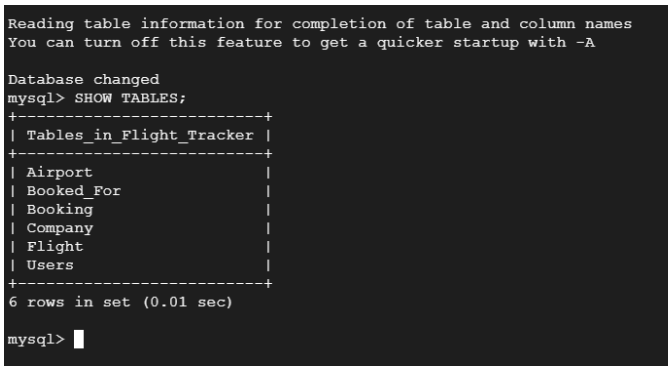
Stage 3: Database Implementation and Indexing

GCP Setup

Database Implementation

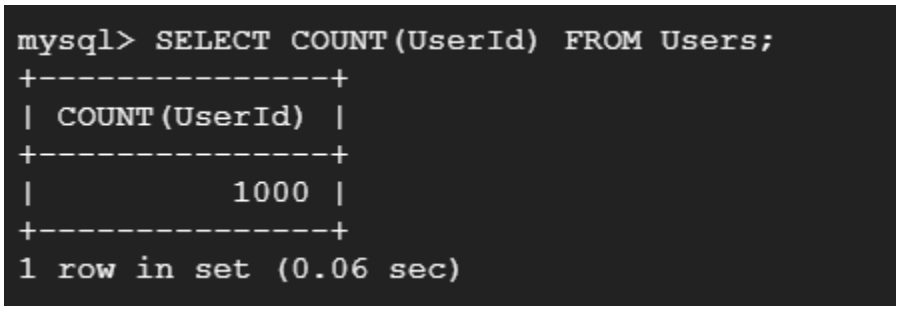
Screenshot of database connection:



DDL Commands:

```
CREATE TABLE Users (
        UserId INT PRIMARY KEY,
        FirstName  VARCHAR(250),
        LastName VARCHAR(250),
        AirportID INT NOT NULL,
        FOREIGN KEY (AirportID) REFERENCES Airport(AirportID) ON DELETE CASCADE
);
```

Count Query:



```
CREATE TABLE Airport (
        AirportID INT PRIMARY KEY,
        AirportName VARCHAR(250)
);
```

Count Query:

```
mysql> SELECT COUNT(AirportID) FROM Airport;
+------------------+
| COUNT(AirportID) |
+------------------+
|               29 |
+------------------+
1 row in set (0.05 sec)
```

```
CREATE TABLE Company (
        CompanyID INT PRIMARY KEY,
        CompanyName VARCHAR(250)
);
```

Count Query:

```
mysql> SELECT COUNT(CompanyID) FROM Company;
+------------------+
| COUNT(CompanyID) |
+------------------+
|               34 |
+------------------+
1 row in set (0.03 sec)
```

```
CREATE TABLE Booking (
        SavedFlightID INT PRIMARY KEY,
        OrderTime DATETIME,
        UserID INT NOT NULL,
        FlightID INT NOT NULL,
        FOREIGN KEY (UserID) REFERENCES Users(UserID) ON DELETE CASCADE,
        FOREIGN KEY (FlightID) REFERENCES Flight(FlightID) ON DELETE CASCADE
);
```

Count Query:

```
mysql> SELECT COUNT(SavedFlightID) FROM Booking;
+----------------------+
| COUNT(SavedFlightID) |
+----------------------+
|                 1500 |
+----------------------+
1 row in set (0.03 sec)
```

```
CREATE TABLE Booked_For (
        SavedFlightID INT,
        FlightID INT,
        PRIMARY KEY (SavedFlightID, FlightID),
        FOREIGN KEY (SavedFlightID) REFERENCES Booking(SavedFlightID) ON DELETE
        CASCADE,
        FOREIGN KEY (FlightID) REFERENCES Flight(FlightID) ON DELETE CASCADE
);
```

Count Query:

```
mysql> SELECT COUNT(SavedFlightID) FROM Booked_For;
+----------------------+
| COUNT(SavedFlightID) |
+----------------------+
|                 2000 |
+----------------------+
1 row in set (0.17 sec)
```

```
CREATE TABLE Flight (
        FlightID INT PRIMARY KEY,
        Departure INT NOT NULL,
        Destination INT NOT NULL,
        FlightPrice REAL,
        TimeOfYear DATE,
        CompanyID INT NOT NULL,
        FOREIGN KEY (Departure) REFERENCES Airport(AirportID) ON DELETE CASCADE,
        FOREIGN KEY (Destination) REFERENCES Airport(AirportID) ON DELETE CASCADE,
        FOREIGN KEY (CompanyID) REFERENCES Company(CompanyID) ON DELETE
        CASCADE
);
```

Count Query:

```
mysql> SELECT COUNT(FlightID) FROM Flight;
+-----------------+
| COUNT(FlightID) |
+-----------------+
|            1000 |
+-----------------+
1 row in set (0.00 sec)
```

