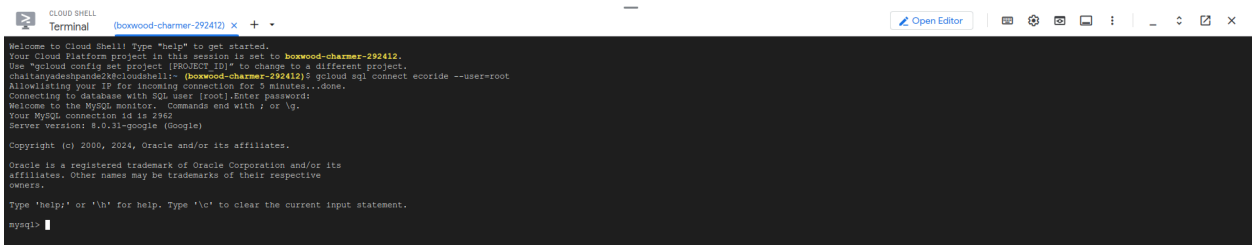
EcoRide: A Car Rental with Environmental Rewards System (Stage3)

DDL Commands

```
CREATE TABLE User (
    userId INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    contact VARCHAR(15) NOT NULL,
    email VARCHAR(50) NOT NULL
);

CREATE TABLE UserRoles (
    userRoleId INT PRIMARY KEY,
    userId INT NOT NULL,
    role ENUM('buyer', 'seller') NOT NULL,
    FOREIGN KEY(userId) REFERENCES User(userId) ON DELETE CASCADE
);

CREATE TABLE Car (
    carId INT PRIMARY KEY,
    userId INT NOT NULL,
    price INT NOT NULL,
    mileage DECIMAL(4, 2) NOT NULL,
    availability BOOLEAN NOT NULL,
    carCompany VARCHAR(50) NOT NULL,
    carModel VARCHAR(50) NOT NULL,
    FOREIGN KEY(userId) REFERENCES User(userId) ON DELETE CASCADE
);

CREATE TABLE Booking (
    bookingId INT PRIMARY KEY,
    carId INT NOT NULL,
    userId INT NOT NULL,
    startDate DATETIME NOT NULL,
    endDate DATETIME NOT NULL,
    startMileage DECIMAL(4, 2) NOT NULL,
    endMileage DECIMAL(4, 2) NOT NULL,
    FOREIGN KEY(carId) REFERENCES Car(carId) ON DELETE CASCADE,
    FOREIGN KEY(userId) REFERENCES User(userId) ON DELETE CASCADE
);
```

```
CREATE TABLE EcoPoints (
   ecoPointsId INT PRIMARY KEY,
   userId INT NOT NULL,
   points INT NOT NULL,
   FOREIGN KEY(userId) REFERENCES User(userId) ON DELETE CASCADE
);

CREATE TABLE CarRating (
   ratingId INT PRIMARY KEY,
   carId INT NOT NULL,
   userId INT NOT NULL,
   bookingId INT NOT NULL,
   ratingValue DECIMAL(2, 1),
   FOREIGN KEY(carId) REFERENCES Car(carId) ON DELETE CASCADE,
   FOREIGN KEY(userId) REFERENCES User(userId) ON DELETE CASCADE
);
```




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

mysql> SELECT COUNT(*) FROM UserRoles;
+----------+
| COUNT(*) |
+----------+
|     1020 |
+----------+
1 row in set (0.09 sec)

mysql> SELECT COUNT(*) FROM Booking;
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.02 sec)
```

```
mysql> SELECT COUNT(*) FROM Car;
+----------+
| COUNT(*) |
+----------+
|      507 |
+----------+
1 row in set (0.01 sec)

mysql> SELECT COUNT(*) FROM CarRating;
+----------+
| COUNT(*) |
+----------+
|      400 |
+----------+
1 row in set (0.01 sec)

mysql> SELECT COUNT(*) FROM EcoPoints;
+----------+
| COUNT(*) |
+----------+
|      450 |
+----------+
1 row in set (0.05 sec)
```

Indexing

1. Query to Retrieve Average Rating for Each Car Company

This query retrieves the average rating for each car company, considering only available cars. It uses a join between Car and CarRating, and aggregates the data by carId, carCompany, carModel

