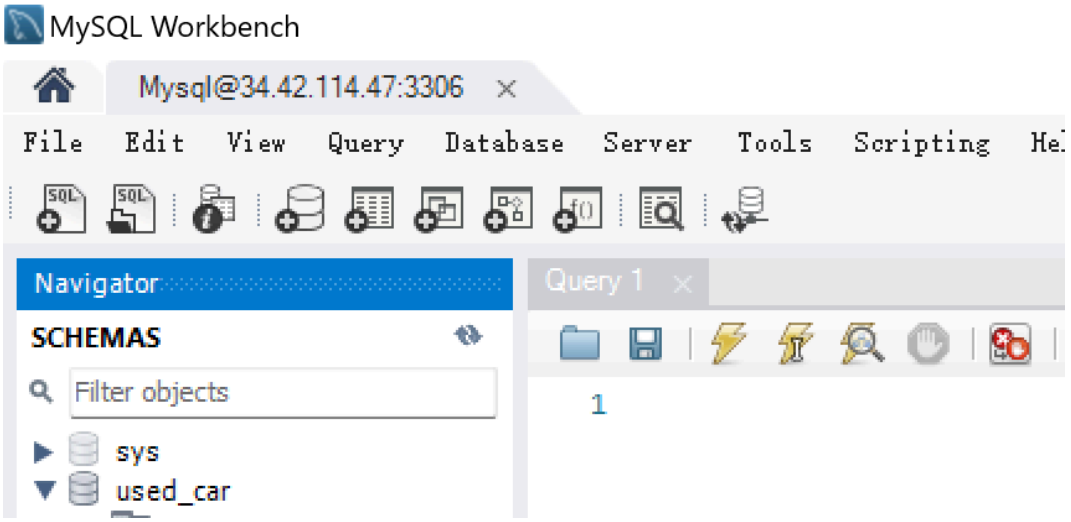
# Stage 3: Database Implementation and Indexing

## I. Database implementation

### access to the database(from MySQL Workbench)




### DDL commands

- **Table Location**

```sql
CREATE TABLE Location (
    id INT PRIMARY KEY,
    name  VARCHAR(100) UNIQUE,
    state_id INT,
    state_code INT,
    state_name VARCHAR(100),
    country_id INT,
    country_code CHAR(2),
    country_name VARCHAR(100),
    latitude DECIMAL(9,6),
    longitude DECIMAL(9,6),
    wikiDataId VARCHAR(20)
);
```

100% 19:4

Result Grid Filter Rows: Search Export:

| Field | Type | Null | Key | Default | Extra |
|---|---|---|---|---|---|
| id | int | NO | PRI | NULL | |
| name | varchar(100) | YES | UNI | NULL | |
| state_id | int | YES | | NULL | |
| state_code | int | YES | | NULL | |
| state_name | varchar(100) | YES | | NULL | |
| country_id | int | YES | | NULL | |
| country_code | char(2) | YES | | NULL | |
| country_name | varchar(100) | YES | | NULL | |
| latitude | decimal(9,6) | YES | | NULL | |
| longitude | decimal(9,6) | YES | | NULL | |
| wikiDataId | varchar(20) | YES | | NULL | |

- **Table User**

```
CREATE TABLE User (
    UserID INT PRIMARY KEY,
    IsSeller BOOLEAN,
    Name VARCHAR(30),
    Email VARCHAR(40),
    BirthDate DATETIME,
    Password VARCHAR(50),
    Locations VARCHAR(50),
    Phone VARCHAR(30),
    FOREIGN KEY (Locations) REFERENCES Location(name)
);
```



100% 19:4

Result Grid Filter Rows: Search Export:

| Field | Type | Null | Key | Default | Extra |
|---|---|---|---|---|---|
| id | int | NO | PRI | NULL | |
| name | varchar(100) | YES | UNI | NULL | |
| state_id | int | YES | | NULL | |
| state_code | int | YES | | NULL | |
| state_name | varchar(100) | YES | | NULL | |
| country_id | int | YES | | NULL | |
| country_code | char(2) | YES | | NULL | |
| country_name | varchar(100) | YES | | NULL | |
| latitude | decimal(9,6) | YES | | NULL | |
| longitude | decimal(9,6) | YES | | NULL | |
| wikiDataId | varchar(20) | YES | | NULL | |

- **Table Brand**

Query 1

```
1  CREATE TABLE Brand (
2  Maker VARCHAR(30) PRIMARY KEY,
3  Founder VARCHAR(30),
4  CompanyType VARCHAR(30),
5  Headquarters VARCHAR(30),
6  FoundedYear INT
7  );
```

Result Grid | Filter Rows: | Export: | Wrap Cell

| Field | Type | Null | Key | Default | Extra |
|---|---|---|---|---|---|
| Maker | varchar(30) | NO | PRI | NULL | |
| Founder | varchar(30) | YES | | NULL | |
| CompanyType | varchar(30) | YES | | NULL | |
| Headquarters | varchar(30) | YES | | NULL | |
| Producta | varchar(50) | YES | | NULL | |
| FoundedYear | int | YES | | NULL | |
| Industry | varchar(50) | YES | | NULL | |
| ProductionOutput | varchar(50) | YES | | NULL | |
| Revenue | int | YES | | NULL | |
| OperatingIncome | int | YES | | NULL | |
| NetIncome | int | YES | | NULL | |
| TotalAsset | int | YES | | NULL | |
| TotalEquity | int | YES | | NULL | |
| NumberOfEmplo... | int | YES | | NULL | |