Advanced SQL Commands and Indexing Analysis

Variables:
SET @UserID = #;
SET @FirstName = 'name';
SET @LastName = 'name';

<u>SQL Command 1</u>
// returning all the flights for a specific route that begin at the user's main airport ordering from
// lowest to highest price
SET @UserID = 3;

SELECT f.FlightPrice, dep.AirportID, dest.AirportID
FROM Flight f
JOIN Airport dep ON f.Departure = dep.AirportID
JOIN Airport dest ON f.Destination = dest.AirportID
WHERE f.Departure IN (SELECT u.AirportID
FROM Users u
WHERE u.UserID = @UserID)
ORDER BY f.FlightPrice DESC;

Query execution:

```
mysql> SELECT  f.FlightPrice, dep.AirportID, dest.AirportID FROM Flight f JOIN Airport dep ON f.Departure = dep.AirportID JOIN Airport dest ON f.Destination = dest.AirportID WHERE f.Departure IN
 (SELECT u.AirportID  FROM Users u    WHERE u.UserID = @UserID) ORDER BY f.FlightPrice DESC;
+-------------+-----------+-----------+
| FlightPrice | AirportID | AirportID |
+-------------+-----------+-----------+
|     1937.88 |        14 |        12 |
|     1883.65 |        14 |         5 |
|     1759.31 |        14 |         4 |
|      1713.2 |        14 |        28 |
|     1709.17 |        14 |         9 |
|     1692.42 |        14 |        26 |
|     1609.86 |        14 |        12 |
|     1602.18 |        14 |        13 |
|     1583.84 |        14 |        25 |
|     1511.81 |        14 |        19 |
|     1480.78 |        14 |         6 |
|      1463.9 |        14 |        28 |
|     1447.37 |        14 |        10 |
|     1304.96 |        14 |        13 |
|     1212.24 |        14 |        18 |
|     1209.38 |        14 |         6 |
|      1160.4 |        14 |        28 |
|     1127.92 |        14 |         6 |
|     1037.38 |        14 |        25 |
|      945.38 |        14 |         8 |
|      826.38 |        14 |         9 |
|      769.55 |        14 |        19 |
|      732.09 |        14 |        13 |
|       730.1 |        14 |        21 |
|      725.18 |        14 |        22 |
|      676.93 |        14 |        26 |
|      635.29 |        14 |         5 |
|      606.13 |        14 |        26 |
|      492.03 |        14 |        10 |
|      471.03 |        14 |        10 |
|      450.51 |        14 |         2 |
```

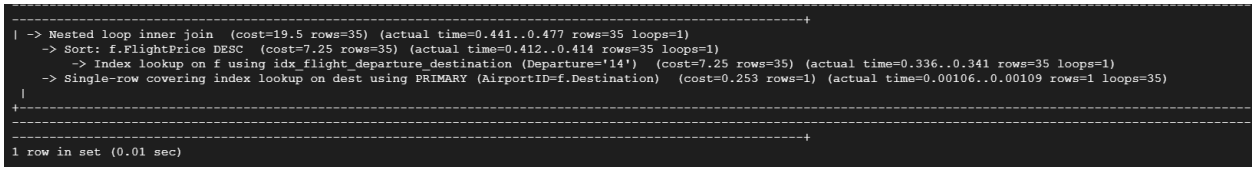
Indexing Analysis:

