

CarQuest: A Student-Centric Used Car Database Solution

Arjun Srinivasan

Balaji Hariharan

Sai Teja Billakanti

I. INTRODUCTION

Reliable, economical, and environmentally friendly transportation options are in high demand in today's changing automotive market, which has resulted in a sharp increase in the appeal of used vehicles. But for prospective buyers, sorting through the wide selection of available cars can be a difficult undertaking. It might take a lot of time and energy to sort through internet listings, find reliable information, and come to a well-informed selection. Our idea aims to introduce a state-of-the-art platform into the used car industry in order to tackle this difficulty. This platform makes use of a strong database system to expedite the search procedure and give consumers an easy-to-use interface. Our goal is to improve data security, accessibility, and accuracy by switching from traditional spreadsheet-based methods to a dynamic database architecture.

II. PHASE I OVERVIEW

In the initial phase of Car Quest development, our focus was on meticulously cleaning and refining the dataset to ensure its integrity and reliability for subsequent stages. A thorough assessment for BCNF (Boyce-Codd Normal Form) compliance was conducted, a pivotal step in optimizing the database. With the data now in impeccable condition, we are prepared to advance to the next phase, which involves the implementation of SQL queries and the construction of the website. This phase marks a substantial stride toward achieving a robust and functional platform.

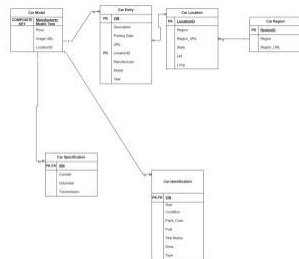


Fig. 1. ER Diagram of CarQuest

III. DATABASE SCHEMA

A. Car_Region Table

```
CREATE TABLE Car_Region ( RegionID
SERIAL PRIMARY KEY, Region
VARCHAR DEFAULT NULL,
Region_URL VARCHAR DEFAULT
NULL );
```

B. Car_Location Table

```
CREATE TABLE Car_Location (
LocationID SERIAL PRIMARY KEY,
RegionID INTEGER DEFAULT 0, State
VARCHAR(2) DEFAULT NULL, Latitude
DECIMAL(9,6) DEFAULT 0.0,

Longitude DECIMAL(9,6) DEFAULT
0.0,
FOREIGN KEY (RegionID) REFERENCES
Car_Region(RegionID) );
```

C. Car_Model Table

```
CREATE TABLE Car_Model (
ModelID SERIAL PRIMARY KEY,
Manufacturer VARCHAR DEFAULT
NULL,
ModelName VARCHAR DEFAULT
NULL, Year INTEGER DEFAULT 0 );
```

D. Car_Entry Table

```
CREATE TABLE Car_Entry ( VIN
VARCHAR PRIMARY KEY, Description
VARCHAR DEFAULT NULL, PostingDate
DATE DEFAULT NULL,
URL VARCHAR DEFAULT NULL,
LocationID INTEGER DEFAULT 0,
ModelID INTEGER DEFAULT 0,
FOREIGN KEY (LocationID)
REFERENCES Car_Location(LocationID),
FOREIGN KEY (ModelID) REFERENCES
Car_Model(ModelID)
);
```

E. Car_Identification Table

```
CREATE TABLE Car_Identification (
VIN VARCHAR PRIMARY KEY, Size
VARCHAR DEFAULT NULL, Condition
VARCHAR DEFAULT NULL, PaintColor
VARCHAR DEFAULT NULL, Fuel
VARCHAR DEFAULT NULL, TitleStatus
VARCHAR DEFAULT NULL, Drive
VARCHAR DEFAULT NULL, Type
VARCHAR DEFAULT NULL,
FOREIGN KEY (VIN) REFERENCES
Car_Entry(VIN)
);
```

F. Car_Specification Table

```
CREATE TABLE Car_Specification (
VIN VARCHAR PRIMARY KEY,
Cylinder VARCHAR DEFAULT NULL,
Odometer INTEGER DEFAULT 0,
Transmission VARCHAR DEFAULT
NULL,
FOREIGN KEY (VIN) REFERENCES
Car_Entry(VIN)
);
```

IV. HANDLING QUERY EXECUTION USING INDEXING STRATEGIES

A. Challenges Faced

1) Handling the Original Large Dataset: Since the original large dataset represented 'used car' data in a scattered manner, our goal was to simplify access for a layman user to this data. To achieve this, we classified the entire dataset into manageable smaller tables based on the region/location where the cars are available. This modularization aimed to split the large chunk of data into smaller, meaningful tables such as Car_Region, Car_Model, Car_Specification, and Car_Identification.

