops=38)
            -> Materialize  (cost=0..0 rows=0) (actual time=4.72..4.72 rows=197 loops=1)
                -> Filter: (city_avg_max between 15 and 25)  (actual time=4.42..4.58 rows=197 loops=1)
                    -> Table scan on <temporary>  (actual time=4.42..4.48 rows=500 loops=1)
                        -> Aggregate using temporary table  (actual time=4.42..4.42 rows=500 loops=1)
                            -> Nested loop inner join  (cost=795 rows=3130) (actual time=0.027..1.85 rows=3500 loops=1)
                                -> Table scan on l  (cost=49.8 rows=488) (actual time=0.0168..0.138 rows=500 loops=1)
                                -> Covering index lookup on w using idx_weatherdaily_loc_temp (location_id=l.location_id)  (cost=0.888 rows=6.41) (actual time=0.00222..0.00297 rows=7 loops=500)
```

The index did not provide an improvement over the previous indexes and the baseline (still at cost = 1750). This might be because the aggregation comes before the HAVING statement. The subquery computes AVG(max_temp_c) per location and only after that applies HAVING. An index can't pre-filter on an aggregate it hasn't computed yet, so rows must still be read and grouped, meaning no work/cost is being removed or rendered redundant from this index. This shows that grouping shape limits index utility as our subquery groups by l.name, l.location_id, and l.country, which don’t match the ones in weatherdaily.



```
EXPLAIN ANALYZE
SELECT
  FlightOption.flight_id,
  FlightOption.flight_number,
  FlightOption.price,
  FlightOption.currency,
  FlightOption.depart_time,
  FlightOption.arrive_time,
  Location.name    AS destination_city,
  Location.country AS destination_country
FROM FlightOption
JOIN flight_origin  ON   flight_origin.flight_id = FlightOption.flight_id
JOIN flight_destination  ON flight_destination.flight_id = FlightOption.flight_id
JOIN Location  ON Location.location_id = flight_destination.location_id
JOIN
(SELECT l.name,
        l.location_id,
        l.country,
        ROUND(AVG(w.max_temp_c), 1) AS city_avg_max
FROM Location AS l
JOIN WeatherDaily AS w ON l.location_id = w.location_id
GROUP BY  l.name,l.location_id ,l.country
HAVING city_avg_max BETWEEN 15 AND 25) AS good_temp_places
ON good_temp_places.location_id = flight_destination.location_id
WHERE flight_origin.location_id IN (
  SELECT location_id FROM Location WHERE country='United States'
)ORDER BY FlightOption.price ASC
LIMIT 15;
```

Advanced Query #4:

Before adding any indexes, we ran the flight query using the EXPLAIN ANALYZE command to measure its performance. The output showed a series of nested loop joins with an overall cost of 2257, primarily between the FlightOption, flight_origin, and flight_destination tables. The plan indicated that the query began with a full table scan on the Location table (aliased as l2), followed by multiple index lookups on the flight_destination, FlightOption, and flight_origin tables. Since most later joins were already using primary-key lookups, the main source of inefficiency was the initial table scan and repeated lookups through the flight_destination join, suggesting that indexing the join attributes could help reduce the overall query cost.

First Index Test: CREATE INDEX idx_flight_destination_fid ON flight_destination(flight_id);

After creating an index on flight_destination(flight_id), we reran EXPLAIN ANALYZE on the same query to measure performance changes. The overall join cost remained at 2257, indicating that the new index did not significantly affect the optimizer's join strategy. The plan still began with a full table scan on the Location table (aliased as l2), followed by a covering index lookup on flight_destination using location_id. This suggests that MySQL continued to join through location_id rather than leveraging the new index on flight_id.

