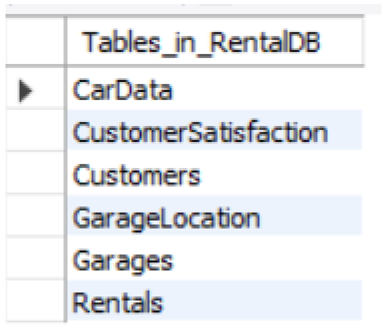
Create a markdown or pdf file called “Database Design” in the doc folder of your GitHub repo.

Part 1:

1. Implement at least five main tables. These tables represent the relational schema you provided in Stage 2.



Tables_in_RentalDB
CarData
CustomerSatisfaction
Customers
GarageLocation
Garages
Rentals

2. In the Database Design markdown or pdf, provide the Data Definition Language (DDL) commands you used to create each of these tables in the database.

   CREATE TABLE Customers (CustomerId INT, Name VARCHAR(100), Email VARCHAR(100), PhoneNumber VARCHAR(15), Age INT);

   CREATE TABLE Rentals (RentalId INT, CarId INT, CustomerId INT, StartTime DATETIME, EndTime DATETIME, GarageId INT, TotalCost DECIMAL(10,2), Miles INT, Purpose VARCHAR(255));

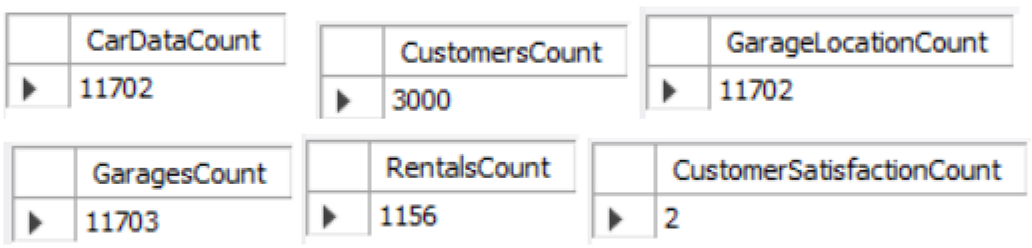
   CREATE TABLE CustomerSatisfaction (RentalId INT, CustomerId INT, Rating DECIMAL(9,6), Comments TEXT);

   CREATE TABLE Garages (GarageId INT, Profit DECIMAL(10,2), Capacity INT, Latitude DECIMAL(9,6), Longitude DECIMAL(9,6));

   CREATE TABLE CarData (CarId INT, GarageId INT, HourlyRate DECIMAL(10,2), Make VARCHAR(100), Model VARCHAR(100), Year INT, Availability BOOLEAN);

   CREATE TABLE GarageLocation (Latitude DECIMAL(9,6), Longitude DECIMAL(9,6), City VARCHAR(100));

3. Insert data into these tables. You should insert at least 1000 rows in three different tables each. Try to use real data, but if you cannot find a good dataset for a particular table, you may use auto-generated data.



CarDataCount
11702

CustomersCount
3000

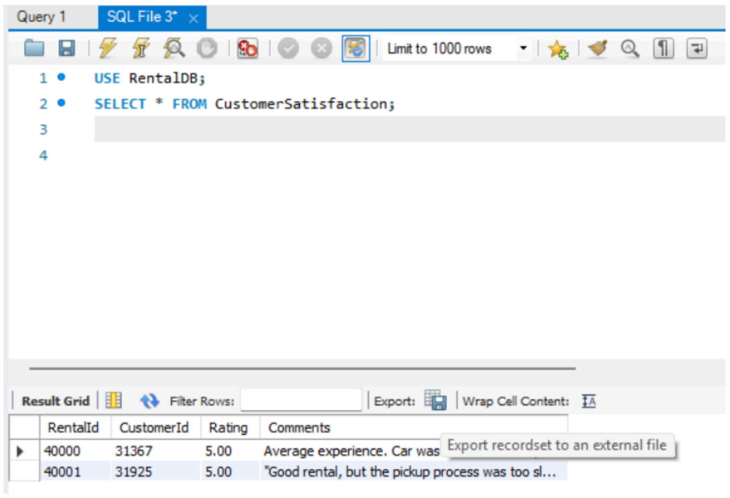
GarageLocationCount
11702

GaragesCount
11703

RentalsCount
1156

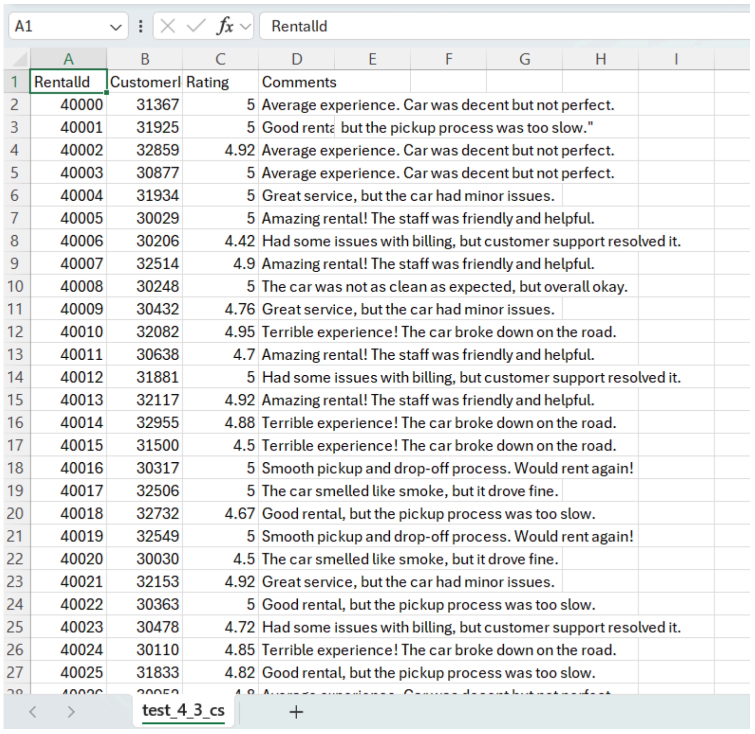
CustomerSatisfactionCount
2

Our CustomerSatisfaction table includes at least a 1000 rows and was imported correctly into our Google Cloud Storage bucket. Unfortunately, when performing the count query, the output reported only 2 rows. However, when we export the output of the SELECT * FROM CustomerSatisfaction query to an external file, all 1000 rows of the data are present. We are unsure what the issue is and have asked multiple TAs at office hours for help.



```
USE RentalDB;
SELECT * FROM CustomerSatisfaction;
```

RentalId	CustomerId	Rating	Comments
40000	31367	5.00	Average experience. Car was
40001	31925	5.00	"Good rental, but the pickup process was too sl...



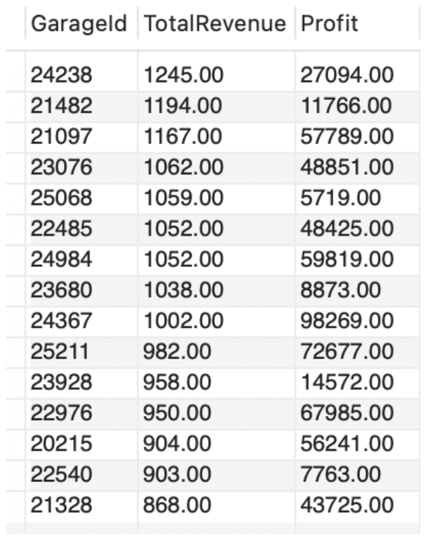
	A	B	C	D
1	RentalId	CustomerI	Rating	Comments
2	40000	31367	5	Average experience. Car was decent but not perfect.
3	40001	31925	5	Good renta but the pickup process was too slow."
4	40002	32859	4.92	Average experience. Car was decent but not perfect.
5	40003	30877	5	Average experience. Car was decent but not perfect.
6	40004	31934	5	Great service, but the car had minor issues.
7	40005	30029	5	Amazing rental! The staff was friendly and helpful.
8	40006	30206	4.42	Had some issues with billing, but customer support resolved it.
9	40007	32514	4.9	Amazing rental! The staff was friendly and helpful.
10	40008	30248	5	The car was not as clean as expected, but overall okay.
11	40009	30432	4.76	Great service, but the car had minor issues.
12	40010	32082	4.95	Terrible experience! The car broke down on the road.
13	40011	30638	4.7	Amazing rental! The staff was friendly and helpful.
14	40012	31881	5	Had some issues with billing, but customer support resolved it.
15	40013	32117	4.92	Amazing rental! The staff was friendly and helpful.
16	40014	32955	4.88	Terrible experience! The car broke down on the road.
17	40015	31500	4.5	Terrible experience! The car broke down on the road.
18	40016	30317	5	Smooth pickup and drop-off process. Would rent again!
19	40017	32506	5	The car smelled like smoke, but it drove fine.
20	40018	32732	4.67	Good rental, but the pickup process was too slow.
21	40019	32549	5	Smooth pickup and drop-off process. Would rent again!
22	40020	30030	4.5	The car smelled like smoke, but it drove fine.
23	40021	32153	4.92	Great service, but the car had minor issues.
24	40022	30363	5	Good rental, but the pickup process was too slow.
25	40023	30478	4.72	Had some issues with billing, but customer support resolved it.
26	40024	30110	4.85	Terrible experience! The car broke down on the road.
27	40025	31833	4.82	Good rental, but the pickup process was too slow.