B. Usage of Indexing

1) Car_Region Table: In the creation of the Car_Region table, we have the implicit index column RegionID. This is a key column that is unique to each row in the table and thus helps to quickly locate and access the data.

2) Car_Location Table:

- **Primary Key Index:** LocationID serves as the primary key, acting as an implicit index.
- **Foreign Key Index:** RegionID is used to refer to the Car_Region table, functioning as a foreign key index that speeds up join operations between the Car_Region and Car_Location tables.

3) Car_Model Table:

- **Primary Key Index:** The primary key on ModelID simplifies the search for car models.

4) Car_Entry Table:

- **Primary Key Index:** The Vehicle Identification Number (VIN) is the unique identifier for each car, ensuring effective retrieval of each car entry listing.
- **Foreign Key Index:** The foreign keys LocationID and ModelID reference the Car_Location and Car_Model tables, respectively.

C. Execution of Complex Queries

```

26 SELECT
27   cm.modelName, cm.manufacturer,
28   ce.description, ce.postingDate,
29   ce.cyl, cr.cyl, lmp.cyl, ce.gasType,
30   ci.cl.vin, ci.condition,
31   ci.paintColor, ci.fuel,
32   ci.transmission, ci.drive,
33   ci.tp.type, ci.state,
34   ci.latITUDE, ci.longITUDE,
35   cm.year, cr.region
36 FROM
37   cm_entry ce
38 JOIN
39   cm_model cm ON ce.modelId = cm.modelId
40 JOIN
41   ce_location cl ON ce.locationId = cl.locationId
42 JOIN
43   cr_region cr ON cl.regionId = cr.regionId
44 JOIN
45   ci_carIdentification ci ON ce.vin = ci.vin
46 JOIN
47   ce_identification ce_i ON ce.vin = ce_i.vin
48 --WHERE
49   --(cm.modelName='golf' and ci.state='ca')
50
51

```

Fig. 2. Before Indexing

Data Output:

Query History

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

174

175

176

177

178

179

180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

212

213

214

215

216

217

218

219

220

221

222

223

224

225

226

227

228

229

230

231

232

233

234

235

236

237

238

239

240

241

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

261

262

263

264

265

266

267

268

269

270

271

272

273

274

275

276

277

278

279

280

281

282

283

284

285

286

287

288

289

290

291

292

293

294

295

296

297

298

299

300

301

302

303

304

305

306

307

308

309

310

311

312

313

314

315

316

317

318

319

320

321

322

323

324

325

326

327

328

329

330

331

332

333

334

335

336

337

338

339

340

341

342

343

344

345

346

347

348

349

350

351

352

353

354

355

356

357

358

359

360

361

362

363

364

365

366

367

368

369

370

371

372

373

374

375

376

377

378

379

380

381

382

383

384

385

386

387

388

389

390

391

392

393

394

395

396

397

398

399

400

401

402

403

404

405

406

407

408

409

410

411

412

413

414

415

416

417

418

419

420

421

422

423

424

425

426

427

428

429

430

431

432

433

434

435

436

437

438

439

440

441

442

443

444

445

446

447

448

449

450

451

452

453

454

455

456

457

458

459

460

461

462

463

464

465

466

467

468

469

470

471

472

473

474

475

476

477

478

Fig. 3. Before Indexing(Observe the Query Exceution time-00:00:01.156

Prior to the implementation of indexing, the query execution time stood at 00:00:01.156 time units. This duration represents the time taken to retrieve and process data from the database without the benefits of indexing optimization. Recognizing the necessity to improve query performance, indexing concepts were adopted to enhance the efficiency of data retrieval operations. Subsequent sections will provide detailed insights into the specific indexing strategies employed and their consequential impact on reducing query execution times.

```
CREATE INDEX idx_car_entry_modelid ON Car_Entry(ModelID);
CREATE INDEX idx_car_entry_locationid ON Car_Entry(LocationID);
CREATE INDEX idx_car_location_regionid ON Car_Location(RegionID);
```

