Stage_3:

We have hosted our database on GCP and created and have been working with our tables in GCP:

```
mysql> show tables;
+----------------------+
| Tables_in_MONGOKINGS |
+----------------------+
| Booking_Reservations |
| Car_Rental_Info      |
| Car_Theft            |
| Customer_Info        |
| Insurance_Detail     |
| Rating_and_Reviews   |
+----------------------+
6 rows in set (0.01 sec)
```

Here, we can clearly see that our GCP is properly set up and we have made the necessary tables in the database with the given constraints and requirements described in the stage 2.

We have the DDL command for the following 6 tables here:

1.Booking_Reservations:

```
CREATE TABLE Booking_Reservations (
    Booking_Id INT NOT NULL,
    Booking_Duration INT NOT NULL,
    Car_Id INT NOT NULL,
    Customer_Id INT NOT NULL,
    Payment INT NOT NULL,
    start_date DATE,
    end_date DATE,
    PRIMARY KEY (Booking_Id),
    FOREIGN KEY (Car_Id)
        REFERENCES Car_Rental_Info (Car_Id)
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    FOREIGN KEY (Customer_Id)
        REFERENCES Customer_Info (Customer_Id)
        ON UPDATE CASCADE
        ON DELETE CASCADE
);
```

```
mysql> select count(*) from Booking_Reservations;
+----------+
| count(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.00 sec)
```

Here, the count of Booking_Reservations is 1000.

Car_Rental_Info:

```
CREATE TABLE Car_Rental_Info (
    Car_Id INT Primary Key,
    FuelType VARCHAR(255),
    State VARCHAR(255),
    City VARCHAR(255),
    Vehicle_Year INT,
    Number_of_trips INT,
    Vehicle_Type VARCHAR(255),
    Vehicle_Make VARCHAR(255),
    Vehicle_Model VARCHAR(50),
    Daily_Price INT,
);
```

```
mysql> select count(*) from Car_Rental_Info;
+----------+
| count(*) |
+----------+
|     3093 |
+----------+
1 row in set (0.01 sec)
```

Here, we have the count of elements in Car_Rental_Info as 3093.

Car_Theft:

```
CREATE TABLE Car_Theft (
    State VARCHAR(255) NOT NULL,
    Model_Make VARCHAR(255) NOT NULL,
    Year INT NOT NULL,
    Number_Thefts INT,
    Criminal_Id INT PRIMARY KEY
);
```

```
mysql> select count(*) from Car_Theft;
+----------+
| count(*) |
+----------+
|      509 |
+----------+
1 row in set (0.00 sec)
```

Herem the count of our table is 509.

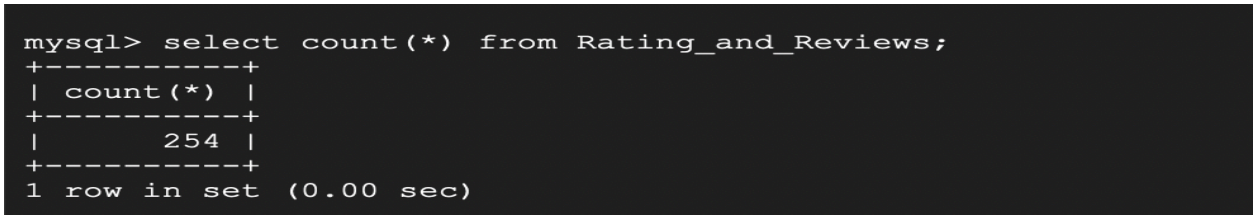
Customer_Info:

```
CREATE TABLE Customer_Info (
    Customer_Id INT Primary Key,
    Name VARCHAR(255),
    Phone BIGINT,
    City VARCHAR(255),
    Age INT,
    Number_of_rentals INT,
);
```

Here, we can see here that count(*) is 1000.

Rating_and_Reviews:

```
CREATE TABLE Rating_and_Reviews (
Booking_Id INT,
rating INT,
Review varchar(255),
date_published DATE,
date_modified DATE,
PRIMARY KEY(Booking_Id),
FOREIGN KEY(Booking_Id)
 REFERENCES Booking_Reservations (Booking_Id)
       ON UPDATE CASCADE
       ON DELETE CASCADE
);
```



Here, the count(*) of the entity is 254.

Insurance_val:

```
Create TABLE  Insurance_Detail (
Age int primary key,
Insurance_val int
);
```

```
mysql> select count(*) from Insurance_Detail;
+-----------+
| count(*)  |
+-----------+
|        60 |
+-----------+
1 row in set (0.00 sec)
```

Here, the count(*) of the entity is 60.

So, here we clearly have 3 entities having at least 1000 rows here.

Advanced queries and indexing:

1.Customer tries to find out the total payment, which product of number_of_days * daily price, car_theft data (group by state), along with a deposit if the that have greater than 25 thefts on average record for the state

```
SELECT br.Booking_Id, ci.Name AS Customer_Name, ci.Customer_Id, cri.State,
(br.Booking_Duration * cri.Daily_Price) AS Rental_Cost, ((br.Booking_Duration *
cri.Daily_Price) + 100 * COALESCE((SELECT 1
                    From Car_Theft sub_ct
                    Where cri.State = sub_ct.State
                     GROUP BY sub_ct.State
                    having avg(sub_ct.Number_Thefts) > 25), 0)
)  +  (SELECT id.Insurance_Val
    FROM Insurance_Detail id
    WHERE id.Age = ci.Age)  AS Total_Payment, (SELECT id.Insurance_Val
    FROM Insurance_Detail id
    WHERE id.Age = ci.Age) AS Insurance_val  FROM Booking_Reservations br
JOIN Car_Rental_Info cri ON br.Car_Id = cri.Car_Id
JOIN Customer_Info ci ON br.Customer_Id = ci.Customer_Id
WHERE ci.Customer_Id = 139256;
```

```
+------------+---------------+-------------+-------+-------------+---------------+---------------+
| Booking_Id | Customer_Name  | Customer_Id | State | Rental_Cost | Total_Payment | Insurance_val |
+------------+---------------+-------------+-------+-------------+---------------+---------------+
|     964090 | Edik Kingsmill |      139256 | OH    |         885 |          1002 |            17 |
|     965527 | Edik Kingsmill |      139256 | NC    |        3280 |          3397 |            17 |
+------------+---------------+-------------+-------+-------------+---------------+---------------+
2 rows in set (0.01 sec)
```