Advanced Queries

Query 1: This query displays the GarageId, total revenue, and profit of each garage from the Rentals and Garages table. It only outputs Garages that have a total cost of greater than $800.

Attempt 2:

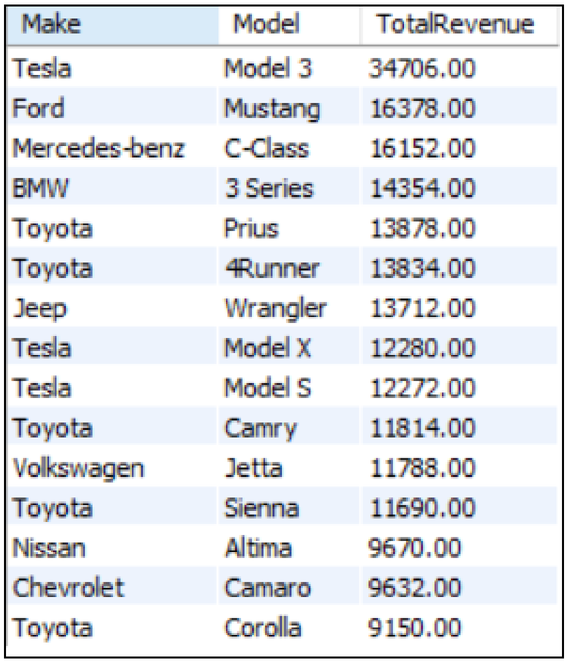
```
SELECT g.GarageId, SUM(r.TotalCost) AS TotalRevenue, g.Profit
FROM Rentals r
JOIN Garages g ON r.GarageId = g.GarageId
GROUP BY g.GarageId, g.Profit
HAVING SUM(r.TotalCost) > 800
ORDER BY TotalRevenue DESC
LIMIT 15;
```



GarageId	TotalRevenue	Profit
24238	1245.00	27094.00
21482	1194.00	11766.00
21097	1167.00	57789.00
23076	1062.00	48851.00
25068	1059.00	5719.00
22485	1052.00	48425.00
24984	1052.00	59819.00
23680	1038.00	8873.00
24367	1002.00	98269.00
25211	982.00	72677.00
23928	958.00	14572.00
22976	950.00	67985.00
20215	904.00	56241.00
22540	903.00	7763.00
21328	868.00	43725.00

Query 2: This query displays the total revenue for each car with specific makes and models in descending order.

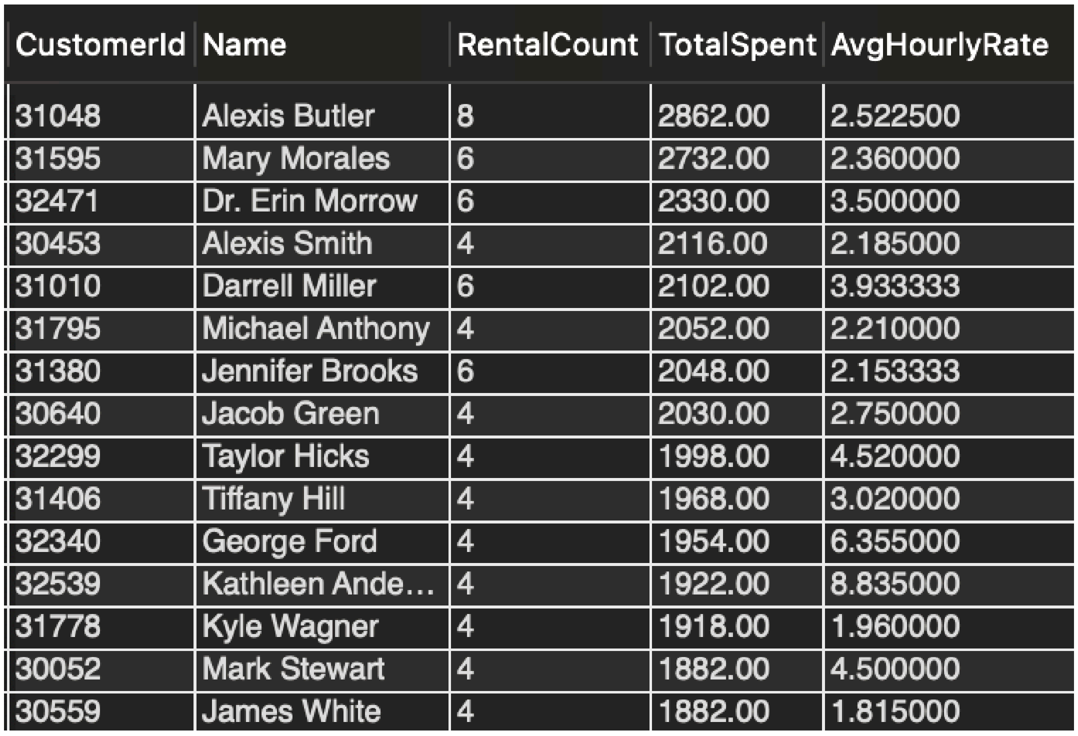
```
SELECT cd.Make, cd.Model, SUM(r.TotalCost) AS TotalRevenue
FROM CarData cd
JOIN Rentals r ON cd.CarId = r.CarId
GROUP BY cd.Make, cd.Model
ORDER BY TotalRevenue DESC
LIMIT 15;
```



Make	Model	TotalRevenue
Tesla	Model 3	34706.00
Ford	Mustang	16378.00
Mercedes-benz	C-Class	16152.00
BMW	3 Series	14354.00
Toyota	Prius	13878.00
Toyota	4Runner	13834.00
Jeep	Wrangler	13712.00
Tesla	Model X	12280.00
Tesla	Model S	12272.00
Toyota	Camry	11814.00
Volkswagen	Jetta	11788.00
Toyota	Sienna	11690.00
Nissan	Altima	9670.00
Chevrolet	Camaro	9632.00
Toyota	Corolla	9150.00

Query 3: This query displays the total number of rentals and the total spent on rentals by each customer and their average hourly rate of their rentals sorted by descending order.

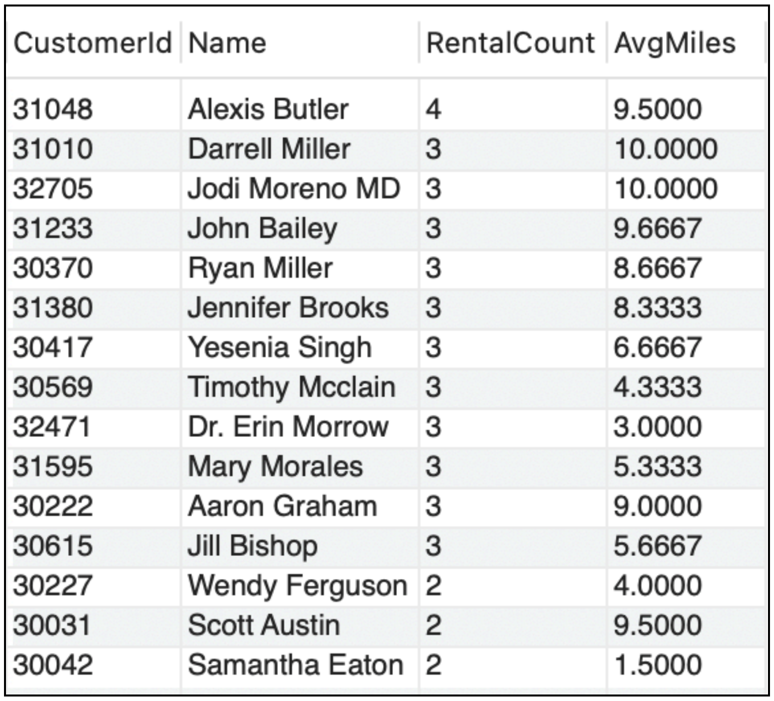
```
SELECT c.CustomerId, c.Name, COUNT(r.RentalId) AS RentalCount, SUM(r.TotalCost) AS
TotalSpent, AVG(cd.HourlyRate) AS AvgHourlyRate
FROM Customers c
JOIN Rentals r ON c.CustomerId = r.CustomerId
JOIN CarData cd ON r.CarId = cd.CarId
GROUP BY c.CustomerId, c.Name
ORDER BY TotalSpent DESC;
LIMIT 15;
```



CustomerId	Name	RentalCount	TotalSpent	AvgHourlyRate
31048	Alexis Butler	8	2862.00	2.522500
31595	Mary Morales	6	2732.00	2.360000
32471	Dr. Erin Morrow	6	2330.00	3.500000
30453	Alexis Smith	4	2116.00	2.185000
31010	Darrell Miller	6	2102.00	3.933333
31795	Michael Anthony	4	2052.00	2.210000
31380	Jennifer Brooks	6	2048.00	2.153333
30640	Jacob Green	4	2030.00	2.750000
32299	Taylor Hicks	4	1998.00	4.520000
31406	Tiffany Hill	4	1968.00	3.020000
32340	George Ford	4	1954.00	6.355000
32539	Kathleen Ande…	4	1922.00	8.835000
31778	Kyle Wagner	4	1918.00	1.960000
30052	Mark Stewart	4	1882.00	4.500000
30559	James White	4	1882.00	1.815000