Fig. 4. Applying Indexing Bases on the complex Query seen above

Following the implementation of indexing, there was a noticeable improvement in query performance, with the execution time reduced to 00:00:01.090 time units. This signifies a positive impact on the efficiency of data retrieval operations

Fig. 5. Result after Applying Indexing(Observe the Reduction in Execution time)

compared to the previous duration of 00:00:01.156 time units. The optimization introduced by indexing has contributed to a more streamlined and expedited querying process, enhancing the overall responsiveness of the database.

V. DATABASE OPERATIONS: SELECT

A. List All Cars with Their Location

```
--List all cars with their location
select CE.VIN, CE.Description, CL.State, CR.Region
from Car_Entry CE
join Car_Location CL on CE.LocationID=CL.LocationID
join Car_Region CR on CL.RegionID=CR.RegionID
```

Fig. 6. All Cars with their Locations

1) Explanation: This SQL query employs the JOIN operation to retrieve information about cars along with their respective locations. It involves three tables: Car_Entry (CE), Car_Location (CL), and Car_Region (CR). The query links entries in the Car_Entry table to their corresponding locations using the LocationID column and then associates the locations with their regions through the RegionID column. The result includes columns such as Vehicle Identification Number (VIN), Description, State, and Region for each car.

Fig. 7. Results of All Cars with their Locations

B. Find Cars with Specific Manufacturer in a Given Region

1) Explanation: This SQL query retrieves information about cars with a specific manufacturer (in this case, 'ford') located in a given region (in this case, 'albany'). It involves

```
--Find cars with specific manufacturer in a given region
select CE.VIN, CM.Manufacturer, CM.ModelName, CR.Region
from Car_Entry CE
join Car_Model CM on CE.ModelID=CM.ModelID
join Car_Location CL on CE.LocationID=CL.LocationID
join Car_Region CR on CL.RegionID=CR.RegionID
where CM.Manufacturer='ford' and region='albany'
```

Fig. 8. Find Cars with Specific Manufacturer in a Given Region

joining four tables: Car_Entry (CE), Car_Model (CM), Car_Location (CL), and Car_Region (CR) to link car entries with their models, locations, and regions. The WHERE clause filters the results based on the specified manufacturer and region, providing a targeted list of cars meeting these criteria.

Fig. 9. Results for Cars with Specific Manufacturer in a Given Region

C. Average Price of Each Car Model

```
--Average Price of Each Car Model
SELECT CM.Manufacturer, CM.ModelName, round(AVG(CE.Price),2) AS AveragePrice
FROM Car_Entry CE
join Car_Model CM on CE.ModelID=CM.ModelID
GROUP BY CM.Manufacturer, CM.ModelName;
```

Fig. 10. Find Average Price of Each Car Model

1) Explanation: This SQL query calculates the average price for each car model by joining the Car_Entry (CE) and Car_Model (CM) tables based on the ModelID. The AVG function is used to compute the average price, and the result is rounded to two decimal places using the ROUND function. The GROUP BY clause ensures that the calculation is performed for each unique combination of manufacturer and model name, providing a summary of average prices for different car models.

D. Top 5 Expensive Car Models in Each Region

1) Explanation: This SQL query retrieves the top 5 most expensive car models in each region. It utilizes the ROW_NUMBER() window function to assign a rank to each car entry based on its price within each region. The outer query then filters the results to include only the top 5 expensive car models in each region, ordered by region and price in descending order.