Our query gets the booking id and customer details in the form of customer name and customer id, while also calculating the total payment of a booking through getting the state at which the car rental will be from, the total payment of the booking, as well as the insurance value. We use multiple joins and subqueries in order to do this. One subquery that we use is to calculate whether we should increase the payment of the rental based on the number of thefts in the area, which we determine through finding if the number of thefts in the state is greater than 25 or not, with 25 being our threshold for whether we should add another $100 to the payment due to the possibility of needing more insurance. We also have another subquery for for the insurance value, which we need to determine the total payment value, and we obtain this directly from our Insurance_Detail table; this value is determined by the age of the person, so in this subquery we make sure that the age of the person is the same as the insurance value for the age in the Insurance_Detail table. We also have a third subquery for returning the insurance value as well, so that the user can know how much insurance is being factored in. Additionally, we use a JOIN for Car_Rental_Info, to find the matching car info for the car in the booking. We

also use another JOIN to get the related customer information for the car in the booking as well. We also use a WHERE clause for the customer id, which we use to find the bookings that have been made by this customer. We can use this WHERE clause to find different Customer_Id details as and when we need. As our output is based on the booking of the Customer we will have less than 15 entries in the output.

This is our cost for the command:

```
| -> Nested loop inner join  (cost=1.47 rows=2) (actual time=0.162..0.205 rows=2 loops=1)
    -> Index lookup on br using Customer_Id (Customer_Id=139256)  (cost=0.7 rows=2) (actual time=0.107..0.112 rows=2 loops=1)
    -> Single-row index lookup on cri using PRIMARY (Car_Id=br.Car_Id)  (cost=0.336 rows=1) (actual time=0.0334..0.0336 rows=1 loops=2)
-> Select #2 (subquery in projection; dependent)
    -> Filter: (avg(sub_ct.Number_Thefts) > 25)  (cost=10.9 rows=1) (actual time=1.55..1.55 rows=1 loops=2)
        -> Aggregate: avg(sub_ct.Number_Thefts)  (cost=10.9 rows=1) (actual time=0.779..0.78 rows=1 loops=2)
            -> Filter: (cri.State = sub_ct.State)  (cost=5.84 rows=50.9) (actual time=0.715..0.771 rows=10 loops=2)
                -> Table scan on sub_ct  (cost=5.84 rows=509) (actual time=0.115..0.271 rows=509 loops=2)
-> Select #3 (subquery in projection; dependent)
    -> Rows fetched before execution  (cost=0..0 rows=1) (actual time=90e-6..143e-6 rows=1 loops=2)
-> Select #4 (subquery in projection; dependent)
    -> Rows fetched before execution  (cost=0..0 rows=1) (actual time=62e-6..104e-6 rows=1 loops=2)
 |
```

There are ways to reduce the cost because we are using nested loop inner joins as well as some subqueries that may not be necessary.

a. CREATE INDEX idx_car_theft_state ON Car_Theft(State);

```
| -> Nested loop inner join  (cost=2.69 rows=2) (actual time=0.067..0.0998 rows=2 loops=1)
    -> Index lookup on br using Customer_Id (Customer_Id=139256)  (cost=0.7 rows=2) (actual time=0.0456..0.05 rows=2 loops=1)
    -> Single-row index lookup on cri using PRIMARY (Car_Id=br.Car_Id)  (cost=0.943 rows=1) (actual time=0.0227..0.0229 rows=1 loops=2)
-> Select #2 (subquery in projection; dependent)
    -> Filter: (avg(sub_ct.Number_Thefts) > 25)  (cost=4.25 rows=3.16) (actual time=0.426..0.426 rows=1 loops=2)
        -> Group aggregate: avg(sub_ct.Number_Thefts)  (cost=4.25 rows=3.16) (actual time=0.0805..0.081 rows=1 loops=2)
            -> Index lookup on sub_ct using idx_car_theft_state (State=cri.State)  (cost=3.25 rows=9.98) (actual time=0.0503..0.0629 rows=10 loops=2)
-> Select #3 (subquery in projection; dependent)
    -> Rows fetched before execution  (cost=0..0 rows=1) (actual time=73e-6..118e-6 rows=1 loops=2)
-> Select #4 (subquery in projection; dependent)
    -> Rows fetched before execution  (cost=0..0 rows=1) (actual time=54e-6..80.5e-6 rows=1 loops=2)
 |
```

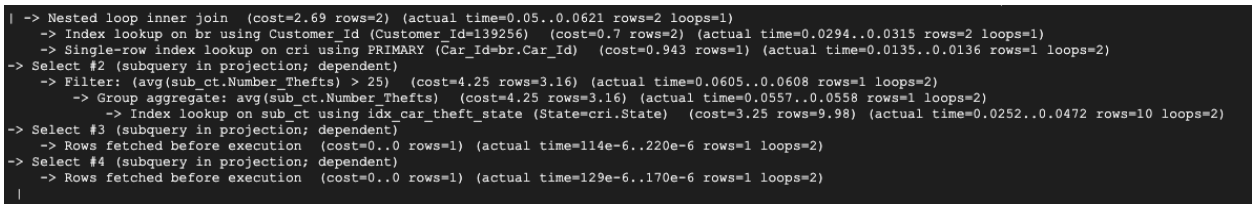
As we can see, creating this index dropped the subquery cost by a significant amount. We can see that the group aggregate now has a cost of 4.25 opposed to the 10.9 cost from before. Additionally, the cost of the index lookup (3.25) now replaces the previous filter (5.84) and table scan (5.84) costs.This is because sub_ct.Number_Thefts now only explores 9.98 rows, when before it was exploring 509 because it was doing a table scan, while now we're directly indexing, showing the improved efficiency of this query. This improvement is due to the fact that before, the query had to go through and explore all of the different Car_Theft values to see which, if any, were greater than 25. However, now the query can directly go through and index the values that represent this condition, making the query a lot more efficient.