- **Table Car**

```sql
CREATE TABLE Car (
CarId INT AUTO_INCREMENT PRIMARY KEY,
Make VARCHAR(300),
Model VARCHAR(300),
CarTitle VARCHAR(200),
CarSubTitle VARCHAR(500),
CarPrice INT,
ManufactureYear INT,
BodyType VARCHAR(300),
Mileage INT,
EngineVolume DECIMAL(10, 2),
EngineSize DECIMAL(10, 2),
TransmissionType VARCHAR(300),
FuelType VARCHAR(200),
TotalPreviousOwners INT,
FOREIGN KEY (Make) REFERENCES Brand(Maker)
);
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

| Field | Type | Null | Key | Default | Extra |
|---|---|---|---|---|---|
| CarId | int | NO | PRI | NULL | auto_increment |
| Make | varchar(300) | YES | MUL | NULL | |
| Model | varchar(300) | YES | | NULL | |
| CarTitle | varchar(200) | YES | UNI | NULL | |
| CarSubTitle | varchar(500) | YES | | NULL | |
| CarPrice | int | YES | | NULL | |
| ManufactureYear | int | YES | | NULL | |
| BodyType | varchar(300) | YES | | NULL | |
| Mileage | int | YES | | NULL | |
| EngineVolume | decimal(10,2) | YES | | NULL | |
| EngineSize | decimal(10,2) | YES | | NULL | |
| TransmissionType | varchar(300) | YES | | NULL | |
| FuelType | varchar(200) | YES | | NULL | |
| TotalPreviousO... | int | YES | | NULL | |

- **Table Advertisement**

```
CREATE TABLE Advertisement (
AdvertisementId INT AUTO_INCREMENT PRIMARY KEY,
CarId INT,
UserId INT,
CarAttentionGrabber VARCHAR(300),
FinanceAvailable BOOLEAN,
Discounted BOOLEAN,
FOREIGN KEY (CarId) REFERENCES Car(CarId),
FOREIGN KEY (UserId) REFERENCES User(UserId)
);
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

| Field | Type | Null | Key | Default | Extra |
|---|---|---|---|---|---|
| AdvertisementId | int | NO | PRI | NULL | auto_increment |
| CarId | int | YES | MUL | NULL | |
| UserId | int | YES | MUL | NULL | |
| CarAttentionGrabber | varchar(300) | YES | | NULL | |
| FinanceAvailable | tinyint(1) | YES | | NULL | |
| Discounted | tinyint(1) | YES | | NULL | |

- **Table Rating**

Open a script file in this editor

```
CREATE TABLE Rating(
RatingId INT AUTO_INCREMENT PRIMARY KEY,
CarTitle VARCHAR(100),
Price VARCHAR(50),
OverallRating DECIMAL(2,1),
ExteriorRating DECIMAL(2,1),
InteriorRating DECIMAL(2,1),
RideQuality DECIMAL(2,1),
Foreign Key (CarTitle) references Car(CarTitle)
);
```

Result Grid | Filter Rows: | Export: | Wrap Cell Conten

| Field | Type | Null | Key | Default | Extra |
|---|---|---|---|---|---|
| RatingId | int | NO | PRI | NULL | auto_increment |
| CarTitle | varchar(100) | YES | MUL | NULL | |
| Price | varchar(50) | YES | | NULL | |
| OverallRating | decimal(2,1) | YES | | NULL | |
| ExteriorRating | decimal(2,1) | YES | | NULL | |
| InteriorRating | decimal(2,1) | YES | | NULL | |
| RideQuality | decimal(2,1) | YES | | NULL | |

YES

## Data volume

```
1 •  select count(*) from Location;
```

Result Grid | Filter Rows: | Export:

| count(*) |
| --- |
| 128905 |

```
1 •  select count(*) from User;
```

Result Grid | Filter Rows:

| count(*) |
| --- |
| 1000 |

```
1 •  select count(*) from Brand;
```

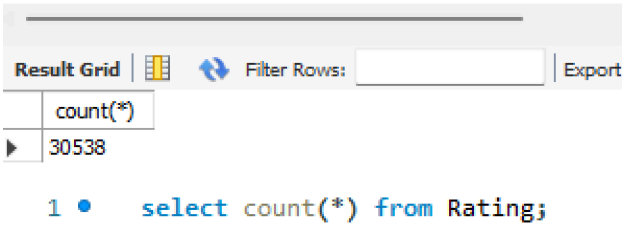
Result Grid | Filter Rows:

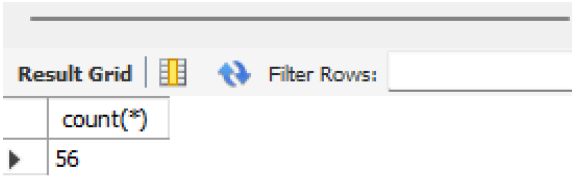
| count(*) |
| --- |
| 132 |

```
1 •  select count(*) from Car;
```

Result Grid | Filter Rows:

| count(*) |
| --- |
| 3669 |


## II. Advanced Queries

### Query 1: Average Reviews of Different Brands

- **Techniques used**: `JOIN`, `GROUP BY`
- **Purpose**: Calculate the average overall rating for each car brand based on user reviews.

```
mysql> SELECT c.Make, AVG(r.OverallRating) AS AvgOverallRating
    -> FROM Car c
    -> JOIN Rating r ON c.CarTitle = r.CarTitle
    -> GROUP BY c.Make
    -> ORDER BY AvgOverallRating DESC
    -> LIMIT 15;
+--------------+------------------+
| Make         | AvgOverallRating |
+--------------+------------------+
| Volkswagen   |          5.00000 |
| Jaguar       |          5.00000 |
| Audi         |          4.45000 |
| Lamborghini  |          4.10000 |
| MINI         |          4.00000 |
| Isuzu        |          3.80000 |
| MG           |          3.70000 |
| Hyundai      |          2.61250 |
| Jeep         |          2.46667 |
| Porsche      |          2.00000 |
| Toyota       |          1.66667 |
| SKODA        |          1.33333 |
| Land Rover   |          0.71429 |
| BMW          |          0.62500 |
| Aston Martin |          0.00000 |
+--------------+------------------+
15 rows in set (0.03 sec)
```