E. Finding Cars with Specific Color and Condition

1) Explanation: This SQL query retrieves information about cars with a specific color ('blue') and

	manufacturer character varying	modelname character varying	averageprice numeric
1	NaN	JT2bg22k5y0513201	3950.00
2	bmw	7 series	23711.43
3	nissan	altima 2.5 4dr sedan	12149.50
4	volvo	c30 t5 2dr hatchback	12500.00
5	ford	econoline e150	8750.00
6	volkswagen	passat s	6270.00
7	NaN	64 Hawk	18500.00
8	toyota	scion	5225.00
9	NaN	Mobility Ventures MV-1 SE Wheel	20900.00
10	ford	transit wagon xl	28995.00
11	volkswagen	passat 2.0t	3000.00
12	porsche	panamera sedan 4d	34990.00
13	NaN	avana Cargo Van	23995.00
14	gmc	terrain awd 4dr sle	24502.00
15	bmw	z4 2.5i	7500.00
16	gmc	yukon xl slt 1500	18995.00
17	ford	f-450 platinum	112500.00
18	subaru	legacy 2.5i sedan 4d	20290.00
19	volvo	s80	6190.71
Total rows: 1000 of 7326			Query complete 00:00:00.124

Fig. 11. Results for Average Price of Each Car Model

```
--Top 5 Expensive Car Models in Each Region
SELECT Region,Manufacturer,ModelName,Price
FROM
(SELECT
  CR.Region,
  CM.Manufacturer,
  CM.ModelName,
  CE.Price,
  ROW_NUMBER() OVER (PARTITION BY CR.Region ORDER BY CE.Price DESC) as rn
FROM Car_Entry CE
JOIN Car_Model CM ON CE.ModelID = CM.ModelID
JOIN Car_Location CL ON CE.LocationID = CL.LocationID
JOIN Car_Region CR ON CL.RegionID = CR.RegionID) as RankedCars
where rn<=5
order by Region,Price DESC
```

Fig. 12. Find Top 5 Expensive Car Models in Each Region

condition ('new'). It involves joining the Car_Entry, Car_Identification, and Car_Model tables to link car entries with their identification details and model information. The WHERE clause filters the results based on the specified paint color and condition.

F. Cars with Mileage Over 100,000 by Manufacturer

```
--Cars with Mileage Over 100,000 by Manufacturer
SELECT CM.Manufacturer, COUNT(*) AS HighMileageCars
FROM Car_Entry CE
join Car_Model CM on CE.ModelID=CM.ModelID
JOIN Car_Specification CS ON CE.VIN = CS.VIN
WHERE CS.Odometer > 100000
GROUP BY CM.Manufacturer;
```

Fig. 16. Results for Cars with Mileage Over 100,000 by Manufacturer

1) Explanation: This SQL query counts the number of cars with mileage exceeding 100,000 by manufacturer. It involves joining the Car_Entry (CE), Car_Model (CM), and Car_Specification (CS) tables to link car entries with their models and specifications. The WHERE clause filters for cars with an odometer reading over 100,000 miles. The COUNT function is then used to tally the number of high-mileage cars for each manufacturer, providing insights into the distribution of cars with significant mileage.

	region character varying	manufacturer character varying	modelname character varying	price numeric
1	atlanta	jeep	wrangler	119000
2	atlanta	NaN	HEARSTER jet	119000
3	atlanta	NaN	2018 Polaris/11000	119000
4	atlanta	NaN	Man Test	100000
5	atlanta	NaN	Man Test	100000
6	atlanta / carolina	chevrolet	corvette	100000
7	atlanta / carolina	chevrolet	corvette	100000
8	atlanta / carolina	jeep	grand cherokee 4x4	69900
9	atlanta / carolina	chevrolet	corvette	69900
10	atlanta / carolina	gmc	acadia 2500hd	61740
11	atlanta	honda	360 models	59900
12	atlanta	chevrolet	corvette z06	60000
13	atlanta	chevrolet	corvette z06	82000
14	atlanta	NaN	NaN	NaN
Total rows: 245 of 245. Query complete 00:00:00.188				

Fig. 13. Results for Top 5 Expensive Car Models in Each Region

```
--Find Cars with Specific Color and Condition
SELECT CE.VIN, CM.ModelName, CE.Description, CI.PaintColor, CI.Condition
FROM Car_Entry CE
JOIN Car_Identification CI ON CE.VIN = CI.VIN
JOIN Car_Model CM ON CE.ModelID=CM.ModelID
WHERE CI.PaintColor = 'blue' AND CI.Condition = 'new';
```