b. CREATE INDEX idx_customer_age ON Customer_Info(Age);

```
| -> Nested loop inner join  (cost=2.69 rows=2) (actual time=0.387..0.487 rows=2 loops=1)
    -> Index lookup on br using Customer_Id (Customer_Id=139256)  (cost=0.7 rows=2) (actual time=0.264..0.274 rows=2 loops=1)
    -> Single-row index lookup on cri using PRIMARY (Car_Id=br.Car_Id)  (cost=0.943 rows=1) (actual time=0.0754..0.0791 rows=1 loops=2)
-> Select #2 (subquery in projection; dependent)
    -> Filter: (avg(sub_ct.Number_Thefts) > 25)  (cost=4.25 rows=3.16) (actual time=0.289..0.29 rows=1 loops=2)
        -> Group aggregate: avg(sub_ct.Number_Thefts)  (cost=4.25 rows=3.16) (actual time=0.273..0.273 rows=1 loops=2)
            -> Index lookup on sub_ct using idx_car_theft_state (State=cri.State)  (cost=3.25 rows=9.98) (actual time=0.117..0.216 rows=10 loops=2)
-> Select #3 (subquery in projection; dependent)
    -> Rows fetched before execution  (cost=0..0 rows=1) (actual time=0.00498..0.00503 rows=1 loops=2)
-> Select #4 (subquery in projection; dependent)
    -> Rows fetched before execution  (cost=0..0 rows=1) (actual time=64.5e-6..96e-6 rows=1 loops=2)
 |
```

With this index, the goal was to index directly on the age of the customer, so that we would only have the relevant ages that would somehow affect the query. However, there seems to be no change in cutting cost for our query. This could be due to the fact that age and Insurance_Val are correlated, and since Number_Thefts is also correlated with Insurance_Val and already fully optimized, a speed up to age wouldn't necessarily change the optimization of this query. In other words, age is already highly optimized, and since now we have all of the indices for the relevant Car_Theft values (where the value is greater than 25), this indexing doesn't change the cost of the query. Aside from this however, the indexing of age would mean that we can index for the age that is directly equal to the corresponding Insurance_Val, but the fact that the query isn't cutting cost means that all of the possible optimizations have already been done to the age-related subqueries.

c. CREATE INDEX idx_customer_name ON Customer_Info(Name);



With the index for customer name, the goal was to cut cost in finding the relevant name in the query, allowing for more general improved efficiency. However, this also ended up not being the case for this query for similar reasons to the last query; all of these subqueries and the join are already optimized, so even trying to make a change like this will not result in a cutting of cost because the necessary indexing to cut as much cost as possible has already been applied.

Similarly, indexing on Car_Theft(Number_Theft) does not reduce the cost. So, for this query indexing on Car_Theft(State) produced the best result.

2. Based on the ratings of the car, provide the best possible option for the customer:

```
SELECT cri.Car_Id, cri.City, cri.Vehicle_Type, cri.Daily_Price, AVG(rr.Rating) AS
Average_Rating
FROM Booking_Reservations br JOIN Rating_and_Reviews rr ON br.Booking_Id =
rr.Booking_id JOIN Car_Rental_Info cri ON br.Car_Id = cri.Car_Id
WHERE cri.City = 'Atlanta'
GROUP BY cri.Car_Id
HAVING Average_Rating >= 3
ORDER BY Average_Rating DESC;
```

The command consists of multiple joins, a group by clause, and a having clause and is used to find the best possible car option in Atlanta, or any other cities, based on customer ratings. The query retrieves details like Car_Id, City, Vehicle_Type, and Daily_Price for users' convenience in choosing. By grouping the data by Car_Id, we calculate the average rating for each car and then use the having clause to filter out any car with an average rating of 3 or below. The results are also ordered in descending order of the average rating, ensuring that the highest-rated cars appear at the top of the list. It's useful for users who believe other users' ratings and want the top-rated car options in desired cities.

Car_Id	City	Vehicle_Type	Daily_Price	Average_Rating
7678058	Atlanta	car	250	6.0000
9451897	Atlanta	suv	140	6.0000
1103338	Atlanta	car	300	6.0000
5801653	Atlanta	car	153	6.0000
6830247	Atlanta	suv	172	6.0000
15071013	Atlanta	suv	105	6.0000
499566	Atlanta	car	189	5.2500
10167863	Atlanta	suv	84	5.0000
1886829	Atlanta	suv	299	5.0000
3659271	Atlanta	car	99	5.0000
15406472	Atlanta	suv	150	5.0000
5878338	Atlanta	car	130	5.0000
8814550	Atlanta	car	87	5.0000
6580855	Atlanta	car	65	4.6667
2052844	Atlanta	suv	85	4.5000

15 rows in set (0.01 sec)

Original command has the following cost:

```
| -> Sort: Average_Rating DESC  (actual time=1.74..1.74 rows=36 loops=1)
    -> Filter: (Average_Rating >= 3)  (actual time=1.67..1.69 rows=36 loops=1)
        -> Table scan on <temporary>  (actual time=1.66..1.67 rows=53 loops=1)
            -> Aggregate using temporary table  (actual time=1.66..1.66 rows=53 loops=1)
                -> Nested loop inner join  (cost=213 rows=25.4) (actual time=0.207..1.52 rows=97 loops=1)
                    -> Nested loop inner join  (cost=115 rows=254) (actual time=0.143..0.777 rows=254 loops=1)
                        -> Table scan on rr  (cost=26.2 rows=254) (actual time=0.104..0.138 rows=254 loops=1)
                        -> Single-row index lookup on br using PRIMARY (Booking_Id=rr.Booking_Id)  (cost=0.25 rows=1) (actual time=0.00205..0.00208 rows=1 loops=254)
                    -> Filter: (cri.City = 'Atlanta')  (cost=0.286 rows=0.1) (actual time=0.00276..0.0028 rows=0.382 loops=254)
                        -> Single-row index lookup on cri using PRIMARY (Car_Id=br.Car_Id)  (cost=0.286 rows=1) (actual time=0.00246..0.00249 rows=1 loops=254)
|
```

We could notice huge costs due to nested loop inner joins.

