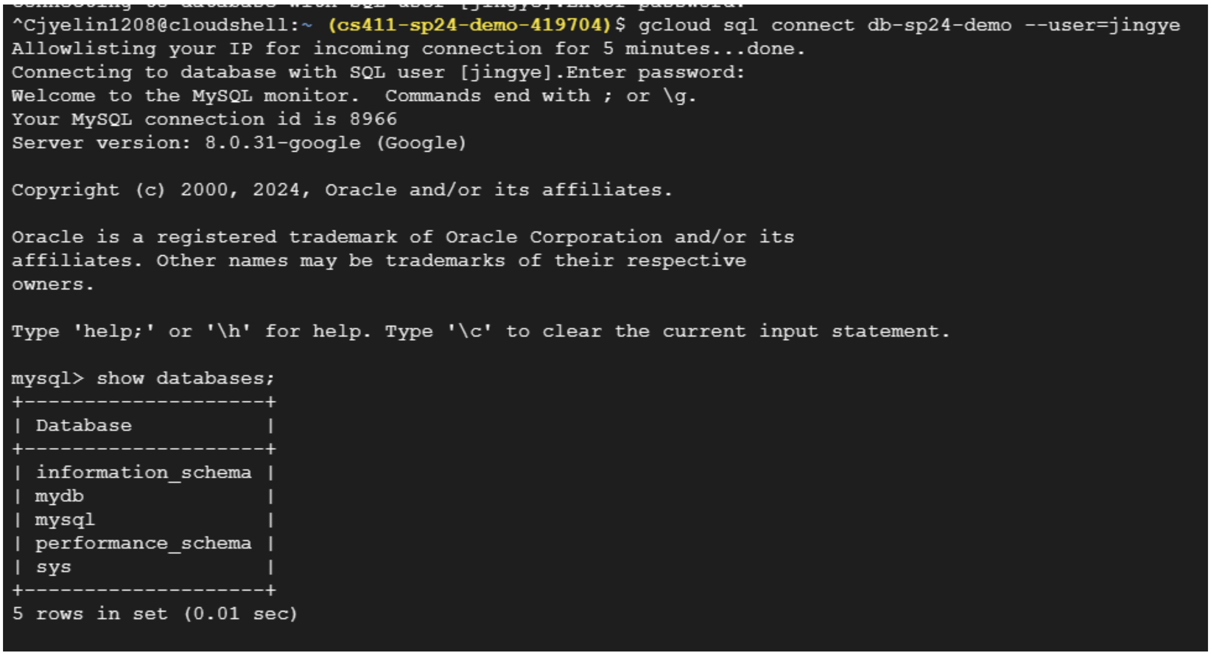
Connecting to GCP

```
gcloud sql connect db-sp24-demo --user=jingye;

show databases;

use mydb;

show tables;
```



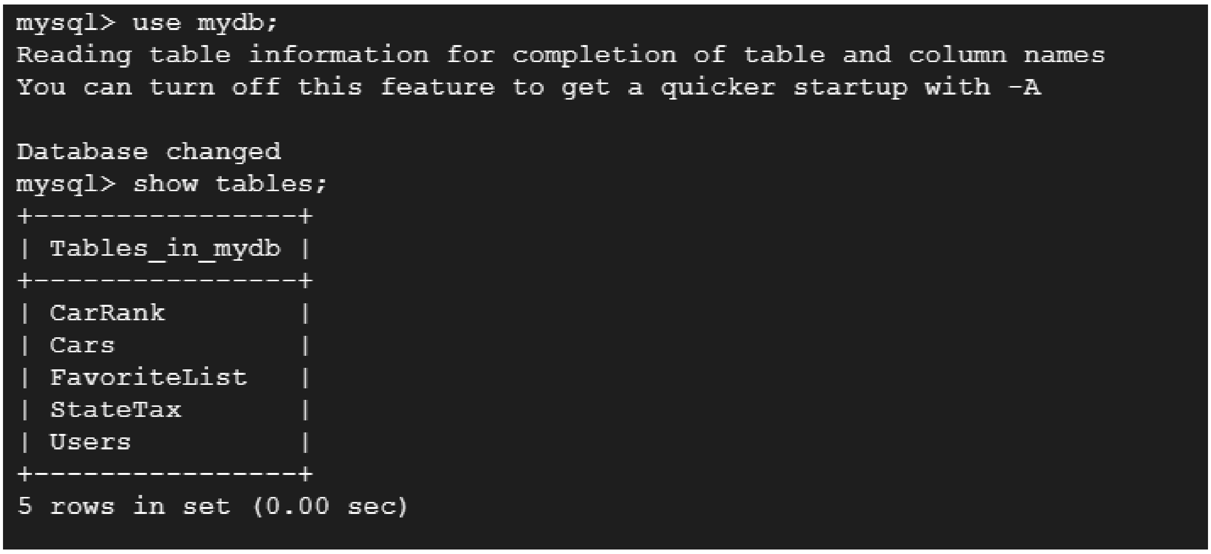
```
^Cjyelin1208@cloudshell:~ (cs411-sp24-demo-419704)$ gcloud sql connect db-sp24-demo --user=jingye
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [jingye].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8966
Server version: 8.0.31-google (Google)

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

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

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

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mydb               |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.01 sec)
```



```
mysql> use mydb;
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_mydb |
+----------------+
| CarRank        |
| Cars           |
| FavoriteList   |
| StateTax       |
| Users          |
+----------------+
5 rows in set (0.00 sec)
```

DDL Commands

-CarRank Table

```
CREATE TABLE CarRank(
    CarID DOUBLE,
    Model VARCHAR(50),
    Jan DOUBLE,
    Feb DOUBLE,
    Mar DOUBLE,
    Apr DOUBLE,
    May DOUBLE,
    Jun DOUBLE,
    Jul DOUBLE,
    Aug DOUBLE,
    Sep DOUBLE,
    Oct DOUBLE,
    Nov DOUBLE,
    Dec DOUBLE,
    PRIMARY KEY (CarID ),
    FOREIGN KEY (CarID) REFERENCES Cars(CarID)
);
```

-Cars Table

```
CREATE TABLE Cars(
CarID DOUBLE,
CarName VARCHAR(10),
Year DOUBLE,
SellingPrice DOUBLE,
Fuel VARCHAR(10),
Transmission VARCHAR(10),
PRIMARY KEY (userID),
FOREIGN KEY (stateID) REFERENCES StateTax(stateID)
);
```

-FavoriteList Table

```
CREATE TABLE FavoriteList(
ListID DOUBLE,
CarID DOUBLE,
UserID DOUBLE,
PRIMARY KEY (ListID),
FOREIGN KEY (CarID) REFERENCES Cars(CarID),
FOREIGN KEY (UserID) REFERENCES Users(UserID)
);
```

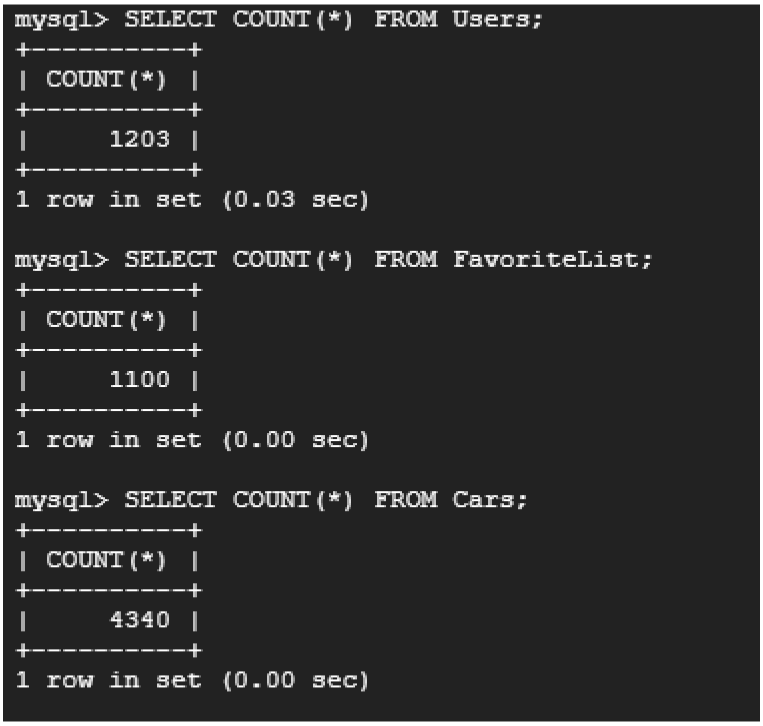
-StateTax Table

```
CREATE TABLE StateTax(
State VARCHAR(50),
StateID DOUBLE,
TaxRate DOUBLE,
PRIMARY KEY (StateID),
FOREIGN KEY (StateID) REFERENCES Users(StateID)
);
```

-Users Table

```
CREATE TABLE Users(
userID DOUBLE,
userName VARCHAR(10),
gender VARCHAR(10),
stateID DOUBLE,
PRIMARY KEY (userID),
FOREIGN KEY (StateID) REFERENCES StateTax(StateID)
);
```

Inserting Data



```
mysql> SELECT COUNT(*) FROM Users;
+----------+
| COUNT(*) |
+----------+
|     1203 |
+----------+
1 row in set (0.03 sec)

mysql> SELECT COUNT(*) FROM FavoriteList;
+----------+
| COUNT(*) |
+----------+
|     1100 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT COUNT(*) FROM Cars;
+----------+
| COUNT(*) |
+----------+
|     4340 |
+----------+
1 row in set (0.00 sec)
```

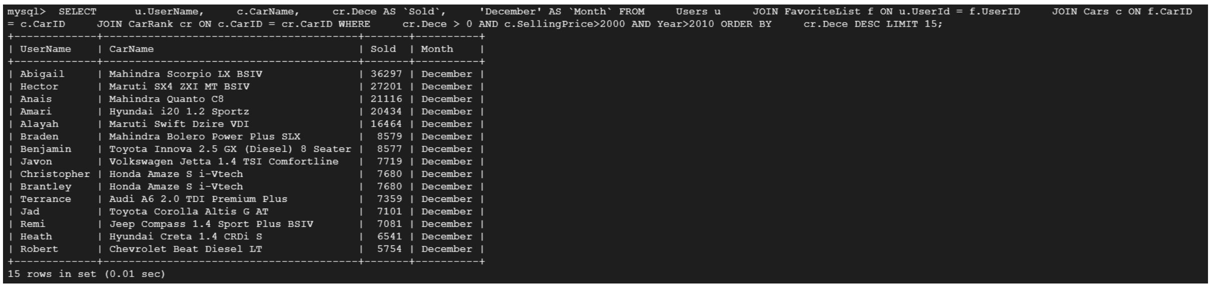
Advanced Queries

1.