Fig. 14. Finding Cars with Specific Color and Condition

	manufacturer character varying	highmileagecars bigint
1	chevrolet	2533
2	mazda	164
3	audi	113
4	acura	120
5	nissan	625
6	mini	44
7	mercedes-benz	239
8	chrysler	237
9	ram	640
10	kia	239
11	bmw	282
12	pontiac	143
13	infiniti	109
14	volkswagen	205
15	NaN	581
16	ford	3047
17	harley-davidson	3
18	jaguar	18
19	fiat	2

Fig. 17. Results for Cars with Mileage Over 100,000 grouped by Manufacturer

G. Latest Models Available in Each State

```
--Latest Models Available in Each State
SELECT CL.State, MAX(CM.Year) AS LatestModelYear
FROM Car_Model CM
join Car_Entry CE on CE.ModelID=CM.ModelID
JOIN Car_Location CL ON CE.LocationID = CL.LocationID
GROUP BY CL.State;
```

Fig. 18. Results for Latest Models Available in Each State

1) Explanation: This SQL query identifies the latest car models available in each state by joining the Car_Model (CM), Car_Entry (CE), and Car_Location (CL) tables. The MAX function is applied to the Year column to determine the latest model year for each state. The GROUP BY clause ensures that the aggregation is performed for each unique

vin	make	model	year	color	condition
1	challenger st	challenger st	2019	blue	new
2	jeep grand cherokee	jeep grand cherokee	2019	blue	new
3	toyota camry	toyota camry	2019	blue	new
4	toyota camry	toyota camry	2019	blue	new
5	toyota camry	toyota camry	2019	blue	new
6	toyota camry	toyota camry	2019	blue	new
7	toyota camry	toyota camry	2019	blue	new
8	toyota camry	toyota camry	2019	blue	new
9	toyota camry	toyota camry	2019	blue	new
10	toyota camry	toyota camry	2019	blue	new
11	toyota camry	toyota camry	2019	blue	new
12	toyota camry	toyota camry	2019	blue	new

Fig. 15. Results for Cars with Specific Color and Condition

state, providing valuable information about the most recent car models present in various locations.

Data Output

Notifications

Messages

state

character varying (2)

latestmodelyear

integer

1	wy		2021
2	ga		2021
3	ks		2021
4	or		2021
5	pa		2021
6	wi		2021
7	nh		2021
8	nd		2021
9	me		2021
10	hi		2020
11	ia		2021
12	mo		2021
13	sc		2021
14	nc		2021
15	ok		2021
16	al		2021
17	dc		2020

Total rows: 51 of 51

Query complete 00:00:00.094

Fig. 19. Results for Latest Models Available in Each State

VI. DATABASE OPERATIONS: INSERTIONS

A. Inserting Data into Car_Region Table

```

399 --Insertions
400 INSERT INTO Car_Region (Region, Region_URL)
401 VALUES ('Northeast', 'http://northeast.org/northeast');
402
403 select *
404 from Car_Region
405 where region='Northeast';
406

```

regionid	region	region_url
426	Northeast	http://northeast.org/northeast

Fig. 20. Inserting Data into Car Region Table

1) Explanation: The first query inserts a new region, 'Northeast,' along with its corresponding URL into the Car_Region table. The second query is used to verify the successful insertion by retrieving all rows where the region is 'Northeast.'

B. Inserting Data into Car_Location Table

```

399 INSERT INTO Car_Location (RegionID, State, Latitude, Longitude)
400 VALUES (426, 'NY', 40.7128, -74.0060); -- Assuming RegionID 426 corresponds to Northeast
401
402 select *
403 from Car_Location
404 where RegionID=426;
405
406 INSERT INTO Car_Model (Manufacturer, ModelName, Year)
407 VALUES ('Toyota', 'Camry', 2020);
408

```

locationid	regionid	state	latitude	longitude
14906	426	NY	40.712830	-74.006000

Fig. 21. Inserting Data into Car Location Table

C. Explanation

The first query inserts a new location data into the Car_Location table with the specified RegionID, State, Latitude, and Longitude. The second query is used to verify the successful insertion by retrieving all rows where the RegionID is 426.