a.Create index cri_city on Car_Rental_Info(City)

```
| -> Sort: Average_Rating DESC  (actual time=1.99..2 rows=36 loops=1)
    -> Filter: (Average_Rating >= 3)  (actual time=1.4..1.93 rows=36 loops=1)
        -> Stream results  (cost=81.9 rows=87.7) (actual time=1.26..1.91 rows=53 loops=1)
            -> Group aggregate: avg(rr.rating)  (cost=81.9 rows=87.7) (actual time=0.216..0.826 rows=53 loops=1)
                -> Nested loop inner join  (cost=73.1 rows=87.7) (actual time=0.152..0.72 rows=97 loops=1)
                    -> Nested loop inner join  (cost=42.5 rows=87.7) (actual time=0.137..0.503 rows=97 loops=1)
                        -> Index lookup on cri using cri_city (City='Atlanta')  (cost=20.4 rows=53) (actual time=0.117..0.297 rows=53 loops=1)
                        -> Covering index lookup on br using idx_booking_car_id (Car_Id=cri.Car_Id)  (cost=0.253 rows=1.65) (actual time=0.00248..0.0036 rows=1.83 loops=53)
                    -> Single-row index lookup on rr using PRIMARY (Booking_Id=br.Booking_Id)  (cost=0.251 rows=1) (actual time=0.00199..0.00203 rows=1 loops=97)
|
```

Initially, the command needs to perform a full table scan to locate the desired records where City matches. We could notice an expensive nested loop joins in costs and the use of a temporary table for aggregating the ratings. After creating the index on Car_Rental_Info(City), the cost dropped significantly from 213+115+26.2+... to 81.9+81.9+73.1+42.5+20.4.... Indexing allows a quicker filter for 'Atlanta' records. It significantly reduced the number of rows processed during the join and grouping steps. The nested loop join on Car_Rental_Info particularly has a lower cost.

b. Create index cri_type on Car_Rental_Info(Vehicle_Type)

```
| -> Sort: Average_Rating DESC  (actual time=2.74..2.74 rows=36 loops=1)
    -> Filter: (Average_Rating >= 3)  (actual time=2.68..2.7 rows=36 loops=1)
        -> Table scan on <temporary>  (actual time=2.67..2.67 rows=53 loops=1)
            -> Aggregate using temporary table  (actual time=2.66..2.66 rows=53 loops=1)
                -> Nested loop inner join  (cost=213 rows=25.4) (actual time=0.185..1.55 rows=97 loops=1)
                    -> Nested loop inner join  (cost=115 rows=254) (actual time=0.139..0.81 rows=254 loops=1)
                        -> Table scan on rr  (cost=26.2 rows=254) (actual time=0.0989..0.14 rows=254 loops=1)
                        -> Single-row index lookup on br using PRIMARY (Booking_Id=rr.Booking_Id)  (cost=0.25 rows=1) (actual time=0.00232..0.00244 rows=1 loops=254)
                    -> Filter: (cri.City = 'Atlanta')  (cost=0.286 rows=0.1) (actual time=0.00274..0.00278 rows=0.382 loops=254)
                        -> Single-row index lookup on cri using PRIMARY (Car_Id=br.Car_Id)  (cost=0.286 rows=1) (actual time=0.00242..0.00245 rows=1 loops=254)
|
```

This index doesn’t help in overall cost at all. It might be because the filter condition is on city and the joins are based on Car_Id. Therefore, the new index on Vehicle_Type is not being utilized in the query execution. Also it can be slower as an added index may introduce some overhead in the decision-making. Since it will not contribute to filtering or join improvements, this may led to increase in execution time.

c.Create index review_rating on Rating_and_Reviews(Rating)

```
| -> Sort: Average_Rating DESC  (actual time=3.16..3.17 rows=36 loops=1)
    -> Filter: (Average_Rating >= 3)  (actual time=3.09..3.11 rows=36 loops=1)
        -> Table scan on <temporary>  (actual time=1.59..1.61 rows=53 loops=1)
            -> Aggregate using temporary table  (actual time=1.59..1.59 rows=53 loops=1)
                -> Nested loop inner join  (cost=213 rows=25.4) (actual time=0.136..1.45 rows=97 loops=1)
                    -> Nested loop inner join  (cost=115 rows=254) (actual time=0.0878..0.687 rows=254 loops=1)
                        -> Covering index scan on rr using review_rating  (cost=26.2 rows=254) (actual time=0.0656..0.0989 rows=254 loops=1)
                        -> Single-row index lookup on br using PRIMARY (Booking_Id=rr.Booking_Id)  (cost=0.25 rows=1) (actual time=0.00211..0.00214 rows=1 loops=254)
                    -> Filter: (cri.City = 'Atlanta')  (cost=0.286 rows=0.1) (actual time=0.00284..0.00287 rows=0.382 loops=254)
                        -> Single-row index lookup on cri using PRIMARY (Car_Id=br.Car_Id)  (cost=0.286 rows=1) (actual time=0.00253..0.00256 rows=1 loops=254)
|
```

This index doesn’t help reduce cost as the query's condition is on cri.City, which relies on index cri_City and not on the Rating values in the table. Also, the query uses grouping to find the average rating and sorting by Average_Rating. These operations use a temporary table where

all the grouped results are stored and then sorted. The cost of these operations tends to dominate, so even if the index on Rating speeds up the access on that particular table, it doesn't have much influence on the overall execution cost.

Therefore, index cri_city would be the choice to improve the performance of costs, given reasons mentioned above.