### Query 2: Discounted cars priced above average

- **Techniques used**: `JOIN`, `Subquery`
- **Purpose**: Identify cars with discounts that are still priced above the average price of all cars.

```
mysql> SELECT a.AdvertisementId, c.CarTitle, c.CarPrice
    -> FROM Advertisement a
    -> JOIN Car c ON a.CarId = c.CarId
    -> WHERE a.Discounted = 1
    ->   AND c.CarPrice > (
    ->     SELECT AVG(CarPrice) FROM Car
    ->   )
    -> ORDER BY c.CarPrice DESC
    -> LIMIT 15;
+-----------------+------------------------------+----------+
| AdvertisementId | CarTitle                     | CarPrice |
+-----------------+------------------------------+----------+
|            4224 | BMW M8 Competition Convertible |   148555 |
|            4226 | BMW M8 Competition Convertible |   148555 |
|            4242 | BMW M8 Gran Coupe              |   147070 |
|            4233 | BMW M8 Competition Gran Coupe  |   141465 |
|            4234 | BMW M8 Competition Gran Coupe  |   141465 |
|            4235 | BMW M8 Competition Gran Coupe  |   141465 |
|            4237 | BMW M8 Competition Gran Coupe  |   141465 |
|            4238 | BMW M8 Competition Gran Coupe  |   141465 |
|            4822 | BMW X6 M Competition           |   136855 |
|            4825 | BMW X6 M Competition           |   136855 |
|            4228 | BMW M8 Competition Coupe       |   133297 |
|            4229 | BMW M8 Competition Coupe       |   133297 |
|            4230 | BMW M8 Competition Coupe       |   133297 |
|            4231 | BMW M8 Competition Coupe       |   133297 |
|            4232 | BMW M8 Competition Coupe       |   133297 |
+-----------------+------------------------------+----------+
15 rows in set (0.03 sec)
```

### Query 3: Find the average price at which each brand is listed in ads

- **Techniques used**: `JOIN`, `GROUP BY`
- **Purpose**: Compute the average listing price for each brand based on all ads.

```
mysql> SELECT c.Make, AVG(c.CarPrice) AS AvgListedPrice
    -> FROM Car c
    -> JOIN Advertisement a ON c.CarId = a.CarId
    -> GROUP BY c.Make
    -> ORDER BY AvgListedPrice DESC
    -> LIMIT 15;
+--------------+----------------+
| Make         | AvgListedPrice |
+--------------+----------------+
| Bugatti      |    895000.0000 |
| Lamborghini  |    251352.0462 |
| Ferrari      |    236756.1200 |
| McLaren      |    217940.0000 |
| Aston Martin |    173017.1207 |
| Rolls-Royce  |    139725.7895 |
| De Tomaso    |    124995.0000 |
| Porsche      |    121404.6435 |
| Radical      |    120000.0000 |
| Bentley      |    111094.9545 |
| Bac          |     99990.0000 |
| Bowler       |     98491.0000 |
| Crendon      |     94748.0000 |
| Factory Five |     89950.0000 |
| Shelby       |     79995.0000 |
+--------------+----------------+
15 rows in set (0.07 sec)
```

### Query4: High-price SUVs above brand average

- **Techniques used**: `Subquery`, filtering conditions
- **Purpose**: List SUV models with engine volume > 1.4L that are priced above their brand's average.

```
mysql> SELECT c.CarTitle, c.Make, c.CarPrice
    -> FROM Car c
    -> WHERE c.CarPrice > (
    ->     SELECT AVG(c2.CarPrice)
    ->     FROM Car c2
    ->     WHERE c2.Make = c.Make
    -> ) AND EngineVolume > 1.4 AND BodyType = 'suv'
    -> ORDER BY c.CarPrice DESC
    -> LIMIT 15;
+---------------------------------------+--------------+----------+
| CarTitle                              | Make         | CarPrice |
+---------------------------------------+--------------+----------+
| Rolls-Royce Cullinan Black Badge      | Rolls-Royce  |   379950 |
| Rolls-Royce Cullinan                  | Rolls-Royce  |   299000 |
| Mercedes-Benz Maybach Gls Class       | Mercedes-Benz |   249987 |
| Mercedes-Benz G Series                | Mercedes-Benz |   229996 |
| Bentley Bentayga S                    | Bentley      |   204890 |
| Bentley Bentayga                      | Bentley      |   194995 |
| Ford F150                             | Ford         |   187140 |
| Mercedes-Benz G63 AMG                 | Mercedes-Benz |   187000 |
| Bentley Bentayga Hybrid               | Bentley      |   182950 |
| Bentley Bentayga Mulliner             | Bentley      |   175890 |
| Land Rover New Range Rover            | Land Rover   |   159891 |
| BMW ALPINA XB7                        | BMW          |   157768 |
| BMW X7                                | BMW          |   150665 |
| Land Rover Range Rover Efi            | Land Rover   |   150000 |
| Mercedes-Benz G Class Amg Station Wagon | Mercedes-Benz |   149506 |
+---------------------------------------+--------------+----------+
15 rows in set (0.16 sec)
```

### Query 5: Overall Ratings of Car Models Priced Above the Average

- **Techniques used**: `JOIN`, `Subquery`, `ORDER BY`

- **Purpose**: Evaluate whether expensive cars have better overall ratings.

```
mysql> SELECT c.CarTitle, c.CarPrice, r.OverallRating
    -> FROM Car c
    -> JOIN Rating r ON c.CarTitle = r.CarTitle
    -> WHERE c.CarPrice > (
    ->     SELECT AVG(CarPrice) FROM Car
    -> )
    -> ORDER BY OverallRating DESC, EngineSize
    -> LIMIT 15;
+------------------------------+----------+---------------+
| CarTitle                     | CarPrice | OverallRating |
+------------------------------+----------+---------------+
| Land Rover Range Rover Evoque |    52500 |           5.0 |
| Lamborghini Huracan Evo      |   229000 |           4.1 |
| Porsche 911                  |    72000 |           4.0 |
| Porsche Cayenne              |    76648 |           4.0 |
| Jeep Compass                 |    40099 |           3.7 |
| Jeep Compass                 |    40099 |           3.7 |
| Hyundai IONIQ 5              |    39985 |           0.0 |
| Land Rover Range Rover Velar |    39600 |           0.0 |
| Land Rover Range Rover Velar |    39600 |           0.0 |
| Land Rover Defender          |    69995 |           0.0 |
| Land Rover Defender          |    69995 |           0.0 |
| Lexus ES                     |    39995 |           0.0 |
| Volvo XC60                   |    39705 |           0.0 |
| Porsche Macan                |    63090 |           0.0 |
| Jeep Wrangler                |    59752 |           0.0 |
+------------------------------+----------+---------------+
15 rows in set (0.01 sec)
```

### Query 6: Low-Mileage Cars with Fewer Owners, Ordered by Earliest Manufacture Year

- **Techniques used**: `Subquery`, `GROUP BY`, `ORDER BY`

- **Purpose**: Find used cars with mileage below average and fewer than 3 previous owners, ordered by the earliest manufacture year.