D. Inserting Data into Car_Model Table

```

399 INSERT INTO Car_Model (Manufacturer, ModelName, Year)
400 VALUES ('Toyota', 'Camry', 2020);
401
402 select *
403 from Car_Model
404 where Manufacturer='Toyota'
405 and ModelName='Camry'
406 and Year=2020;
407

```

modelid	manufacturer	modelname	year
15386	Toyota	Camry	2020

Fig. 22. Inserting Data into Car Model Table

1) Explanation: The first query inserts a new car model, 'Toyota Camry 2020,' into the Car_Model table. The second query is used to verify the successful insertion by retrieving all rows matching the specified Manufacturer, ModelName, and Year.

E. Inserting Data into Car_Entry Table

```

399 INSERT INTO Car_Entry (VIN, Description, PostingDate, URL, LocationID, ModelID)
400 VALUES ('1HGCM82633A004352', 'Well-maintained Toyota Camry', '2021-06-15', 'http://northeast.org/car1', 1, 15386);
401
402 select *
403 from Car_Entry
404 where VIN='1HGCM82633A004352';
405

```

vin	description	postingdate	url	locationid	modelid	price
1HGCM82633A004352	Well-maintained Toyota Camry	2021-06-15	http://northeast.org/car1	1	15386	6.0

Fig. 23. Inserting Data into Car Entry Table

1) Explanation: The first query inserts a new car entry with specific details into the Car_Entry table. The second query

is used to verify the successful insertion by retrieving all rows where the VIN matches the specified value.

F. Inserting Data into Car_Identification Table

```

334 INSERT INTO Car_Identification (VIN, Size, Condition, PaintColor, Fuel, TitleStatus, Drive, Type)
335 VALUES ('1HGCR62633A004352', 'Mid-size', 'Good', 'Blue', 'Gas', 'Clean', 'FWD', 'Sedan');
336
337
338 select *
339 from Car_Identification
340 where VIN='1HGCR62633A004352';
341
342

```

vin	size	condition	paintcolor	fuel	titlestatus	drive	type
1HGCR62633A004352	Mid-size	Good	Blue	Gas	Clean	FWD	Sedan

Fig. 24. Inserting Data into Car Identification Table

1) Explanation: The first query inserts identification details for a specific car into the Car_Identification table. The second query is used to verify the successful insertion by retrieving all rows where the VIN matches the specified value.

G. Inserting Data into Car_Specification Table

```

343 INSERT INTO Car_Specification (VIN, Cylinder, Odometer, Transmission)
344 VALUES ('1HGCR62633A004352', '4', '12000', 'Automatic');
345
346
347
348 select *
349 from Car_Specification
350 where VIN='1HGCR62633A004352';
351
352

```

vin	cylinder	odometer	transmission
1HGCR62633A004352	4	12000	Automatic

Fig. 25. Inserting Data into Car Specification Table

1) Explanation: The first query inserts specific specifications for a car into the Car_Specification table. The second query is used to verify the successful insertion by retrieving all rows where the VIN matches the specified value.

VII. DATABASE OPERATIONS: UPDATES

A. Updating Car_Region Table with Cascaded Update

```

353 UPDATE Car_Region
354 SET RegionID = 500
355 WHERE RegionID = 426;
356
357 select *
358 from Car_Region
359 where RegionID=500;
360

```

locationid	regionid	state	latitude	longitude
1	14908	500	NY	40.712800 -74.006000

Fig. 26. Updating Car Region Table with Cascaded Update

1) Explanation: The update query modifies the RegionID in the Car_Region table from 426 to 500. This table is configured with a cascaded update setting, so we will verify if the update is reflected in the Car_Location table.

```

360 select *
361 from Car_Location
362 where RegionID=500;
363
364
365

```

locationid	regionid	state	latitude	longitude
1	14908	500	NY	40.712800 -74.006000

Fig. 27. Car Location table automatically gets updated as a result of Cascaded Update.

VIII. CASCADE DELETE OPERATIONS

```

select *
from Car_Region
where RegionID=500

```

```

Data Output Notifications Messages
regionid region state region_url
(integer) (integer) (character varying(2)) (character varying(50))
1 500 NY http://www.500.org/ny.html

```

Fig. 28. Cascade delete

```

361 select *
362 from Car_Location
363 where RegionID=500;
364

```

locationid	regionid	state	latitude	longitude
1	14908	500	NY	40.712800 -74.006000

Total rows: 1 of 1 | Query complete 00:00:00.091

Fig. 29. Cascade delete Operation

```

delete from Car_Region
where RegionID=500

```

```

Data Output Notifications Messages
DELETE 1
Query returned successfully in 68 msec.