Query 4: This query returns the average miles of all trips rented by each customer.

```
SELECT c.CustomerId, c.Name,
    COUNT(r.RentalId) AS RentalCount,
    AVG(r.Miles) AS AvgMiles
FROM Customers c
JOIN Rentals r ON c.CustomerId = r.CustomerId
GROUP BY c.CustomerId, c.Name
ORDER BY RentalCount DESC
LIMIT 15;
```



CustomerId	Name	RentalCount	AvgMiles
31048	Alexis Butler	4	9.5000
31010	Darrell Miller	3	10.0000
32705	Jodi Moreno MD	3	10.0000
31233	John Bailey	3	9.6667
30370	Ryan Miller	3	8.6667
31380	Jennifer Brooks	3	8.3333
30417	Yesenia Singh	3	6.6667
30569	Timothy Mcclain	3	4.3333
32471	Dr. Erin Morrow	3	3.0000
31595	Mary Morales	3	5.3333
30222	Aaron Graham	3	9.0000
30615	Jill Bishop	3	5.6667
30227	Wendy Ferguson	2	4.0000
30031	Scott Austin	2	9.5000
30042	Samantha Eaton	2	1.5000

Part 2 Indexing: As a team, for each advanced query:

1. Use the EXPLAIN ANALYZE command to measure your advanced query performance before adding indexes.

   <u>Query 1:</u>

   Original cost = 1.76 e +12

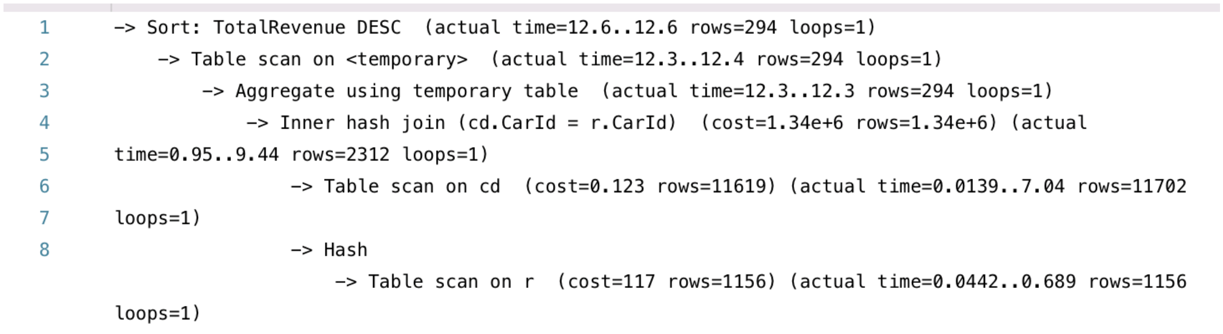
   EXPLAIN ANALYZE Output:

```
-> Sort: TotalRevenue DESC  (actual time=996..996 rows=25 loops=1)
    -> Filter: (`sum(r.TotalCost)` > 800)  (actual time=995..995 rows=25 loops=1)
        -> Table scan on <temporary>  (actual time=995..995 rows=1053 loops=1)
            -> Aggregate using temporary table  (actual time=995..995 rows=1053 loops=1)
                -> Nested loop inner join  (cost=5621 rows=12687) (actual time=0.0344..993 rows=1156
loops=1)
                    -> Filter: (g.GarageId is not null)  (cost=1180 rows=11557) (actual
time=0.0151..732 rows=11703 loops=1)
                        -> Table scan on g  (cost=1180 rows=11557) (actual time=0.0144..731
rows=11703 loops=1)
                    -> Index lookup on r using idx_rentals_garageId (GarageId=g.GarageId)
(cost=0.274 rows=1.1) (actual time=0.022..0.0221 rows=0.0988 loops=11703)
```

   <u>Query 2:</u>

   Original cost = 1.34e+6

   EXPLAIN ANALYZE Output:

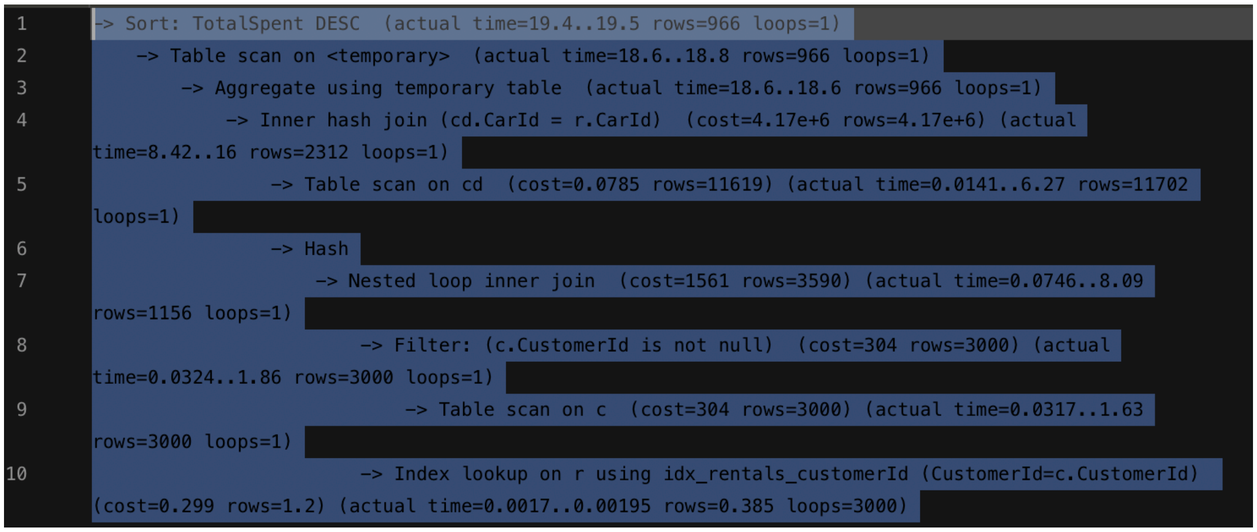


```
-> Sort: TotalRevenue DESC  (actual time=12.6..12.6 rows=294 loops=1)
    -> Table scan on <temporary>  (actual time=12.3..12.4 rows=294 loops=1)
        -> Aggregate using temporary table  (actual time=12.3..12.3 rows=294 loops=1)
            -> Inner hash join (cd.CarId = r.CarId)  (cost=1.34e+6 rows=1.34e+6) (actual
time=0.95..9.44 rows=2312 loops=1)
                -> Table scan on cd  (cost=0.123 rows=11619) (actual time=0.0139..7.04 rows=11702
loops=1)
                -> Hash
                    -> Table scan on r  (cost=117 rows=1156) (actual time=0.0442..0.689 rows=1156
loops=1)
```

<u>Query 3:</u>

Original cost = 4.17e+6

EXPLAIN ANALYZE Output:

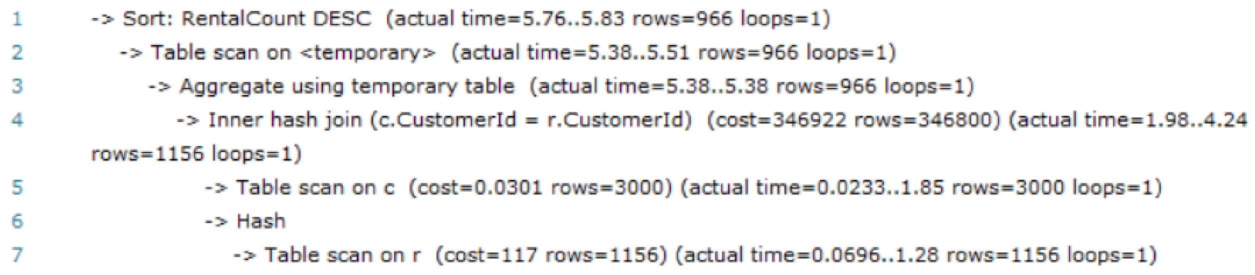


```
-> Sort: TotalSpent DESC  (actual time=19.4..19.5 rows=966 loops=1)
    -> Table scan on <temporary>  (actual time=18.6..18.8 rows=966 loops=1)
        -> Aggregate using temporary table  (actual time=18.6..18.6 rows=966 loops=1)
            -> Inner hash join (cd.CarId = r.CarId)  (cost=4.17e+6 rows=4.17e+6) (actual
time=8.42..16 rows=2312 loops=1)
                -> Table scan on cd  (cost=0.0785 rows=11619) (actual time=0.0141..6.27 rows=11702
loops=1)
                -> Hash
                    -> Nested loop inner join  (cost=1561 rows=3590) (actual time=0.0746..8.09
rows=1156 loops=1)
                        -> Filter: (c.CustomerId is not null)  (cost=304 rows=3000) (actual
time=0.0324..1.86 rows=3000 loops=1)
                            -> Table scan on c  (cost=304 rows=3000) (actual time=0.0317..1.63
rows=3000 loops=1)
                        -> Index lookup on r using idx_rentals_customerId (CustomerId=c.CustomerId)
(cost=0.299 rows=1.2) (actual time=0.0017..0.00195 rows=0.385 loops=3000)
```