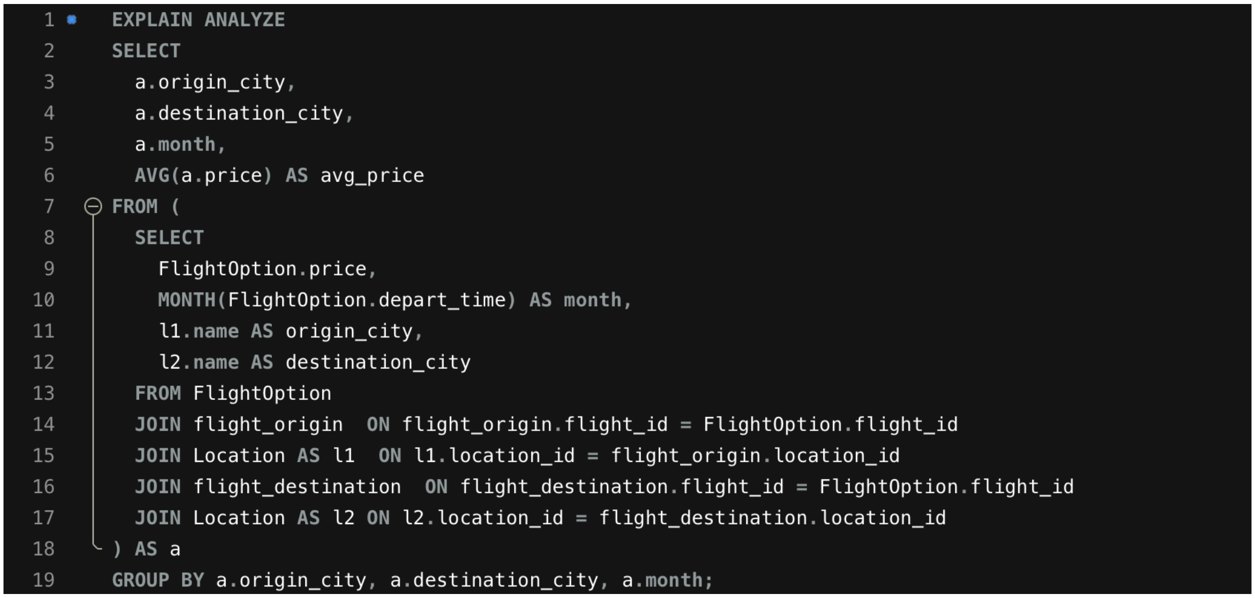
Second Index Test: CREATE INDEX idx_flight_origin_fid ON flight_origin(flight_id);

We then added an index on flight_origin(flight_id) and reran EXPLAIN ANALYZE. The overall join cost stayed at 2257, and the plan structure was identical to the previous one: the query still started with a full table scan on Location (l2), then used the existing covering index on flight_destination via location_id, followed by single-row lookups on FlightOption, flight_origin, and Location (l1). This shows that indexing flight_origin(flight_id) did not change the optimizer's chosen path.

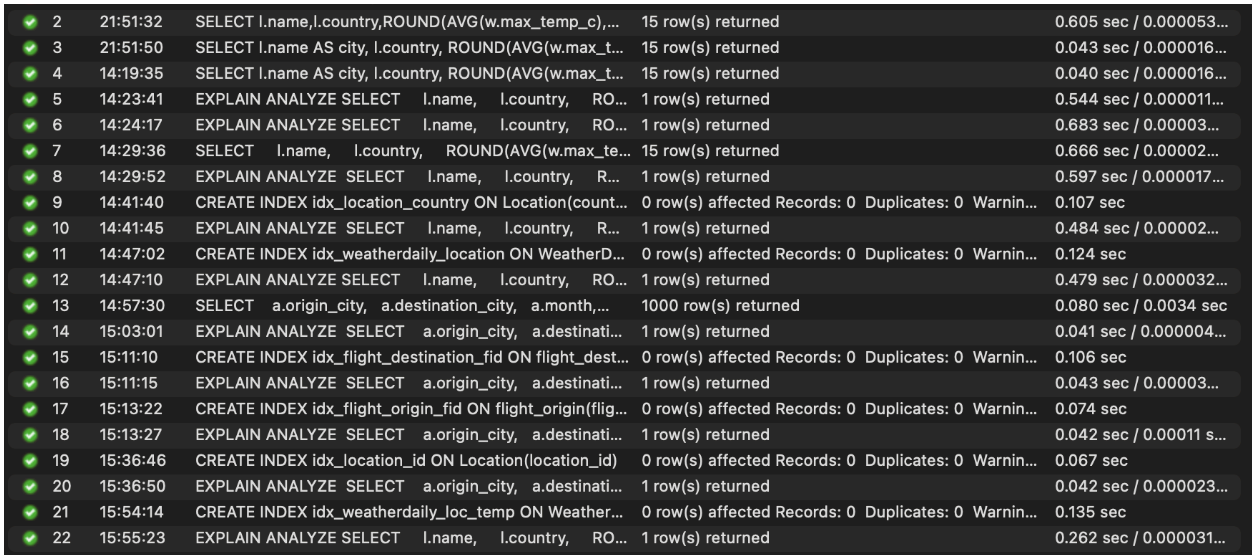
Third Index Test: CREATE INDEX idx_location_id ON Location(location_id);

Finally, we added an index on the Location table to target the part of the plan that was still doing a full table scan (l2). After creating the index and rerunning EXPLAIN ANALYZE, the overall join cost remained at 2257 and the plan continued to start with a table scan on l2, followed by the same sequence of index lookups on flight_destination, FlightOption, flight_origin, and Location (l1). This shows that, for this dataset, MySQL prefers a full scan of Location because the table is small and fully read anyway, so adding an index on the join column did not yield an improvement.