3. Based on number of previous rentals and also if he has rented a particular car before, we can decide to give him a special discount:

```
SELECT  c.Customer_Id, c.Name, car.Car_Id,
car.Vehicle_Make, car.Vehicle_Model, car.Daily_Price,
car.Daily_Price * (1 -  0.09 *  exists (SELECT 1 from Booking_Reservations b
WHERE b.Customer_Id = c.Customer_Id and b.Car_Id  = car.Car_Id and b.Booking_Duration >
5)) as discount_price
FROM Customer_Info c
LEFT JOIN Booking_Reservations br on c.Customer_Id = br.Customer_Id
LEFT JOIN Car_Rental_Info car on car.Car_Id = br.Car_Id where car.Car_Id = 21255  and
c.Number_of_Rentals > 1;
```

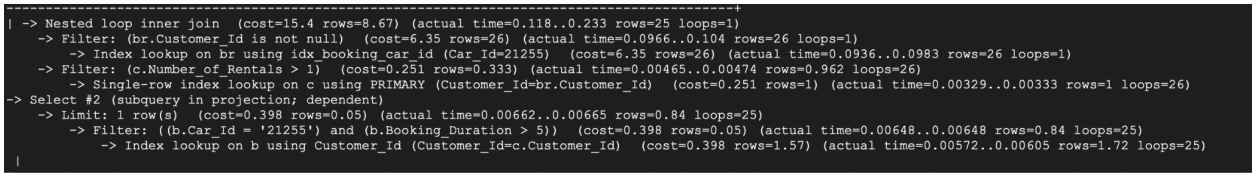
Here is what the query returns for a particular Car_Id.

Customer_Id	Name	Car_Id	Vehicle_Make	Vehicle_Model	Daily_Price	discount_price
164463	Johnna Rennenbach	21255	Dodge	Challenger	80	72.80
112430	Delano Coopman	21255	Dodge	Challenger	80	72.80
150751	Lyndy Gittoes	21255	Dodge	Challenger	80	72.80
177239	Terri Drain	21255	Dodge	Challenger	80	80.00
174275	Myer Duesbury	21255	Dodge	Challenger	80	72.80
182588	Jamil Welberry	21255	Dodge	Challenger	80	72.80
122350	Thor Adamowitz	21255	Dodge	Challenger	80	72.80
175161	Reinaldos Bearcroft	21255	Dodge	Challenger	80	80.00
168549	Merrily Benitti	21255	Dodge	Challenger	80	72.80
193132	Cherye Vanes	21255	Dodge	Challenger	80	80.00
126277	Rora Prator	21255	Dodge	Challenger	80	72.80
133106	Tobias Lewson	21255	Dodge	Challenger	80	72.80
160736	Arlette Kislingbury	21255	Dodge	Challenger	80	72.80
177254	Tessy Hoggins	21255	Dodge	Challenger	80	72.80
139256	Edik Kingsmill	21255	Dodge	Challenger	80	72.80

When a customer wants to rent a new car, we decide to give him a discount based on certain conditions. This query gives us customer and car rental details, by using a conditional discount based on the previous number of bookings of the customer. It selects the customer's ID and name from the Customer_Info table, along with the car_Id, Car.Vehicle_Make, Car.Vehicle_Model, and Car.Daily_price from Car_Rental_Info. A left join is used to join Customer_Info with Booking_Reservations on Customer_Id, ensuring that all customers meeting the criteria are taken in, even if they don't have an current reservation. Another left join

with Car_Rental_Info is performed to get us the car details. This query also makes sure that the number of previous rentals are greater than 1 (we can change this based on demand and supply) and we make our search based on a particular car_Id, over here = 21255. Then, we give the discount to the customer on the basis if he has rented the same car for more than 5 days. We use an exists query to search for the same.

Our cost analysis before we do indexing.



```
-> Nested loop inner join  (cost=15.4 rows=8.67) (actual time=0.118..0.233 rows=25 loops=1)
    -> Filter: (br.Customer_Id is not null)  (cost=6.35 rows=26) (actual time=0.0966..0.104 rows=26 loops=1)
        -> Index lookup on br using idx_booking_car_id (Car_Id=21255)  (cost=6.35 rows=26) (actual time=0.0936..0.0983 rows=26 loops=1)
    -> Filter: (c.Number_of_Rentals > 1)  (cost=0.251 rows=0.333) (actual time=0.00465..0.00474 rows=0.962 loops=26)
        -> Single-row index lookup on c using PRIMARY (Customer_Id=br.Customer_Id)  (cost=0.251 rows=1) (actual time=0.00329..0.00333 rows=1 loops=26)
-> Select #2 (subquery in projection; dependent)
    -> Limit: 1 row(s)  (cost=0.398 rows=0.05) (actual time=0.00662..0.00665 rows=0.84 loops=25)
        -> Filter: ((b.Car_Id = '21255') and (b.Booking_Duration > 5))  (cost=0.398 rows=0.05) (actual time=0.00648..0.00648 rows=0.84 loops=25)
            -> Index lookup on b using Customer_Id (Customer_Id=c.Customer_Id)  (cost=0.398 rows=1.57) (actual time=0.00572..0.00605 rows=1.72 loops=25)
```

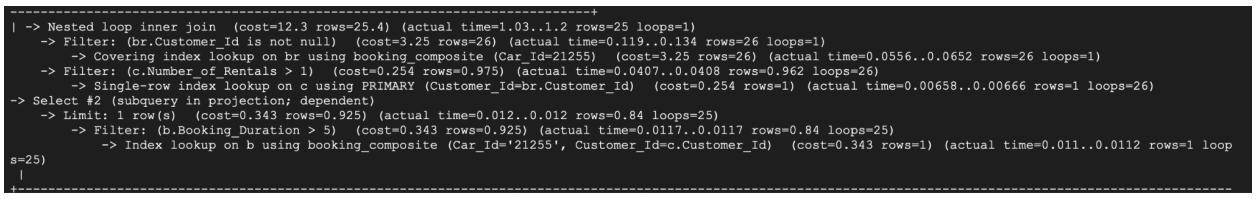
Here, above we can have a look at the overall cost analysis of our query before we index certain of the attributes.

The analysis shows a nested loop join with a cost of 15.4 driving the overall expense, followed by two filters at 6.35 each that create bottlenecks. Creating indexes on Customer_Id and Car_Id would likely improve performance by reducing these high costs in the execution path.