Before indexing:

```
                                                                                 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------+
| -> Nested loop inner join  (cost=19.5 rows=35) (actual time=0.119..0.153 rows=35 loops=1)
    -> Sort: f.FlightPrice DESC  (cost=7.25 rows=35) (actual time=0.112..0.114 rows=35 loops=1)
        -> Index lookup on f using Departure (Departure='14')  (cost=7.25 rows=35) (actual time=0.0831..0.0884 rows=35 loops=1)
    -> Single-row covering index lookup on dest using PRIMARY (AirportID=f.Destination)  (cost=0.253 rows=1) (actual time=858e-6..887e-6 rows=1 loops=35)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

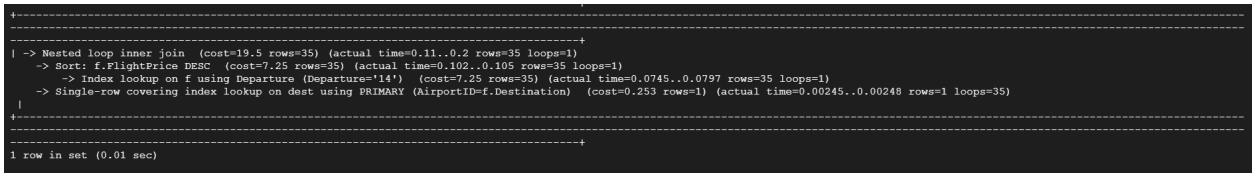
Using the following three index commands:

1. CREATE INDEX idx_flight_departure_destination ON Flight (Departure, Destination);



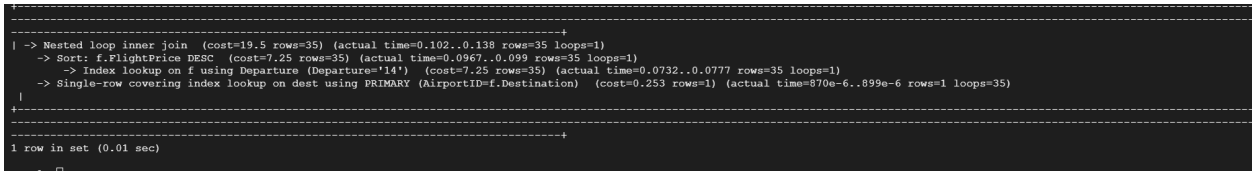
   The computational costs for indexing on flights' Departure and Destination is the same as the original query. This is most likely due to the low amount of data involved. For example, there is only 1 row where AirportID=f.Destination, and therefore, the cost of an index lookup is very low (.253) since each AirportID is unique. There are only 35 rows (flights) that match the criteria, and because of the low amount of data, the cost is low and easy to index on (same as original cost).

2. CREATE INDEX idx_users_userid ON Users(FirstName, LastName);



   Again, since there are not a lot of rows in the final query (35) and not a lot of complicated filtering, the computational cost with indexing is still low at 19.5 and 7.25 (same as original). A lot of costs involved with filtering like for AirportID=f.Destination is fast (cost of .253) since there is only 1 airport that we are looking for (which in this case has AirportId = 14).

3. CREATE INDEX idx_users_firstname ON Users (FirstName);



   Again, since there are not a lot of rows in the final query (35) and not a lot of complicated filtering, the computational cost with indexing on FirstName is still low at 19.5 and 7.25 (same as original). Queries with not a lot of rows tend not to have a lot of computational cost due to the small dataset size, meaning that even without complex indexing, the system can process the data efficiently. Therefore, there is no need to index on FirstName, or any other attribute due to the limited number of rows.

SQL Command 2
// finds the cheapest flight per company
SELECT c.CompanyName, MIN(f.FlightPrice) AS CheapestFlightPrice
FROM Company c JOIN Flight f ON c.CompanyID = f.CompanyID
WHERE f.TimeOfYear = 3
GROUP BY c.CompanyName
ORDER BY CheapestFlightPrice;

Query Execution:

```
    -> GROUP BY c.CompanyName
    -> ORDER BY CheapestFlightPrice;
+-------------------------+---------------------+
| CompanyName             | CheapestFlightPrice |
+-------------------------+---------------------+
| Mesa Airlines           |               54.75 |
| Luxair                  |               58.16 |
| ViaAir                  |               63.48 |
| Boutique Air            |               72.92 |
| Allegiant Air           |               87.36 |
| American Airlines       |              117.34 |
| Envoy Air               |              119.63 |
| British Airways         |              126.33 |
| GoJet Airlines          |              127.76 |
| JetBlue Airways         |              133.39 |
| KLM                     |              145.35 |
| Singapore Airlines      |              146.44 |
| Republic Airways        |              172.39 |
| United Airlines         |              181.23 |
| Spirit Airlines         |              188.94 |
| Alaska Airlines         |              194.22 |
| Lufthansa               |              196.83 |
| Air Choice One          |              198.52 |
| Emirates                |              206.59 |
| Endeavor Air            |              222.94 |
| Southern Airways Express |             231.14 |
| Frontier Airlines       |              237.94 |
| Twin Cities Air Service |               283.1 |
| Air France              |              304.81 |
| Southwest Airlines      |              311.02 |
| Qatar Airways           |              384.27 |
| Contour Airlines        |              401.25 |
| Hawaiian Airlines       |              402.59 |
| Compass Airlines        |              450.51 |
| Silver Airways          |              506.75 |
| Cape Air                |              691.67 |
| Sun Country Airlines    |              717.98 |
| Delta Air Lines         |             1249.28 |
| SkyWest Airlines        |             1417.52 |
```

Indexing Analysis:
Before indexing:

```
----------------------------------------------------------------------------------------+
| -> Sort: CheapestFlightPrice  (actual time=2.84..2.84 rows=34 loops=1)
    -> Table scan on <temporary>  (actual time=2.8..2.81 rows=34 loops=1)
        -> Aggregate using temporary table  (actual time=2.8..2.8 rows=34 loops=1)
            -> Nested loop inner join  (cost=136 rows=100) (actual time=0.153..1.28 rows=244 loops=1)
                -> Filter: (f.TimeOfYear = 3)  (cost=101 rows=100) (actual time=0.0891..0.535 rows=244 loops=1)
                    -> Table scan on f  (cost=101 rows=1000) (actual time=0.0836..0.45 rows=1000 loops=1)
                -> Single-row index lookup on c using PRIMARY (CompanyID=f.CompanyID)  (cost=0.251 rows=1) (actual time=0.00249..0.00253 rows=1 loops=244)
 |
+----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

Using the following three index commands:

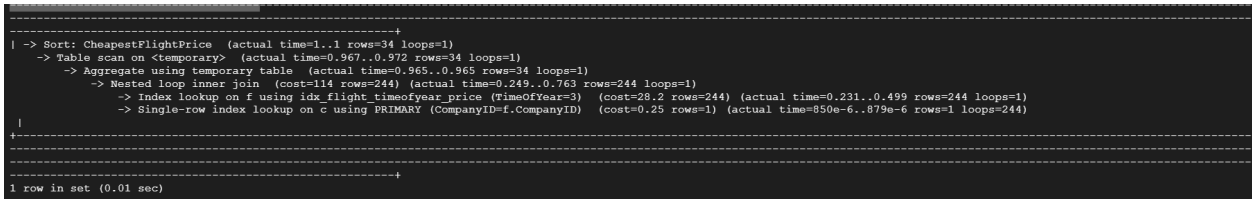
1. CREATE INDEX idx_flight_price ON Flight (FlightPrice, Departure);

```
-------------------------------------------------------------+
| -> Sort: CheapestFlightPrice  (actual time=0.988..0.99 rows=34 loops=1)
    -> Table scan on <temporary>  (actual time=0.954..0.959 rows=34 loops=1)
        -> Aggregate using temporary table  (actual time=0.952..0.952 rows=34 loops=1)
            -> Nested loop inner join  (cost=114 rows=244) (actual time=0.32..0.763 rows=244 loops=1)
                -> Index lookup on f using idx_flight_timeofyear_price (TimeOfYear=3)  (cost=28.2 rows=244) (actual time=0.252..0.478 rows=244 loops=1)
                -> Single-row index lookup on c using PRIMARY (CompanyID=f.CompanyID)  (cost=0.25 rows=1) (actual time=746e-6..774e-6 rows=1 loops=244)
 |
+-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
-------------------------------------------------------------+
1 row in set (0.00 sec)
```

The computational costs for indexing on flights' FlightPrice is different from the original query overall. However, when doing an index lookup for CompanyID=f.CompanyID, the computational cost is very low (and the same as the original) since there is only 1 unique row involved (only 1 company which we are looking for). Indexing on FlightPrice reduces

overall computational cost (114 < 136) in this query by allowing the database to quickly find the MIN(f.FlightPrice) per company using the index, rather than scanning all rows. This improves the performance of both the MIN() and sorting ORDER BY steps. Without the index, the database would need to scan the entire Flight table, filter rows, aggregate the prices, and then perform a sort, all of which are more resource-intensive.

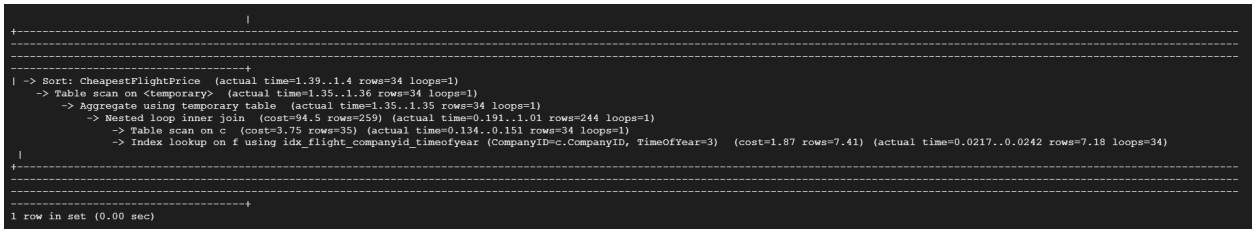
2. CREATE INDEX idx_flight_timeofyear_price ON Flight (TimeOfYear, FlightPrice);



```
| -> Sort: CheapestFlightPrice  (actual time=1..1 rows=34 loops=1)
    -> Table scan on <temporary>  (actual time=0.967..0.972 rows=34 loops=1)
        -> Aggregate using temporary table  (actual time=0.965..0.965 rows=34 loops=1)
            -> Nested loop inner join  (cost=114 rows=244) (actual time=0.249..0.763 rows=244 loops=1)
                -> Index lookup on f using idx_flight_timeofyear_price (TimeOfYear=3)  (cost=28.2 rows=244) (actual time=0.231..0.499 rows=244 loops=1)
                -> Single-row index lookup on c using PRIMARY (CompanyID=f.CompanyID)  (cost=0.25 rows=1) (actual time=850e-6..879e-6 rows=1 loops=244)
 |
1 row in set (0.01 sec)
```

The computational costs for indexing on flights' FlightPrice and TimeOfYear is different from the original query overall. However, when doing an index lookup for CompanyID=f.CompanyID, the computational cost is very low (and the same as the original) since there is only 1 unique row involved (only 1 company which we are looking for). Indexing on FlightPrice and TimeOfYear reduces overall computational cost (114 < 136) in this query by allowing the database to quickly find the MIN(f.FlightPrice) per company using the index, rather than scanning all rows. This improves the performance of both the MIN() and sorting ORDER BY steps. Without the index, the database would need to scan the entire Flight table, filter rows, aggregate the prices, and then perform a sort, all of which are more resource-intensive.