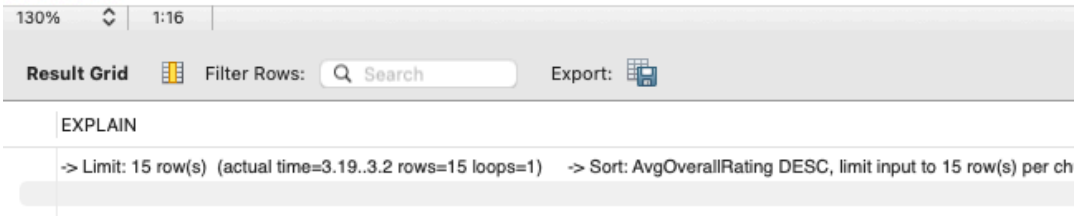
```
mysql> SELECT CarTitle
    -> FROM Car
    -> WHERE Mileage < (
    ->     SELECT AVG(Mileage)
    ->     FROM Car
    -> )
    -> AND TotalPreviousOwners < 3
    -> GROUP BY CarTitle
    -> ORDER BY MIN(ManufactureYear)
    -> LIMIT 15;
+--------------------------+
| CarTitle                 |
+--------------------------+
| Audi A5 Coupe            |
| Audi RS Q3               |
| BMW 7 Series Saloon (LWB)|
| Audi S3 Saloon           |
| Audi Tts Coupe           |
| BMW 2 Series Gran Coupe  |
| BMW 7 Series Saloon      |
| Abarth 695C              |
| Abarth Abarth 500        |
| Alpine Alpine            |
| BMW 4 Series Coupe       |
| Audi TT Roadster         |
| Audi S8                  |
| BMW 8 Series Coupe       |
| BMW ALPINA B8            |
+--------------------------+
15 rows in set (0.13 sec)
```

# III. Indexing Analysis

## A. Query 1

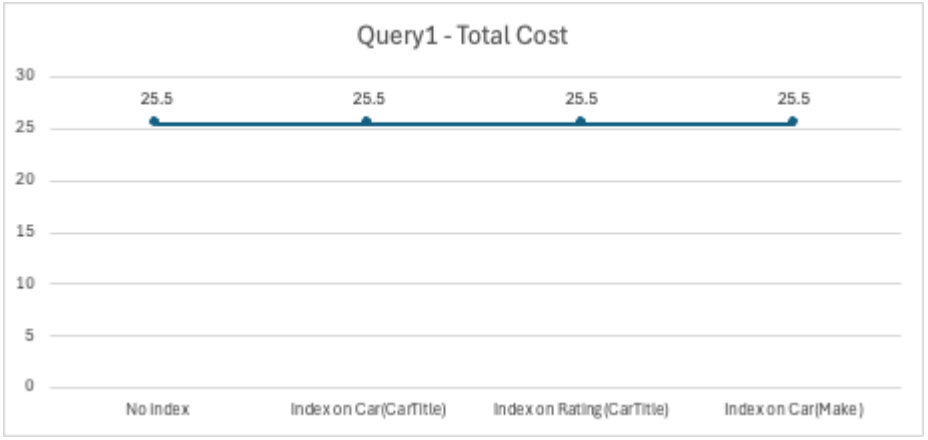
### Screenshot for Analyzing the Query

```
18 •  EXPLAIN ANALYZE
19    SELECT c.Make, AVG(r.OverallRating) AS AvgOverallRating
20    FROM Car c
21    JOIN Rating r ON c.CarTitle = r.CarTitle
22    GROUP BY c.Make
23    ORDER BY AvgOverallRating DESC
24    LIMIT 15;
```




### Indexing Strategies Explored

We experimented with indexing different attributes that appeared in the `JOIN`, `WHERE`, `GROUP BY`, or `HAVING` clauses. Below are the results for each configuration:




### Performance Observations

Surprisingly, all indexing configurations yielded the **same query cost of 25.5**, indicating **no observable performance gain or degradation** regardless of which attribute was indexed.