Advanced features: Joining Multiple Relations & Conditional Filtering and Ordering

This SQL query is designed to retrieve a list of the top 15 favorite cars more expensive than 2000 and later than 2010 based on their sales in December.

```
SELECT
    u.UserName,
    c.CarName,
    cr.Dece AS `Sold`,
    'December' AS `Month`
FROM
    Users u
    JOIN FavoriteList f ON u.UserId = f.UserID
    JOIN Cars c ON f.CarID = c.CarID
    JOIN CarRank cr ON c.CarID = cr.CarID
WHERE
    cr.Dece > 0 AND c.SellingPrice>2000 AND Year>2010
ORDER BY
    cr.Dece DESC
LIMIT 15;
```



```
mysql>  SELECT      u.UserName,     c.CarName,     cr.Dece AS `Sold`,     'December' AS `Month` FROM     Users u     JOIN FavoriteList f ON u.UserId = f.UserID     JOIN Cars c ON f.CarID
= c.CarID     JOIN CarRank cr ON c.CarID = cr.CarID WHERE     cr.Dece > 0 AND c.SellingPrice>2000 AND Year>2010 ORDER BY     cr.Dece DESC LIMIT 15;
+-------------+----------------------------------------+-------+----------+
| UserName    | CarName                                | Sold  | Month    |
+-------------+----------------------------------------+-------+----------+
| Abigail     | Mahindra Scorpio LX BSIV               | 36297 | December |
| Hector      | Maruti SX4 ZXI MT BSIV                 | 27201 | December |
| Anais       | Mahindra Quanto C8                     | 21116 | December |
| Amari       | Hyundai i20 1.2 Sportz                 | 20434 | December |
| Alayah      | Maruti Swift Dzire VDI                 | 16464 | December |
| Braden      | Mahindra Bolero Power Plus SLX         |  8579 | December |
| Benjamin    | Toyota Innova 2.5 GX (Diesel) 8 Seater |  8577 | December |
| Javon       | Volkswagen Jetta 1.4 TSI Comfortline   |  7719 | December |
| Christopher | Honda Amaze S i-Vtech                  |  7680 | December |
| Brantley    | Honda Amaze S i-Vtech                  |  7680 | December |
| Terrance    | Audi A6 2.0 TDI Premium Plus           |  7359 | December |
| Jad         | Toyota Corolla Altis G AT              |  7101 | December |
| Remi        | Jeep Compass 1.4 Sport Plus BSIV       |  7081 | December |
| Heath       | Hyundai Creta 1.4 CRDi S               |  6541 | December |
| Robert      | Chevrolet Beat Diesel LT               |  5754 | December |
+-------------+----------------------------------------+-------+----------+
15 rows in set (0.01 sec)
```

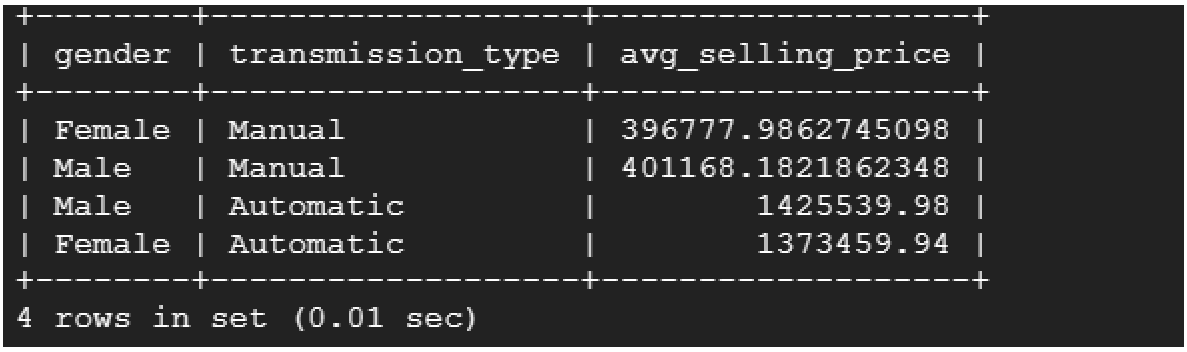
2.

Advanced features: Joining Multiple Relations & Aggregation with GROUP BY & Subqueries

This query is to calculate the average selling price of vehicles that are more expensive than 100,000 (manual transmission or automatic transmission) preferred by users of different genders.

```
SELECT
  u.gender,
  CASE
    WHEN c.Transmission = 'Manual' THEN 'Manual'
    WHEN c.Transmission = 'Automatic' THEN 'Automatic'
    ELSE 'Other'
  END AS transmission_type,
  AVG(c.SellingPrice) AS avg_selling_price
```

```
FROM
  FavoriteList f
JOIN
  Cars c ON f.CarID = c.CarID
JOIN
  Users u ON f.userID = u.userID
GROUP BY
  u.gender,
  Transmission_type
Having avg_selling_price > 100000;
```

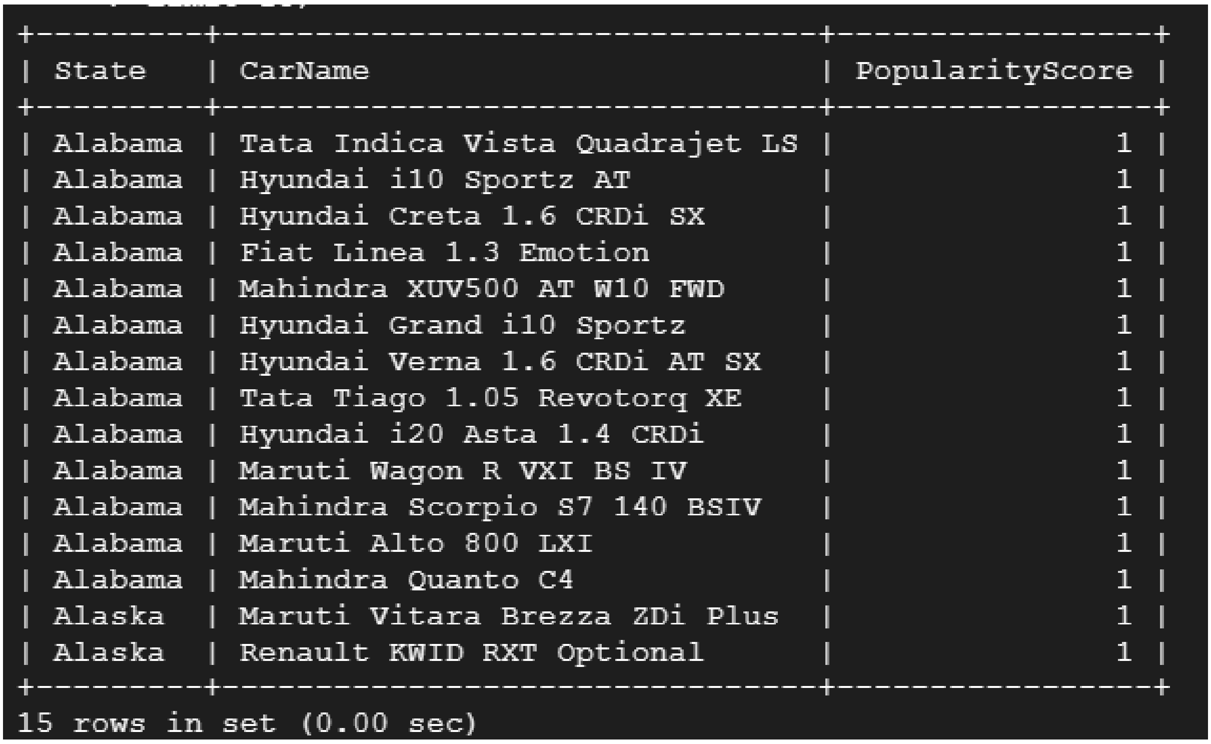


```
+--------+-------------------+-------------------+
| gender | transmission_type | avg_selling_price |
+--------+-------------------+-------------------+
| Female | Manual            | 396777.9862745098 |
| Male   | Manual            | 401168.1821862348 |
| Male   | Automatic         |        1425539.98 |
| Female | Automatic         |        1373459.94 |
+--------+-------------------+-------------------+
4 rows in set (0.01 sec)
```

3. Advanced features: Aggregation via Group By & Joining Multiple Relations
This query combines information from three tables to find the most popular cars by state.

```
SELECT
    StateTax.State,
    Cars.CarName,
    COUNT(FavoriteList.CarID) AS PopularityScore
FROM
    FavoriteList
JOIN
    Users ON FavoriteList.UserID = Users.userID
JOIN
    StateTax ON Users.stateID = StateTax.StateID
JOIN
    Cars ON FavoriteList.CarID = Cars.CarID
WHERE Cars.Year  > 2010
GROUP BY
    StateTax.State, Cars.CarName
ORDER BY
    StateTax.State,  PopularityScore DESC
```