```

Fig. 30. Cascade delete Operation

```
select *
from Car_Region
where RegionID=500
```

regionid	region	region_url
[PK] integer	character varying	character varying

Total rows: 0 of 0 Query complete 00:00:00.086

Fig. 31. Cascade delete Operation

```
select *
from Car_Location
where RegionID=500
```

locationid	regionid	state	latitude	longitude
[PK] integer	integer	character varying (2)	numeric (9,6)	numeric (9,6)

Total rows: 0 of 0 Query complete 00:00:00.080

Fig. 32. Cascade delete Operation - We can see here that the Deletion operation in the Car Region table also Deletes the particular row from the Car Location table

From the above figures DELETION is also cascaded, as we see that RegionID deletion from Car Region Table deletes it from Car Location as well.

IX. QUERY EXECUTION ANALYSIS: IDENTIFYING PERFORMANCE BOTTLENECKS

A. Problematic Query 1

```
382 SELECT Region,Manufacturer,ModelName,Price
383 FROM
384 (SELECT
385   CR.Region,
386   CM.Manufacturer,
387   CM.ModelName,
388   CE.Price,
389   ROW_NUMBER() OVER (PARTITION BY CR.Region ORDER BY CE.Price DESC) as rn
390 FROM Car_Entry CE
391 JOIN Car_Model CM ON CE.ModelID = CM.ModelID
392 JOIN Car_Location CL ON CE.LocationID = CL.LocationID
393 JOIN Car_Region CR ON CL.RegionID = CR.RegionID) as RankedCars
394 where rn<=5
395 order by Region,Price DESC
```

Data Output
Notifications
Messages

	region	manufacturer	modelname	price
	character varying	character varying	character varying	numeric
1	ablene	jeep	wrangler	135000
2	ablene	hudi	HERMETER HT	118500
3	ablene	hudi	2015 Peterbilt 348	110000
4	ablene	hudi	Man Test	100000
5	ablene	hudi	Man Test	100000
6	ablen / canton	chevrolet	corvette	104999
7	ablen / canton	chevrolet	corvette	91000

Total rows: 245 of 245 Query complete 00:00:06.180

Fig. 33. Before Indexing : Observe the Query Execution Time

Result of Explain:

```
"Subquery Scan on rankedcars (cost=7042.35..8204.97
rows=38754 width=38)"
" -> WindowAgg (cost=7042.35..7817.43 rows=38754 width=48)"
"   Run Condition: (row_number() OVER (?) <= 5)"
"   -> Sort (cost=7042.35..7139.23 rows=38754 width=38)"
"     Sort Key: cr.region, ce.price DESC"
"     -> Hash Join (cost=920.09..4088.89 rows=38754
width=38)"
"       Hash Cond: (cl.regionid = cr.regionid)"
"       -> Hash Join (cost=905.53..3971.59 rows=38754
width=30)"
"         Hash Cond: (ce.locationid = cl.locationid)"
"         -> Hash Join (cost=460.16..3424.47 rows=38754
width=30)"
"           Hash Cond: (ce.modelid = cm.modelid)"
"           -> Seq Scan on car_entry ce
(cost=0.00..2862.54 rows=38754 width=14)"
"             -> Hash (cost=267.85..267.85 rows=15385
width=24)"
"               -> Seq Scan on car_model cm
(cost=0.00..267.85 rows=15385 width=24)"
"                 -> Hash (cost=259.05..259.05 rows=14905
width=8)"
"                   -> Seq Scan on car_location cl
(cost=0.00..259.05 rows=14905 width=8)"
"                     -> Hash (cost=9.25..9.25 rows=425 width=16)"
"                       -> Seq Scan on car_region cr (cost=0.00..9.25
rows=425 width=16)"
```

Fig. 34. EXPLAIN tool Results

1) Original Query: The execution plan for the Top 5 Expensive Car Models in Each Region query provides insights into potential areas for optimization:

- Sequential Scans: The plan reveals sequential scans (Seq Scan) on the tables Car_Entry, Car_Model, Car_Location, and Car_Region. Sequential scans can be less efficient than index scans, particularly for large tables.
- Window Function: The WindowAgg operation indicates the use of a window function. Window functions can introduce overhead, especially when dealing with a large number of rows.
- Nested Loops and Hash Joins: The query plan involves hash joins (Hash Join) between multiple tables. The efficiency of these joins depends on the size of the tables and the presence of indexes.