**Why indexing didn't help:**

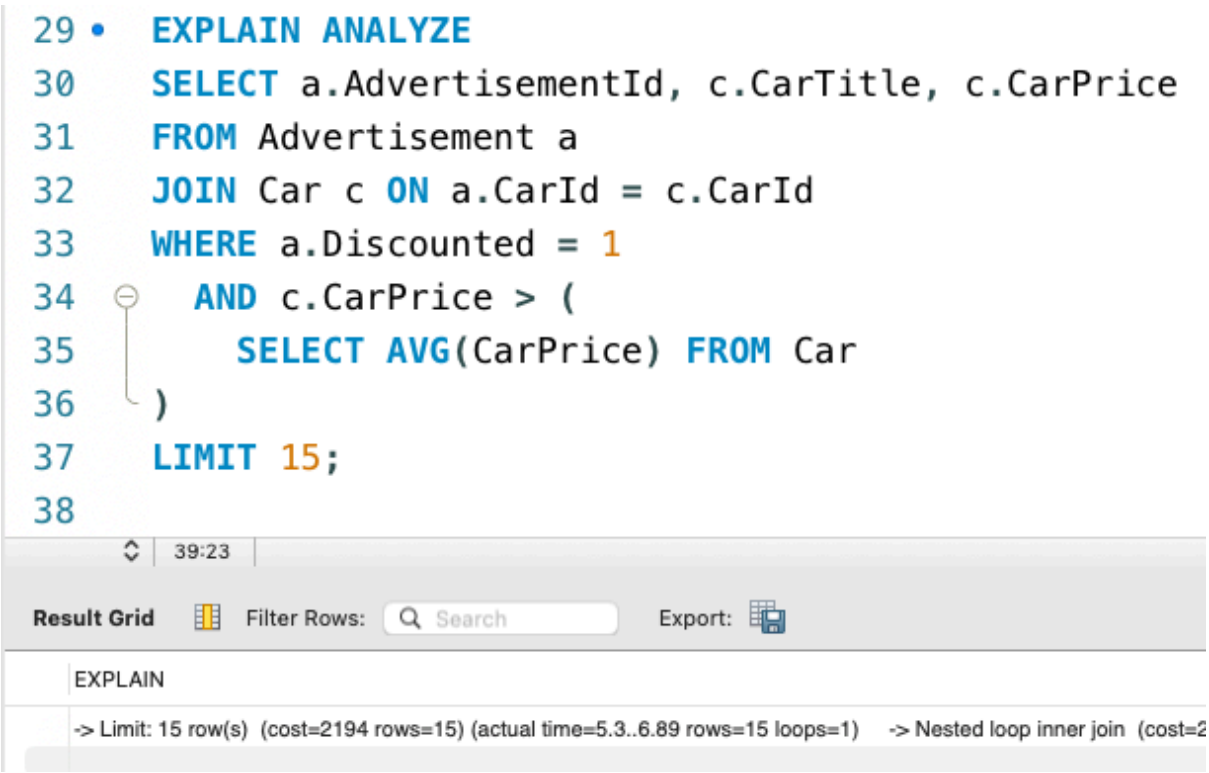
**Small Dataset Size:** The Rating table has only 56 data, which is too small to perform cost variation with different indexing. As a result, the dataset is small enough that the query planner determines a full table scan is more efficient than using indexes.

### Final Index Design Decision

Given that no index yielded performance improvement, **no additional indexes are selected** for this specific query. We opt to avoid unnecessary indexing to save on write/update overhead and storage cost.

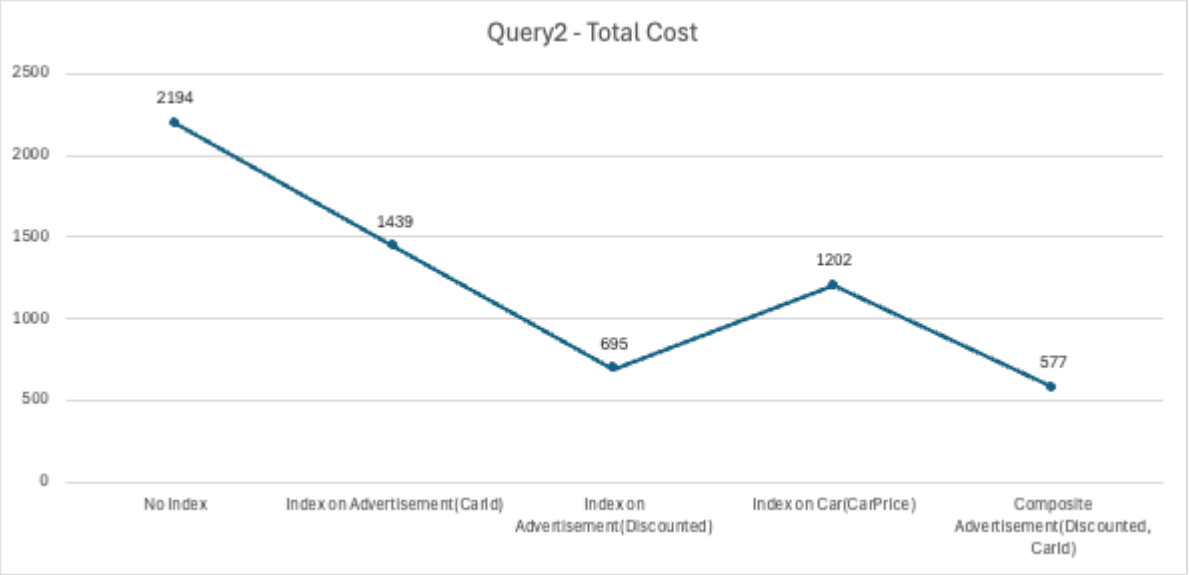
## B. Query 2

### Screenshot for Analyzing the Query




### Indexing Strategies Explored

We experimented with indexing different attributes that appeared in the `JOIN`, `WHERE`, `GROUP BY`, or `HAVING` clauses. Below are the results for each configuration:




**Performance Observations**

The results clearly show that **adding indexes significantly reduces the query cost**.

- Indexing **`Advertisement(CarId)`** alone reduces the cost to **1439**, as it optimizes the `JOIN`.
- Indexing **`Advertisement(Discounted)`** reduces it further to **695**, helping the `WHERE` clause filter on the discounted flag.
- Indexing **`Car(CarPrice)`** improves filtering for price-based logic in both the `WHERE` clause and the subquery.
- The best result was achieved using a **composite index on `Advertisement(Discounted, CarId)`**, dropping the cost to **577** — likely due to MySQL taking advantage of multi-column filtering and join optimization.