<u>Query 4:</u>

Original cost = 346922

EXPLAIN ANALYZE Output:



```
-> Sort: RentalCount DESC  (actual time=5.76..5.83 rows=966 loops=1)
    -> Table scan on <temporary>  (actual time=5.38..5.51 rows=966 loops=1)
        -> Aggregate using temporary table  (actual time=5.38..5.38 rows=966 loops=1)
            -> Inner hash join (c.CustomerId = r.CustomerId)  (cost=346922 rows=346800) (actual time=1.98..4.24
rows=1156 loops=1)
                -> Table scan on c  (cost=0.0301 rows=3000) (actual time=0.0233..1.85 rows=3000 loops=1)
                -> Hash
                    -> Table scan on r  (cost=117 rows=1156) (actual time=0.0696..1.28 rows=1156 loops=1)
```

2. Explore tradeoffs of adding different indices to different attributes on your advanced queries. For each indexing design you try, use the EXPLAIN ANALYZE command to measure the query performance after adding the indices. Report the performance gains or degradations for each indexing configuration you tried. You may use a line chart to visualize your results if you wish.
    a. Tips: Don't index primary keys, don't index attributes that don't appear in your query. We recommend trying to index JOIN attributes or attributes that appear in the WHERE, GROUP BY, or HAVING clauses. If there aren't enough indexable attributes in your advanced query, you will need to modify it so that there are.

Query 1:

Original cost = 5621

Indexing design 1:

- **Cost = 5621**
- This cost is the same as the cost of the original query.
- CREATE INDEX idx_rentals_garage ON Rentals(GarageId);
- Optimizes the JOIN between Rentals and Garages, reducing lookup time.
- Speeds up queries by allowing the database to quickly find matching GarageId values instead of performing a full table scan.
- Reduces the number of comparisons needed in the join operation.

Command:

```sql
USE RentalDB;
CREATE INDEX idx_rentals_garage ON Rentals(GarageId);
EXPLAIN ANALYZE
SELECT g.GarageId, SUM(r.TotalCost) AS TotalRevenue, g.Profit
FROM Rentals r
JOIN Garages g ON r.GarageId = g.GarageId
GROUP BY g.GarageId, g.Profit
HAVING SUM(r.TotalCost) > 800
ORDER BY TotalRevenue DESC
LIMIT 15;
```

Output:

```
-> Limit: 15 row(s)  (actual time=27.8..27.8 rows=15 loops=1)
    -> Sort: TotalRevenue DESC  (actual time=27.8..27.8 rows=15 loops=1)
        -> Filter: (`sum(r.TotalCost)` > 800)  (actual time=27.6..27.8 rows=25 loops=1)
            -> Table scan on <temporary>  (actual time=27.5..27.7 rows=1053 loops=1)
                -> Aggregate using temporary table  (actual time=27.5..27.5 rows=1053 loops=1)
                    -> Nested loop inner join  (cost=5621 rows=12687) (actual time=0.105..26
rows=1156 loops=1)
                        -> Filter: (g.GarageId is not null)  (cost=1180 rows=11557) (actual
time=0.0315..7.64 rows=11703 loops=1)
                            -> Table scan on g  (cost=1180 rows=11557) (actual time=0.0283..6.71
rows=11703 loops=1)
                        -> Index lookup on r using idx_rentals_garage (GarageId=g.GarageId)
(cost=0.274 rows=1.1) (actual time=0.00137..0.00144 rows=0.0988 loops=11703)
```

Indexing design 2:

- **Cost = 9674**

- This cost is higher than the cost of the original query.
- CREATE INDEX idx_rentals_garage_totalcost_endtime ON Rentals(GarageId, TotalCost, EndTime);
- Fully optimizes the query by covering JOIN (GarageId), aggregation (SUM(TotalCost)), and sorting (ORDER BY TotalRevenue DESC).
- The database can retrieve, aggregate, and sort rows faster, minimizing sorting overhead.
- Reduces the need for expensive temporary sorting operations.

Command:

```sql
USE RentalDB;
SHOW INDEX FROM Rentals;
CREATE INDEX idx_rentals_garage_totalcost_endtime ON Rentals(GarageId, TotalCost, EndTime);
EXPLAIN ANALYZE
SELECT g.GarageId, SUM(r.TotalCost) AS TotalRevenue, g.Profit
FROM Rentals r
JOIN Garages g ON r.GarageId = g.GarageId
GROUP BY g.GarageId, g.Profit
HAVING SUM(r.TotalCost) > 800
ORDER BY TotalRevenue DESC
LIMIT 15;
```

Output:

```
-> Limit: 15 row(s)  (actual time=32.6..32.6 rows=15 loops=1)
    -> Sort: TotalRevenue DESC  (actual time=32.6..32.6 rows=15 loops=1)
        -> Filter: (`sum(r.TotalCost)` > 800)  (actual time=32.3..32.5 rows=25 loops=1)
            -> Table scan on <temporary>  (actual time=32.3..32.4 rows=1053 loops=1)
                -> Aggregate using temporary table  (actual time=32.3..32.3 rows=1053 loops=1)
                    -> Nested loop inner join  (cost=9674 rows=12687) (actual time=0.0527..30.1
rows=1156 loops=1)
                        -> Filter: (g.GarageId is not null)  (cost=1180 rows=11557) (actual
time=0.0295..8.61 rows=11703 loops=1)
                            -> Table scan on g  (cost=1180 rows=11557) (actual time=0.0288..7.48
rows=11703 loops=1)
                        -> Covering index lookup on r using idx_rentals_garage_totalcost_endtime
(GarageId=g.GarageId)  (cost=0.625 rows=1.1) (actual time=0.00163..0.0017 rows=0.0988 loops=11703)
```

Indexing design 3:

- **Cost = 9673**
- This cost is higher than the cost of the original query.
- CREATE INDEX idx_rentals_garage_totalcost ON Rentals(GarageId, TotalCost);
- Improves both JOIN and GROUP BY operations by making lookups on GarageId and SUM(TotalCost) more efficient.
- Allows the database to use an index-only scan, reducing disk I/O.
- Reduces temporary table creation for aggregation.

Command:

```sql
USE RentalDB;
SHOW INDEX FROM Rentals;
CREATE INDEX idx_rentals_garage_totalcost ON Rentals(GarageId, TotalCost);
EXPLAIN ANALYZE
SELECT g.GarageId, SUM(r.TotalCost) AS TotalRevenue, g.Profit
FROM Rentals r
JOIN Garages g ON r.GarageId = g.GarageId
GROUP BY g.GarageId, g.Profit
HAVING SUM(r.TotalCost) > 800
ORDER BY TotalRevenue DESC
LIMIT 15;
```

Output:

```
-> Limit: 15 row(s)  (actual time=25.1..25.1 rows=15 loops=1)
    -> Sort: TotalRevenue DESC  (actual time=25.1..25.1 rows=15 loops=1)
        -> Filter: (`sum(r.TotalCost)` > 800)  (actual time=24.8..25 rows=25 loops=1)
            -> Table scan on <temporary>  (actual time=24.8..24.9 rows=1053 loops=1)
                -> Aggregate using temporary table  (actual time=24.8..24.8 rows=1053 loops=1)
                    -> Nested loop inner join  (cost=9673 rows=12687) (actual time=0.0545..23.8
rows=1156 loops=1)
                        -> Filter: (g.GarageId is not null)  (cost=1180 rows=11557) (actual
time=0.0293..7.26 rows=11703 loops=1)
                            -> Table scan on g  (cost=1180 rows=11557) (actual time=0.0286..6.38
rows=11703 loops=1)
                        -> Covering index lookup on r using idx_rentals_garage_totalcost
(GarageId=g.GarageId)  (cost=0.625 rows=1.1) (actual time=0.00124..0.00129 rows=0.0988 loops=11703)
```

Query 2:

Original cost = 1.34 x 10^6

Indexing design 1:

CREATE INDEX idx_cardata_make_model ON CarData(Make, Model);

- Cost = 9610
- This cost is smaller than the cost before adding the index.
- Allows faster lookups by speeding up GROUP BY on Make and Model
- Also reduces sorting cost in ORDER BY if the database uses this index

Command:

```sql
USE RentalDB;

CREATE INDEX idx_cardata_make_model ON CarData(Make, Model);
EXPLAIN ANALYZE

SELECT cd.Make, cd.Model, SUM(r.TotalCost) AS TotalRevenue
FROM CarData cd
JOIN Rentals r ON cd.CarId = r.CarId
GROUP BY cd.Make, cd.Model
ORDER BY TotalRevenue DESC
```

