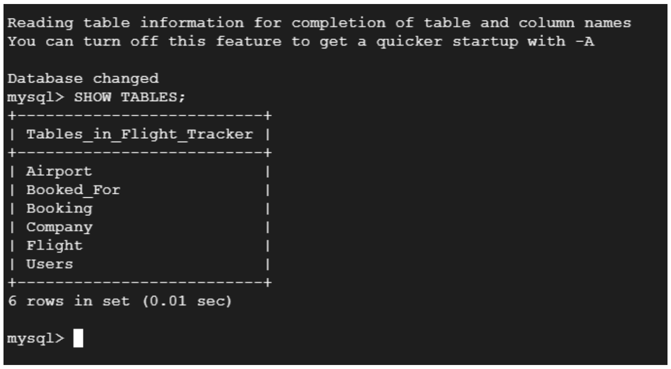
Stage 3: Database Implementation and Indexing

GCP Setup

Database Implementation

Screenshot of database connection:

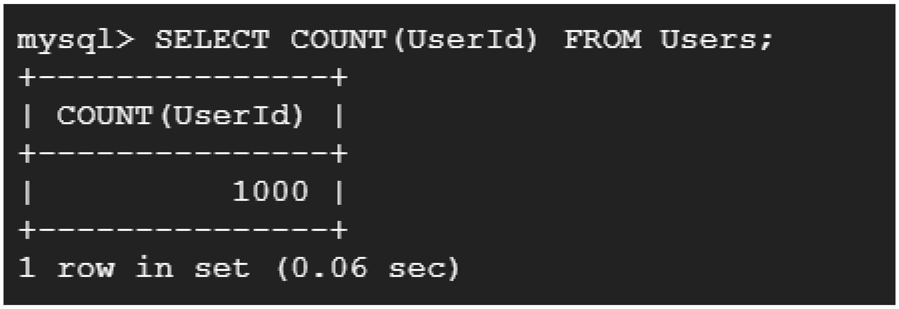


```
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SHOW TABLES;
+--------------------------+
| Tables_in_Flight_Tracker |
+--------------------------+
| Airport                  |
| Booked_For               |
| Booking                  |
| Company                  |
| Flight                   |
| Users                    |
+--------------------------+
6 rows in set (0.01 sec)

mysql> 
```

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
);
```

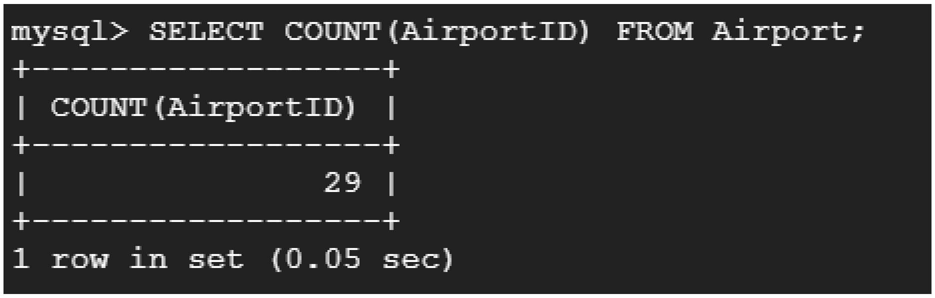
Count Query:



```
mysql> SELECT COUNT(UserId) FROM Users;
+---------------+
| COUNT(UserId) |
+---------------+
|          1000 |
+---------------+
1 row in set (0.06 sec)
```

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

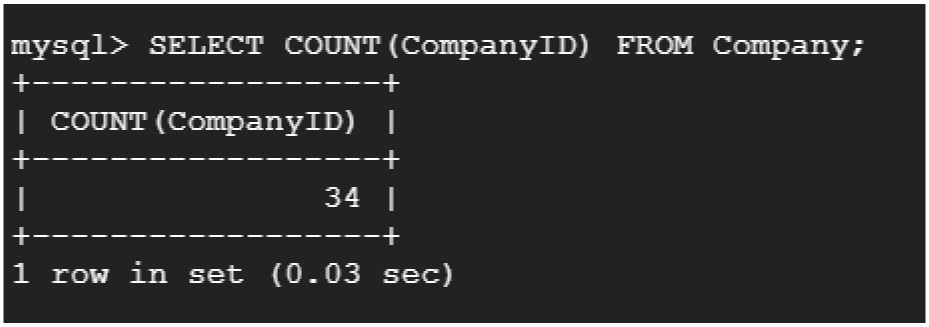
Count Query:



```
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)
);
```

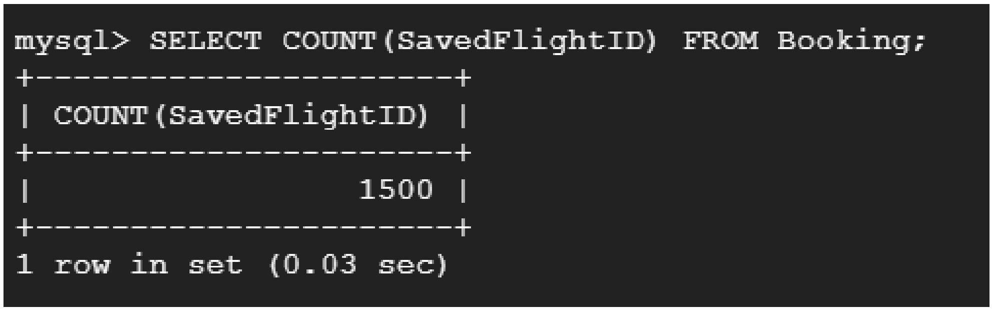
Count Query:



```
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
);
```

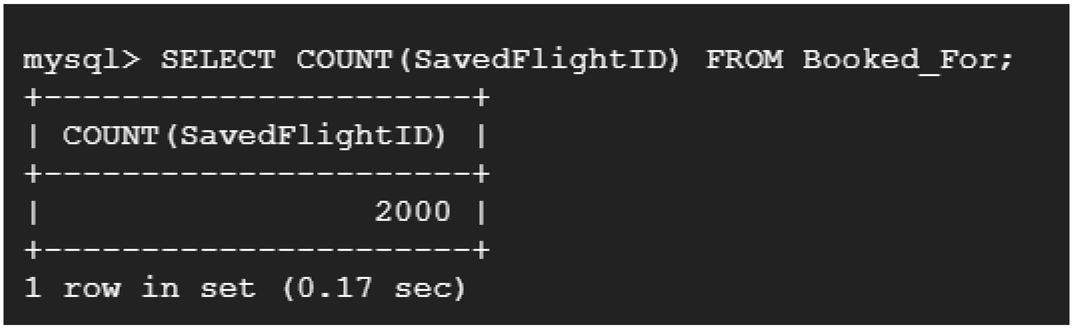
Count Query:



```
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
);
```

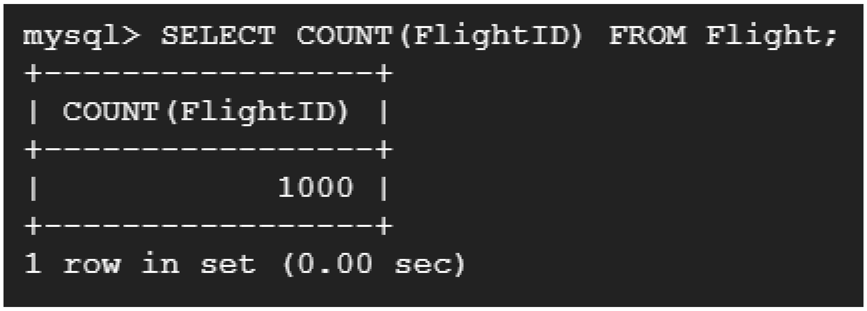
Count Query:



```
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)
```

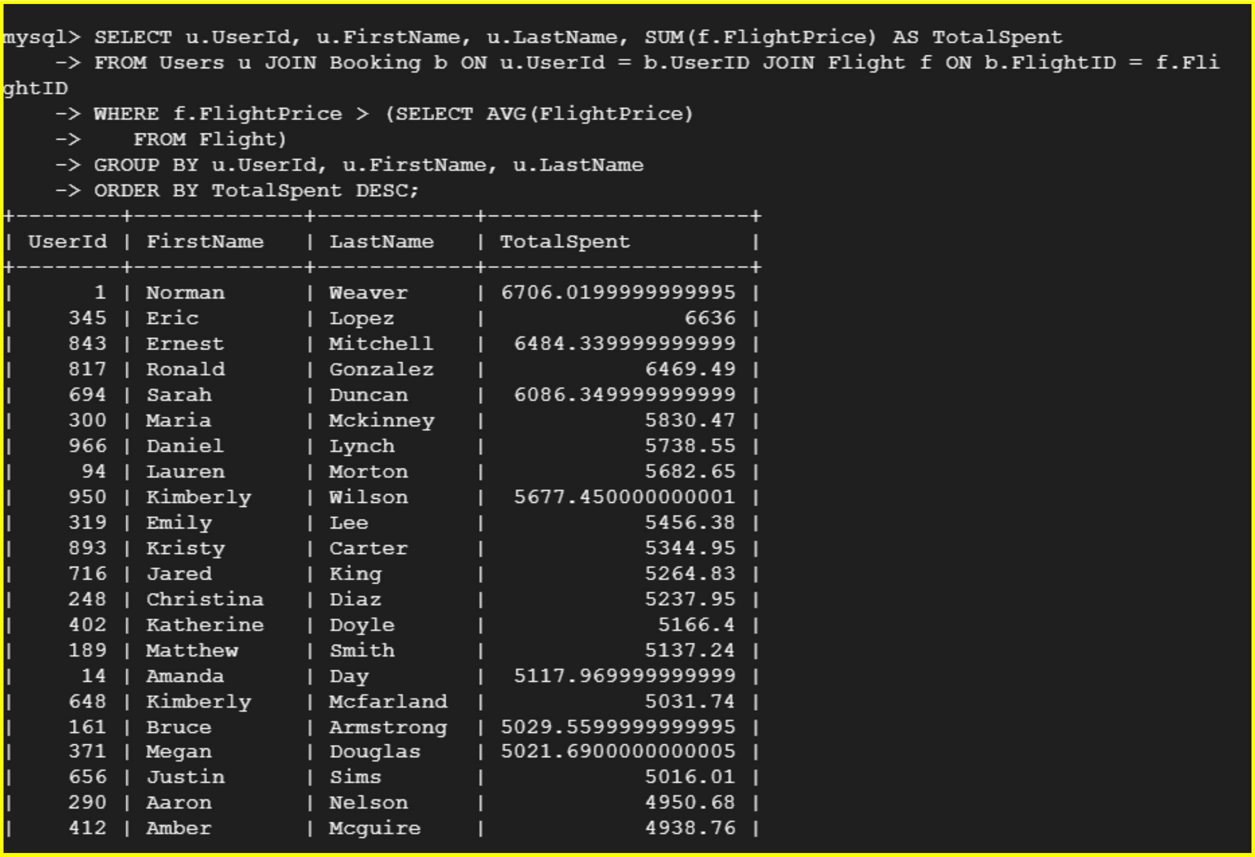
Advanced SQL Commands and Indexing Analysis

Variables:
SET @UserID = #;
SET @FirstName = ‘name’;
SET @LastName = ‘name’;

SQL Command 1

```
// query that orders users based on how much they spent on flights (only counting flights that
// cost more than the average price) – could be useful for reward program
SELECT u.UserId, u.FirstName, u.LastName, SUM(f.FlightPrice) AS TotalSpent
FROM Users u JOIN Booking b ON u.UserId = b.UserID JOIN Flight f ON b.FlightID = f.FlightID
WHERE f.FlightPrice > (SELECT AVG(FlightPrice)
                          FROM Flight)
GROUP BY u.UserId, u.FirstName, u.LastName
ORDER BY TotalSpent DESC;
```

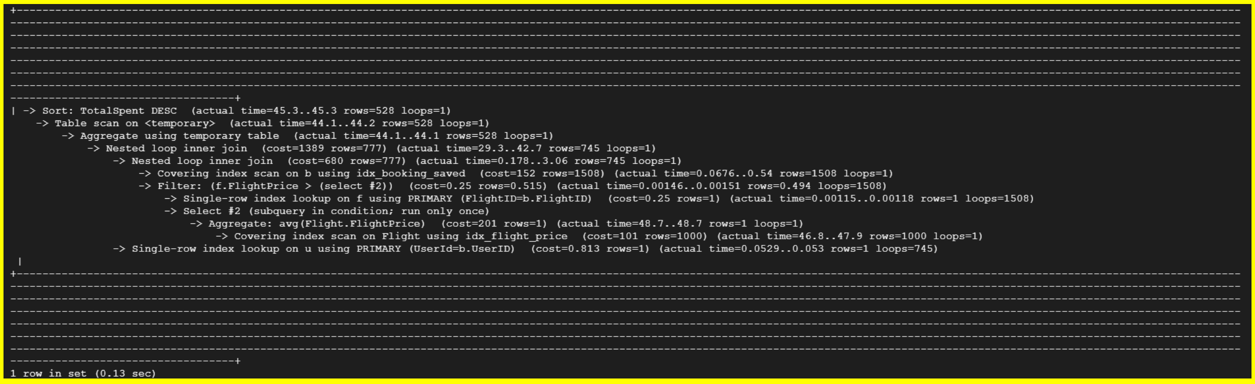
Query execution:



```
mysql> SELECT u.UserId, u.FirstName, u.LastName, SUM(f.FlightPrice) AS TotalSpent
    -> FROM Users u JOIN Booking b ON u.UserId = b.UserID JOIN Flight f ON b.FlightID = f.Fli
ghtID
    -> WHERE f.FlightPrice > (SELECT AVG(FlightPrice)
    ->    FROM Flight)
    -> GROUP BY u.UserId, u.FirstName, u.LastName
    -> ORDER BY TotalSpent DESC;
+--------+-------------+------------+--------------------+
| UserId | FirstName   | LastName   | TotalSpent         |
+--------+-------------+------------+--------------------+
|      1 | Norman      | Weaver     | 6706.0199999999995 |
|    345 | Eric        | Lopez      |               6636 |
|    843 | Ernest      | Mitchell   |  6484.339999999999 |
|    817 | Ronald      | Gonzalez   |            6469.49 |
|    694 | Sarah       | Duncan     |  6086.349999999999 |
|    300 | Maria       | Mckinney   |            5830.47 |
|    966 | Daniel      | Lynch      |            5738.55 |
|     94 | Lauren      | Morton     |            5682.65 |
|    950 | Kimberly    | Wilson     |  5677.450000000001 |
|    319 | Emily       | Lee        |            5456.38 |
|    893 | Kristy      | Carter     |            5344.95 |
|    716 | Jared       | King       |            5264.83 |
|    248 | Christina   | Diaz       |            5237.95 |
|    402 | Katherine   | Doyle      |             5166.4 |
|    189 | Matthew     | Smith      |            5137.24 |
|     14 | Amanda      | Day        |  5117.969999999999 |
|    648 | Kimberly    | Mcfarland  |            5031.74 |
|    161 | Bruce       | Armstrong  | 5029.5599999999995 |
|    371 | Megan       | Douglas    | 5021.6900000000005 |
|    656 | Justin      | Sims       |            5016.01 |
|    290 | Aaron       | Nelson     |            4950.68 |
|    412 | Amber       | Mcguire    |            4938.76 |
```

Indexing Analysis:

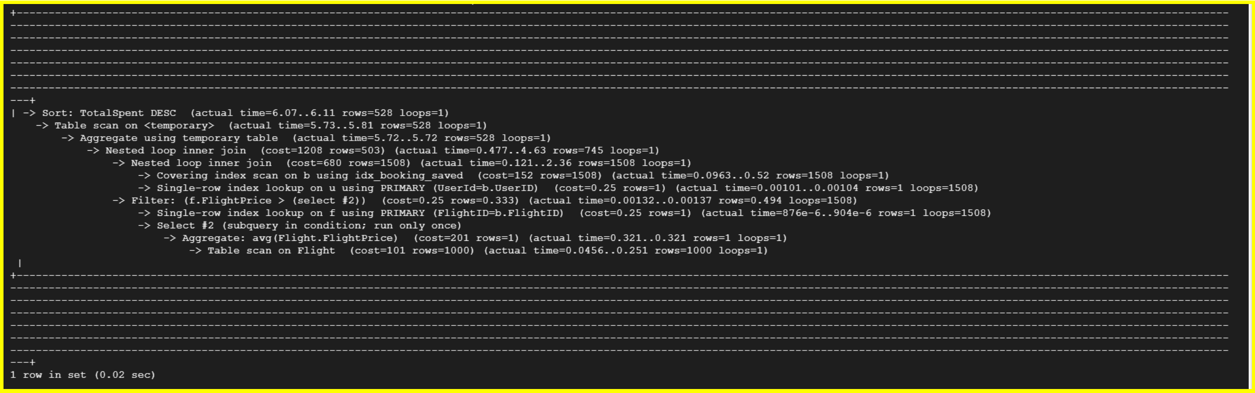
Before indexing:



```
+--------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
-----------------------------------+
| -> Sort: TotalSpent DESC  (actual time=45.3..45.3 rows=528 loops=1)
    -> Table scan on <temporary>  (actual time=44.1..44.2 rows=528 loops=1)
        -> Aggregate using temporary table  (actual time=44.1..44.1 rows=528 loops=1)
            -> Nested loop inner join  (cost=1389 rows=777) (actual time=29.3..42.7 rows=745 loops=1)
                -> Nested loop inner join  (cost=680 rows=777) (actual time=0.178..3.06 rows=745 loops=1)
                    -> Covering index scan on b using idx_booking_saved  (cost=152 rows=1508) (actual time=0.0676..0.54 rows=1508 loops=1)
                    -> Filter: (f.FlightPrice > (select #2))  (cost=0.25 rows=0.515) (actual time=0.00146..0.00151 rows=0.494 loops=1508)
                        -> Single-row index lookup on f using PRIMARY (FlightID=b.FlightID)  (cost=0.25 rows=1) (actual time=0.00115..0.00118 rows=1 loops=1508)
                        -> Select #2 (subquery in condition; run only once)
                            -> Aggregate: avg(Flight.FlightPrice)  (cost=201 rows=1) (actual time=48.7..48.7 rows=1 loops=1)
                                -> Covering index scan on Flight using idx_flight_price  (cost=101 rows=1000) (actual time=46.8..47.9 rows=1000 loops=1)
                -> Single-row index lookup on u using PRIMARY (UserId=b.UserID)  (cost=0.813 rows=1) (actual time=0.0529..0.053 rows=1 loops=745)
 |
+--------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
-----------------------------------+
1 row in set (0.13 sec)
```

Using the following three index commands:

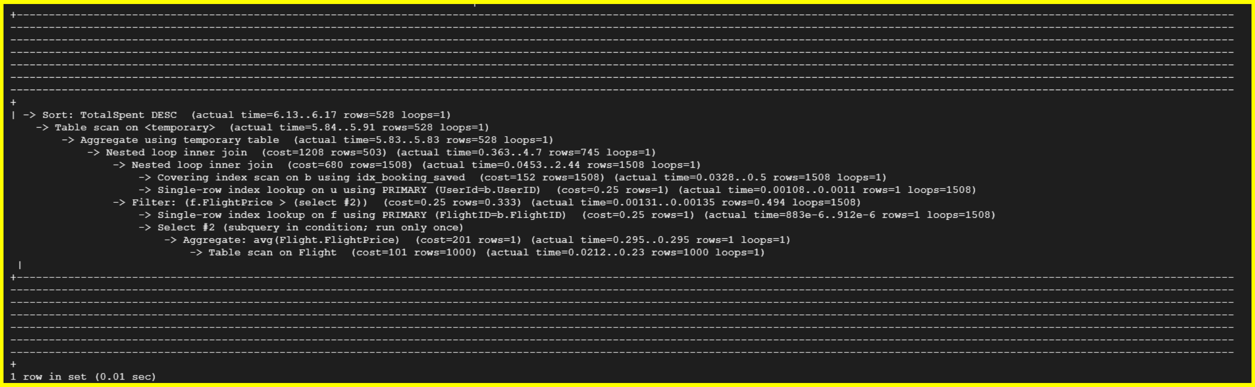
1. CREATE INDEX idx_flight_price ON Flight(FlightPrice);



```
+--------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---+
| -> Sort: TotalSpent DESC  (actual time=6.07..6.11 rows=528 loops=1)
    -> Table scan on <temporary>  (actual time=5.73..5.81 rows=528 loops=1)
        -> Aggregate using temporary table  (actual time=5.72..5.72 rows=528 loops=1)
            -> Nested loop inner join  (cost=1208 rows=503) (actual time=0.477..4.63 rows=745 loops=1)
                -> Nested loop inner join  (cost=680 rows=1508) (actual time=0.121..2.36 rows=1508 loops=1)
                    -> Covering index scan on b using idx_booking_saved  (cost=152 rows=1508) (actual time=0.0963..0.52 rows=1508 loops=1)
                    -> Single-row index lookup on u using PRIMARY (UserId=b.UserID)  (cost=0.25 rows=1) (actual time=0.00101..0.00104 rows=1 loops=1508)
                -> Filter: (f.FlightPrice > (select #2))  (cost=0.25 rows=0.333) (actual time=0.00132..0.00137 rows=0.494 loops=1508)
                    -> Single-row index lookup on f using PRIMARY (FlightID=b.FlightID)  (cost=0.25 rows=1) (actual time=876e-6..904e-6 rows=1 loops=1508)
                    -> Select #2 (subquery in condition; run only once)
                        -> Aggregate: avg(Flight.FlightPrice)  (cost=201 rows=1) (actual time=0.321..0.321 rows=1 loops=1)
                            -> Table scan on Flight  (cost=101 rows=1000) (actual time=0.0456..0.251 rows=1000 loops=1)
 |
+--------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---+
1 row in set (0.02 sec)
```

The computational cost for indexing on FlightPrice is different from the original query. For example, in the original query the first nested loop inner join had a cost of 1389 and after applying the index it had a cost of 1208. The second nested loop inner join, on the other hand, had the same cost before and after the index (680). This means that indexing on FlightPrice resulted in a slightly more efficient search. However, it is limited because flight prices are very unique and vary a lot from flight to flight. Therefore, in order to isolate flight prices, you would still need to scan through the entire database which means this wouldn't be a very efficient choice of index.

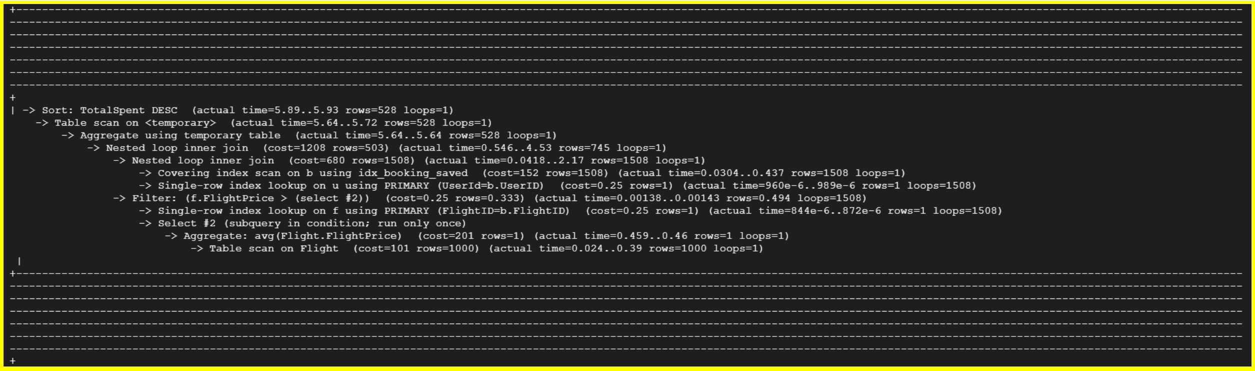
2. CREATE INDEX idx_users_userid ON Users(FirstName);