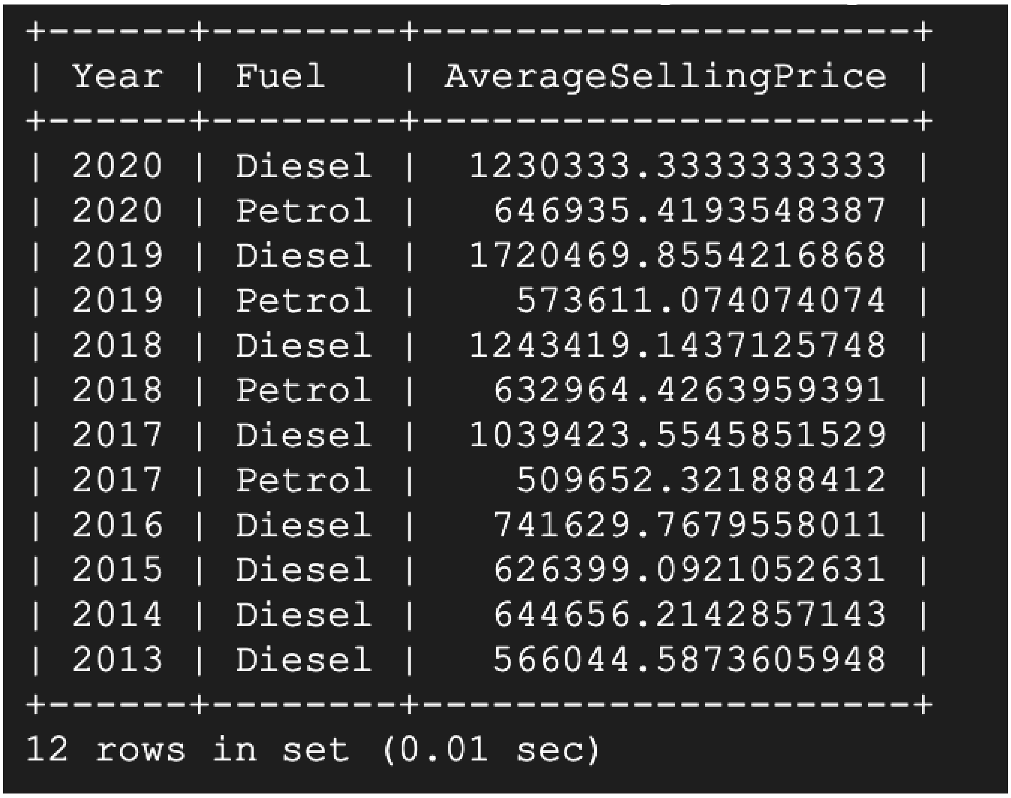
Limit 15;



```
+---------+--------------------------------+-----------------+
| State   | CarName                        | PopularityScore |
+---------+--------------------------------+-----------------+
| Alabama | Tata Indica Vista Quadrajet LS |               1 |
| Alabama | Hyundai i10 Sportz AT          |               1 |
| Alabama | Hyundai Creta 1.6 CRDi SX      |               1 |
| Alabama | Fiat Linea 1.3 Emotion         |               1 |
| Alabama | Mahindra XUV500 AT W10 FWD     |               1 |
| Alabama | Hyundai Grand i10 Sportz       |               1 |
| Alabama | Hyundai Verna 1.6 CRDi AT SX   |               1 |
| Alabama | Tata Tiago 1.05 Revotorq XE    |               1 |
| Alabama | Hyundai i20 Asta 1.4 CRDi      |               1 |
| Alabama | Maruti Wagon R VXI BS IV       |               1 |
| Alabama | Mahindra Scorpio S7 140 BSIV   |               1 |
| Alabama | Maruti Alto 800 LXI            |               1 |
| Alabama | Mahindra Quanto C4             |               1 |
| Alaska  | Maruti Vitara Brezza ZDi Plus  |               1 |
| Alaska  | Renault KWID RXT Optional      |               1 |
+---------+--------------------------------+-----------------+
15 rows in set (0.00 sec)
```

4. Advanced features: Aggregation via Group By & AVG Function & Having Condition & Order By
This query combines information from three tables to find the most popular cars by state.

```
SELECT
    Year,
    Fuel,
    AVG(SellingPrice) AS AverageSellingPrice
FROM
    Cars
GROUP BY
    Year, Fuel
HAVING
    AVG(SellingPrice) > (SELECT AVG(SellingPrice) FROM Cars WHERE Year <
YEAR(CURRENT_DATE()))
ORDER BY
    Year DESC, AverageSellingPrice DESC;
```



```
+------+--------+--------------------+
| Year | Fuel   | AverageSellingPrice |
+------+--------+--------------------+
| 2020 | Diesel |  1230333.3333333333 |
| 2020 | Petrol |   646935.4193548387 |
| 2019 | Diesel |  1720469.8554216868 |
| 2019 | Petrol |    573611.074074074 |
| 2018 | Diesel |  1243419.1437125748 |
| 2018 | Petrol |   632964.4263959391 |
| 2017 | Diesel |  1039423.5545851529 |
| 2017 | Petrol |    509652.321888412 |
| 2016 | Diesel |   741629.7679558011 |
| 2015 | Diesel |   626399.0921052631 |
| 2014 | Diesel |   644656.2142857143 |
| 2013 | Diesel |   566044.5873605948 |
+------+--------+--------------------+
12 rows in set (0.01 sec)
```

5. Advanced features: SUM function and Ordering

This query displays car details along with their total sales ranking score from January to December.

```
SELECT
   Cars.CarID,
   Cars.CarName,
   Cars.Year,
   Cars.SellingPrice,
   Cars.Fuel,
   Cars.Transmission,
   (SELECT SUM(Jan + Feb + Mar + Apr + May + Jun + Jul + Aug + Sep + Oct + Nov + Dece)
    FROM CarRank
    WHERE CarRank.CarID = Cars.CarID) AS TotalRankScore
FROM
   Cars
ORDER BY TotalRankScore DESC
LIMIT 15;
```

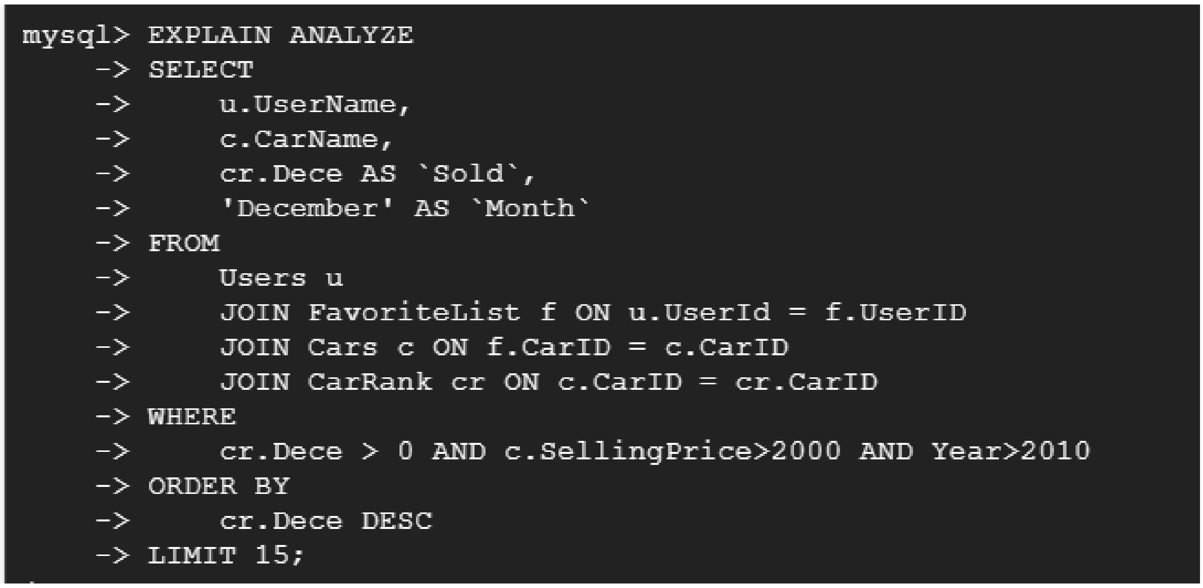
6. Advanced features: Joining Multiple Relations & Conditional filtering

This query shows detailed car information in a user’s favorite list, including how much tax they’d pay based on their state.

```
SELECT
    Users.userName,
    Cars.CarName,
    Cars.Year,
    Cars.SellingPrice,
    StateTax.TaxRate,
    (Cars.SellingPrice * (StateTax.TaxRate / 100)) AS TaxAmount
FROM
    FavoriteList
JOIN
    Cars ON FavoriteList.CarID = Cars.CarID
JOIN
    Users ON FavoriteList.UserID = Users.userID
JOIN
    StateTax ON Users.stateID = StateTax.StateID
WHERE
    FavoriteList.UserID = 311
Limit 15;
```

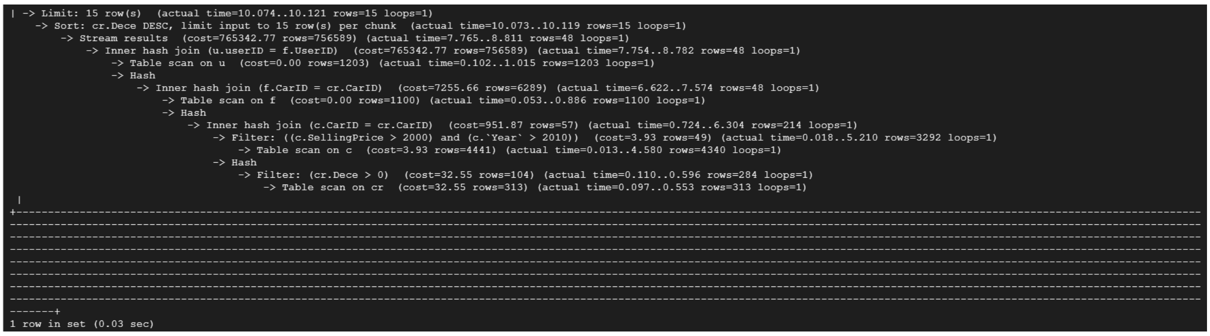
Indexing (for the first four queries)