```
SELECT
   c.carId,
   c.carCompany,
   c.carModel,
   ROUND(AVG(cr.ratingValue), 2) AS averageRating
FROM
   Car AS c
JOIN
   CarRating AS cr ON c.carId = cr.carId
WHERE
   c.availability = TRUE AND c.carCompany IN ('Tesla', 'Audi','BMW')
GROUP BY
   c.carId, c.carCompany, c.carModel;
```

Output:

```
mysql> SELECT      c.carId,      c.carCompany,      c.carModel,     ROUND(AVG(cr.ratingValue), 2) AS averageRating FROM      Car AS c JOIN      CarRating AS cr ON c.carId = cr.carId WHERE
 c.availability = TRUE AND c.carCompany IN ('Tesla', 'Audi','BMW') GROUP BY      c.carId, c.carCompany, c.carModel LIMIT 15;
+-------+------------+--------------------+---------------+
| carId | carCompany | carModel           | averageRating |
+-------+------------+--------------------+---------------+
|    52 | Audi       | RS 7               |          4.95 |
|    65 | Tesla      | Model Mazda 3      |          5.00 |
|    74 | BMW        | 24                 |          4.90 |
|    79 | Tesla      | Model Mazda 3      |          4.95 |
|    83 | Tesla      | Model X            |          4.80 |
|    88 | BMW        | M4                 |          4.85 |
|    90 | BMW        | Mazda 5 Series     |          5.00 |
|   114 | BMW        | X4                 |          5.00 |
|   149 | Tesla      | Model Mazda 3      |          4.85 |
|   168 | Tesla      | Model Mazda 3      |          5.00 |
|   170 | Tesla      | Model S            |          5.00 |
|   187 | BMW        | Mazda 3 Series     |          4.80 |
|   208 | BMW        | 4 Series Gran Coupe |          5.00 |
|   230 | BMW        | Mazda 3 Series     |          5.00 |
|   236 | BMW        | Mazda 3 Series     |          4.70 |
+-------+------------+--------------------+---------------+
15 rows in set (0.01 sec)
```

DEFAULT:

Cost = 0.26

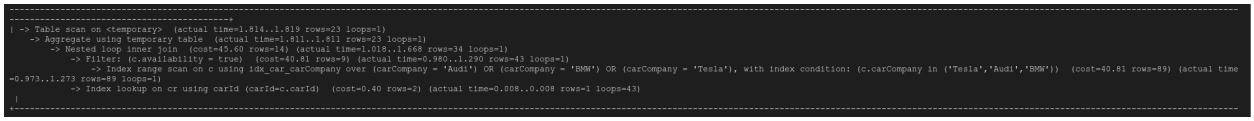
```
-> Table scan on <temporary>  (actual time=0.730..0.734 rows=23 loops=1)
    -> Aggregate using temporary table  (actual time=0.728..0.728 rows=23 loops=1)
        -> Nested loop inner join  (cost=57.02 rows=15) (actual time=0.167..0.532 rows=34 loops=1)
            -> Filter: ((c.availability = true) and (c.carCompany in ('Tesla','Audi','BMW')))  (cost=51.70 rows=15) (actual time=0.114..0.366 rows=43 loops=1)
                -> Table scan on c  (cost=51.70 rows=507) (actual time=0.089..0.261 rows=507 loops=1)
            -> Index lookup on cr using carId (carId=c.carId)  (cost=0.26 rows=1) (actual time=0.003..0.004 rows=1 loops=43)
```

CREATE INDEX idx_car_availability ON Car (availability);

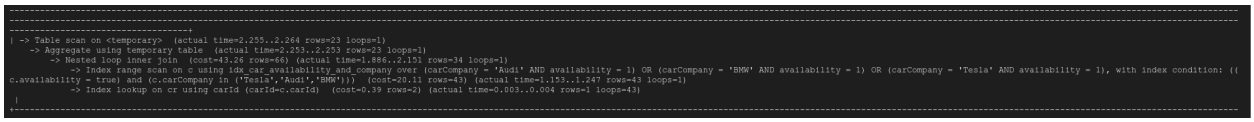
Cost = 0.39