```
+--------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
+
| -> Sort: TotalSpent DESC  (actual time=6.13..6.17 rows=528 loops=1)
    -> Table scan on <temporary>  (actual time=5.84..5.91 rows=528 loops=1)
        -> Aggregate using temporary table  (actual time=5.83..5.83 rows=528 loops=1)
            -> Nested loop inner join  (cost=1208 rows=503) (actual time=0.363..4.7 rows=745 loops=1)
                -> Nested loop inner join  (cost=680 rows=1508) (actual time=0.0453..2.44 rows=1508 loops=1)
                    -> Covering index scan on b using idx_booking_saved  (cost=152 rows=1508) (actual time=0.0328..0.5 rows=1508 loops=1)
                    -> Single-row index lookup on u using PRIMARY (UserId=b.UserID)  (cost=0.25 rows=1) (actual time=0.00108..0.0011 rows=1 loops=1508)
                -> Filter: (f.FlightPrice > (select #2))  (cost=0.25 rows=0.333) (actual time=0.00131..0.00135 rows=0.494 loops=1508)
                    -> Single-row index lookup on f using PRIMARY (FlightID=b.FlightID)  (cost=0.25 rows=1) (actual time=883e-6..912e-6 rows=1 loops=1508)
                    -> Select #2 (subquery in condition; run only once)
                        -> Aggregate: avg(Flight.FlightPrice)  (cost=201 rows=1) (actual time=0.295..0.295 rows=1 loops=1)
                            -> Table scan on Flight  (cost=101 rows=1000) (actual time=0.0212..0.23 rows=1000 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
+
1 row in set (0.01 sec)
```

Indexing on FirstName resulted in a computational cost slightly different from the original query. It follows a similar pattern as the previous index. In the original query the first nested loop inner join had a cost of 1389 and after applying the index it had a cost of 1208. The second nested loop inner join, on the other hand, had the same cost before and after the index (680). This means that indexing on FirstName resulted in a slightly more efficient search. But, it is limited because just like flight prices, first names are also pretty unique. If our database doesn't have many people with the same first name, this index won't improve efficiency that much. In other words, you would still need to scan through the entire database to return the first names which means this wouldn't be a very efficient choice of index. That being said, this is most likely the best choice of index out of the three we have because first names are most likely to have the most overlap when compared to flight prices and last names (people can share common first names).

3. CREATE INDEX idx_users_firstname ON Users (LastName);



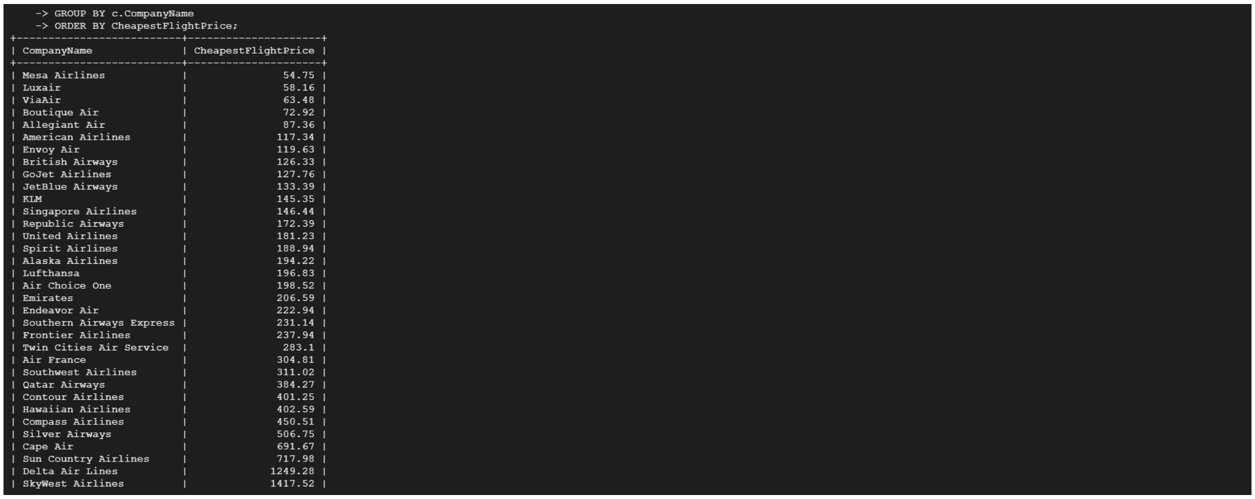
```
+--------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
+
| -> Sort: TotalSpent DESC  (actual time=5.89..5.93 rows=528 loops=1)
    -> Table scan on <temporary>  (actual time=5.64..5.72 rows=528 loops=1)
        -> Aggregate using temporary table  (actual time=5.64..5.64 rows=528 loops=1)
            -> Nested loop inner join  (cost=1208 rows=503) (actual time=0.546..4.53 rows=745 loops=1)
                -> Nested loop inner join  (cost=680 rows=1508) (actual time=0.0418..2.17 rows=1508 loops=1)
                    -> Covering index scan on b using idx_booking_saved  (cost=152 rows=1508) (actual time=0.0304..0.437 rows=1508 loops=1)
                    -> Single-row index lookup on u using PRIMARY (UserId=b.UserID)  (cost=0.25 rows=1) (actual time=960e-6..989e-6 rows=1 loops=1508)
                -> Filter: (f.FlightPrice > (select #2))  (cost=0.25 rows=0.333) (actual time=0.00138..0.00143 rows=0.494 loops=1508)
                    -> Single-row index lookup on f using PRIMARY (FlightID=b.FlightID)  (cost=0.25 rows=1) (actual time=844e-6..872e-6 rows=1 loops=1508)
                    -> Select #2 (subquery in condition; run only once)
                        -> Aggregate: avg(Flight.FlightPrice)  (cost=201 rows=1) (actual time=0.459..0.46 rows=1 loops=1)
                            -> Table scan on Flight  (cost=101 rows=1000) (actual time=0.024..0.39 rows=1000 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
+
```

Indexing on LastName resulted in a computational cost slightly different from the original query. It follows a similar pattern as the previous index. In the original query the first nested loop inner join had a cost of 1389 and after applying the index it had a cost of 1208. The second nested loop inner join, on the other hand, had the same cost before and after the index (680). This means that indexing on LastName resulted in a slightly more efficient search. It is limited because just like the first name, last names are very unique and can vary from user to user. If our database doesn't have many people with the same last name (which is common), this index won't improve efficiency that much. In other words, you would still need to scan through the entire database to return the last names which means this wouldn't be a very efficient choice of index.

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;

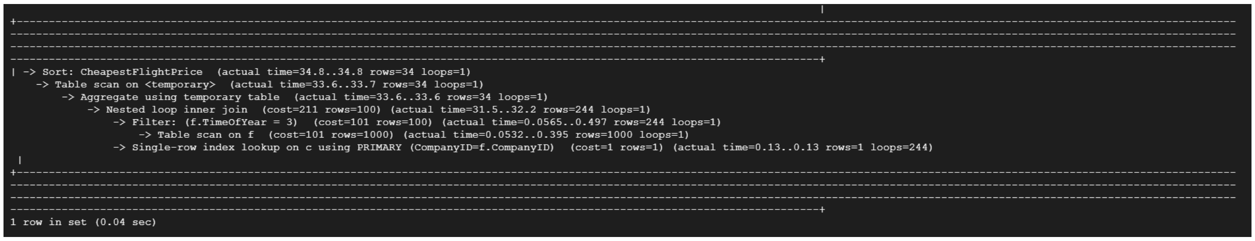
Query Execution:



```
    -> 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:

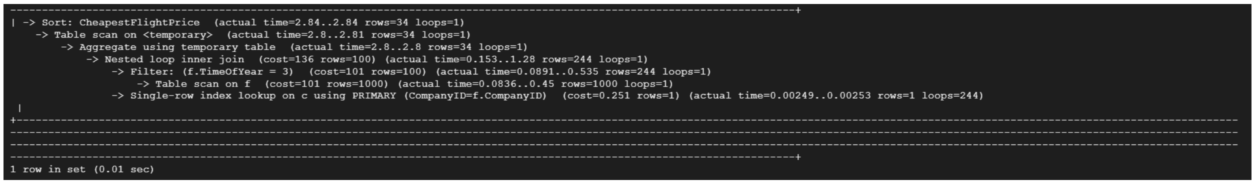
Before indexing:



```
                                                                                                                               |
+----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------+
| -> Sort: CheapestFlightPrice  (actual time=34.8..34.8 rows=34 loops=1)
    -> Table scan on <temporary>  (actual time=33.6..33.7 rows=34 loops=1)
        -> Aggregate using temporary table  (actual time=33.6..33.6 rows=34 loops=1)
            -> Nested loop inner join  (cost=211 rows=100) (actual time=31.5..32.2 rows=244 loops=1)
                -> Filter: (f.TimeOfYear = 3)  (cost=101 rows=100) (actual time=0.0565..0.497 rows=244 loops=1)
                    -> Table scan on f  (cost=101 rows=1000) (actual time=0.0532..0.395 rows=1000 loops=1)
                -> Single-row index lookup on c using PRIMARY (CompanyID=f.CompanyID)  (cost=1 rows=1) (actual time=0.13..0.13 rows=1 loops=244)
 |
+----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.04 sec)
```

Using the following three index commands:

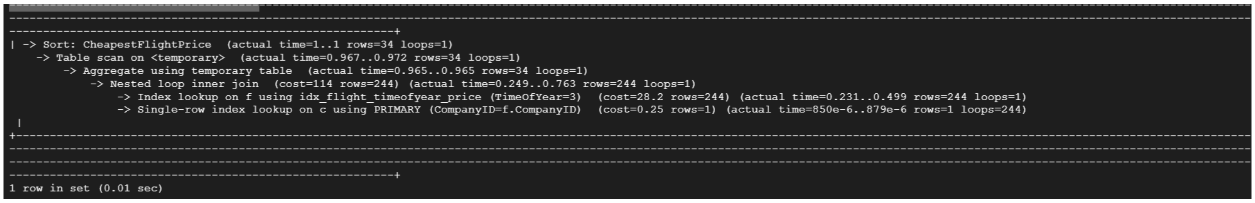
1. CREATE INDEX idx_flight_price ON Flight (FlightPrice);



```
--------------------------------------------------------------------------------------------------------------------+
| -> 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)
```

Indexing on FlightPrice resulted in a more efficient query, with a decreased computational cost. For the first nested loop inner join, the original query had a cost of 211, whereas the query after indexing had a cost of 136. The filter for TimeOfYear = 3 had a computational cost of 101 for before and after the index. The decrease in computation cost suggests that indexing on FlightPrice improves the speed of the query and helps narrow down the search by location relevant rows based on their price. This could be because of the MIN(FlightPrice) which determines the minimum cost across all flight prices. For the filter, there was no change, but this is expected, as we must simply get rid of all rows that do not match the required criteria which is a process that won't be made more efficient by indexing of FlightPrice.

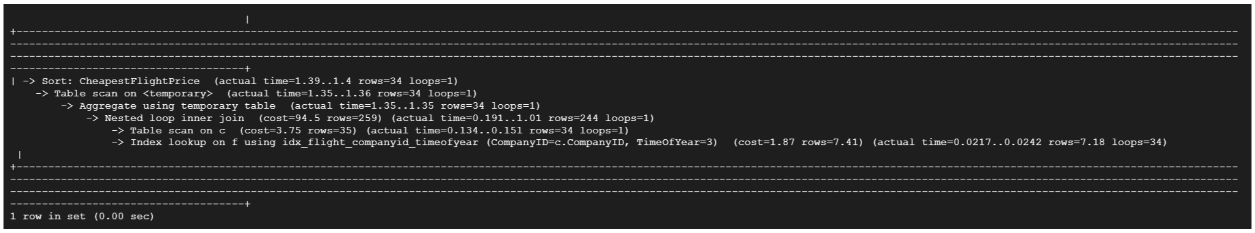
2. CREATE INDEX idx_flight_timeofyear_price ON Flight (TimeOfYear, FlightPrice);



```
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------+
| -> 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);

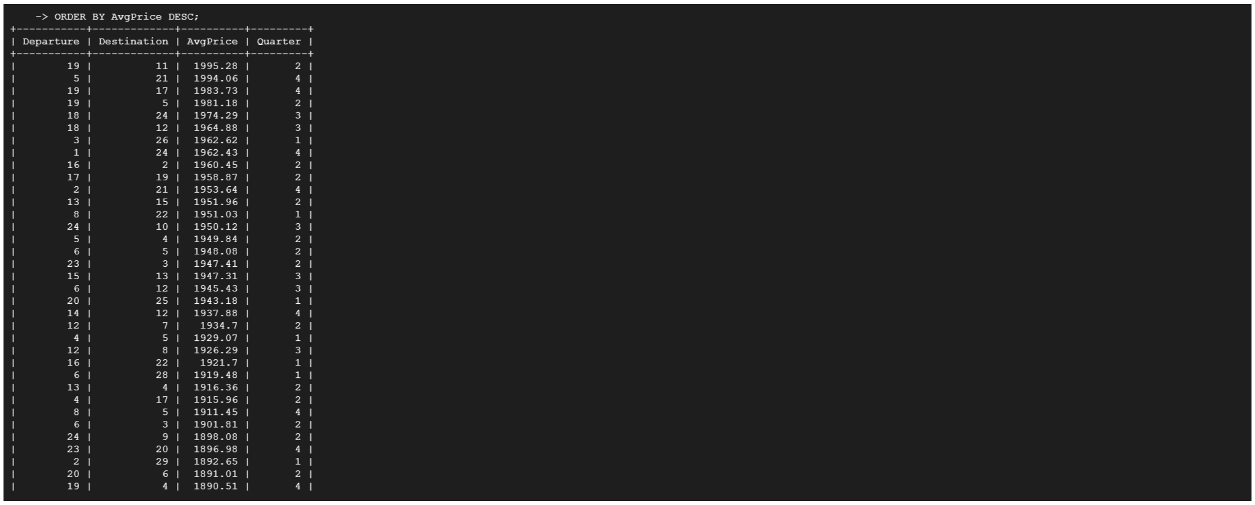


```
                                      |
+------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------+
| -> 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. This index provides the best decrease in cost and should be used.