Query 1



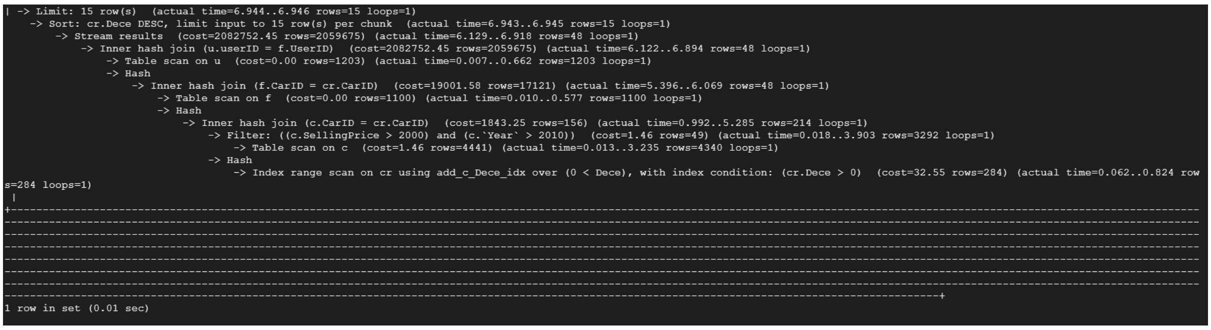
```
mysql> EXPLAIN ANALYZE
    -> SELECT
    ->     u.UserName,
    ->     c.CarName,
    ->     cr.Dece AS `Sold`,
    ->     'December' AS `Month`
    -> FROM
    ->     Users u
    ->     JOIN FavoriteList f ON u.UserId = f.UserID
    ->     JOIN Cars c ON f.CarID = c.CarID
    ->     JOIN CarRank cr ON c.CarID = cr.CarID
    -> WHERE
    ->     cr.Dece > 0 AND c.SellingPrice>2000 AND Year>2010
    -> ORDER BY
    ->     cr.Dece DESC
    -> LIMIT 15;
```

This query has a high cost due to our use of a set operation



```
| -> Limit: 15 row(s)  (actual time=10.074..10.121 rows=15 loops=1)
    -> Sort: cr.Dece DESC, limit input to 15 row(s) per chunk  (actual time=10.073..10.119 rows=15 loops=1)
        -> Stream results  (cost=765342.77 rows=756589) (actual time=7.765..8.811 rows=48 loops=1)
            -> Inner hash join (u.userID = f.UserID)  (cost=765342.77 rows=756589) (actual time=7.754..8.782 rows=48 loops=1)
                -> Table scan on u  (cost=0.00 rows=1203) (actual time=0.102..1.015 rows=1203 loops=1)
                -> Hash
                    -> Inner hash join (f.CarID = cr.CarID)  (cost=7255.66 rows=6289) (actual time=6.622..7.574 rows=48 loops=1)
                        -> Table scan on f  (cost=0.00 rows=1100) (actual time=0.053..0.886 rows=1100 loops=1)
                        -> Hash
                            -> Inner hash join (c.CarID = cr.CarID)  (cost=951.87 rows=57) (actual time=0.724..6.304 rows=214 loops=1)
                                -> Filter: ((c.SellingPrice > 2000) and (c.`Year` > 2010))  (cost=3.93 rows=49) (actual time=0.018..5.210 rows=3292 loops=1)
                                    -> Table scan on c  (cost=3.93 rows=4441) (actual time=0.013..4.580 rows=4340 loops=1)
                                -> Hash
                                    -> Filter: (cr.Dece > 0)  (cost=32.55 rows=104) (actual time=0.110..0.596 rows=284 loops=1)
                                        -> Table scan on cr  (cost=32.55 rows=313) (actual time=0.097..0.553 rows=313 loops=1)
 |
+----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------+
1 row in set (0.03 sec)
```

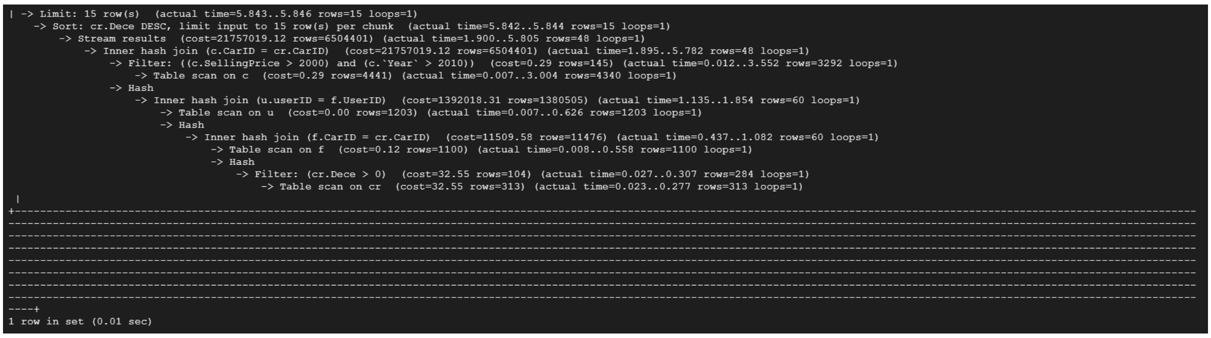
Attempt1: CREATE INDEX add_c_Dece_idx ON CarRank(Dece)



```
| -> Limit: 15 row(s)  (actual time=6.944..6.946 rows=15 loops=1)
    -> Sort: cr.Dece DESC, limit input to 15 row(s) per chunk  (actual time=6.943..6.945 rows=15 loops=1)
        -> Stream results  (cost=2082752.45 rows=2059675) (actual time=6.129..6.918 rows=48 loops=1)
            -> Inner hash join (u.userID = f.UserID)  (cost=2082752.45 rows=2059675) (actual time=6.122..6.894 rows=48 loops=1)
                -> Table scan on u  (cost=0.00 rows=1203) (actual time=0.007..0.662 rows=1203 loops=1)
                -> Hash
                    -> Inner hash join (f.CarID = cr.CarID)  (cost=19001.58 rows=17121) (actual time=5.396..6.069 rows=48 loops=1)
                        -> Table scan on f  (cost=0.00 rows=1100) (actual time=0.010..0.577 rows=1100 loops=1)
                        -> Hash
                            -> Inner hash join (c.CarID = cr.CarID)  (cost=1843.25 rows=156) (actual time=0.992..5.285 rows=214 loops=1)
                                -> Filter: ((c.SellingPrice > 2000) and (c.`Year` > 2010))  (cost=1.46 rows=49) (actual time=0.018..3.903 rows=3292 loops=1)
                                    -> Table scan on c  (cost=1.46 rows=4441) (actual time=0.013..3.235 rows=4340 loops=1)
                                -> Hash
                                    -> Index range scan on cr using add_c_Dece_idx over (0 < Dece), with index condition: (cr.Dece > 0)  (cost=32.55 rows=284) (actual time=0.062..0.824 row
s=284 loops=1)
 |
+----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

This design of the index is able to reduce the time. After utilizing an index scan on the joined table, the query efficiently located relevant data rows. This reduction in the number of times the table needed to be scanned led to a significant improvement in query performance. It reduces the time of scanning the table CarRank from 0.097 to 0.062. And also reduces the general cost in the first line.

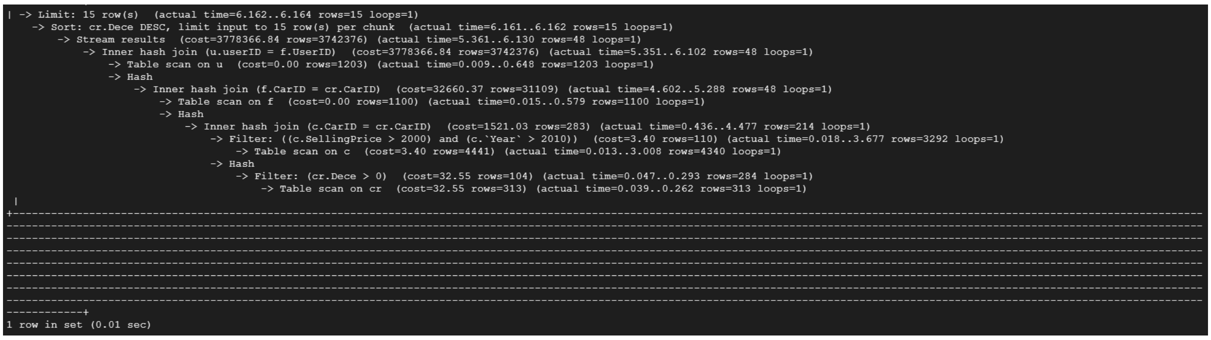
Attempt2: CREATE INDEX add_c_sellprice_idx ON Cars(SellingPrice)