```
-> Table scan on <temporary>  (actual time=0.946..0.950 rows=23 loops=1)
    -> Aggregate using temporary table  (actual time=0.943..0.943 rows=23 loops=1)
        -> Nested loop inner join  (cost=52.23 rows=119) (actual time=0.287..0.840 rows=34 loops=1)
            -> Filter: (c.carCompany in ('Tesla','Audi','BMW'))  (cost=10.71 rows=77) (actual time=0.234..0.587 rows=43 loops=1)
                -> Index lookup on c using idx_car_availability (availability=true)  (cost=10.71 rows=257) (actual time=0.230..0.535 rows=257 loops=1)
            -> Index lookup on cr using carId (carId=c.carId)  (cost=0.39 rows=2) (actual time=0.005..0.006 rows=1 loops=43)
```

CREATE INDEX idx_car_carCompany ON Car (carCompany);
Cost = 0.40



CREATE INDEX idx_car_availability_and_company ON Car (carCompany,availability);
Cost = 0.39



The default behavior of the query shows that it initially searches the Car table using the carId key, resulting in a cost of 0.26. We then created an index on the availability column in the Car table, which increased the cost from 0.26 to 0.39. Adding an index on the carCompany column further raised the cost slightly to 0.40, similar to the previous case. Finally, we applied a composite index on carCompany and availability, as these columns are frequently accessed together in the WHERE clause, resulting in a cost of 0.39. Thus, the default behavior provided the best performance among all indexes tested. This reflects the idea that while adding indexes can be helpful for filtering, they also bring some extra overhead, which can affect performance compared to the efficiency of primary key lookups.

2. Query to Find Cars Eligible for Free Service Based on Trip Count, Average Rating, and Mileage

This query identifies cars eligible for free servicing by analyzing their trip count and average customer ratings. It selects available cars that have completed a multiple of four trips, maintain a top average rating, and have an average mileage above a certain threshold. This approach ensures that only well-reviewed, frequently rented vehicles with good performance qualify for the benefit.

```
SELECT
   c.carId,
   c.carModel,
   COUNT(b.bookingId) AS tripCount,
   ROUND(AVG(cr.ratingValue),2) AS averageRating
FROM
   Car AS c
JOIN
   Booking AS b ON c.carId = b.carId
JOIN
   CarRating AS cr ON c.carId = cr.carId
WHERE
```

```
    c.availability=TRUE
GROUP BY
    c.carId
HAVING
    MOD(COUNT(b.bookingId), 4) = 0
    AND ROUND(AVG(cr.ratingValue), 2) >= 4.5
   AND AVG(c.mileage) > 8;
```

Output:

```
mysql> SELECT      c.carId,     c.carModel,     COUNT(b.bookingId) AS tripCount,     ROUND(AVG(cr.ratingValue),2) AS averageRating FROM      Car AS c JOIN      Booking AS b ON c.carId = b.carI
d JOIN      CarRating AS cr ON c.carId = cr.carId WHERE     c.availability=TRUE GROUP BY      c.carId HAVING      MOD(COUNT(b.bookingId), 4) = 0      AND ROUND(AVG(cr.ratingValue), 2) >= 4.5
  AND AVG(c.mileage) > 8 LIMIT 15;
+-------+---------------+-----------+---------------+
| carId | carModel      | tripCount | averageRating |
+-------+---------------+-----------+---------------+
|    54 | XJ            |         4 |          4.90 |
|    58 | S-Class       |         4 |          4.95 |
|    73 | fortwo        |         4 |          4.75 |
|    75 | Camaro        |         4 |          5.00 |
|    79 | Model Mazda 3 |         4 |          4.95 |
|    88 | M4            |         4 |          4.85 |
|   114 | X4            |        12 |          5.00 |
|   149 | Model Mazda 3 |         4 |          4.85 |
|   230 | Mazda 3 Series |         4 |          5.00 |
|   247 | Boxster       |         8 |          4.85 |
|   250 | Trax          |         4 |          4.95 |
|   273 | Model Mazda 3 |         4 |          4.75 |
|   288 | Navigator     |         4 |          4.70 |
|   289 | Equinox       |         4 |          4.88 |
|   298 | XMazda 5      |         4 |          4.90 |
+-------+---------------+-----------+---------------+
15 rows in set (0.00 sec)
```

DEFAULT
Cost = 0.26