Overall, indexing had little effect on this query's performance. The total join cost remained constant at 2257 across all tests, showing that MySQL's optimizer was already using efficient primary-key lookups and that additional indexes on small or already well-indexed tables did not provide measurable improvements.



```sql
EXPLAIN ANALYZE
SELECT
  a.origin_city,
  a.destination_city,
  a.month,
  AVG(a.price) AS avg_price
FROM (
  SELECT
    FlightOption.price,
    MONTH(FlightOption.depart_time) AS month,
    l1.name AS origin_city,
    l2.name AS destination_city
  FROM FlightOption
  JOIN flight_origin  ON flight_origin.flight_id = FlightOption.flight_id
  JOIN Location AS l1  ON l1.location_id = flight_origin.location_id
  JOIN flight_destination  ON flight_destination.flight_id = FlightOption.flight_id
  JOIN Location AS l2 ON l2.location_id = flight_destination.location_id
) AS a
GROUP BY a.origin_city, a.destination_city, a.month;
```



```
✓  2   21:51:32  SELECT l.name,l.country,ROUND(AVG(w.max_temp_c),...   15 row(s) returned                                      0.605 sec / 0.000053...
✓  3   21:51:50  SELECT l.name AS city, l.country, ROUND(AVG(w.max_t...  15 row(s) returned                                      0.043 sec / 0.000016...
✓  4   14:19:35  SELECT l.name AS city, l.country, ROUND(AVG(w.max_t...  15 row(s) returned                                      0.040 sec / 0.000016...
✓  5   14:23:41  EXPLAIN ANALYZE SELECT   l.name,   l.country,   RO...    1 row(s) returned                                       0.544 sec / 0.000011...
✓  6   14:24:17  EXPLAIN ANALYZE SELECT   l.name,   l.country,   RO...    1 row(s) returned                                       0.683 sec / 0.00003...
✓  7   14:29:36  SELECT   l.name,   l.country,   ROUND(AVG(w.max_te...   15 row(s) returned                                      0.666 sec / 0.00002...
✓  8   14:29:52  EXPLAIN ANALYZE  SELECT   l.name,   l.country,   R...   1 row(s) returned                                       0.597 sec / 0.000017...
✓  9   14:41:40  CREATE INDEX idx_location_country ON Location(count...  0 row(s) affected Records: 0  Duplicates: 0  Warnin...  0.107 sec
✓  10  14:41:45  EXPLAIN ANALYZE  SELECT   l.name,   l.country,   R...   1 row(s) returned                                       0.484 sec / 0.00002...
✓  11  14:47:02  CREATE INDEX idx_weatherdaily_location ON WeatherD...   0 row(s) affected Records: 0  Duplicates: 0  Warnin...  0.124 sec
✓  12  14:47:10  EXPLAIN ANALYZE SELECT   l.name,   l.country,   RO...    1 row(s) returned                                       0.479 sec / 0.000032...
✓  13  14:57:30  SELECT   a.origin_city,  a.destination_city,  a.month,...  1000 row(s) returned                                  0.080 sec / 0.0034 sec
✓  14  15:03:01  EXPLAIN ANALYZE  SELECT   a.origin_city,  a.destinati...  1 row(s) returned                                      0.041 sec / 0.000004...
✓  15  15:11:10  CREATE INDEX idx_flight_destination_fid ON flight_dest...  0 row(s) affected Records: 0  Duplicates: 0  Warnin...  0.106 sec
✓  16  15:11:15  EXPLAIN ANALYZE  SELECT   a.origin_city,  a.destinati...  1 row(s) returned                                      0.043 sec / 0.00003...
✓  17  15:13:22  CREATE INDEX idx_flight_origin_fid ON flight_origin(flig...  0 row(s) affected Records: 0  Duplicates: 0  Warnin...  0.074 sec
✓  18  15:13:27  EXPLAIN ANALYZE  SELECT   a.origin_city,  a.destinati...  1 row(s) returned                                      0.042 sec / 0.00011 s...
✓  19  15:36:46  CREATE INDEX idx_location_id ON Location(location_id)   0 row(s) affected Records: 0  Duplicates: 0  Warnin...  0.067 sec
✓  20  15:36:50  EXPLAIN ANALYZE  SELECT   a.origin_city,  a.destinati...  1 row(s) returned                                      0.042 sec / 0.000023...
✓  21  15:54:14  CREATE INDEX idx_weatherdaily_loc_temp ON Weather...    0 row(s) affected Records: 0  Duplicates: 0  Warnin...  0.135 sec
✓  22  15:55:23  EXPLAIN ANALYZE SELECT   l.name,   l.country,   RO...    1 row(s) returned                                       0.262 sec / 0.000031...
```