Output:

```
-> Sort: TotalRevenue DESC  (actual time=36.3..36.4 rows=294 loops=1)
    -> Table scan on <temporary>  (actual time=36..36.1 rows=294 loops=1)
        -> Aggregate using temporary table  (actual time=36..36 rows=294 loops=1)
            -> Nested loop inner join  (cost=9610 rows=11619) (actual time=0.0724..32.9 rows=2312
loops=1)
                -> Filter: (cd.CarId is not null)  (cost=1186 rows=11619) (actual time=0.0466..8.67
rows=11702 loops=1)
                    -> Table scan on cd  (cost=1186 rows=11619) (actual time=0.0454..7.74 rows=11702
loops=1)
                -> Covering index lookup on r using idx_Rentals_CarId_TotalCost (CarId=cd.CarId)
(cost=0.625 rows=1) (actual time=0.00184..0.00194 rows=0.198 loops=11702)
```

Indexing design 2:

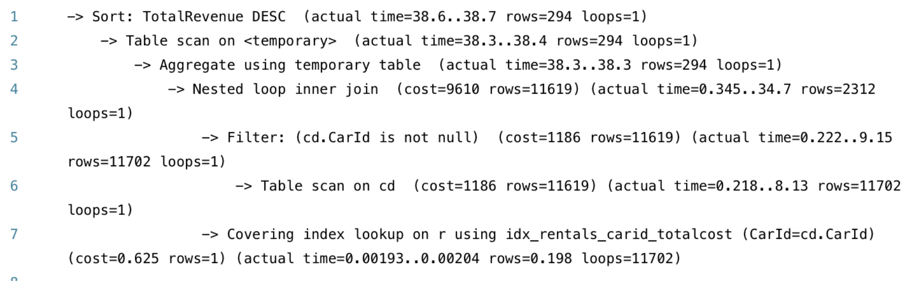
CREATE INDEX idx_rentals_totalcost ON Rentals(TotalCost);

- Cost = 9610
- Speeds up aggregation functions like SUM(TotalCost) since it enables quicker access to cost values
- Reduces full table searches when filtering or sorting by TotalCost
- Improves performance of queries that utilize ORDER BY TotalCost
- Helps speed up and optimize queries that analyze data for revenue of rentals

Command:

```sql
USE RentalDB;
CREATE INDEX idx_rentals_totalcost ON Rentals(TotalCost);
EXPLAIN ANALYZE

SELECT cd.Make, cd.Model, SUM(r.TotalCost) AS TotalRevenue
FROM CarData cd
JOIN Rentals r ON cd.CarId = r.CarId
GROUP BY cd.Make, cd.Model
ORDER BY TotalRevenue DESC
```

Output:



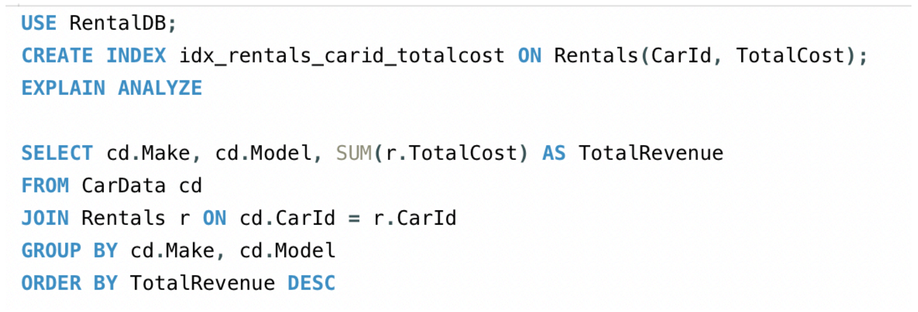
```
-> Sort: TotalRevenue DESC  (actual time=38.6..38.7 rows=294 loops=1)
    -> Table scan on <temporary>  (actual time=38.3..38.4 rows=294 loops=1)
        -> Aggregate using temporary table  (actual time=38.3..38.3 rows=294 loops=1)
            -> Nested loop inner join  (cost=9610 rows=11619) (actual time=0.345..34.7 rows=2312
loops=1)
                -> Filter: (cd.CarId is not null)  (cost=1186 rows=11619) (actual time=0.222..9.15
rows=11702 loops=1)
                    -> Table scan on cd  (cost=1186 rows=11619) (actual time=0.218..8.13 rows=11702
loops=1)
                -> Covering index lookup on r using idx_rentals_carid_totalcost (CarId=cd.CarId)
(cost=0.625 rows=1) (actual time=0.00193..0.00204 rows=0.198 loops=11702)
```

Indexing design 3:

CREATE INDEX idx_rentals_carid_totalcost ON Rentals(CarId, TotalCost);

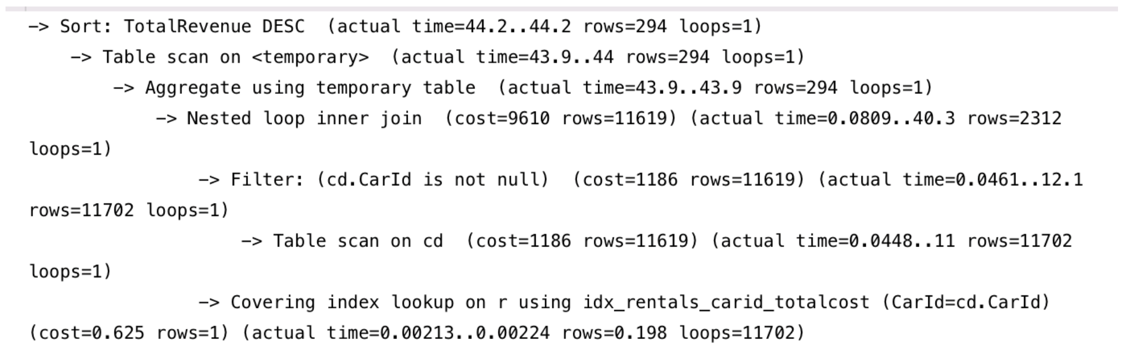
- Cost = 9610
- The primary key already indexes CarId, and adding TotalCost to the index enhances query efficiency since it covered multiple conditions in a single index
- Speeds up aggregation and filtering by indexing by CarId and TotalCost together

Command:



```sql
USE RentalDB;
CREATE INDEX idx_rentals_carid_totalcost ON Rentals(CarId, TotalCost);
EXPLAIN ANALYZE

SELECT cd.Make, cd.Model, SUM(r.TotalCost) AS TotalRevenue
FROM CarData cd
JOIN Rentals r ON cd.CarId = r.CarId
GROUP BY cd.Make, cd.Model
ORDER BY TotalRevenue DESC
```

Output:



```
-> Sort: TotalRevenue DESC  (actual time=44.2..44.2 rows=294 loops=1)
    -> Table scan on <temporary>  (actual time=43.9..44 rows=294 loops=1)
        -> Aggregate using temporary table  (actual time=43.9..43.9 rows=294 loops=1)
            -> Nested loop inner join  (cost=9610 rows=11619) (actual time=0.0809..40.3 rows=2312
loops=1)
                -> Filter: (cd.CarId is not null)  (cost=1186 rows=11619) (actual time=0.0461..12.1
rows=11702 loops=1)
                    -> Table scan on cd  (cost=1186 rows=11619) (actual time=0.0448..11 rows=11702
loops=1)
                -> Covering index lookup on r using idx_rentals_carid_totalcost (CarId=cd.CarId)
(cost=0.625 rows=1) (actual time=0.00213..0.00224 rows=0.198 loops=11702)
```

Query 3:

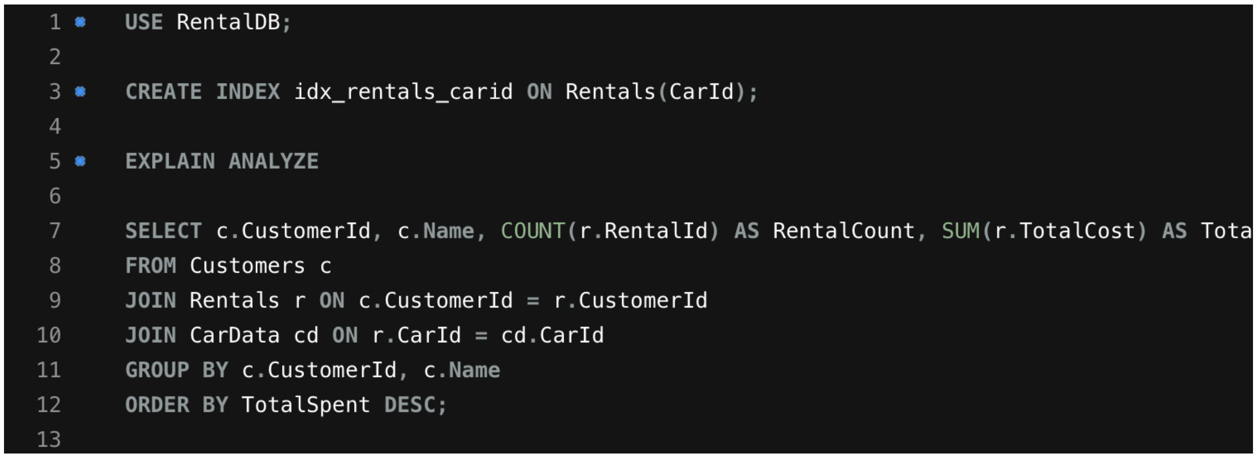
Original cost = 4.17e+6

Indexing design 1:

- **Cost =** 4.17e+6
- This cost is the same as the cost of the original query.
- CREATE INDEX idx_rentals_carid ON Rentals(CarId);
- Optimizes the JOIN between Rentals and CarData, reducing lookup time.