```
| -> Limit: 15 row(s)  (actual time=5.843..5.846 rows=15 loops=1)
    -> Sort: cr.Dece DESC, limit input to 15 row(s) per chunk  (actual time=5.842..5.844 rows=15 loops=1)
        -> Stream results  (cost=21757019.12 rows=6504401) (actual time=1.900..5.805 rows=48 loops=1)
            -> Inner hash join (c.CarID = cr.CarID)  (cost=21757019.12 rows=6504401) (actual time=1.895..5.782 rows=48 loops=1)
                -> Filter: ((c.SellingPrice > 2000) and (c.`Year` > 2010))  (cost=0.29 rows=145) (actual time=0.012..3.552 rows=3292 loops=1)
                    -> Table scan on c  (cost=0.29 rows=4441) (actual time=0.007..3.004 rows=4340 loops=1)
                -> Hash
                    -> Inner hash join (u.userID = f.UserID)  (cost=1392018.31 rows=1380505) (actual time=1.135..1.854 rows=60 loops=1)
                        -> Table scan on u  (cost=0.00 rows=1203) (actual time=0.007..0.626 rows=1203 loops=1)
                        -> Hash
                            -> Inner hash join (f.CarID = cr.CarID)  (cost=11509.58 rows=11476) (actual time=0.437..1.082 rows=60 loops=1)
                                -> Table scan on f  (cost=0.12 rows=1100) (actual time=0.008..0.558 rows=1100 loops=1)
                                -> Hash
                                    -> Filter: (cr.Dece > 0)  (cost=32.55 rows=104) (actual time=0.027..0.307 rows=284 loops=1)
                                        -> Table scan on cr  (cost=32.55 rows=313) (actual time=0.023..0.277 rows=313 loops=1)
 |
+----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
----+
1 row in set (0.01 sec)
```

This index design effectively reduces query execution time. By utilizing an index scan on the joined table, the query efficiently locates relevant data rows. This reduction in the number of times the table needs to be scanned significantly improves query performance. It also minimizes the overall cost, as mentioned in the first line.

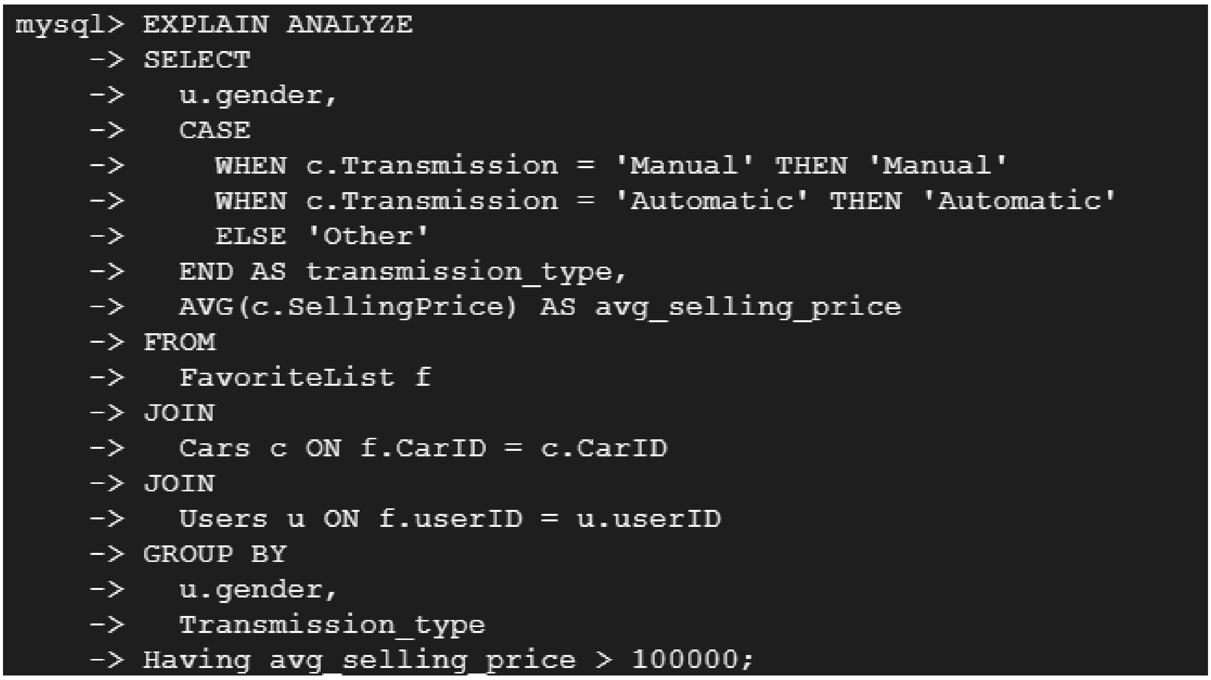
Attempt3: CREATE INDEX add_c_year_idx ON Cars(Year)



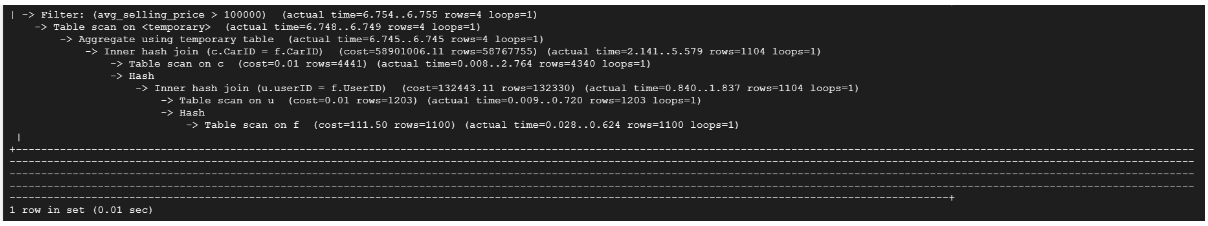
```
| -> Limit: 15 row(s)  (actual time=6.162..6.164 rows=15 loops=1)
    -> Sort: cr.Dece DESC, limit input to 15 row(s) per chunk  (actual time=6.161..6.162 rows=15 loops=1)
        -> Stream results  (cost=3778366.84 rows=3742376) (actual time=5.361..6.130 rows=48 loops=1)
            -> Inner hash join (u.userID = f.UserID)  (cost=3778366.84 rows=3742376) (actual time=5.351..6.102 rows=48 loops=1)
                -> Table scan on u  (cost=0.00 rows=1203) (actual time=0.009..0.648 rows=1203 loops=1)
                -> Hash
                    -> Inner hash join (f.CarID = cr.CarID)  (cost=32660.37 rows=31109) (actual time=4.602..5.288 rows=48 loops=1)
                        -> Table scan on f  (cost=0.00 rows=1100) (actual time=0.015..0.579 rows=1100 loops=1)
                        -> Hash
                            -> Inner hash join (c.CarID = cr.CarID)  (cost=1521.03 rows=283) (actual time=0.436..4.477 rows=214 loops=1)
                                -> Filter: ((c.SellingPrice > 2000) and (c.`Year` > 2010))  (cost=3.40 rows=110) (actual time=0.018..3.677 rows=3292 loops=1)
                                    -> Table scan on c  (cost=3.40 rows=4441) (actual time=0.013..3.008 rows=4340 loops=1)
                                -> Hash
                                    -> Filter: (cr.Dece > 0)  (cost=32.55 rows=104) (actual time=0.047..0.293 rows=284 loops=1)
                                        -> Table scan on cr  (cost=32.55 rows=313) (actual time=0.039..0.262 rows=313 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------+
1 row in set (0.01 sec)
```

This index design significantly reduces query execution time. By employing an index scan on the joined table, the query efficiently identifies relevant data rows. Consequently, the reduced number of table scans results in a substantial improvement in query performance, leading to cost savings overall.

Query 2

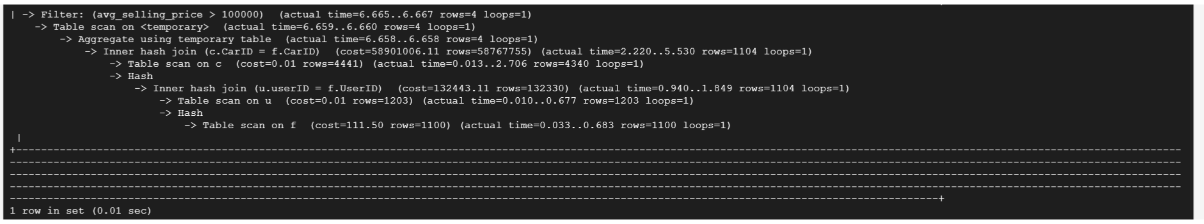


```
mysql> EXPLAIN ANALYZE
    -> SELECT
    ->   u.gender,
    ->   CASE
    ->     WHEN c.Transmission = 'Manual' THEN 'Manual'
    ->     WHEN c.Transmission = 'Automatic' THEN 'Automatic'
    ->     ELSE 'Other'
    ->   END AS transmission_type,
    ->   AVG(c.SellingPrice) AS avg_selling_price
    -> FROM
    ->   FavoriteList f
    -> JOIN
    ->   Cars c ON f.CarID = c.CarID
    -> JOIN
    ->   Users u ON f.userID = u.userID
    -> GROUP BY
    ->   u.gender,
    ->   Transmission_type
    -> Having avg_selling_price > 100000;
```



```
| -> Filter: (avg_selling_price > 100000)  (actual time=6.754..6.755 rows=4 loops=1)
    -> Table scan on <temporary>  (actual time=6.748..6.749 rows=4 loops=1)
        -> Aggregate using temporary table  (actual time=6.745..6.745 rows=4 loops=1)
            -> Inner hash join (c.CarID = f.CarID)  (cost=58901006.11 rows=58767755) (actual time=2.141..5.579 rows=1104 loops=1)
                -> Table scan on c  (cost=0.01 rows=4441) (actual time=0.008..2.764 rows=4340 loops=1)
                -> Hash
                    -> Inner hash join (u.userID = f.UserID)  (cost=132443.11 rows=132330) (actual time=0.840..1.837 rows=1104 loops=1)
                        -> Table scan on u  (cost=0.01 rows=1203) (actual time=0.009..0.720 rows=1203 loops=1)
                        -> Hash
                            -> Table scan on f  (cost=111.50 rows=1100) (actual time=0.028..0.624 rows=1100 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

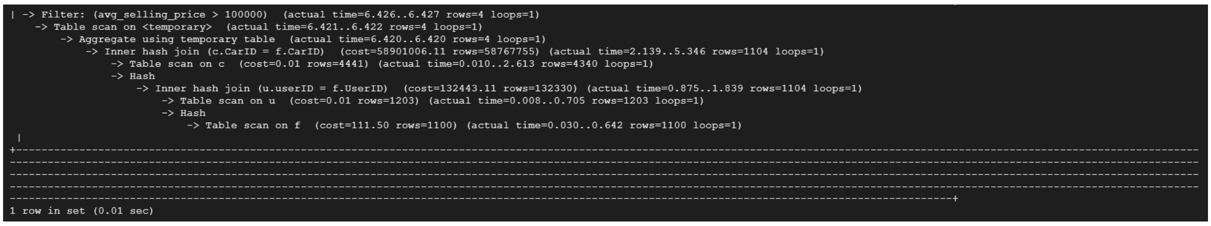
Attempt1: CREATE INDEX add_u_genderid_idx ON Users(gender)



```
| -> Filter: (avg_selling_price > 100000)  (actual time=6.665..6.667 rows=4 loops=1)
    -> Table scan on <temporary>  (actual time=6.659..6.660 rows=4 loops=1)
        -> Aggregate using temporary table  (actual time=6.658..6.658 rows=4 loops=1)
            -> Inner hash join (c.CarID = f.CarID)  (cost=58901006.11 rows=58767755) (actual time=2.220..5.530 rows=1104 loops=1)
                -> Table scan on c  (cost=0.01 rows=4441) (actual time=0.013..2.706 rows=4340 loops=1)
                -> Hash
                    -> Inner hash join (u.userID = f.UserID)  (cost=132443.11 rows=132330) (actual time=0.940..1.849 rows=1104 loops=1)
                        -> Table scan on u  (cost=0.01 rows=1203) (actual time=0.010..0.677 rows=1203 loops=1)
                        -> Hash
                            -> Table scan on f  (cost=111.50 rows=1100) (actual time=0.033..0.683 rows=1100 loops=1)
 |
+------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

There is no significant change in actual time of all and that of aggregate. This might be because gender itself has few different values, making the index no sense. And group by need to scan the table again.

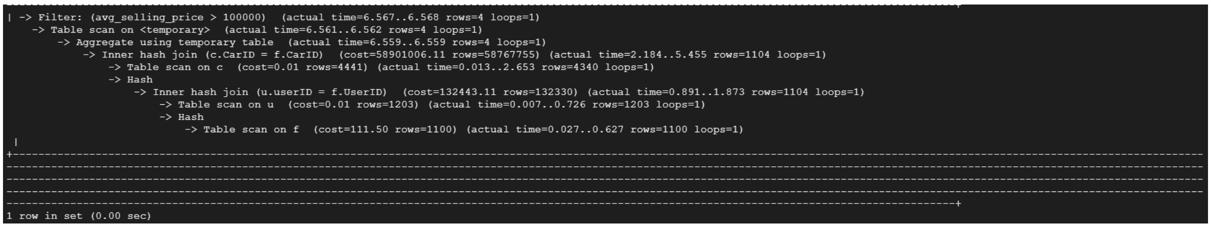
Attempt2: CREATE INDEX add_c_transmission_idx ON Cars(Transmission)