```
-> Filter: (((count(b.bookingId) % 4) = 0) and (round(avg(cr.ratingValue),2) >= 4.5) and (avg(c.mileage) > 8))  (actual time=2.847..2.916 rows=22 loops=1)
    -> Table scan on <temporary>  (actual time=2.828..2.861 rows=116 loops=1)
        -> Aggregate using temporary table  (actual time=2.826..2.826 rows=116 loops=1)
            -> Nested loop inner join  (cost=199.67 rows=94) (actual time=0.111..2.115 rows=400 loops=1)
                -> Nested loop inner join  (cost=180.25 rows=40) (actual time=0.096..1.188 rows=209 loops=1)
                    -> Table scan on cr  (cost=40.25 rows=400) (actual time=0.064..0.248 rows=400 loops=1)
                    -> Filter: (c.availability = true)  (cost=0.25 rows=0.1) (actual time=0.002..0.002 rows=1 loops=400)
                        -> Single-row index lookup on c using PRIMARY (carId=cr.carId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=400)
                -> Covering index lookup on b using carId (carId=cr.carId)  (cost=0.26 rows=2) (actual time=0.003..0.004 rows=2 loops=209)
```

CREATE INDEX idx_availability ON Car (availability);
Cost = 0.25

```
-> Filter: (((count(b.bookingId) % 4) = 0) and (round(avg(cr.ratingValue),2) >= 4.5) and (avg(c.mileage) > 8))  (cost=452.12 rows=930) (actual time=1.000..2.721 rows=22 loops=1)
    -> Group aggregate: avg(c.mileage), avg(cr.ratingValue), count(b.bookingId), avg(cr.ratingValue), count(b.bookingId)  (cost=452.12 rows=930) (actual time=0.292..1.987 rows=116 loops=1)
        -> Nested loop inner join  (cost=359.09 rows=930) (actual time=0.273..1.787 rows=400 loops=1)
            -> Nested loop inner join  (cost=167.08 rows=395) (actual time=0.264..1.155 rows=209 loops=1)
                -> Index lookup on c using idx_availability (availability=true)  (cost=28.70 rows=257) (actual time=0.167..0.433 rows=257 loops=1)
                -> Index lookup on cr using carId (carId=c.carId)  (cost=0.39 rows=2) (actual time=0.002..0.003 rows=1 loops=257)
            -> Covering index lookup on b using carId (carId=c.carId)  (cost=0.25 rows=2) (actual time=0.002..0.003 rows=2 loops=209)
```

CREATE INDEX idx_availability_mileage ON Car (availability,mileage);
Cost = 0.25

```
-> Filter: (((count(b.bookingId) % 4) = 0) and (round(avg(cr.ratingValue),2) >= 4.5) and (avg(c.mileage) > 8))  (actual time=3.332..3.395 rows=22 loops=1)
    -> Table scan on <temporary>  (actual time=3.305..3.339 rows=116 loops=1)
        -> Aggregate using temporary table  (actual time=3.301..3.301 rows=116 loops=1)
            -> Nested loop inner join  (cost=359.09 rows=930) (actual time=0.289..2.547 rows=400 loops=1)
                -> Nested loop inner join  (cost=167.08 rows=395) (actual time=0.262..1.689 rows=209 loops=1)
                    -> Index lookup on c using idx_availability_mileage (availability=true)  (cost=28.70 rows=257) (actual time=0.238..0.560 rows=257 loops=1)
                    -> Index lookup on cr using carId (carId=c.carId)  (cost=0.39 rows=2) (actual time=0.004..0.004 rows=1 loops=257)
                -> Covering index lookup on b using carId (carId=c.carId)  (cost=0.25 rows=2) (actual time=0.003..0.004 rows=2 loops=209)
```

CREATE INDEX idx_ratingValue ON CarRating (ratingValue);
Cost = 0.26