**Final Index Design Decision**

We selected the **composite index on `Advertisement(Discounted, CarId)`** as the optimal index for this query.
It provides the lowest cost by efficiently supporting:

- Filtering by `Discounted`
- The `JOIN` with `Car` on `CarId`

### C. Query 3

#### Screenshot for Analyzing the Query

```
37    EXPLAIN ANALYZE
38    SELECT c.Make, AVG(c.CarPrice) AS AvgPrice
39    FROM Advertisement a
40    JOIN Car c ON a.CarId = c.CarId
41    GROUP BY c.Make
42    HAVING AvgPrice > (
43        SELECT AVG(CarPrice) FROM Car
44    )
45    ORDER BY AvgPrice DESC
46    LIMIT 15;
47
```

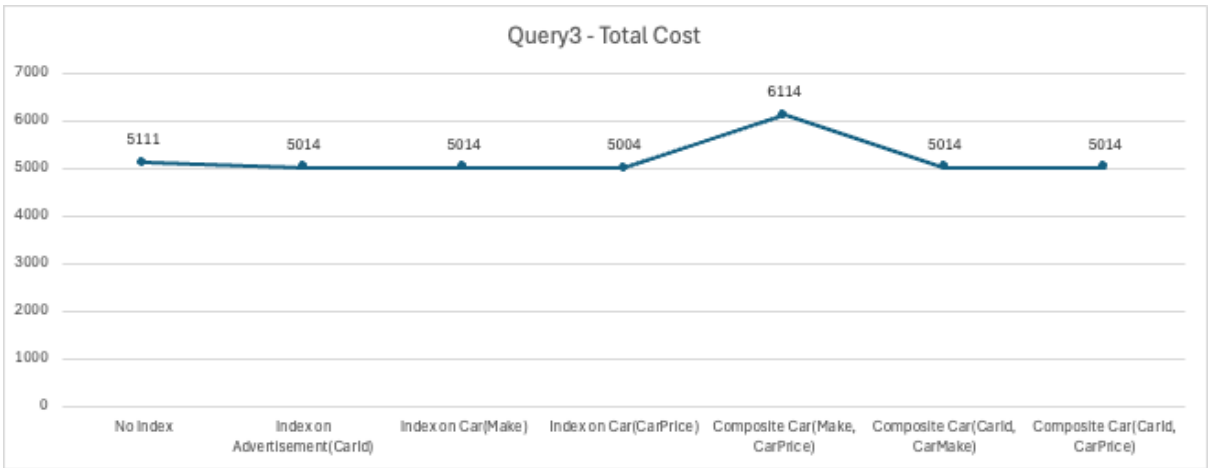
130% 1:31

Result Grid Filter Rows: Search Export:

| EXPLAIN |
| --- |
| -> Limit: 15 row(s) (actual time=25.5..25.5 rows=15 loops=1) -> Sort: AvgPrice DESC (actua |

#### Indexing Strategies Explored

We experimented with indexing different attributes that appeared in the `JOIN`, `WHERE`, `GROUP BY`, or `HAVING` clauses. Below are the results for each configuration:



### Performance Observations

Overall, **none of the indexing strategies significantly improved the query performance**. The slight drop from 5111 to 5004 with the `Car(CarPrice)` index was the best observed improvement, but it's too minor to justify the index alone.

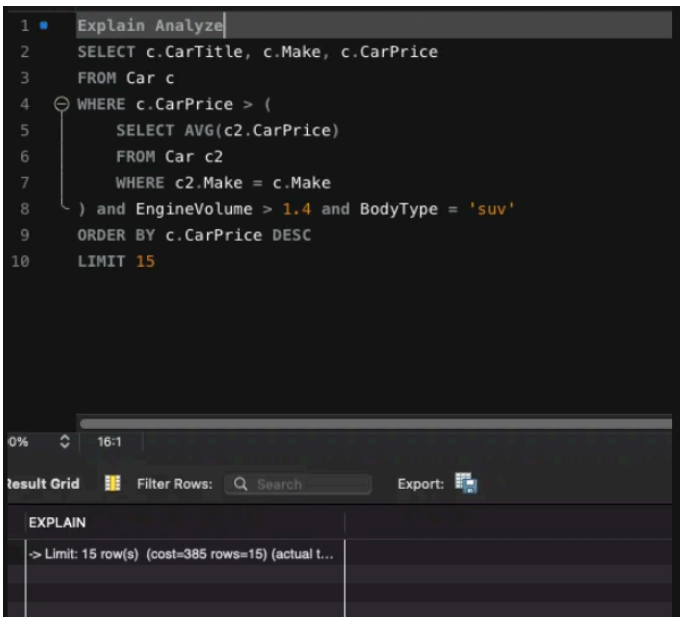
- **Indexing `Car(CarPrice)`** helped slightly, likely due to its use in the subquery and `AVG()` computation.
- Composite indexes such as `Car(Make, CarPrice)` actually **worsened** performance to **6114**, suggesting the planner did not find them helpful, or they added overhead.
- Other composite indexes such as `Car(CarId, Make)` or `Car(CarId, CarPrice)` made **no difference** compared to individual indexes.

### Final Index Design Decision

Since **none of the indexes produced meaningful performance improvements**, we recommend **not adding any new index** for this query.
We will retain the default (or possibly `Car(CarPrice)`) only if needed elsewhere in the workload.