Lets move on to index few of the attributes here:

1. We try to use composite index on Booking_Reservation on (Car_Id, Customer_Id)

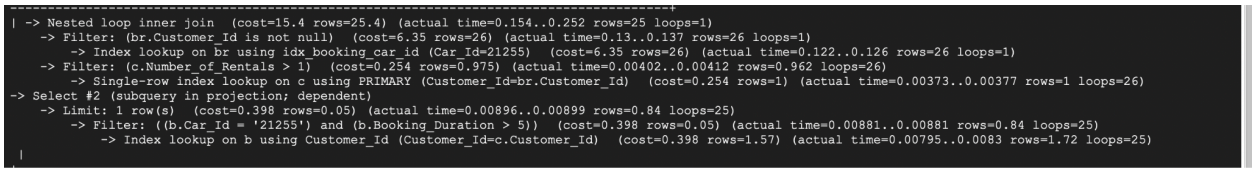
CREATE INDEX booking_composite ON Booking_Reservations (Car_Id, Customer_Id);



```
-> Nested loop inner join  (cost=12.3 rows=25.4) (actual time=1.03..1.2 rows=25 loops=1)
    -> Filter: (br.Customer_Id is not null)  (cost=3.25 rows=26) (actual time=0.119..0.134 rows=26 loops=1)
        -> Covering index lookup on br using booking_composite (Car_Id=21255)  (cost=3.25 rows=26) (actual time=0.0556..0.0652 rows=26 loops=1)
    -> Filter: (c.Number_of_Rentals > 1)  (cost=0.254 rows=0.975) (actual time=0.0407..0.0408 rows=0.962 loops=26)
        -> Single-row index lookup on c using PRIMARY (Customer_Id=br.Customer_Id)  (cost=0.254 rows=1) (actual time=0.00658..0.00666 rows=1 loops=26)
-> Select #2 (subquery in projection; dependent)
    -> Limit: 1 row(s)  (cost=0.343 rows=0.925) (actual time=0.012..0.012 rows=0.84 loops=25)
        -> Filter: (b.Booking_Duration > 5)  (cost=0.343 rows=0.925) (actual time=0.0117..0.0117 rows=0.84 loops=25)
            -> Index lookup on b using booking_composite (Car_Id='21255', Customer_Id=c.Customer_Id)  (cost=0.343 rows=1) (actual time=0.011..0.0112 rows=1 loop
s=25)
```

Here, we can see that using a composite index on Car_Id, Customer_Id helps us reduce the cost of the nested loop inner join from 15.4 to 12.3. As in our query we are joining the 3 tables using these attributes and doing searches based on them, indexing them allows us to reduce the cost of the nested loop inner join. Along with this, the cost of the filter br.Customer_id is null reduced by 50%. Our indexing of br.Customer_Id leads to this reduction in cost, as now we have indexed the Customer_Id.

2. We will try to index Number_of_Rentals here

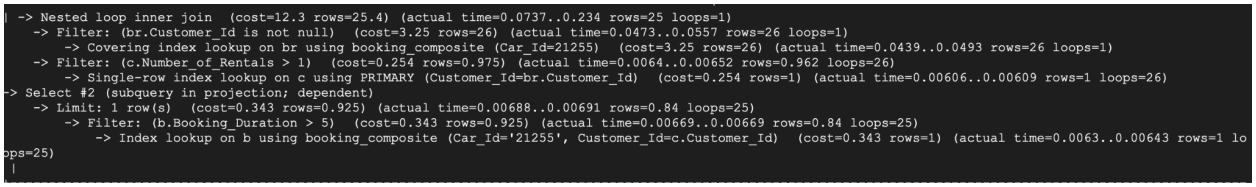


```
-> Nested loop inner join  (cost=15.4 rows=25.4) (actual time=0.154..0.252 rows=25 loops=1)
    -> Filter: (br.Customer_Id is not null)  (cost=6.35 rows=26) (actual time=0.13..0.137 rows=26 loops=1)
        -> Index lookup on br using idx_booking_car_id (Car_Id=21255)  (cost=6.35 rows=26) (actual time=0.122..0.126 rows=26 loops=1)
    -> Filter: (c.Number_of_Rentals > 1)  (cost=0.254 rows=0.975) (actual time=0.00402..0.00412 rows=0.962 loops=26)
        -> Single-row index lookup on c using PRIMARY (Customer_Id=br.Customer_Id)  (cost=0.254 rows=1) (actual time=0.00373..0.00377 rows=1 loops=26)
-> Select #2 (subquery in projection; dependent)
    -> Limit: 1 row(s)  (cost=0.398 rows=0.05) (actual time=0.00896..0.00899 rows=0.84 loops=25)
        -> Filter: ((b.Car_Id = '21255') and (b.Booking_Duration > 5))  (cost=0.398 rows=0.05) (actual time=0.00881..0.00881 rows=0.84 loops=25)
            -> Index lookup on b using Customer_Id (Customer_Id=c.Customer_Id)  (cost=0.398 rows=1.57) (actual time=0.00795..0.0083 rows=1.72 loops=25)
```

Over here, we can see that indexing based on Number_of_Rentals does not lead to reduction in the overall cost of the filter as well the following (upper) analysis. So, here we can see that as the Number_of_Rentals is being used within a filter indexing it does not help us much in reducing the cost of our query. As it is being used within a filter analysis, its cost is being overpowered by the cost of operations like joins. This can be seen from the fact that the cost of the filter checking for booking_duration is a mere 0.398 before and after filtering.

3. Now, we will finally try to index Booking_Duration which is attribute of the Booking_Reservation table here:

CREATE INDEX duration_index on Booking_Reservations(Booking_Duration)



```
-> Nested loop inner join  (cost=12.3 rows=25.4) (actual time=0.0737..0.234 rows=25 loops=1)
    -> Filter: (br.Customer_Id is not null)  (cost=3.25 rows=26) (actual time=0.0473..0.0557 rows=26 loops=1)
        -> Covering index lookup on br using booking_composite (Car_Id=21255)  (cost=3.25 rows=26) (actual time=0.0439..0.0493 rows=26 loops=1)
    -> Filter: (c.Number_of_Rentals > 1)  (cost=0.254 rows=0.975) (actual time=0.0064..0.00652 rows=0.962 loops=26)
        -> Single-row index lookup on c using PRIMARY (Customer_Id=br.Customer_Id)  (cost=0.254 rows=1) (actual time=0.00606..0.00609 rows=1 loops=26)
-> Select #2 (subquery in projection; dependent)
    -> Limit: 1 row(s)  (cost=0.343 rows=0.925) (actual time=0.00688..0.00691 rows=0.84 loops=25)
        -> Filter: (b.Booking_Duration > 5)  (cost=0.343 rows=0.925) (actual time=0.00669..0.00669 rows=0.84 loops=25)
            -> Index lookup on b using booking_composite (Car_Id='21255', Customer_Id=c.Customer_Id)  (cost=0.343 rows=1) (actual time=0.0063..0.00643 rows=1 lo
ops=25)
```

Over here, we can see that we had minute reductions in cost but overall the cost of the major components of our query remains the same. This can be attributed to the fact the highest cost of our query is from the loop joins which have already been optimized by a composite index before. So, there is not major reductions on cost

So, overall here, we created and checked 3 different indexes and got varying results for all the 3 indexes. The index on Booking_Reservations (Car_Id, Customer_Id) produced the best results.