```
-> Filter: (((count(b.bookingId) % 4) = 0) and (round(avg(cr.ratingValue),2) >= 4.5) and (avg(c.mileage) > 8))  (actual time=1.709..1.757 rows=22 loops=1)
    -> Table scan on <temporary>  (actual time=1.693..1.716 rows=116 loops=1)
        -> Aggregate using temporary table  (actual time=1.692..1.692 rows=116 loops=1)
            -> Nested loop inner join  (cost=199.67 rows=94) (actual time=0.066..1.260 rows=400 loops=1)
                -> Nested loop inner join  (cost=180.25 rows=40) (actual time=0.056..0.730 rows=209 loops=1)
                    -> Table scan on cr  (cost=40.25 rows=400) (actual time=0.034..0.137 rows=400 loops=1)
                    -> Filter: (c.availability = true)  (cost=0.25 rows=0.1) (actual time=0.001..0.001 rows=1 loops=400)
                        -> Single-row index lookup on c using PRIMARY (carId=cr.carId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=400)
                -> Covering index lookup on b using carId (carId=cr.carId)  (cost=0.26 rows=2) (actual time=0.002..0.002 rows=2 loops=209)
```

The default query behavior initially searches the Booking table using the carId key. We then created an index on the availability column in the Car table, which offered only a slight improvement, reducing the cost by 0.01. After that, we added a composite index on availability and mileage in the Car table, but this did not reduce the cost further; it remained the same as with the index on availability alone. We also tried indexing on the rating column, but the optimizer continued to prefer the primary index, likely based on current selectivity and performance statistics. Overall, we consistently achieved the best performance when the index included availability.

3. Find Budget-Friendly Cars with Above Average Mileage

This query identifies cars that are both budget-friendly and offer above-average mileage compared to other available vehicles. It selects the carId of those cars priced below the average of all available cars while ensuring their mileage exceeds the average mileage of the same pool.

```
SELECT
   c.carId
FROM
   Car AS c
WHERE
   c.price < (SELECT AVG(price)
              FROM Car
              WHERE availability = TRUE)
   AND c.mileage > (SELECT AVG(mileage)
                    FROM Car
                    WHERE availability = TRUE)
   AND c.availability = TRUE;
```

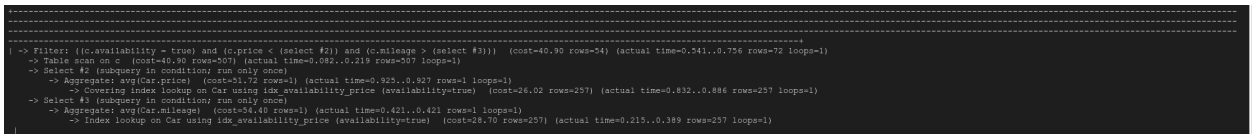
Output:

```
mysql> SELECT      c.carId FROM      Car AS c WHERE      c.price < (SELECT AVG(price)                FROM Car                WHERE availability = TRUE)     AND c.mileage > (SELECT AVG(mile
age)                    FROM Car                       WHERE availability = TRUE)     AND c.availability = TRUE LIMIT 15;
+-------+
| carId |
+-------+
|    14 |
|    15 |
|    36 |
|    46 |
|    56 |
|    72 |
|    73 |
|    74 |
|    76 |
|   111 |
|   121 |
|   132 |
|   180 |
|   188 |
|   211 |
+-------+
15 rows in set (0.01 sec)
```

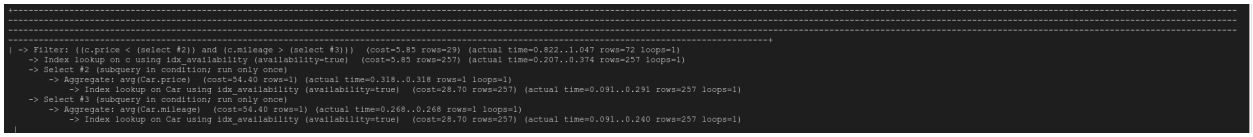
DEFAULT:
Cost: 51.70