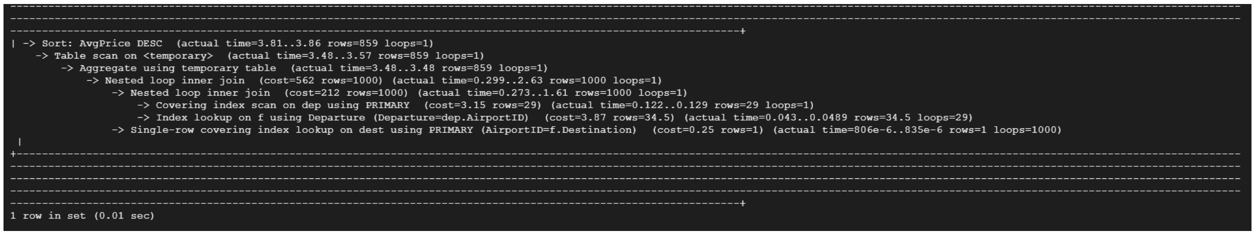
SQL Command 3
// calculates the average flight price for each departure to destination grouped by quarter
// (time of year) and sorts them from prices low to high
SELECT f.Departure, f.Destination, ROUND(AVG(f.FlightPrice), 2) AS AvgPrice, f.TimeOfYear AS Quarter
FROM Flight f
JOIN Airport dep ON f.Departure = 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 |
```

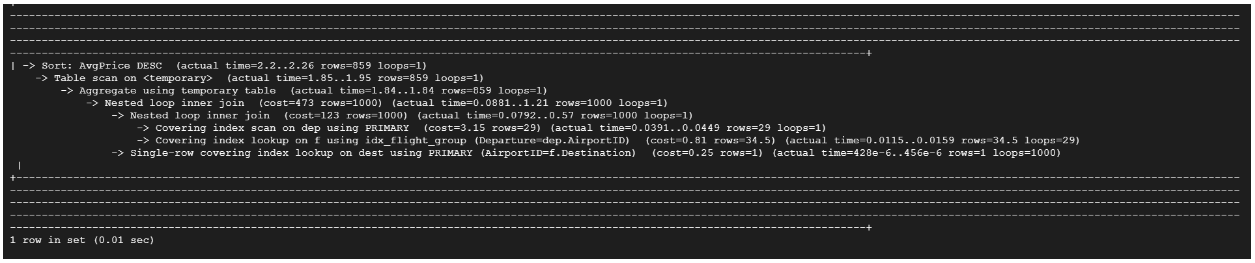
Indexing Analysis:
Before indexing:



```
-------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------+
| -> 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:

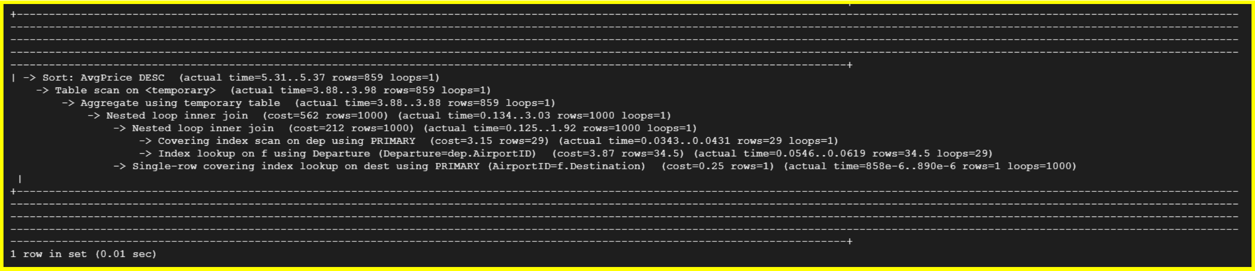
1. CREATE INDEX idx_flight_group ON Flight (Departure, Destination, TimeOfYear, FlightPrice);



```
---------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------+
| -> Sort: AvgPrice DESC  (actual time=2.2..2.26 rows=859 loops=1)
    -> Table scan on <temporary>  (actual time=1.85..1.95 rows=859 loops=1)
        -> Aggregate using temporary table  (actual time=1.84..1.84 rows=859 loops=1)
            -> Nested loop inner join  (cost=473 rows=1000) (actual time=0.0881..1.21 rows=1000 loops=1)
                -> Nested loop inner join  (cost=123 rows=1000) (actual time=0.0792..0.57 rows=1000 loops=1)
                    -> Covering index scan on dep using PRIMARY  (cost=3.15 rows=29) (actual time=0.0391..0.0449 rows=29 loops=1)
                    -> Covering index lookup on f using idx_flight_group (Departure=dep.AirportID)  (cost=0.81 rows=34.5) (actual time=0.0115..0.0159 rows=34.5 loops=29)
                -> Single-row covering index lookup on dest using PRIMARY (AirportID=f.Destination)  (cost=0.25 rows=1) (actual time=428e-6..456e-6 rows=1 loops=1000)
 |
+--------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

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. Since this index provided a decrease in computational cost, it is the best choice and should be used, as the other two indices do not.

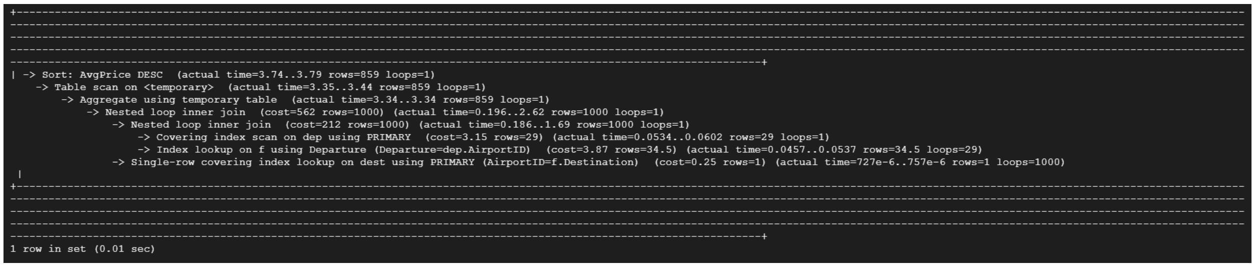
2. CREATE INDEX idx_flight_timeofyear ON Flight (TimeOfYear);



```
+--------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------+
| -> Sort: AvgPrice DESC  (actual time=5.31..5.37 rows=859 loops=1)
    -> Table scan on <temporary>  (actual time=3.88..3.98 rows=859 loops=1)
        -> Aggregate using temporary table  (actual time=3.88..3.88 rows=859 loops=1)
            -> Nested loop inner join  (cost=562 rows=1000) (actual time=0.134..3.03 rows=1000 loops=1)
                -> Nested loop inner join  (cost=212 rows=1000) (actual time=0.125..1.92 rows=1000 loops=1)
                    -> Covering index scan on dep using PRIMARY  (cost=3.15 rows=29) (actual time=0.0343..0.0431 rows=29 loops=1)
                    -> Index lookup on f using Departure (Departure=dep.AirportID)  (cost=3.87 rows=34.5) (actual time=0.0546..0.0619 rows=34.5 loops=29)
                -> Single-row covering index lookup on dest using PRIMARY (AirportID=f.Destination)  (cost=0.25 rows=1) (actual time=858e-6..890e-6 rows=1 loops=1000)
 |
+--------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