4. Looking up the car by state and making sure that the car has not been booked for those particular dates. (group by state, join booking_reservation and car_rental_info):

```
SELECT c.state, COUNT(distinct c.car_id) as total_cars,
(COUNT(distinct c.car_id) - COUNT(distinct b.car_id)) as cars_available,
(SELECT avg(c2.daily_price) FROM Car_Rental_Info c2
WHERE c2.car_id NOT IN (
SELECT car_id
FROM Booking_Reservations
WHERE end_date >= '2023-08-02' and start_date <= '2023-08-25')
and c2.state = c.state) as avg_price
FROM Car_Rental_Info c
LEFT JOIN Booking_Reservations b on b.car_id = c.car_id
and b.end_date >= '2023-08-02' and b.start_date <= '2023-08-25'
GROUP BY c.state
ORDER BY (total_cars - cars_available) DESC;
```

Here, we have an advanced query that consists of joins, subqueries and group by clause. Let's move on to explain what our query is trying to do here. We are trying to find the total numbers of registered cars in a state, followed by checking the total available cars between 2 dates and the average price of the car of these available cars. One important thing to note here is that the avg_price is not the average price of the cars in the state but it is the average price of the available cars, which are not booked during these particular dates. This query is extremely useful for the administrator as he can use it to see the overall trends of the available cars and their average price during peak specific dates and decide discounts, offers, addition of new cars in the region based on this. We are ordering by the region where most cars are booked. This query can also be used on the customer_interface to provide details regarding important details to decide about booking a car

```
mysql> select c.state, count(distinct c.car_id) as total_cars, (count(distinct c.car_id) - count(distinct b.car_id)) as cars_available, (select avg(c2.daily_pri
ce) from Car_Rental_Info c2 where c2.car_id not in (select car_id from Booking_Reservations where end_date >= '2023-08-02' and start_date <= '2023-08-25') and c
2.state = c.state) as avg_price from Car_Rental_Info c left join Booking_Reservations b on b.car_id = c.car_id and b.end_date >= '2023-08-02' and b.start_date <
= '2023-08-25' group by c.state ORDER BY (total_cars - cars_available) DESC limit 15 ;
+-------+-----------+---------------+-----------+
| state | total_cars | cars_available | avg_price |
+-------+-----------+---------------+-----------+
| TX    |        256 |           225 |   89.5022 |
| GA    |        132 |           103 |  139.1165 |
| NC    |        118 |           108 |  100.6204 |
| OH    |         54 |            45 |  108.2444 |
| TN    |         78 |            69 |   96.3478 |
| CO    |        107 |           102 |  104.6569 |
| MD    |         50 |            45 |   97.7556 |
| VA    |         80 |            76 |   91.9342 |
| CT    |         12 |             9 |  207.4444 |
| SC    |         29 |            26 |  101.8462 |
| ID    |         22 |            19 |   81.0526 |
| NM    |         22 |            20 |   75.8000 |
| DC    |          9 |             7 |  123.8571 |
| WA    |         78 |            76 |   82.0000 |
| FL    |        416 |           415 |  108.4940 |
+-------+-----------+---------------+-----------+
15 rows in set (0.08 sec)
```

Now, we can move on to doing the indexing section for this particular query here:

Here, we will index non primary key based attributes present in our keys. Before indexing, we need to check the costs associated with our query.

```
-------+
-> Sort: (total_cars - cars_available) DESC  (actual time=87.4..87.4 rows=46 loops=1)
  -> Stream results  (cost=2586 rows=55.3) (actual time=6.16..87.2 rows=46 loops=1)
      -> Group aggregate: count(distinct b.Car_Id), count(distinct c.Car_Id), count(distinct c.Car_Id)  (cost=2586 rows=55.3) (actual time=3.55..14.1 rows=46
oops=1)
          -> Nested loop left join  (cost=2080 rows=5053) (actual time=3.45..12.5 rows=3117 loops=1)
              -> Sort: c.State  (cost=312 rows=3055) (actual time=3.4..3.87 rows=3093 loops=1)
                  -> Table scan on c  (cost=312 rows=3055) (actual time=0.518..1.58 rows=3093 loops=1)
              -> Filter: ((b.end_date >= '2023-08-02') and (b.start_date <= '2023-08-25'))  (cost=0.414 rows=1.65) (actual time=0.00252..0.00261 rows=0.0466 l
oops=3093)
                  -> Index lookup on b using Car_Id (Car_Id=c.Car_Id)  (cost=0.414 rows=1.65) (actual time=0.00202..0.00222 rows=0.323 loops=3093)

-------+
row in set, 1 warning (0.13 sec)
```

We can see we have high costs based on end_date and start_date filters, scan on c, sorting by our state and group aggregation.

Lets attempt to reduce the cost by creating indexes.