```
| -> Filter: (avg_selling_price > 100000)  (actual time=6.426..6.427 rows=4 loops=1)
    -> Table scan on <temporary>  (actual time=6.421..6.422 rows=4 loops=1)
        -> Aggregate using temporary table  (actual time=6.420..6.420 rows=4 loops=1)
            -> Inner hash join (c.CarID = f.CarID)  (cost=58901006.11 rows=58767755) (actual time=2.139..5.346 rows=1104 loops=1)
                -> Table scan on c  (cost=0.01 rows=4441) (actual time=0.010..2.613 rows=4340 loops=1)
                -> Hash
                    -> Inner hash join (u.userID = f.UserID)  (cost=132443.11 rows=132330) (actual time=0.875..1.839 rows=1104 loops=1)
                        -> Table scan on u  (cost=0.01 rows=1203) (actual time=0.008..0.705 rows=1203 loops=1)
                        -> Hash
                            -> Table scan on f  (cost=111.50 rows=1100) (actual time=0.030..0.642 rows=1100 loops=1)
 |
+------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

There is no significant change in actual time of all and that of aggregate. This might be because transmission itself has few different values, making the index no sense. And group by need to scan the table again.

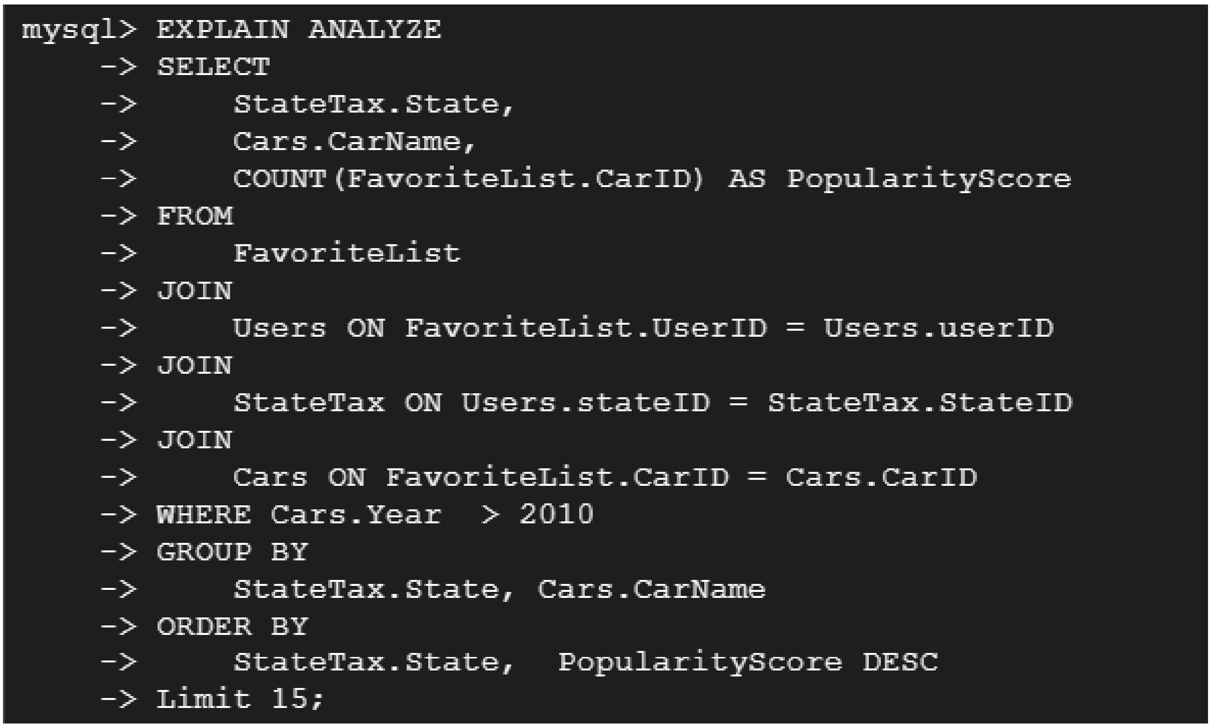
Attempt3: CREATE INDEX add_c_sellprice_idx ON Cars(SellingPrice)



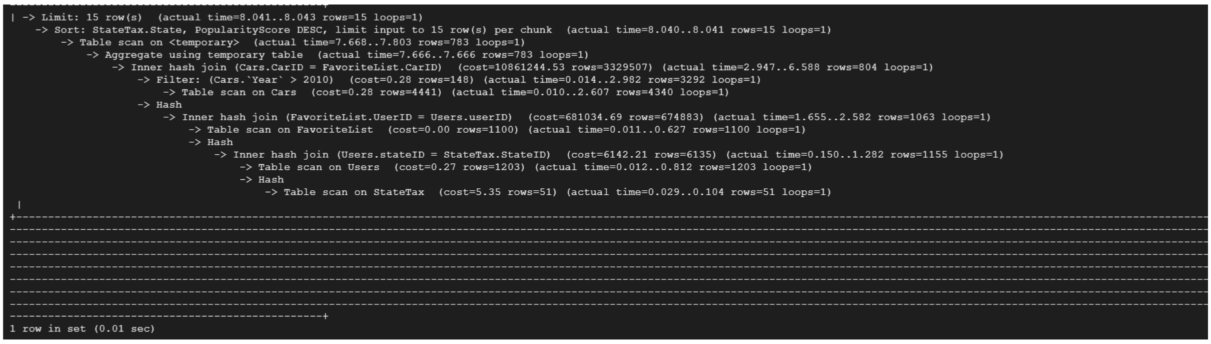
```
| -> Filter: (avg_selling_price > 100000)  (actual time=6.567..6.568 rows=4 loops=1)
    -> Table scan on <temporary>  (actual time=6.561..6.562 rows=4 loops=1)
        -> Aggregate using temporary table  (actual time=6.559..6.559 rows=4 loops=1)
            -> Inner hash join (c.CarID = f.CarID)  (cost=58901006.11 rows=58767755) (actual time=2.184..5.455 rows=1104 loops=1)
                -> Table scan on c  (cost=0.01 rows=4441) (actual time=0.013..2.653 rows=4340 loops=1)
                -> Hash
                    -> Inner hash join (u.userID = f.UserID)  (cost=132443.11 rows=132330) (actual time=0.891..1.873 rows=1104 loops=1)
                        -> Table scan on u  (cost=0.01 rows=1203) (actual time=0.007..0.726 rows=1203 loops=1)
                        -> Hash
                            -> Table scan on f  (cost=111.50 rows=1100) (actual time=0.027..0.627 rows=1100 loops=1)
 |
+------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

There is no significant change in actual time of all and that of aggregate. This might be because the average selling price will be calculated again, which means the origin value should be scanned from the table.

Query 3

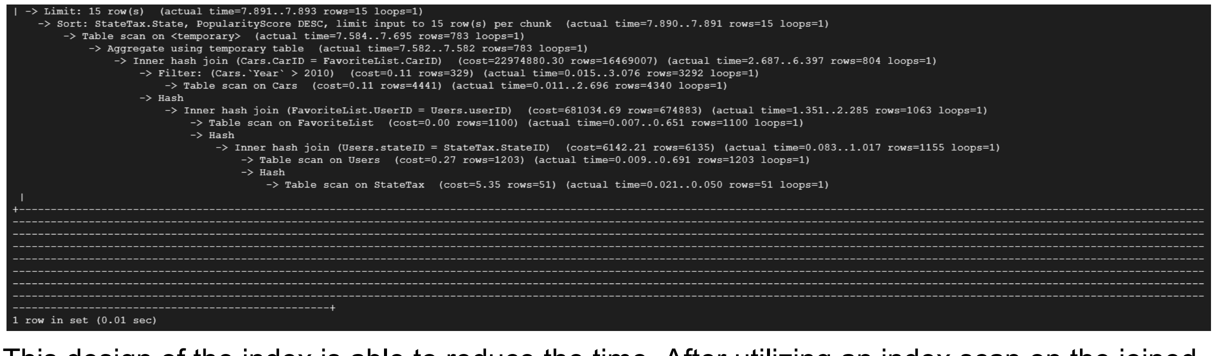


```
mysql> EXPLAIN ANALYZE
    -> SELECT
    ->     StateTax.State,
    ->     Cars.CarName,
    ->     COUNT(FavoriteList.CarID) AS PopularityScore
    -> FROM
    ->     FavoriteList
    -> JOIN
    ->     Users ON FavoriteList.UserID = Users.userID
    -> JOIN
    ->     StateTax ON Users.stateID = StateTax.StateID
    -> JOIN
    ->     Cars ON FavoriteList.CarID = Cars.CarID
    -> WHERE Cars.Year  > 2010
    -> GROUP BY
    ->     StateTax.State, Cars.CarName
    -> ORDER BY
    ->     StateTax.State,  PopularityScore DESC
    -> Limit 15;
```