3. CREATE INDEX idx_flight_companyid_timeofyear ON Flight (CompanyName, TimeOfYear);



```
| -> Sort: CheapestFlightPrice  (actual time=1.39..1.4 rows=34 loops=1)
    -> Table scan on <temporary>  (actual time=1.35..1.36 rows=34 loops=1)
        -> Aggregate using temporary table  (actual time=1.35..1.35 rows=34 loops=1)
            -> Nested loop inner join  (cost=94.5 rows=259) (actual time=0.191..1.01 rows=244 loops=1)
                -> Table scan on c  (cost=3.75 rows=35) (actual time=0.134..0.151 rows=34 loops=1)
                -> Index lookup on f using idx_flight_companyid_timeofyear (CompanyID=c.CompanyID, TimeOfYear=3)  (cost=1.87 rows=7.41) (actual time=0.0217..0.0242 rows=7.18 loops=34)
 |
1 row in set (0.00 sec)
```

Indexing on CompanyName and TimeOfYear was more efficient yielding a computational cost of 94.5 for the nested loop inner join (compared to a cost of 136 for the original). Because we are trying to filter our query using a TimeOfYear = 3, indexing with this attribute makes it much more efficient in cutting out rows for flights. We can see in the output from the original query that the filter itself had a cost of 101, which we were able to dramatically reduce by using this indexing structure.

SQL Command 3

```
// calculates the average flight price for each departure to destination grouped by quarter
// (time of year) and sorts them from prices low to high
SELECT f.Departure, f.Destination,  ROUND(AVG(f.FlightPrice), 2) AS AvgPrice, f.TimeOfYear
AS Quarter
FROM Flight f
JOIN Airport dep ON f.Departure = dep.AirportID
JOIN Airport dest ON f.Destination = dest.AirportID
GROUP BY f.Departure, f.Destination, Quarter
ORDER BY AvgPrice DESC;
```

Query Execution:

```
    -> ORDER BY AvgPrice DESC;
+-----------+-------------+----------+---------+
| Departure | Destination | AvgPrice | Quarter |
+-----------+-------------+----------+---------+
|        19 |          11 |  1995.28 |       2 |
|         5 |          21 |  1994.06 |       4 |
|        19 |          17 |  1983.73 |       4 |
|        19 |           5 |  1981.18 |       2 |
|        18 |          24 |  1974.29 |       3 |
|        18 |          12 |  1964.88 |       3 |
|         3 |          26 |  1962.62 |       1 |
|         1 |          24 |  1962.43 |       4 |
|        16 |           2 |  1960.45 |       2 |
|        17 |          19 |  1958.87 |       2 |
|         2 |          21 |  1953.64 |       4 |
|        13 |          15 |  1951.96 |       2 |
|         8 |          22 |  1951.03 |       1 |
|        24 |          10 |  1950.12 |       3 |
|         5 |           4 |  1949.84 |       2 |
|         6 |           5 |  1948.08 |       2 |
|        23 |           3 |  1947.41 |       2 |
|        15 |          13 |  1947.31 |       3 |
|         6 |          12 |  1945.43 |       3 |
|        20 |          25 |  1943.18 |       1 |
|        14 |          12 |  1937.88 |       4 |
|        12 |           7 |   1934.7 |       2 |
|         4 |           5 |  1929.07 |       1 |
|        12 |           8 |  1926.29 |       3 |
|        16 |          22 |   1921.7 |       1 |
|         6 |          28 |  1919.48 |       1 |
|        13 |           4 |  1916.36 |       2 |
|         4 |          17 |  1915.96 |       2 |
|         8 |           5 |  1911.45 |       4 |
|         6 |           3 |  1901.81 |       2 |
|        24 |           9 |  1898.08 |       2 |
|        23 |          20 |  1896.98 |       4 |
|         2 |          29 |  1892.65 |       1 |
|        20 |           6 |  1891.01 |       2 |
|        19 |           4 |  1890.51 |       4 |
```

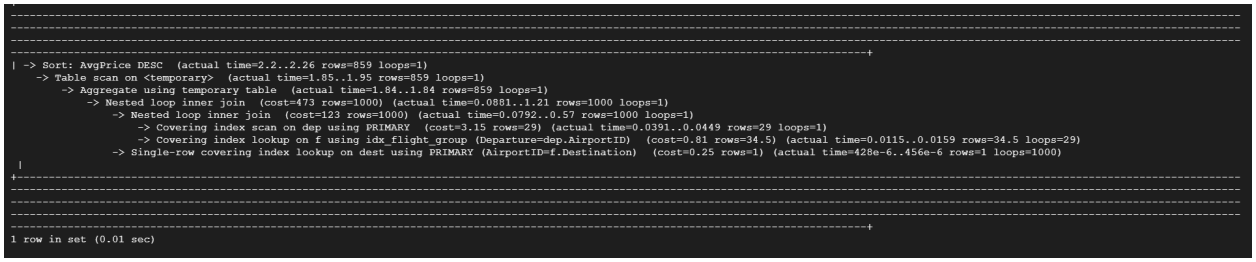
Indexing Analysis:

Before indexing:

```
-> Sort: AvgPrice DESC  (actual time=3.81..3.86 rows=859 loops=1)
    -> Table scan on <temporary>  (actual time=3.48..3.57 rows=859 loops=1)
        -> Aggregate using temporary table  (actual time=3.48..3.48 rows=859 loops=1)
            -> Nested loop inner join  (cost=562 rows=1000) (actual time=0.299..2.63 rows=1000 loops=1)
                -> Nested loop inner join  (cost=212 rows=1000) (actual time=0.273..1.61 rows=1000 loops=1)
                    -> Covering index scan on dep using PRIMARY  (cost=3.15 rows=29) (actual time=0.122..0.129 rows=29 loops=1)
                    -> Index lookup on f using Departure (Departure=dep.AirportID)  (cost=3.87 rows=34.5) (actual time=0.043..0.0489 rows=34.5 loops=29)
                -> Single-row covering index lookup on dest using PRIMARY (AirportID=f.Destination)  (cost=0.25 rows=1) (actual time=806e-6..835e-6 rows=1 loops=1000)
 |
1 row in set (0.01 sec)
```

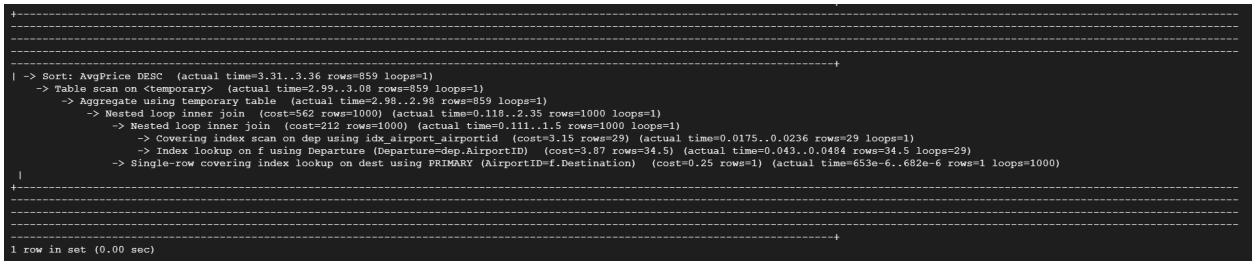
Using the following three index commands:

1. CREATE INDEX idx_flight_group ON Flight (Departure, Destination, TimeOfYear, FlightPrice);



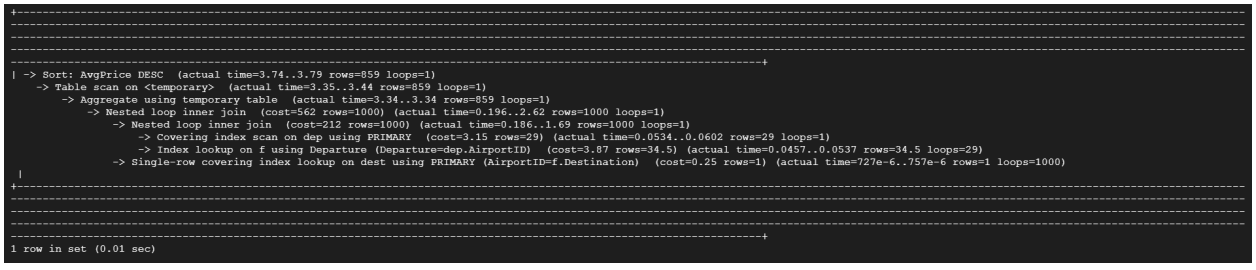
The index on (Departure, Destination, TimeOfYear, FlightPrice) is more efficient than non indexing (cost of 473 with index < cost of 562 without index) because if CompanyID is unique, it allows the database to quickly filter by TimeOfYear and join the Company and Flight tables, which is guaranteed to be unique for each company. This lessens the number of rows to process. Since the query is grouped by CompanyName and aggregates the minimum flight price, this index directly optimizes the query, making the query faster than if an index were created solely on FlightPrice, which isn't as relevant for filtering or joining.

2. CREATE INDEX idx_airport_airportid ON Airport (AirportName);



Indexing on AirportName does not change the efficiency of the query. As we can see, the query before indexing had a cost of 562 (for the outer nested loop inner join) and 212 (for the inner nested loop inner join). When we apply the index, we also get the same results: 562 and 212 respectively. The reason this is most likely due to the fact that we do not filter or join on AirportName in the original query, so creating an index will have no effect. The main cost of this query is driven by the GROUP BY and ORDER BY which do not operate on AirportName, so this is why we see no change in the cost. Therefore, this is not an efficient/beneficial index and we should not use it.

3. CREATE INDEX idx_flight__destination_price ON Flight (Destination, FlightPrice);



Indexing on Destination and FlightPrice also seems to have little to no change on the efficiency of the query. As mentioned before, the query without indexing had a cost of 562 (for the outer nested loop inner join) and 212 (for the inner nested loop inner join).

When we apply the index, we get 562 and 212 again. Although we do use Destination in the JOIN in the original query (when we JOIN ON destination and departure), we are only optimizing the search for Destination in this query which is not as efficient since destination and departure come in pairs. Also, since the query needs to compute the average of the FlightPrice, it still needs to scan through the entire column of values, so indexing it doesn't help as much. Therefore, indexing on these values is not very beneficial and Destination and FlightPrice should not be used as an index.