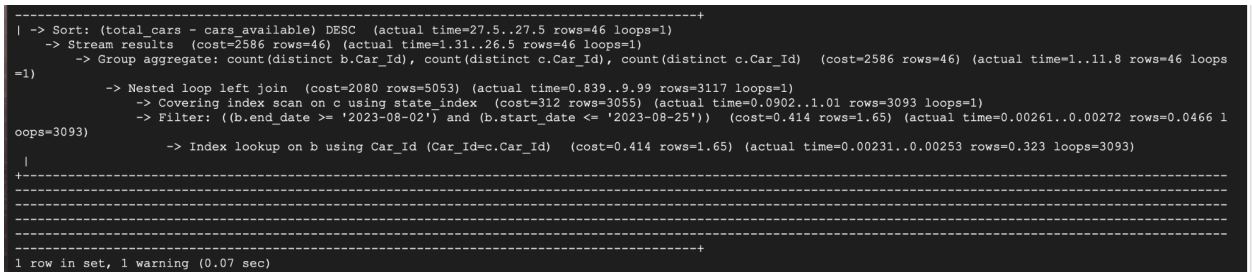
1. Create index state_index on Car_Rental_Info(state)

```
+-----------------------------------------------------------------------------------------+
| -> Sort: (total_cars - cars_available) DESC  (actual time=43.7..43.7 rows=46 loops=1)
    -> Stream results  (cost=2586 rows=46) (actual time=0.618..43.5 rows=46 loops=1)
        -> Group aggregate: count(distinct b.Car_Id), count(distinct c.Car_Id), count(distinct c.Car_Id)  (cost=2586 rows=46) (actual time=0.306..18.3 rows=46 l
oops=1)
            -> Nested loop left join  (cost=2080 rows=5053) (actual time=0.115..15.6 rows=3117 loops=1)
                -> Covering index scan on c using state_index  (cost=312 rows=3055) (actual time=0.0884..2.28 rows=3093 loops=1)
                -> Filter: ((b.end_date >= '2023-08-02') and (b.start_date <= '2023-08-25'))  (cost=0.414 rows=1.65) (actual time=0.00389..0.00403 rows=0.0466 l
oops=3093)
                    -> Index lookup on b using Car_Id (Car_Id=c.Car_Id)  (cost=0.414 rows=1.65) (actual time=0.00344..0.00376 rows=0.323 loops=3093)
 |
+-----------------------------------------------------------------------------------------+
```

In the above explain analysis, we can see a reduction in the cost. Here, the cost on index lookup and filter on dates remains the same. We have an additional covering index scan on c but that cost gets negated by the table scan on c in our unindexed query. In the unindexed query we have an additional sorting cost of 312 which is not present in our indexed query on state as we use indexing. Everything else has similar costs. So, overall, in our indexed query here, we have decreased the cost. This can be explained by the fact that we have index scan over a full table scan here.

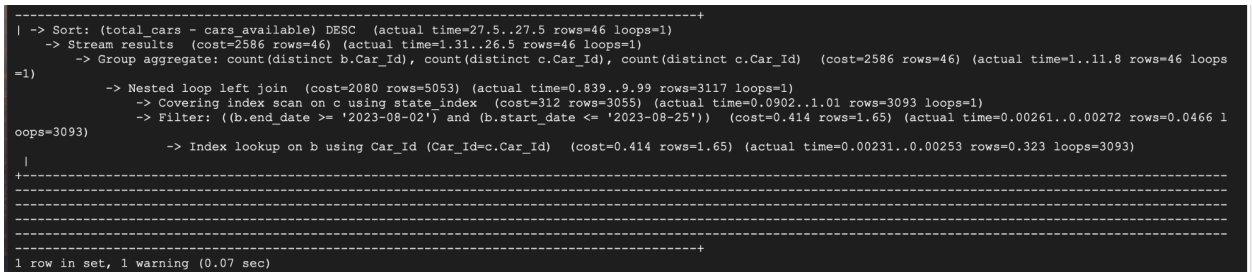
Now let's move and try to index based on another variable.

2. Create index daily_price_index on Car_Rental_Info(daily_price)



Here is the explain analyze command after we index on daily price. The cost of most of the operations remain the same. We can observe this here because we are using daily_price within an aggregate function and this doesn't add much to the overall cost of the mentioned operations. It however leads to improved time performance in the long run.

3. Create index start_date_index and end_date_index on Booking_Reservations(start_date) and Booking_Reservations(end_date)



```
-----------------------------------------------------------------------------------------+
| -> Sort: (total_cars - cars_available) DESC  (actual time=27.5..27.5 rows=46 loops=1)
    -> Stream results  (cost=2586 rows=46) (actual time=1.31..26.5 rows=46 loops=1)
       -> Group aggregate: count(distinct b.Car_Id), count(distinct c.Car_Id), count(distinct c.Car_Id)  (cost=2586 rows=46) (actual time=1..11.8 rows=46 loops
=1)
          -> Nested loop left join  (cost=2080 rows=5053) (actual time=0.839..9.99 rows=3117 loops=1)
             -> Covering index scan on c using state_index  (cost=312 rows=3055) (actual time=0.0902..1.01 rows=3093 loops=1)
             -> Filter: ((b.end_date >= '2023-08-02') and (b.start_date <= '2023-08-25'))  (cost=0.414 rows=1.65) (actual time=0.00261..0.00272 rows=0.0466 l
oops=3093)
                -> Index lookup on b using Car_Id (Car_Id=c.Car_Id)  (cost=0.414 rows=1.65) (actual time=0.00231..0.00253 rows=0.323 loops=3093)
 |
+-----------------------------------------------------------------------------------------+
1 row in set, 1 warning (0.07 sec)
```

Here is the explain analyze command after we index on start and end date. We can see that there is no change in the overall cost and row analysis. The cost remains around 0.414 on the filter based on the end date and start date compared to a parameter. This can be attributed to the fact that the cost is already very low for the filters and adding the indexing of them is not leading to much reduction in cost lookups. The major cost are from the group by and joins based on the primary key here.

So, here, we have overall tried 3 different indexing and found and explained those varied results in detail here. The index on Car_Rental_Info(state) led to reduction in cost.

Conclusion:

With all of our queries and the setup of the table, we've better streamlined the process for customers to query the data in our database. This has been done through optimizations such as adding an index for the number of car thefts when the customer tries to get their total payment and adding a composite index for the booking in order to decide if the customer will get a special discount. With speed-ups like these, our aim has been to minimize cost while also maximizing efficiency through adding these indexes for what we would assume to be commonly used queries within our database.

Through this, we've built a more optimized system through which data can not only be delivered faster, but also more efficiently, which should be extremely helpful in the long run.