```
-> Filter: ((c.availability = true) and (c.price < (select #2)) and (c.mileage > (select #3)))  (cost=47.19 rows=6) (actual time=0.566..0.839 rows=72 loops=1)
    -> Table scan on c  (cost=47.19 rows=507) (actual time=0.068..0.270 rows=507 loops=1)
    -> Select #2 (subquery in condition; run only once)
        -> Aggregate: avg(Car.price)  (cost=56.77 rows=1) (actual time=0.243..0.243 rows=1 loops=1)
            -> Filter: (Car.availability = true)  (cost=51.70 rows=51) (actual time=0.044..0.213 rows=257 loops=1)
                -> Table scan on Car  (cost=51.70 rows=507) (actual time=0.042..0.172 rows=507 loops=1)
    -> Select #3 (subquery in condition; run only once)
        -> Aggregate: avg(Car.mileage)  (cost=56.77 rows=1) (actual time=0.206..0.206 rows=1 loops=1)
            -> Filter: (Car.availability = true)  (cost=51.70 rows=51) (actual time=0.021..0.177 rows=257 loops=1)
                -> Table scan on Car  (cost=51.70 rows=507) (actual time=0.021..0.129 rows=507 loops=1)
|
```

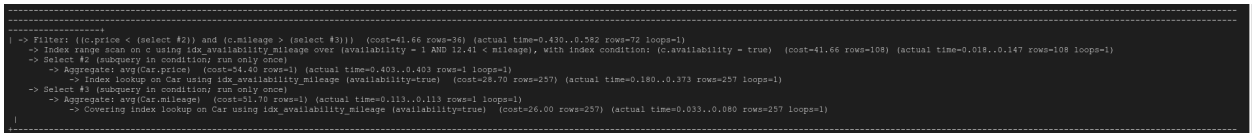
CREATE INDEX idx_availability_price ON Car(availability, price);
Cost: 28.70



CREATE INDEX idx_availability ON Car (availability);
Cost: 28.70



CREATE INDEX idx_availability_mileage ON Car(availability, mileage);
Cost: 26.00



The default indexing setup with only primary and foreign keys resulted in a high cost of 51.70, showing no optimization for availability, price, or mileage conditions in the query. Adding an index on (availability, price) reduced the cost to 28.70 by efficiently filtering records with both conditions. Indexing availability alone yielded a similar improvement (28.70), but the best performance came from a composite index on (availability, mileage), reducing the cost further to 26.00. This index configuration provided the lowest cost by targeting the two attributes used together in the WHERE clause, making it the optimal indexing choice for this query.

4. Top Rated Affordable Cars from Selected Car Company

This query retrieves a list of available cars from some selected car companies that are priced under $20,000, with an average rating of 4.0 or higher. It also includes the total number of bookings for each car, filtering for those with at least two bookings and orders the results by booking count in descending order.

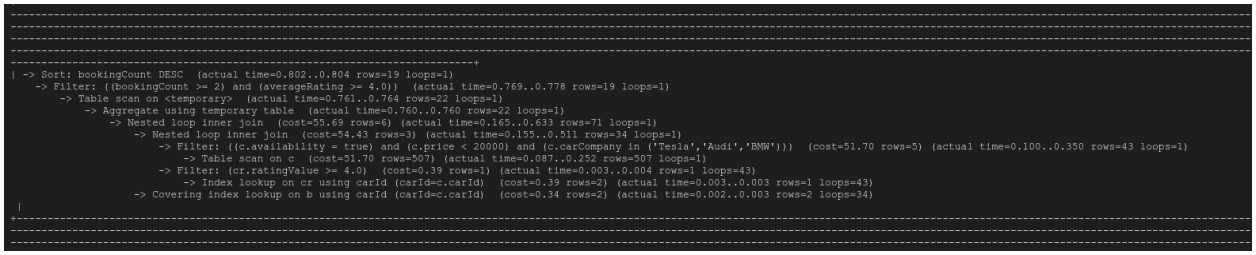
```
SELECT
    c.carId,
    c.carCompany,
    c.carModel,
    COUNT(b.bookingId) AS bookingCount,
    AVG(cr.ratingValue) AS averageRating,
    c.price
FROM
    Car AS c
JOIN
```