SQL Commands 4

```
// returns the most popular flights that users have saved
SELECT b.SavedFlightID, f.Departure, f.Destination, f.FlightPrice, COUNT(b.SavedFlightID) As SavedCount
FROM Booking b
JOIN Flight f USING (FlightID)
JOIN Users u USING (UserID)
JOIN Airport dep ON f.Departure = dep.AirportID
JOIN Airport dest ON f.Destination = dest.AirportID
GROUP BY  b.SavedFlightID, f.Departure, f.Destination, f.FlightPrice
ORDER BY SavedCount DESC;
```

Query Execution:

SavedFlightID	Departure	Destination	FlightPrice	SavedCount
1	24	20	196.84	1
2	10	1	1796.86	1
3	7	1	1265.76	1
4	19	29	1607.59	1
5	2	6	1679.74	1
6	22	24	614.23	1
7	1	27	1963.12	1
8	27	6	116.14	1
9	26	14	1474.38	1
10	21	18	454.26	1
11	5	13	1246.47	1
12	29	28	401.25	1
13	3	28	105.08	1
14	20	14	296.75	1
15	2	9	1370.97	1
16	16	8	449.2	1
17	4	29	1585.76	1
18	20	24	1360.99	1
19	7	10	745.48	1
20	16	21	52.46	1
21	23	15	1400.92	1
22	28	14	1582.59	1
23	2	9	1370.97	1
24	17	2	1126.9	1
25	2	10	159.81	1
26	29	27	645.47	1
27	19	12	1170.11	1
28	1	19	852.72	1
29	21	22	1774.66	1
30	4	13	705.93	1
31	17	15	953.22	1
32	19	28	756.66	1
33	23	1	1650.11	1
34	17	3	1338.85	1
35	16	6	837.1	1
36	29	27	645.47	1

Indexing Analysis:

Before indexing:

```
| -> Sort: SavedCount DESC  (actual time=8.7..8.8 rows=1500 loops=1)
    -> Table scan on <temporary>  (actual time=8.27..8.43 rows=1500 loops=1)
        -> Aggregate using temporary table  (actual time=8.27..8.27 rows=1500 loops=1)
            -> Nested loop inner join  (cost=2251 rows=1500) (actual time=0.0952..6.99 rows=1500 loops=1)
                -> Nested loop inner join  (cost=1726 rows=1500) (actual time=0.089..5.27 rows=1500 loops=1)
                    -> Nested loop inner join  (cost=1201 rows=1500) (actual time=0.0844..4.03 rows=1500 loops=1)
                        -> Nested loop inner join  (cost=676 rows=1500) (actual time=0.0771..2.67 rows=1500 loops=1)
                            -> Table scan on b  (cost=151 rows=1500) (actual time=0.0593..0.609 rows=1500 loops=1)
                            -> Single-row index lookup on f using PRIMARY (FlightID=b.FlightID)  (cost=0.25 rows=1) (actual time=0.00114..0.00117 rows=1 loops=1500)
                        -> Single-row covering index lookup on dep using PRIMARY (AirportID=f.Departure)  (cost=0.25 rows=1) (actual time=684e-6..713e-6 rows=1 loops=1500)
                    -> Single-row covering index lookup on dest using PRIMARY (AirportID=f.Destination)  (cost=0.25 rows=1) (actual time=629e-6..659e-6 rows=1 loops=1500)
                -> Single-row covering index lookup on u using PRIMARY (UserId=b.UserID)  (cost=0.25 rows=1) (actual time=938e-6..967e-6 rows=1 loops=1500)
 |
1 row in set (0.01 sec)
```

Using the following three index commands:

1. CREATE INDEX idx_flight_departure_destination_price ON Flight (Departure, Destination, FlightPrice);

```
| -> Sort: SavedCount DESC  (actual time=7.33..7.44 rows=1500 loops=1)
    -> Table scan on <temporary>  (actual time=6.92..7.07 rows=1500 loops=1)
        -> Aggregate using temporary table  (actual time=6.92..6.92 rows=1500 loops=1)
            -> Nested loop inner join  (cost=2251 rows=1500) (actual time=0.0568..5.87 rows=1500 loops=1)
                -> Nested loop inner join  (cost=1726 rows=1500) (actual time=0.0512..4.51 rows=1500 loops=1)
                    -> Nested loop inner join  (cost=1201 rows=1500) (actual time=0.0473..3.32 rows=1500 loops=1)
                        -> Nested loop inner join  (cost=676 rows=1500) (actual time=0.0408..2.12 rows=1500 loops=1)
                            -> Table scan on b  (cost=151 rows=1500) (actual time=0.0297..0.432 rows=1500 loops=1)
                            -> Single-row index lookup on f using PRIMARY (FlightID=b.FlightID)  (cost=0.25 rows=1) (actual time=914e-6..942e-6 rows=1 loops=1500)
                        -> Single-row covering index lookup on dep using PRIMARY (AirportID=f.Departure)  (cost=0.25 rows=1) (actual time=606e-6..640e-6 rows=1 loops=1500)
                    -> Single-row covering index lookup on dest using PRIMARY (AirportID=f.Destination)  (cost=0.25 rows=1) (actual time=593e-6..621e-6 rows=1 loops=1500)
                -> Single-row covering index lookup on u using PRIMARY (UserId=b.UserID)  (cost=0.25 rows=1) (actual time=709e-6..737e-6 rows=1 loops=1500)
 |
1 row in set (0.01 sec)
```