Indexing on TimeOfYear 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 TimeOfYear 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, and although the GROUP BY does operate on TimeOfYear, the query is not made more efficient by indexing on it , as 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);



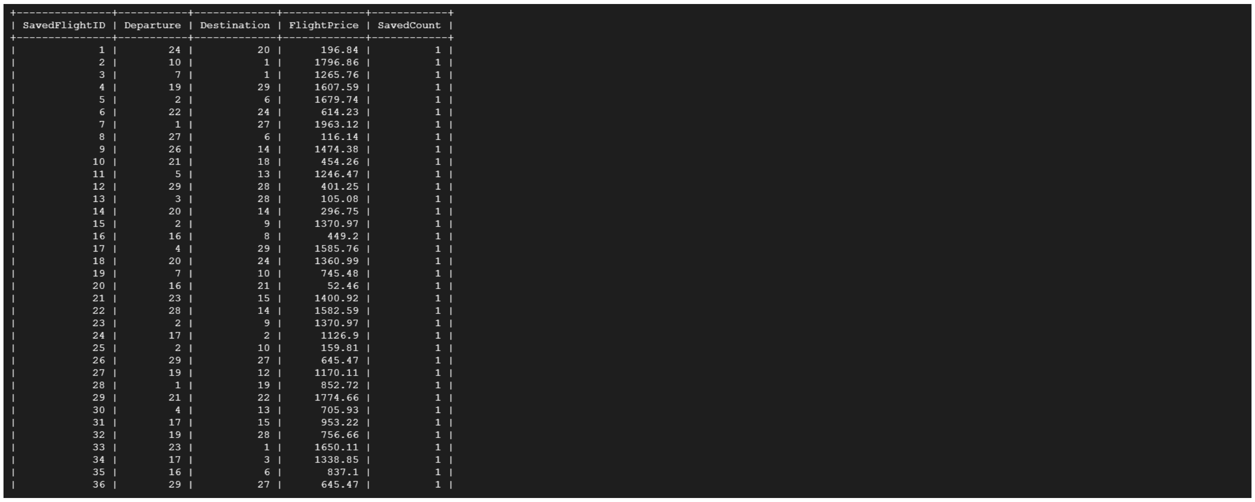
```
+---------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------+
| -> Sort: AvgPrice DESC  (actual time=3.74..3.79 rows=859 loops=1)
    -> Table scan on <temporary>  (actual time=3.35..3.44 rows=859 loops=1)
        -> Aggregate using temporary table  (actual time=3.34..3.34 rows=859 loops=1)
            -> Nested loop inner join  (cost=562 rows=1000) (actual time=0.196..2.62 rows=1000 loops=1)
                -> Nested loop inner join  (cost=212 rows=1000) (actual time=0.186..1.69 rows=1000 loops=1)
                    -> Covering index scan on dep using PRIMARY  (cost=3.15 rows=29) (actual time=0.0534..0.0602 rows=29 loops=1)
                    -> Index lookup on f using Departure (Departure=dep.AirportID)  (cost=3.87 rows=34.5) (actual time=0.0457..0.0537 rows=34.5 loops=29)
                -> Single-row covering index lookup on dest using PRIMARY (AirportID=f.Destination)  (cost=0.25 rows=1) (actual time=727e-6..757e-6 rows=1 loops=1000)
 |
+---------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

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.

<u>SQL Commands 4</u>

```
// 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;
```

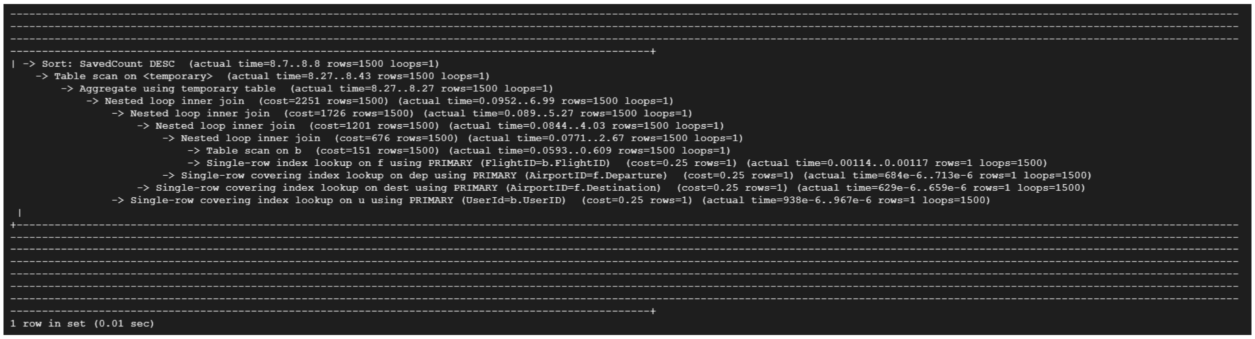
Query Execution:



```
+---------------+-----------+-------------+-------------+------------+
| 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:

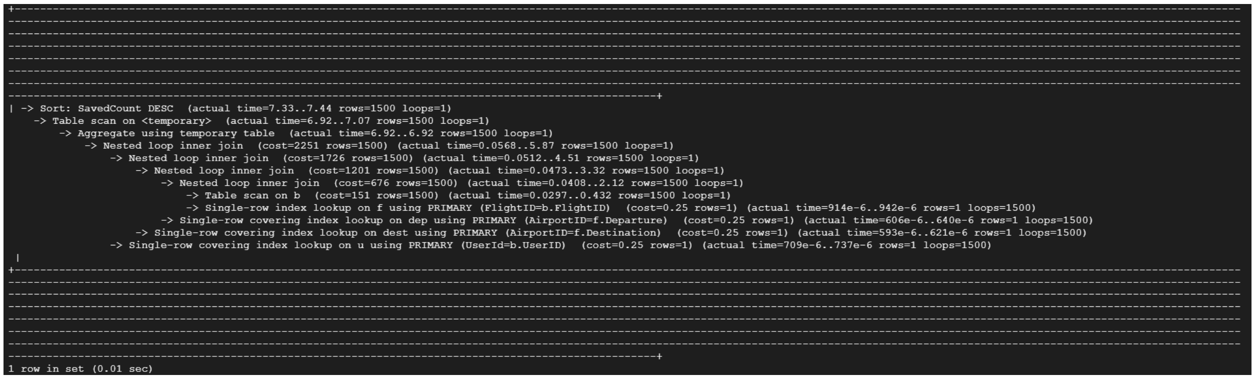
Before indexing:’



```
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
------------------------------------------------------------------------+
| -> 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:

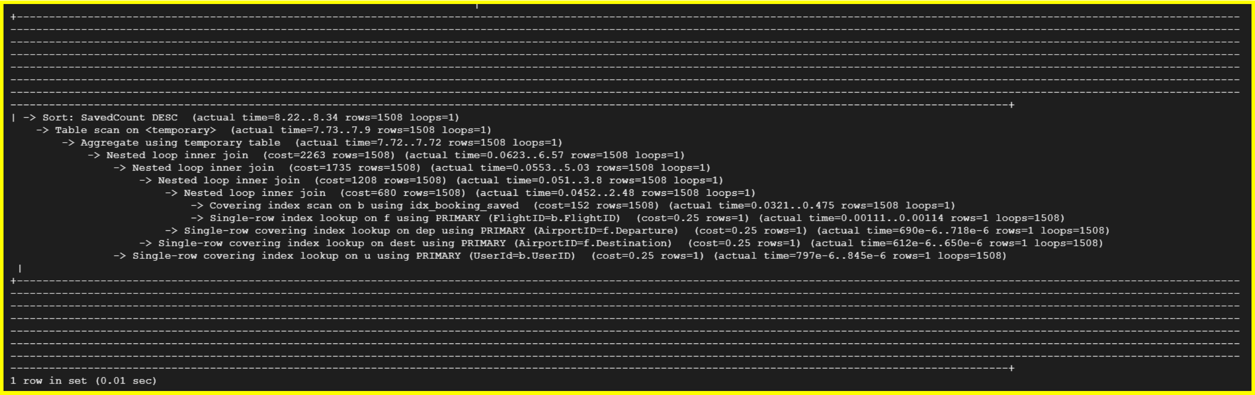
1. CREATE INDEX idx_flight_departure_destination_price ON Flight (Departure, Destination, FlightPrice);