### D. Query 4

### ScreenShot for Analyzing the Query



## Performance Observations

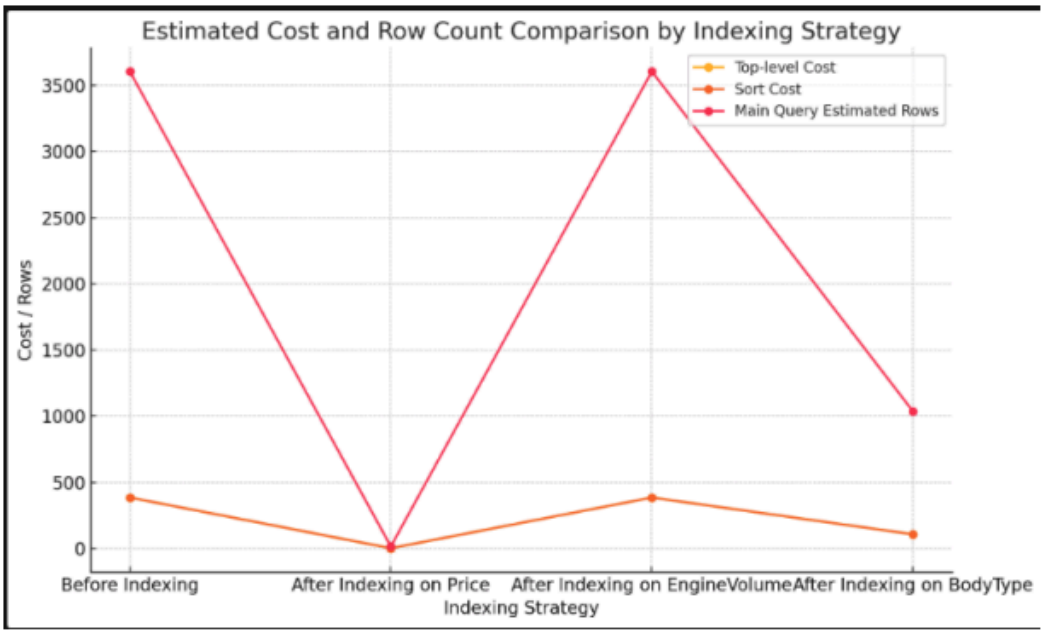
### 1. Before Indexing

- Top-level Cost: `385`
- Scan Type: Full Table Scan
- Filter Estimated Rows: `3606`

### 2. Indexing Experiments

| Index | Top-level Cost | Notes |
|---|---|---|
| On `CarPrice` DESC (`idx_price`) | 1.55 | Dramatic cost reduction. Used index scan. |
| On `EngineVolume` | 385 | No improvement, full table scan remained. |
| On `BodyType` (`idx_BodyType`) | 107 | Partial improvement. Reduced scanned rows to 1038. |

### 3. Analysis




- Indexing CarPrice provided the most dramatic improvement.

- **BodyType indexing helped, but less significantly.**
- **EngineVolume index had no effect, likely due to low selectivity.**

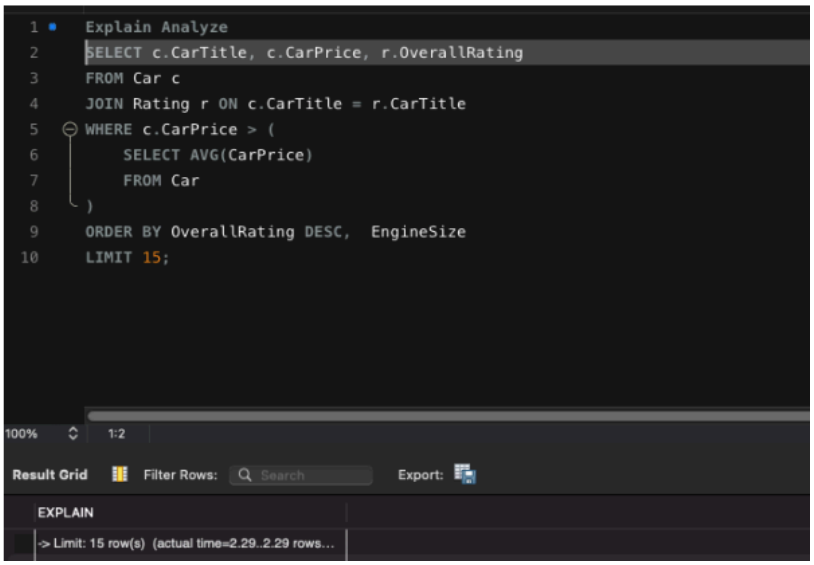
### Final Index Design Decision

**Selected: Car(CarPrice DESC)**

- **Reason: Reduced cost from 385 to 1.55 by leveraging index for filtering and sorting.**

## E. Query 5

### ScreenShot for Analyzing the Query




### Performance Observations

#### 1. Before Indexing

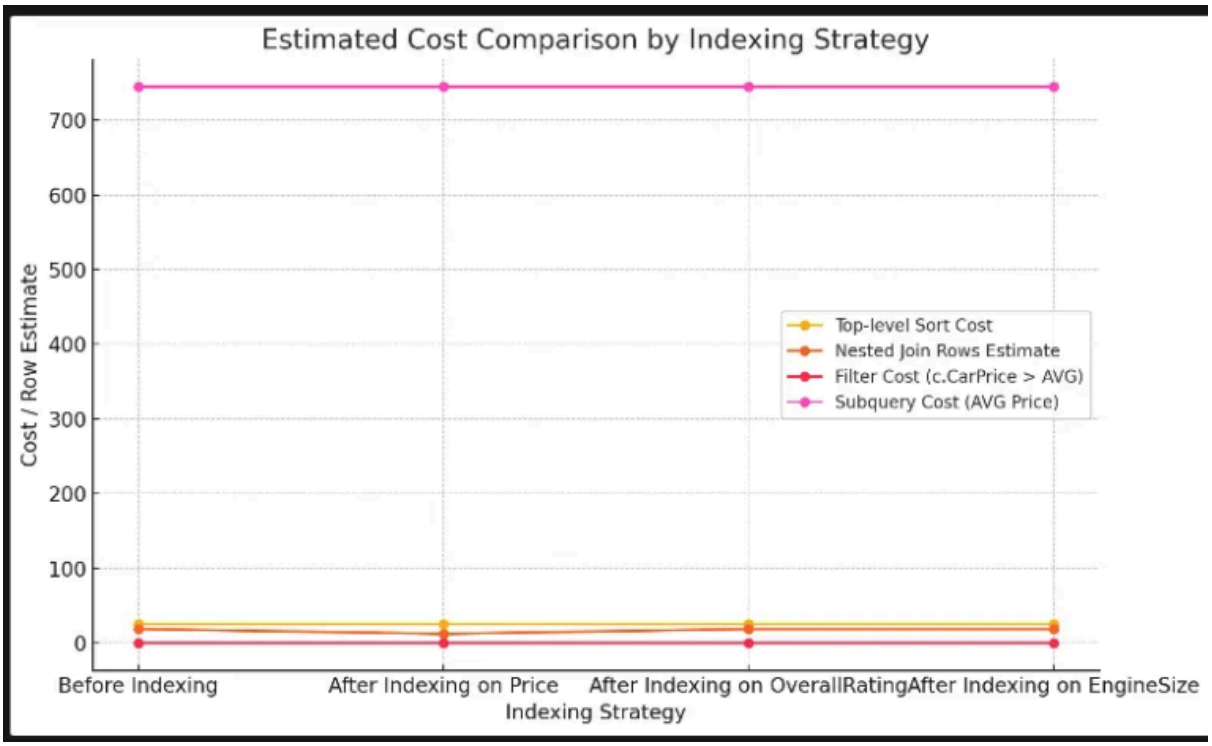
- **Top-level Cost: 25.5**
- **Subquery Cost: 745**
- **Table Scan on Car: 385**