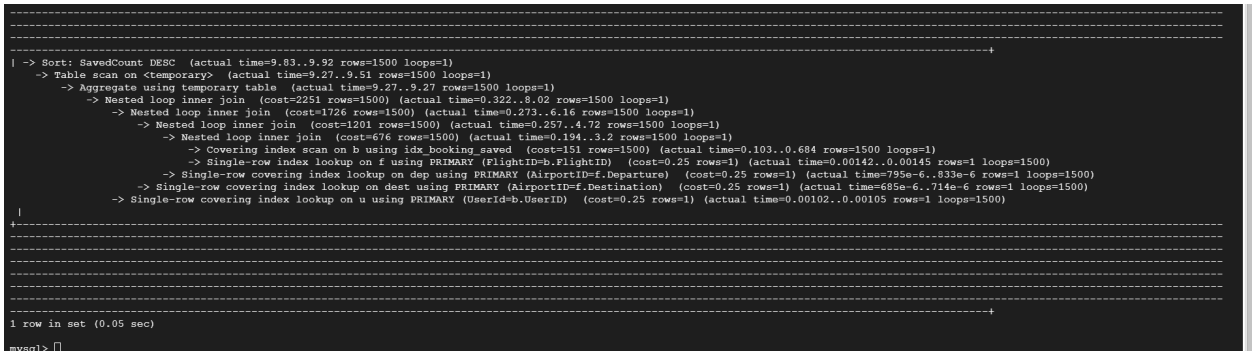
The cost is the same as the original query (very high at 2251), even with the indexing due to the large volume of data. It involves multiple JOIN operations, which combine large tables, resulting in a large combined dataset. The GROUP BY and sorting functions require the database to aggregate many combinations of values which is resource-intensive leading to a high cost for many rows. Indexing on Departure, Destination, and FlightPrice do not change the large overall amount of data and the amount of grouping / ordering between entries.

2. CREATE INDEX idx_customer_id ON Booking(OrderTime);

```
| -> Sort: SavedCount DESC  (actual time=7.99..8.08 rows=1500 loops=1)
    -> Table scan on <temporary>  (actual time=7.55..7.7 rows=1500 loops=1)
        -> Aggregate using temporary table  (actual time=7.55..7.55 rows=1500 loops=1)
            -> Nested loop inner join  (cost=2251 rows=1500) (actual time=0.158..6.42 rows=1500 loops=1)
                -> Nested loop inner join  (cost=1726 rows=1500) (actual time=0.153..4.8 rows=1500 loops=1)
                    -> Nested loop inner join  (cost=1201 rows=1500) (actual time=0.149..3.58 rows=1500 loops=1)
                        -> Nested loop inner join  (cost=676 rows=1500) (actual time=0.142..2.33 rows=1500 loops=1)
                            -> Table scan on b  (cost=151 rows=1500) (actual time=0.12..0.535 rows=1500 loops=1)
                            -> Single-row index lookup on f using PRIMARY (FlightID=b.FlightID)  (cost=0.25 rows=1) (actual time=978e-6..0.00101 rows=1 loops=1500)
                        -> Single-row covering index lookup on dep using PRIMARY (AirportID=f.Departure)  (cost=0.25 rows=1) (actual time=624e-6..652e-6 rows=1 loops=1500)
                    -> Single-row covering index lookup on dest using PRIMARY (AirportID=f.Destination)  (cost=0.25 rows=1) (actual time=603e-6..631e-6 rows=1 loops=1500)
                -> Single-row covering index lookup on u using PRIMARY (UserId=b.UserID)  (cost=0.25 rows=1) (actual time=853e-6..888e-6 rows=1 loops=1500)
 |
1 row in set (0.01 sec)
```

Even with the indexing (CREATE INDEX idx_customer_id ON Booking(OrderTime)), the cost remains the same as the original query (very high at 2251), due to the large volume of data. The query involves multiple JOIN operations, which combine large tables, resulting in a large combined dataset. The GROUP BY and sorting functions require the database to aggregate many combinations of values, which is resource-intensive, leading to a high cost for processing many rows.

3. CREATE INDEX idx_users_lastname ON Users(LastName);



Indexing on LastName does not impact the cost of the query. Before the query, we have a cost of 2251, 1726, 1201, and 676 for the nested loop inner joins (from the outermost to the innermost). After indexing, we have the same values. This is most likely because LastName is not used in the WHERE, JOIN, GROUP BY, or ORDER BY clauses. This means that indexing on LastName will not really speed up the process of computing this query. Therefore, indexing on LastName is not helpful and we should not use it for this index.