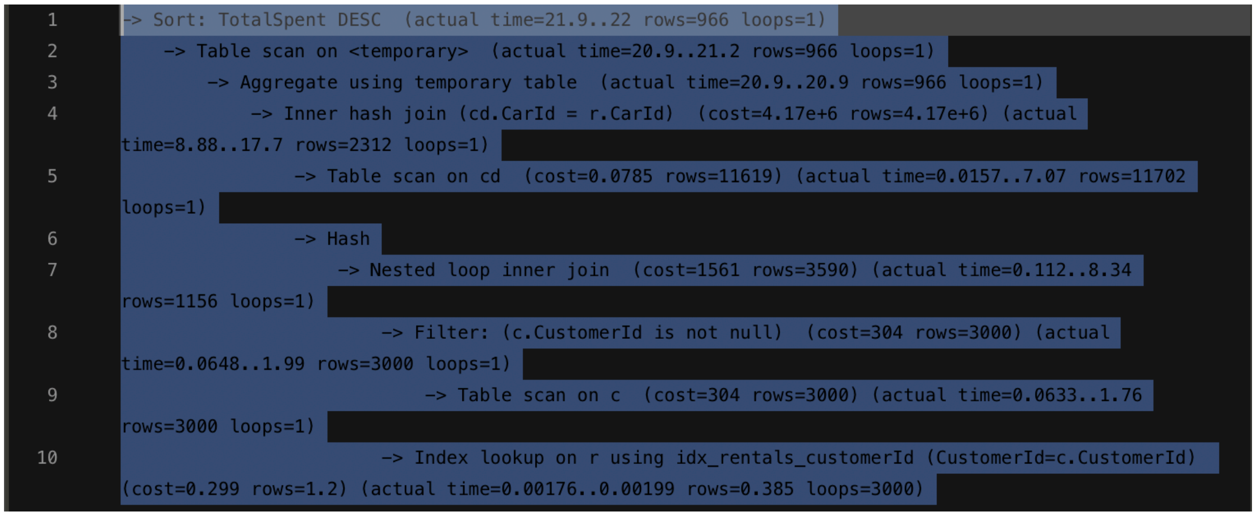
- Speeds up queries by allowing the database to quickly find matching CarId values instead of performing a full table scan.
- Reduces the number of comparisons needed in the join operation.

Command:



```sql
USE RentalDB;

CREATE INDEX idx_rentals_carid ON Rentals(CarId);

EXPLAIN ANALYZE

SELECT c.CustomerId, c.Name, COUNT(r.RentalId) AS RentalCount, SUM(r.TotalCost) AS Tota
FROM Customers c
JOIN Rentals r ON c.CustomerId = r.CustomerId
JOIN CarData cd ON r.CarId = cd.CarId
GROUP BY c.CustomerId, c.Name
ORDER BY TotalSpent DESC;

```

Output:

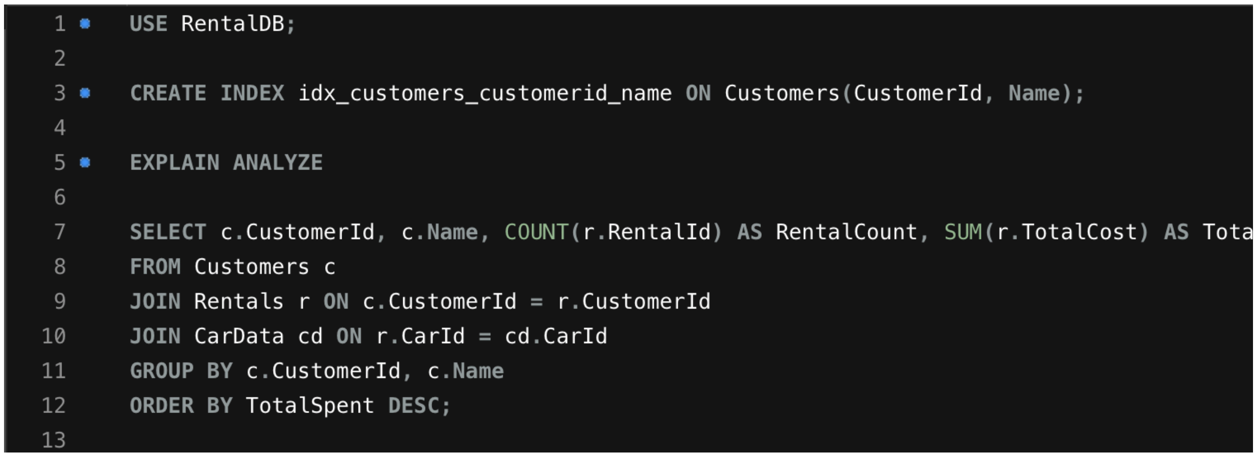


```
-> Sort: TotalSpent DESC  (actual time=21.9..22 rows=966 loops=1)
    -> Table scan on <temporary>  (actual time=20.9..21.2 rows=966 loops=1)
        -> Aggregate using temporary table  (actual time=20.9..20.9 rows=966 loops=1)
            -> Inner hash join (cd.CarId = r.CarId)  (cost=4.17e+6 rows=4.17e+6) (actual
time=8.88..17.7 rows=2312 loops=1)
                -> Table scan on cd  (cost=0.0785 rows=11619) (actual time=0.0157..7.07 rows=11702
loops=1)
                -> Hash
                    -> Nested loop inner join  (cost=1561 rows=3590) (actual time=0.112..8.34
rows=1156 loops=1)
                        -> Filter: (c.CustomerId is not null)  (cost=304 rows=3000) (actual
time=0.0648..1.99 rows=3000 loops=1)
                            -> Table scan on c  (cost=304 rows=3000) (actual time=0.0633..1.76
rows=3000 loops=1)
                        -> Index lookup on r using idx_rentals_customerId (CustomerId=c.CustomerId)
(cost=0.299 rows=1.2) (actual time=0.00176..0.00199 rows=0.385 loops=3000)
```

Indexing design 2:

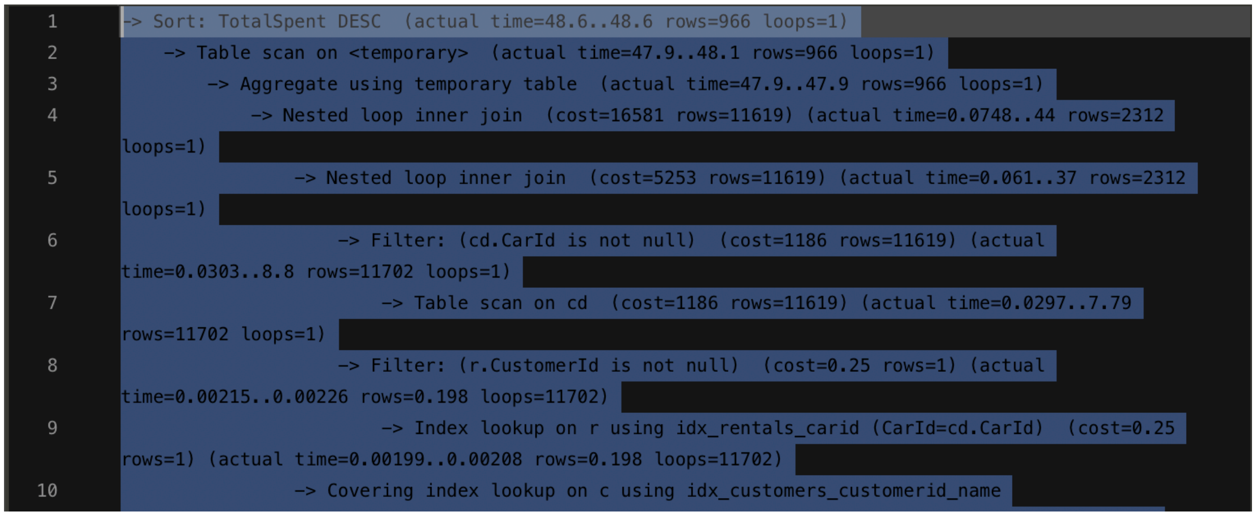
- **Cost = 16581**
- This cost is lower than the cost of the original query.
- CREATE INDEX idx_customers_customerid_name ON Customers(CustomerId, Name);
- Optimizes the GROUP BY of c.CustomerId and c.Name.
- Without the index, the database is forced to do a whole scan of the Customers table to sort and group the results.
- With the index on (CustomerId, Name), the database is able to efficiently retrieve the grouped results because they are already sorted in index order.