```
    Booking AS b ON c.carId = b.carId
JOIN
    CarRating AS cr ON c.carId = cr.carId
WHERE
    c.availability = TRUE
    AND c.price < 20000
    AND cr.ratingValue >= 4.0
    AND c.carCompany IN ('Tesla', 'Audi','BMW')
GROUP BY
    c.carId, c.carCompany, c.carModel
HAVING
    bookingCount >= 2
    AND averageRating >= 4.0
ORDER BY
    bookingCount DESC;
```

Output:

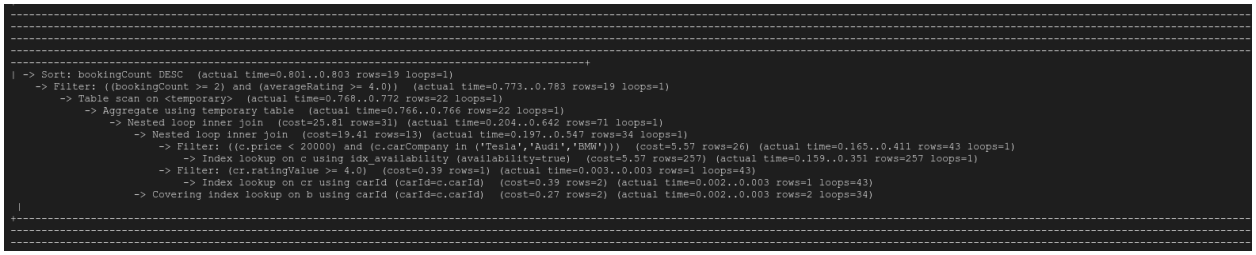
```
mysql> SELECT      c.carId,      c.carCompany,      c.carModel,      COUNT(b.bookingId) AS bookingCount,      AVG(cr.ratingValue) AS averageRating,      c.price FROM      Car AS c JOIN      Booking AS b ON c.carId = b.carId JOIN      CarRati
ng AS cr ON c.carId = cr.carId WHERE      c.availability = TRUE      AND c.price < 20000      AND cr.ratingValue >= 4.0      AND c.carCompany IN ('Tesla', 'Audi','BMW') GROUP BY      c.carId, c.carCompany, c.carModel HAVING      bookingCount
>= 2      AND averageRating >= 4.0 ORDER BY      bookingCount DESC LIMIT 15;
+------+------------+---------------+--------------+---------------+-------+
| carId | carCompany | carModel      | bookingCount | averageRating | price |
+------+------------+---------------+--------------+---------------+-------+
|   114 | BMW        | X4            |           12 |       5.00000 |    60 |
|    74 | BMW        | Z4            |            5 |       4.90000 |    78 |
|   319 | Audi       | A4            |            4 |       4.95000 |    39 |
|   230 | BMW        | Mazda 3 Series |           4 |       5.00000 |    42 |
|   298 | BMW        | XMazda 5      |            4 |       4.90000 |    64 |
|    88 | BMW        | M4            |            4 |       4.85000 |   180 |
|   273 | Tesla      | Model Mazda 3  |           4 |       4.75000 |    99 |
|   149 | Tesla      | Model Mazda 3  |           4 |       4.85000 |   149 |
|    79 | Tesla      | Model Mazda 3  |           4 |       4.95000 |   150 |
|   448 | Tesla      | Model Mazda 3  |           4 |       5.00000 |   172 |
|   438 | BMW        | Mazda 3 Series |           3 |       5.00000 |    49 |
|    52 | Audi       | RS 7          |            2 |       4.95000 |   209 |
|   236 | BMW        | Mazda 3 Series |           2 |       4.70000 |    43 |
|   187 | BMW        | Mazda 3 Series |           2 |       4.80000 |    47 |
|    90 | BMW        | Mazda 5 Series |           2 |       5.00000 |    64 |
+------+------------+---------------+--------------+---------------+-------+
15 rows in set (0.00 sec)
```

DEFAULT:
Cost : 0.34



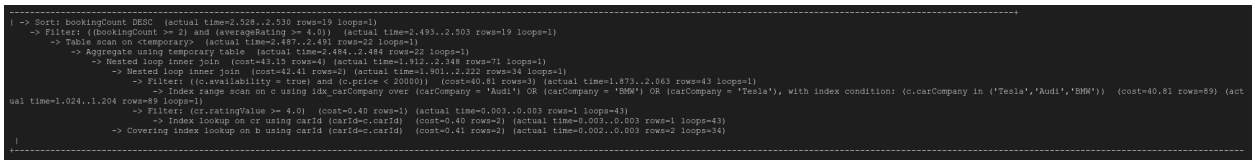
```
-> Sort: bookingCount DESC  (actual time=0.802..0.804 rows=19 loops=1)
    -> Filter: ((bookingCount >= 2) and (averageRating >= 4.0))  (actual time=0.769..0.778 rows=19 loops=1)
        -> Table scan on <temporary>  (actual time=0.761..0.764 rows=22 loops=1)
            -> Aggregate using temporary table  (actual time=0.760..0.760 rows=22 loops=1)
                -> Nested loop inner join  (cost=55.69 rows=6) (actual time=0.165..0.633 rows=71 loops=1)
                    -> Nested loop inner join  (cost=54.43 rows=3) (actual time=0.155..0.511 rows=34 loops=1)
                        -> Filter: ((c.availability = true) and (c.price < 20000) and (c.carCompany in ('Tesla','Audi','BMW')))  (cost=51.70 rows=5) (actual time=0.100..0.350 rows=43 loops=1)
                            -> Table scan on c  (cost=51.70 rows=507) (actual time=0.087..0.252 rows=507 loops=1)
                        -> Filter: (cr.ratingValue >= 4.0)  (cost=0.39 rows=1) (actual time=0.003..0.004 rows=1 loops=43)
                            -> Index lookup on cr using carId (carId=c.carId)  (cost=0.39 rows=2) (actual time=0.003..0.003 rows=1 loops=43)
                    -> Covering index lookup on b using carId (carId=c.carId)  (cost=0.34 rows=2) (actual time=0.002..0.003 rows=2 loops=34)
 |
```

CREATE INDEX idx_availability ON Car (availability);
Cost : 0.27

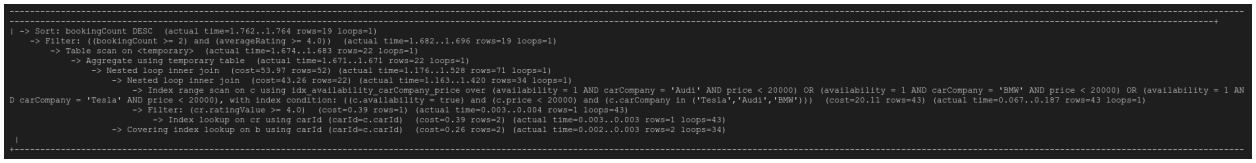