```
| -> Limit: 15 row(s)  (actual time=8.041..8.043 rows=15 loops=1)
    -> Sort: StateTax.State, PopularityScore DESC, limit input to 15 row(s) per chunk  (actual time=8.040..8.041 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=7.668..7.803 rows=783 loops=1)
            -> Aggregate using temporary table  (actual time=7.666..7.666 rows=783 loops=1)
                -> Inner hash join (Cars.CarID = FavoriteList.CarID)  (cost=10861244.53 rows=3329507) (actual time=2.947..6.588 rows=804 loops=1)
                    -> Filter: (Cars.`Year` > 2010)  (cost=0.28 rows=148) (actual time=0.014..2.982 rows=3292 loops=1)
                        -> Table scan on Cars  (cost=0.28 rows=4441) (actual time=0.010..2.607 rows=4340 loops=1)
                    -> Hash
                        -> Inner hash join (FavoriteList.UserID = Users.userID)  (cost=681034.69 rows=674883) (actual time=1.655..2.582 rows=1063 loops=1)
                            -> Table scan on FavoriteList  (cost=0.00 rows=1100) (actual time=0.011..0.627 rows=1100 loops=1)
                            -> Hash
                                -> Inner hash join (Users.stateID = StateTax.StateID)  (cost=6142.21 rows=6135) (actual time=0.150..1.282 rows=1155 loops=1)
                                    -> Table scan on Users  (cost=0.27 rows=1203) (actual time=0.012..0.812 rows=1203 loops=1)
                                    -> Hash
                                        -> Table scan on StateTax  (cost=5.35 rows=51) (actual time=0.029..0.104 rows=51 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------
--------------------------------------------------+
1 row in set (0.01 sec)
```

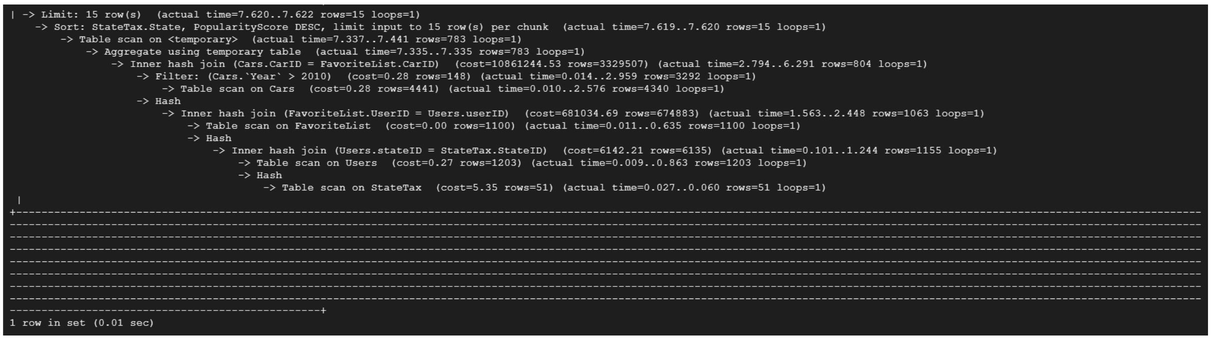
Attempt1: CREATE INDEX add_c_year_idx ON Cars(Year)



```
| -> Limit: 15 row(s)  (actual time=7.891..7.893 rows=15 loops=1)
    -> Sort: StateTax.State, PopularityScore DESC, limit input to 15 row(s) per chunk  (actual time=7.890..7.891 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=7.584..7.695 rows=783 loops=1)
            -> Aggregate using temporary table  (actual time=7.582..7.582 rows=783 loops=1)
                -> Inner hash join (Cars.CarID = FavoriteList.CarID)  (cost=22974880.30 rows=16469007) (actual time=2.687..6.397 rows=804 loops=1)
                    -> Filter: (Cars.`Year` > 2010)  (cost=0.11 rows=329) (actual time=0.015..3.076 rows=3292 loops=1)
                        -> Table scan on Cars  (cost=0.11 rows=4441) (actual time=0.011..2.696 rows=4340 loops=1)
                    -> Hash
                        -> Inner hash join (FavoriteList.UserID = Users.userID)  (cost=681034.69 rows=674883) (actual time=1.351..2.285 rows=1063 loops=1)
                            -> Table scan on FavoriteList  (cost=0.00 rows=1100) (actual time=0.007..0.651 rows=1100 loops=1)
                            -> Hash
                                -> Inner hash join (Users.stateID = StateTax.StateID)  (cost=6142.21 rows=6135) (actual time=0.083..1.017 rows=1155 loops=1)
                                    -> Table scan on Users  (cost=0.27 rows=1203) (actual time=0.009..0.691 rows=1203 loops=1)
                                    -> Hash
                                        -> Table scan on StateTax  (cost=5.35 rows=51) (actual time=0.021..0.050 rows=51 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------
--------------------------------------------------+
1 row in set (0.01 sec)
```

This design of the index is able to reduce the time. After utilizing an index scan on the joined table, the query efficiently located relevant data rows. This reduction in the number of times the table needed to be scanned led to a significant improvement in query performance. It reduces the general cost in the first line.

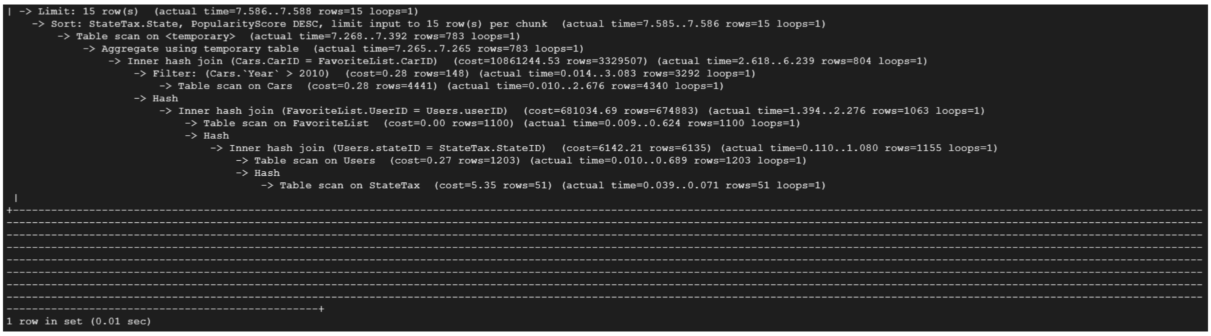
Attempt2: CREATE INDEX add_st_State_idx ON StateTax(State)



```
| -> Limit: 15 row(s)  (actual time=7.620..7.622 rows=15 loops=1)
    -> Sort: StateTax.State, PopularityScore DESC, limit input to 15 row(s) per chunk  (actual time=7.619..7.620 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=7.337..7.441 rows=783 loops=1)
            -> Aggregate using temporary table  (actual time=7.335..7.335 rows=783 loops=1)
                -> Inner hash join (Cars.CarID = FavoriteList.CarID)  (cost=10861244.53 rows=3329507) (actual time=2.794..6.291 rows=804 loops=1)
                    -> Filter: (Cars.`Year` > 2010)  (cost=0.28 rows=148) (actual time=0.014..2.959 rows=3292 loops=1)
                        -> Table scan on Cars  (cost=0.28 rows=4441) (actual time=0.010..2.576 rows=4340 loops=1)
                    -> Hash
                        -> Inner hash join (FavoriteList.UserID = Users.userID)  (cost=681034.69 rows=674883) (actual time=1.563..2.448 rows=1063 loops=1)
                            -> Table scan on FavoriteList  (cost=0.00 rows=1100) (actual time=0.011..0.635 rows=1100 loops=1)
                            -> Hash
                                -> Inner hash join (Users.stateID = StateTax.StateID)  (cost=6142.21 rows=6135) (actual time=0.101..1.244 rows=1155 loops=1)
                                    -> Table scan on Users  (cost=0.27 rows=1203) (actual time=0.009..0.863 rows=1203 loops=1)
                                    -> Hash
                                        -> Table scan on StateTax  (cost=5.35 rows=51) (actual time=0.027..0.060 rows=51 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------+
1 row in set (0.01 sec)
```

This index design significantly reduces query execution time. By employing an index scan on the joined table, the query efficiently identifies relevant data rows. Consequently, the reduced number of table scans results in a substantial improvement in query performance, leading to cost savings overall.

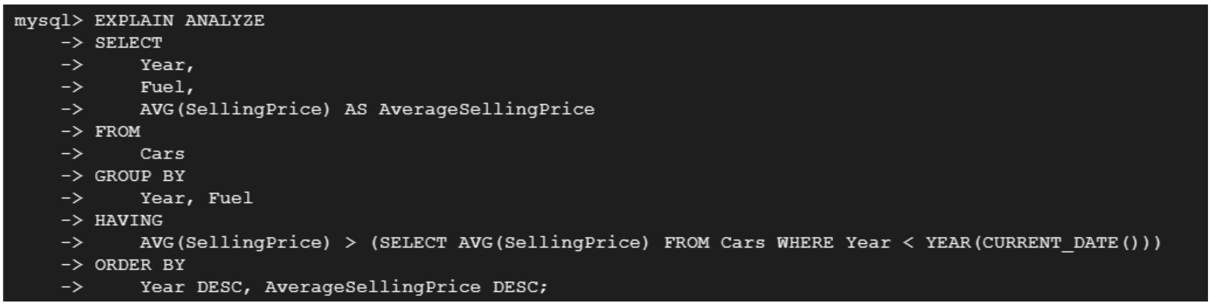
Attempt3: CREATE INDEX add_c_CarName_idx ON Cars(CarName)



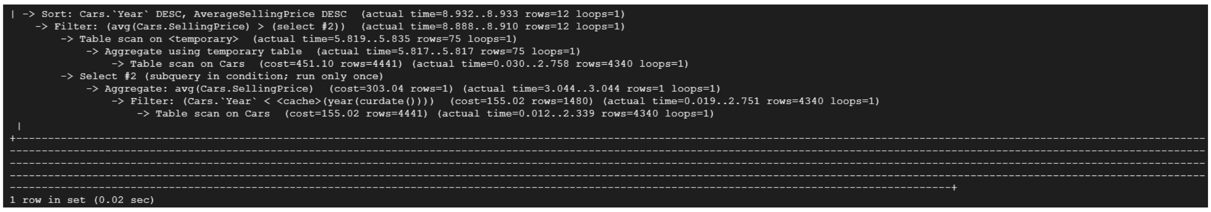
```
| -> Limit: 15 row(s)  (actual time=7.586..7.588 rows=15 loops=1)
    -> Sort: StateTax.State, PopularityScore DESC, limit input to 15 row(s) per chunk  (actual time=7.585..7.586 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=7.268..7.392 rows=783 loops=1)
            -> Aggregate using temporary table  (actual time=7.265..7.265 rows=783 loops=1)
                -> Inner hash join (Cars.CarID = FavoriteList.CarID)  (cost=10861244.53 rows=3329507) (actual time=2.618..6.239 rows=804 loops=1)
                    -> Filter: (Cars.`Year` > 2010)  (cost=0.28 rows=148) (actual time=0.014..3.083 rows=3292 loops=1)
                        -> Table scan on Cars  (cost=0.28 rows=4441) (actual time=0.010..2.676 rows=4340 loops=1)
                    -> Hash
                        -> Inner hash join (FavoriteList.UserID = Users.userID)  (cost=681034.69 rows=674883) (actual time=1.394..2.276 rows=1063 loops=1)
                            -> Table scan on FavoriteList  (cost=0.00 rows=1100) (actual time=0.009..0.624 rows=1100 loops=1)
                            -> Hash
                                -> Inner hash join (Users.stateID = StateTax.StateID)  (cost=6142.21 rows=6135) (actual time=0.110..1.080 rows=1155 loops=1)
                                    -> Table scan on Users  (cost=0.27 rows=1203) (actual time=0.010..0.689 rows=1203 loops=1)
                                    -> Hash
                                        -> Table scan on StateTax  (cost=5.35 rows=51) (actual time=0.039..0.071 rows=51 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------+
1 row in set (0.01 sec)
```

This index design of car name in cars table is similar with car year, which reduced number of table scans.

Query 4

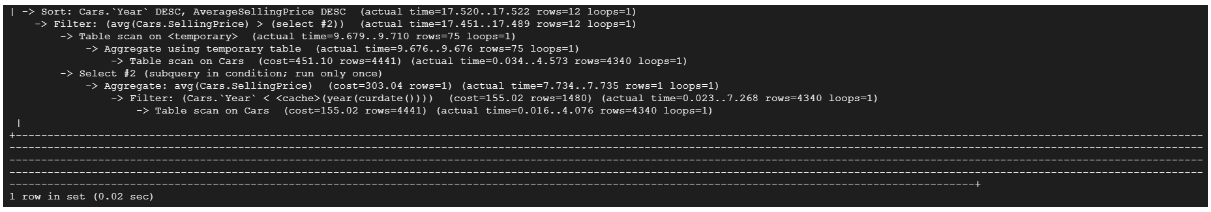