Command:



```sql
USE RentalDB;

CREATE INDEX idx_customers_customerid_name ON Customers(CustomerId, Name);

EXPLAIN ANALYZE

SELECT c.CustomerId, c.Name, COUNT(r.RentalId) AS RentalCount, SUM(r.TotalCost) AS Tota
FROM Customers c
JOIN Rentals r ON c.CustomerId = r.CustomerId
JOIN CarData cd ON r.CarId = cd.CarId
GROUP BY c.CustomerId, c.Name
ORDER BY TotalSpent DESC;

```

Output



```
-> Sort: TotalSpent DESC  (actual time=48.6..48.6 rows=966 loops=1)
    -> Table scan on <temporary>  (actual time=47.9..48.1 rows=966 loops=1)
        -> Aggregate using temporary table  (actual time=47.9..47.9 rows=966 loops=1)
            -> Nested loop inner join  (cost=16581 rows=11619) (actual time=0.0748..44 rows=2312
loops=1)
                -> Nested loop inner join  (cost=5253 rows=11619) (actual time=0.061..37 rows=2312
loops=1)
                    -> Filter: (cd.CarId is not null)  (cost=1186 rows=11619) (actual
time=0.0303..8.8 rows=11702 loops=1)
                        -> Table scan on cd  (cost=1186 rows=11619) (actual time=0.0297..7.79
rows=11702 loops=1)
                    -> Filter: (r.CustomerId is not null)  (cost=0.25 rows=1) (actual
time=0.00215..0.00226 rows=0.198 loops=11702)
                        -> Index lookup on r using idx_rentals_carid (CarId=cd.CarId)  (cost=0.25
rows=1) (actual time=0.00199..0.00208 rows=0.198 loops=11702)
                -> Covering index lookup on c using idx_customers_customerid_name
```

Query 4:

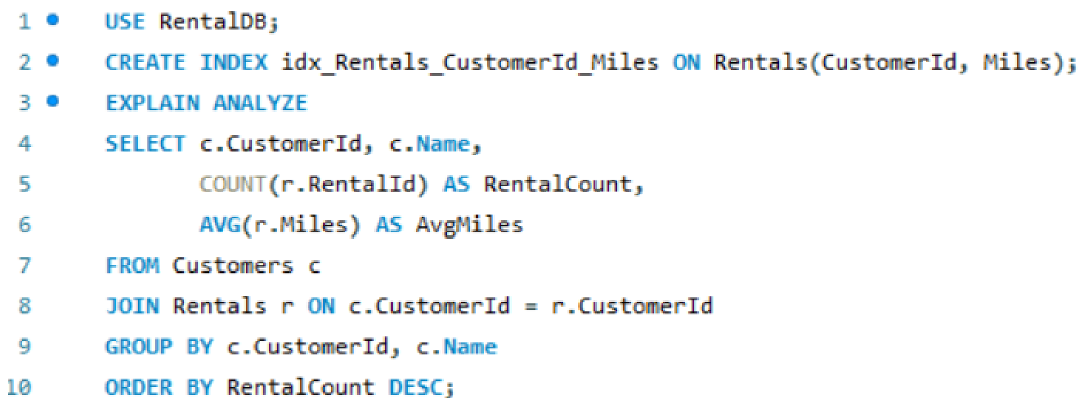
Original cost = 346922

Indexing design 1:

CREATE INDEX idx_Rentals_CustomerId_Miles ON Rentals(CustomerId, Miles);

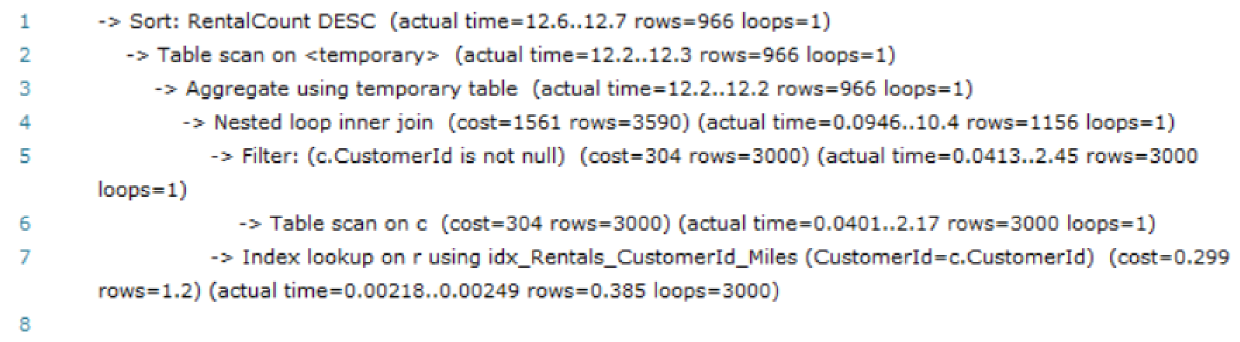
- Cost = 1561

Command:



```sql
USE RentalDB;
CREATE INDEX idx_Rentals_CustomerId_Miles ON Rentals(CustomerId, Miles);
EXPLAIN ANALYZE
SELECT c.CustomerId, c.Name,
       COUNT(r.RentalId) AS RentalCount,
       AVG(r.Miles) AS AvgMiles
FROM Customers c
JOIN Rentals r ON c.CustomerId = r.CustomerId
GROUP BY c.CustomerId, c.Name
ORDER BY RentalCount DESC;
```

Output:



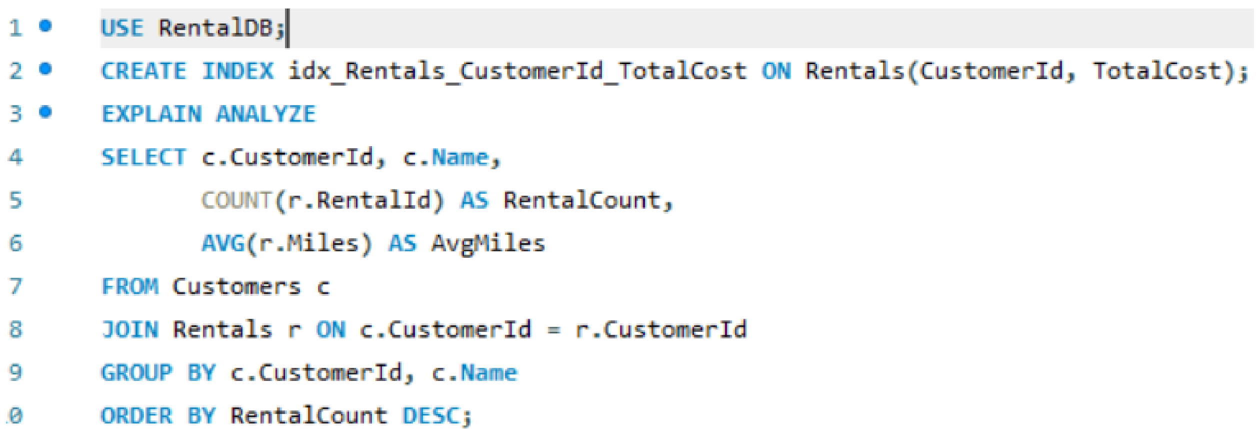
```
-> Sort: RentalCount DESC  (actual time=12.6..12.7 rows=966 loops=1)
    -> Table scan on <temporary>  (actual time=12.2..12.3 rows=966 loops=1)
        -> Aggregate using temporary table  (actual time=12.2..12.2 rows=966 loops=1)
            -> Nested loop inner join  (cost=1561 rows=3590) (actual time=0.0946..10.4 rows=1156 loops=1)
                -> Filter: (c.CustomerId is not null)  (cost=304 rows=3000) (actual time=0.0413..2.45 rows=3000 loops=1)
                    -> Table scan on c  (cost=304 rows=3000) (actual time=0.0401..2.17 rows=3000 loops=1)
                -> Index lookup on r using idx_Rentals_CustomerId_Miles (CustomerId=c.CustomerId)  (cost=0.299 rows=1.2) (actual time=0.00218..0.00249 rows=0.385 loops=3000)
```

Indexing design 2:

CREATE INDEX idx_Rentals_CustomerId_TotalCost ON Rentals(CustomerId, TotalCost);

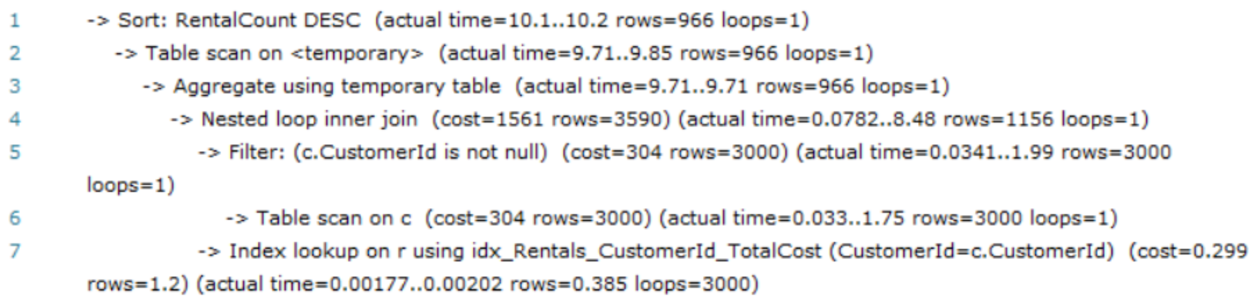
- Cost = 1561

Command:



```
USE RentalDB;
CREATE INDEX idx_Rentals_CustomerId_TotalCost ON Rentals(CustomerId, TotalCost);
EXPLAIN ANALYZE
SELECT c.CustomerId, c.Name,
       COUNT(r.RentalId) AS RentalCount,
       AVG(r.Miles) AS AvgMiles
FROM Customers c
JOIN Rentals r ON c.CustomerId = r.CustomerId
GROUP BY c.CustomerId, c.Name
ORDER BY RentalCount DESC;
```