#### 2. Indexing Experiments

| Index | Top-level Cost | Notes |
|---|---|---|
| On `CarPrice` | 25.5 | Slight improvement in filter rows |

| | | |
|---|---|---|
| | | (from 0.333 → 0.218) |
| On `Rating.CarTitle` | 25.5 | No improvement, Rating table is small. |
| On `EngineSize` | 25.5 | No impact. Not used for filtering. |

### 3. Analysis



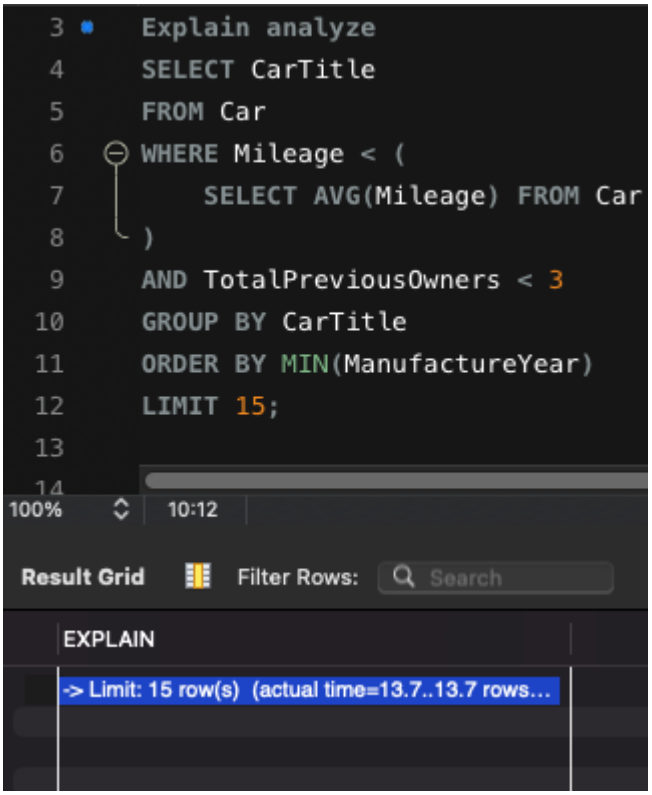

- The index on `CarPrice` improved filter selectivity slightly.
- Other indexes had no visible impact due to small table size or no filtering role.

## Final Index Design Decision

- Selected: `Car(CarPrice)`
- Reason: Slight improvement in join cost and filter accuracy. Others ineffective

### F. Query 6

### ScreenShot for Analyzing the Query




### Observations

#### 1. Before Indexing

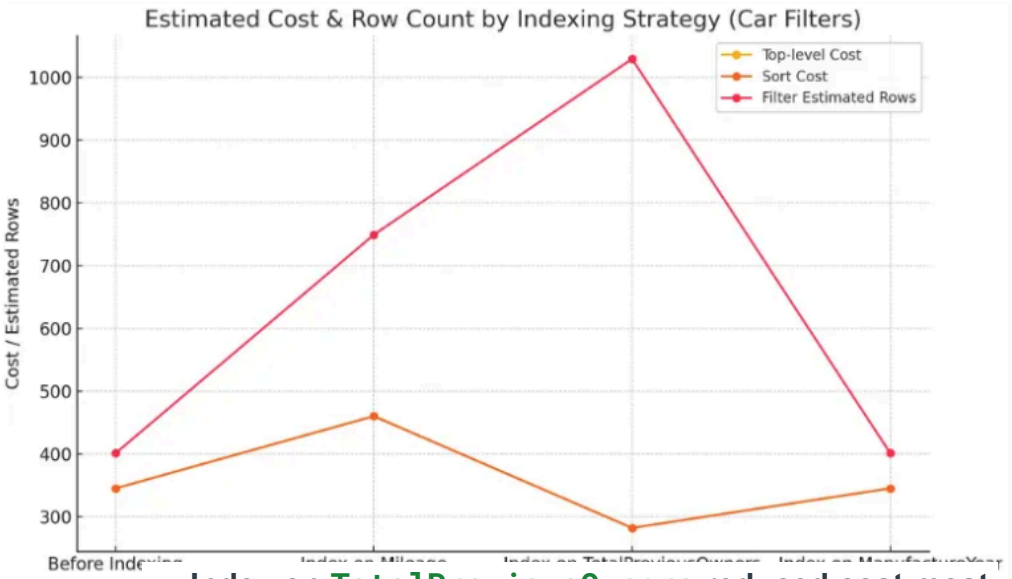
- Top-level Cost: `345`
- Estimated Rows: `401`

#### 2. Indexing Experiments

| Index | Top-level Cost | Filter Rows | Notes |
|---|---|---|---|
| On `Mileage` | 460 | 749 | Cost increased |
| On `TotalPreviousOwners` | 282 | 1049 | Best cost reduction, despite high row count. |
| On `Manufacture` | 345 | 201 | No change. Used in ORDER BY but no |

| | | | |
|---|---|---|---|
| `Year` | | | filter. |

## Analysis




- Index on `TotalPreviousOwners` reduced cost most.
- Index on `Mileage` worsened performance (overhead).
- ManufactureYear index unused for optimization.

## Final Index Design Dicision

- Selected: `Car(TotalPreviousOwners)`
- Reason: Lowered top-level cost from 345 → 282. Best tradeoff overall.