```
mysql> EXPLAIN ANALYZE
    -> SELECT
    ->     Year,
    ->     Fuel,
    ->     AVG(SellingPrice) AS AverageSellingPrice
    -> FROM
    ->     Cars
    -> GROUP BY
    ->     Year, Fuel
    -> HAVING
    ->     AVG(SellingPrice) > (SELECT AVG(SellingPrice) FROM Cars WHERE Year < YEAR(CURRENT_DATE()))
    -> ORDER BY
    ->     Year DESC, AverageSellingPrice DESC;
```



```
| -> Sort: Cars.`Year` DESC, AverageSellingPrice DESC  (actual time=8.932..8.933 rows=12 loops=1)
    -> Filter: (avg(Cars.SellingPrice) > (select #2))  (actual time=8.888..8.910 rows=12 loops=1)
        -> Table scan on <temporary>  (actual time=5.819..5.835 rows=75 loops=1)
            -> Aggregate using temporary table  (actual time=5.817..5.817 rows=75 loops=1)
                -> Table scan on Cars  (cost=451.10 rows=4441) (actual time=0.030..2.758 rows=4340 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(Cars.SellingPrice)  (cost=303.04 rows=1) (actual time=3.044..3.044 rows=1 loops=1)
                -> Filter: (Cars.`Year` < <cache>(year(curdate())))  (cost=155.02 rows=1480) (actual time=0.019..2.751 rows=4340 loops=1)
                    -> Table scan on Cars  (cost=155.02 rows=4441) (actual time=0.012..2.339 rows=4340 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.02 sec)
```

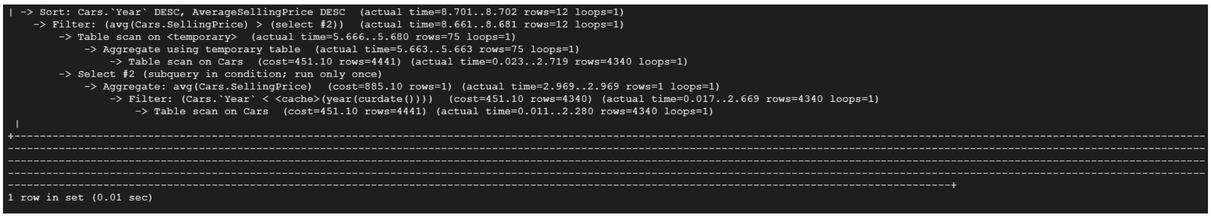
Attempt1: CREATE INDEX add_c_fuel_idx ON Cars(Fuel)



```
| -> Sort: Cars.`Year` DESC, AverageSellingPrice DESC  (actual time=17.520..17.522 rows=12 loops=1)
    -> Filter: (avg(Cars.SellingPrice) > (select #2))  (actual time=17.451..17.489 rows=12 loops=1)
        -> Table scan on <temporary>  (actual time=9.679..9.710 rows=75 loops=1)
            -> Aggregate using temporary table  (actual time=9.676..9.676 rows=75 loops=1)
                -> Table scan on Cars  (cost=451.10 rows=4441) (actual time=0.034..4.573 rows=4340 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(Cars.SellingPrice)  (cost=303.04 rows=1) (actual time=7.734..7.735 rows=1 loops=1)
                -> Filter: (Cars.`Year` < <cache>(year(curdate())))  (cost=155.02 rows=1480) (actual time=0.023..7.268 rows=4340 loops=1)
                    -> Table scan on Cars  (cost=155.02 rows=4441) (actual time=0.016..4.076 rows=4340 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.02 sec)
```

This design increased the total actual time. Since the cars table still needs to be scanned with year and price, this design makes it more complicated and costs more time.

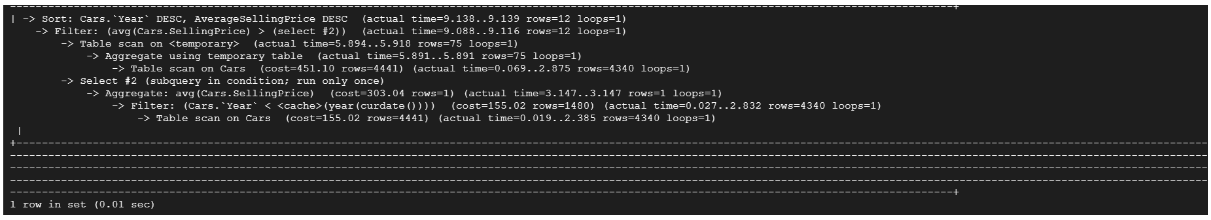
Attempt2: CREATE INDEX add_c_year_idx ON Cars(Year)



```
| -> Sort: Cars.`Year` DESC, AverageSellingPrice DESC  (actual time=8.701..8.702 rows=12 loops=1)
    -> Filter: (avg(Cars.SellingPrice) > (select #2))  (actual time=8.661..8.681 rows=12 loops=1)
        -> Table scan on <temporary>  (actual time=5.666..5.680 rows=75 loops=1)
            -> Aggregate using temporary table  (actual time=5.663..5.663 rows=75 loops=1)
                -> Table scan on Cars  (cost=451.10 rows=4441) (actual time=0.023..2.719 rows=4340 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(Cars.SellingPrice)  (cost=885.10 rows=1) (actual time=2.969..2.969 rows=1 loops=1)
                -> Filter: (Cars.`Year` < <cache>(year(curdate())))  (cost=451.10 rows=4340) (actual time=0.017..2.669 rows=4340 loops=1)
                    -> Table scan on Cars  (cost=451.10 rows=4441) (actual time=0.011..2.280 rows=4340 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

This design does not make improvements. This might be because the table scan cannot be reduced when using the car year index.

Attempt3: CREATE INDEX add_c_sellprice_idx ON Cars(SellingPrice)



```
--------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Sort: Cars.`Year` DESC, AverageSellingPrice DESC  (actual time=9.138..9.139 rows=12 loops=1)
    -> Filter: (avg(Cars.SellingPrice) > (select #2))  (actual time=9.088..9.116 rows=12 loops=1)
        -> Table scan on <temporary>  (actual time=5.894..5.918 rows=75 loops=1)
            -> Aggregate using temporary table  (actual time=5.891..5.891 rows=75 loops=1)
                -> Table scan on Cars  (cost=451.10 rows=4441) (actual time=0.069..2.875 rows=4340 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(Cars.SellingPrice)  (cost=303.04 rows=1) (actual time=3.147..3.147 rows=1 loops=1)
                -> Filter: (Cars.`Year` < <cache>(year(curdate())))  (cost=155.02 rows=1480) (actual time=0.027..2.832 rows=4340 loops=1)
                    -> Table scan on Cars  (cost=155.02 rows=4441) (actual time=0.019..2.385 rows=4340 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

This design of using sell price as index does not make any improvement on the actual time. It does not reduce the time of aggregate process and also no reduce of table scan.