Output:



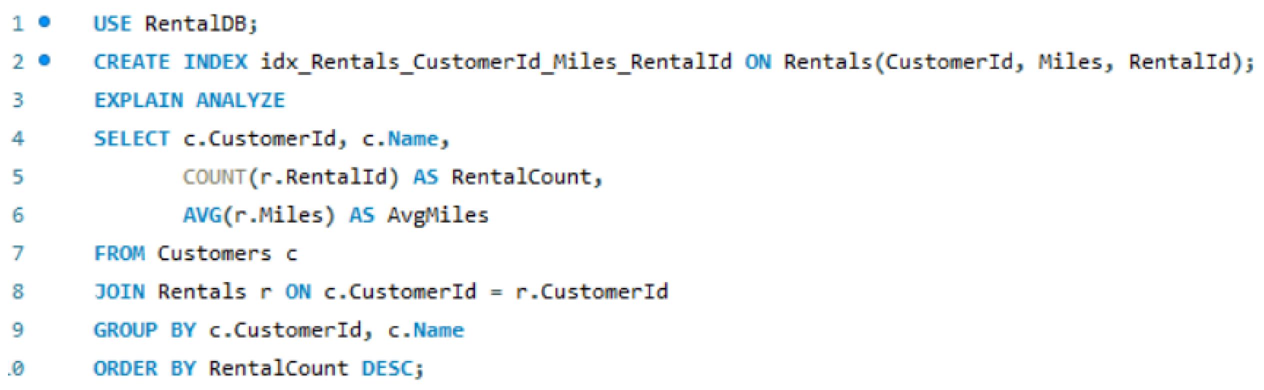
```
-> Sort: RentalCount DESC  (actual time=10.1..10.2 rows=966 loops=1)
    -> Table scan on <temporary>  (actual time=9.71..9.85 rows=966 loops=1)
        -> Aggregate using temporary table  (actual time=9.71..9.71 rows=966 loops=1)
            -> Nested loop inner join  (cost=1561 rows=3590) (actual time=0.0782..8.48 rows=1156 loops=1)
                -> Filter: (c.CustomerId is not null)  (cost=304 rows=3000) (actual time=0.0341..1.99 rows=3000 loops=1)
                    -> Table scan on c  (cost=304 rows=3000) (actual time=0.033..1.75 rows=3000 loops=1)
                -> Index lookup on r using idx_Rentals_CustomerId_TotalCost (CustomerId=c.CustomerId)  (cost=0.299 rows=1.2) (actual time=0.00177..0.00202 rows=0.385 loops=3000)
```

Indexing design 3:

CREATE INDEX idx_Rentals_CustomerId_Miles_RentalId ON Rentals(CustomerId, Miles, RentalId);

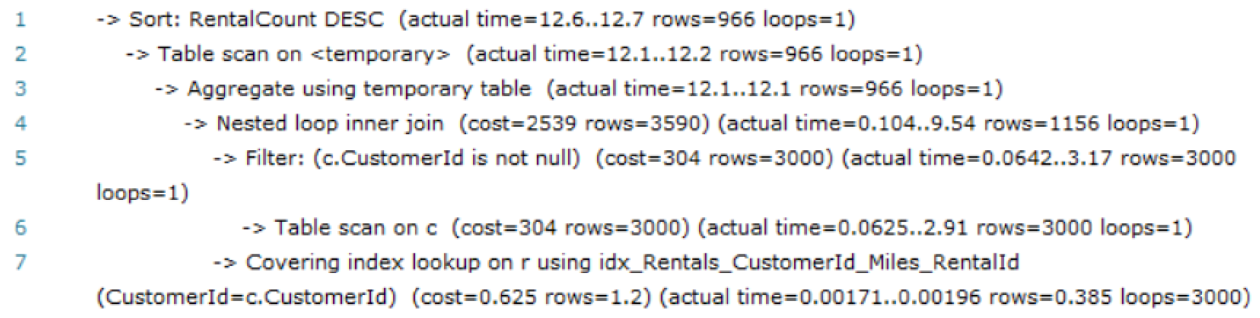
- Cost = 2539

Command:



```sql
USE RentalDB;
CREATE INDEX idx_Rentals_CustomerId_Miles_RentalId ON Rentals(CustomerId, Miles, RentalId);
EXPLAIN ANALYZE
SELECT c.CustomerId, c.Name,
       COUNT(r.RentalId) AS RentalCount,
       AVG(r.Miles) AS AvgMiles
FROM Customers c
JOIN Rentals r ON c.CustomerId = r.CustomerId
GROUP BY c.CustomerId, c.Name
ORDER BY RentalCount DESC;
```

Output:



```
-> Sort: RentalCount DESC  (actual time=12.6..12.7 rows=966 loops=1)
   -> Table scan on <temporary>  (actual time=12.1..12.2 rows=966 loops=1)
      -> Aggregate using temporary table  (actual time=12.1..12.1 rows=966 loops=1)
         -> Nested loop inner join  (cost=2539 rows=3590) (actual time=0.104..9.54 rows=1156 loops=1)
            -> Filter: (c.CustomerId is not null)  (cost=304 rows=3000) (actual time=0.0642..3.17 rows=3000 
loops=1)
               -> Table scan on c  (cost=304 rows=3000) (actual time=0.0625..2.91 rows=3000 loops=1)
            -> Covering index lookup on r using idx_Rentals_CustomerId_Miles_RentalId 
(CustomerId=c.CustomerId)  (cost=0.625 rows=1.2) (actual time=0.00171..0.00196 rows=0.385 loops=3000)
```

3. Report on the final index design you selected and explain why you chose it, referencing the analysis you performed. Note that if you did not find any difference in your results, report that as well. Explain why you think this change in indexing did not bring a better effect on your queries' performance.

   Query 1:

   The final indexing design we selected for this query was indexing design #1:
   CREATE INDEX idx_rentals_garage ON Rentals(GarageId);

   This index was chosen because it provides the same query execution cost as the original query (5621), meaning it optimizes performance without introducing unnecessary drawbacks. By indexing GarageId, the primary attribute used in the JOIN operation, this design allows the database to efficiently locate matching records in the Rentals table, reducing the need for a full table scan. In contrast, indexing designs #2 and #3 resulted in higher costs (9674 and 9673, respectively), indicating that the additional indexed attributes did not improve performance and may have introduced unnecessary complexity. While these other designs aimed to optimize aggregation and sorting, they did not yield the expected benefits and instead increased query cost. Therefore, indexing design #1 is the best choice, as it enhances JOIN efficiency without negatively impacting query performance or reducing performance cost.

   Query 2:

   The final indexing design we selected for this query was indexing design #3:

CREATE INDEX idx_rentals_carid_totalcost ON Rentals(CarId, TotalCost);

This is the most optimal index design because it is efficient for both JOINs and aggregation commands, which decreases the cost of the query from 1.34 x 10^6 to 9610. This composite index enhances the efficiency of data lookups because CarId is the attribute used in the JOIN and TotalCost is aggregated with the SUM function. As a result, users of the query can avoid full table scans. In addition, since the primary key is used in conjunction with TotalCost, it improves the performance by minimizing redundancy and duplication. Although the other two query designs have the same cost, the benefits of design #3 outweigh the others'. In design #1, idx_cardata_make_model helps GROUP BY while idx_rentals_totalcost in design #2 speeds up SUM. Therefore, index design #3 is the most versatile since it optimizes JOIN, filtering, and aggregation by SUM in a step.

Query 3:

The final indexing design we selected for this query was indexing design #2:
CREATE INDEX idx_customers_customerid_name ON Customers(CustomerId, Name);

This is the most optimal index design because it significantly improves the efficiency of the GROUP BY operation, reducing the cost of the query from 4.17 × 10^6 to 16,581. The composite index enhances the performance of the GROUP BY by ensuring that CustomerId and Name are pre-sorted, which ultimately allows the database engine to efficiently group results without having to do a full scan of the table or a taxing or expensive sorting operation. Also, this index reduces how many redundant lookups could originally occur, as CustomerId is frequently used in JOIN statements, and Name is for optimizing sorting within the groups. Compared to design #1 (idx_rentals_carid), which attempted to speed up the JOIN between Rentals and CarData but did not reduce the query cost, indexing design #2 gives the most gain in performance.

Query 4:

The final indexing design we selected for Query 4 was indexing design #2:
idx_Rentals_CustomerId_TotalCost on Rentals(CustomerId, TotalCost).

Query 4 returns the average miles of all trips rented by each customer. The original cost of the query was 346922. This index resulted in a reduced cost of 1561 which was the same as indexing design #1. The cost using indexing designs #1 and #2 were significantly lower than indexing design #3, which had a cost of 2539. By indexing CustomerId and TotalCost, this design optimized the JOIN on CustomerId. This indexing design also improved the efficiency of aggregation clauses within our query like COUNT(RentalId) and AVG(Miles). Indexing design 3 included three columns and introduced unnecessary overhead, so indexing design 2 was better suited to run the queries efficiently.