```
-> Sort: bookingCount DESC  (actual time=0.801..0.803 rows=19 loops=1)
    -> Filter: ((bookingCount >= 2) and (averageRating >= 4.0))  (actual time=0.773..0.783 rows=19 loops=1)
        -> Table scan on <temporary>  (actual time=0.768..0.772 rows=22 loops=1)
            -> Aggregate using temporary table  (actual time=0.766..0.766 rows=22 loops=1)
                -> Nested loop inner join  (cost=25.81 rows=31) (actual time=0.204..0.642 rows=71 loops=1)
                    -> Nested loop inner join  (cost=19.41 rows=13) (actual time=0.197..0.547 rows=34 loops=1)
                        -> Filter: ((c.price < 20000) and (c.carCompany in ('Tesla','Audi','BMW')))  (cost=5.57 rows=26) (actual time=0.165..0.411 rows=43 loops=1)
                            -> Index lookup on c using idx_availability (availability=true)  (cost=5.57 rows=257) (actual time=0.159..0.351 rows=257 loops=1)
                        -> Filter: (cr.ratingValue >= 4.0)  (cost=0.39 rows=1) (actual time=0.003..0.003 rows=1 loops=43)
                            -> Index lookup on cr using carId (carId=c.carId)  (cost=0.39 rows=2) (actual time=0.002..0.003 rows=1 loops=43)
                    -> Covering index lookup on b using carId (carId=c.carId)  (cost=0.27 rows=2) (actual time=0.002..0.003 rows=2 loops=34)
 |
```

CREATE INDEX idx_carCompany ON Car (carCompany);
Cost : 0.41



CREATE INDEX idx_availability_carCompany_price ON Car (availability, carCompany, price);
Cost : 0.26



The default indexing with only primary and foreign keys resulted in a cost of 0.34, indicating no optimization for the query's conditions. Adding an index on availability reduced the cost to 0.27, showing better filtering for available cars. However, indexing carCompany increased the cost to 0.41, suggesting it wasn't effective for this query. The best performance came from the composite index on (availability, carCompany, price), which lowered the cost further to 0.26. This index effectively targeted the key attributes in the WHERE clause, making it the optimal choice for retrieving top-rated affordable cars from selected companies.