```
+---------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
------------------------------------------------------------------------+
| -> 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. Because other indexes are even higher than the original, we should use this index (or no index at all) due to the high volume of entries we are joining and querying.

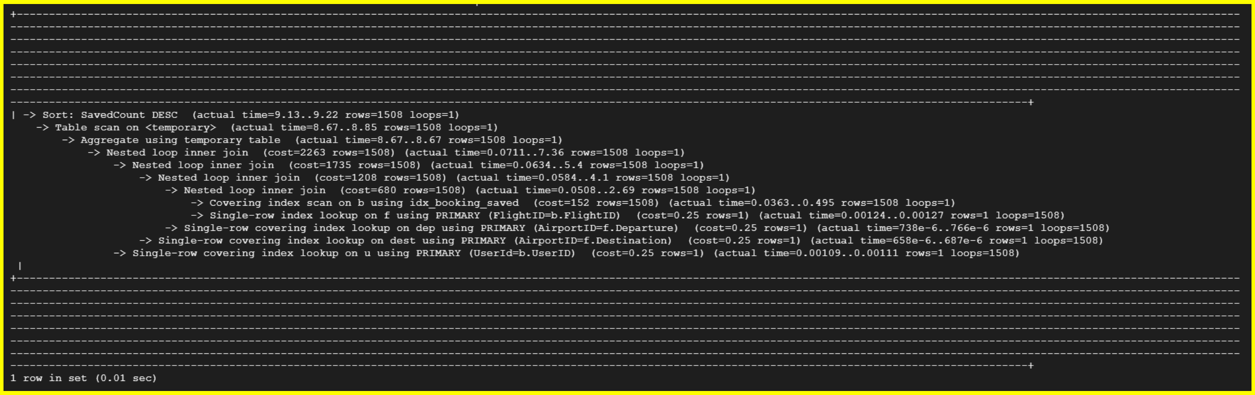
2. CREATE INDEX idx_departure ON Flight(Departure);



```
+-----------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------+
| -> Sort: SavedCount DESC  (actual time=8.22..8.34 rows=1508 loops=1)
    -> Table scan on <temporary>  (actual time=7.73..7.9 rows=1508 loops=1)
        -> Aggregate using temporary table  (actual time=7.72..7.72 rows=1508 loops=1)
            -> Nested loop inner join  (cost=2263 rows=1508) (actual time=0.0623..6.57 rows=1508 loops=1)
                -> Nested loop inner join  (cost=1735 rows=1508) (actual time=0.0553..5.03 rows=1508 loops=1)
                    -> Nested loop inner join  (cost=1208 rows=1508) (actual time=0.051..3.8 rows=1508 loops=1)
                        -> Nested loop inner join  (cost=680 rows=1508) (actual time=0.0452..2.48 rows=1508 loops=1)
                            -> Covering index scan on b using idx_booking_saved  (cost=152 rows=1508) (actual time=0.0321..0.475 rows=1508 loops=1)
                            -> Single-row index lookup on f using PRIMARY (FlightID=b.FlightID)  (cost=0.25 rows=1) (actual time=0.00111..0.00114 rows=1 loops=1508)
                        -> Single-row covering index lookup on dep using PRIMARY (AirportID=f.Departure)  (cost=0.25 rows=1) (actual time=690e-6..718e-6 rows=1 loops=1508)
                    -> Single-row covering index lookup on dest using PRIMARY (AirportID=f.Destination)  (cost=0.25 rows=1) (actual time=612e-6..650e-6 rows=1 loops=1508)
                -> Single-row covering index lookup on u using PRIMARY (UserId=b.UserID)  (cost=0.25 rows=1) (actual time=797e-6..845e-6 rows=1 loops=1508)
 |
+-----------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

The index on Flight(Departure) made the cost even higher than the original query without indexing (2263 vs 2251 for the original). This is most likely due to the fact that a lot of rows are selected and joined and arbitrarily indexing on only 1 of the attributes can make the process of joining a little more unpredictable in terms of accessing certain keys for the joins (less sequential). Since the query touches a lot of rows, using the index actually made things slower compared to just scanning the table normally. In the end, the overhead of using the index wasn't worth it, so the total cost ended up being a little higher. For this reason, we should not use this index.

3. CREATE INDEX idx_destination ON Flight(Destination);



```
+-----------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------+
| -> Sort: SavedCount DESC  (actual time=9.13..9.22 rows=1508 loops=1)
    -> Table scan on <temporary>  (actual time=8.67..8.85 rows=1508 loops=1)
        -> Aggregate using temporary table  (actual time=8.67..8.67 rows=1508 loops=1)
            -> Nested loop inner join  (cost=2263 rows=1508) (actual time=0.0711..7.36 rows=1508 loops=1)
                -> Nested loop inner join  (cost=1735 rows=1508) (actual time=0.0634..5.4 rows=1508 loops=1)
                    -> Nested loop inner join  (cost=1208 rows=1508) (actual time=0.0584..4.1 rows=1508 loops=1)
                        -> Nested loop inner join  (cost=680 rows=1508) (actual time=0.0508..2.69 rows=1508 loops=1)
                            -> Covering index scan on b using idx_booking_saved  (cost=152 rows=1508) (actual time=0.0363..0.495 rows=1508 loops=1)
                            -> Single-row index lookup on f using PRIMARY (FlightID=b.FlightID)  (cost=0.25 rows=1) (actual time=0.00124..0.00127 rows=1 loops=1508)
                        -> Single-row covering index lookup on dep using PRIMARY (AirportID=f.Departure)  (cost=0.25 rows=1) (actual time=738e-6..766e-6 rows=1 loops=1508)
                    -> Single-row covering index lookup on dest using PRIMARY (AirportID=f.Destination)  (cost=0.25 rows=1) (actual time=658e-6..687e-6 rows=1 loops=1508)
                -> Single-row covering index lookup on u using PRIMARY (UserId=b.UserID)  (cost=0.25 rows=1) (actual time=0.00109..0.00111 rows=1 loops=1508)
 |
+-----------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

The index on Flight(Destination) actually made the query cost even higher than the original version without any indexing (2263 vs 2251). This is probably because a lot of rows are being selected and joined, and adding an index on just one attribute can make accessing the rows needed for the joins less predictable and less sequential. Since the query pulls in a lot of rows, using the index ended up making things slower than just doing a normal table scan. In the end, the extra work from the index wasn’t worth it, and the total cost ended up being a bit higher. For this reason, it’s better not to use this index.