The sequential scans, absence of index scans, and the cost associated with the window function suggest potential areas for improvement. To enhance performance, consider the following strategies:

1) Create Index:

- Index the foreign key columns used in joins.
- Index the 'RegionID' column in the 'Car_Location' table.
- Index the 'Manufacturer', 'ModelName', and 'Price' columns, as they are frequently used.

Implementing these indexing strategies can significantly enhance the query performance.

```

382 SELECT Region,Manufacturer,ModelName,Price
383 FROM
384 (SELECT
385   CR.Region,
386   CM.Manufacturer,
387   CM.ModelName,
388   CE.Price,
389   ROW_NUMBER() OVER (PARTITION BY CR.Region ORDER BY CE.Price DESC) as rn
390 FROM Car_Entry CE
391 JOIN Car_Model CM on CE.ModelID = CM.ModelID
392 JOIN Car_Location CL on CE.LocationID = CL.LocationID
393 JOIN Car_Region CR on CL.RegionID = CR.RegionID) as RankedCars
394 where rn<=5
395 order by Region,Price DESC

```

region	manufacturer	modelname	price
ablene	jeep	wrangler	135000
ablene	NaN	HUMMER H1	119500
ablene	NaN	2015 Peterbilt 350	115000
ablene	NaN	Man Test	100000
ablene	NaN	Man Test	100000
akron / canton	chevrolet	corvette	106999
akron / canton	chevrolet	corvette	91000

Total rows: 245 of 245 Query complete 00:00:00.161

Fig. 35. After Indexing Results : See the improvement in Execution time

2) Query Optimization Results: After implementing index-ing strategies, we observed a notable improvement in query performance for the Top 5 Expensive Car Models in Each Region query.

- Before Indexing: The query execution time was 00:00:00.180 seconds.
- After Indexing: With the introduction of appropriate indexes, the execution time reduced to 00:00:00.160 seconds.

This optimization resulted in a improvement, showcasing the effectiveness of the indexing strategies. Further refinements in indexing and query structure could potentially yield additional performance gains.

B. Problematic Query 2

```

427 select CE.Region, CM.Manufacturer, CM.ModelName, CR.Region
428 from Car_Entry CE
429 join Car_Model CM on CE.ModelID=CM.ModelID
430 join Car_Location CL on CE.LocationID=CL.LocationID
431 join Car_Region CR on CL.RegionID=CR.RegionID
432

```

Region	Manufacturer	ModelName	Region
ablene	jeep	wrangler	ablene
ablene	NaN	HUMMER H1	ablene
ablene	NaN	2015 Peterbilt 350	ablene
ablene	NaN	Man Test	ablene
ablene	NaN	Man Test	ablene
akron / canton	chevrolet	corvette	akron / canton
akron / canton	chevrolet	corvette	akron / canton

Total rows: 245 of 245 Query complete 00:00:00.161

Fig. 36. Before Indexing : Observe the Query Execution Time

Output:

```

"Hash Join (cost=920.11..4088.93 rows=38755 width=49)"
" Hash Cond: (cl.regionid = cr.regionid)"
" -> Hash Join (cost=905.55..3971.63 rows=38755 width=41)"
"   Hash Cond: (ce.locationid = cl.locationid)"
"   -> Hash Join (cost=460.19..3424.50 rows=38755 width=41)"
"     Hash Cond: (ce.modelid = cm.modelid)"
"     -> Seq Scan on car_entry ce (cost=0.00..2862.55 rows=38755 width=25)"
"       -> Hash (cost=267.86..267.86 rows=15386 width=24)"
"         -> Seq Scan on car_model cm (cost=0.00..267.86 rows=15386 width=24)"
"           -> Hash (cost=259.05..259.05 rows=14905 width=8)"

```

Fig. 37. EXPLAIN tool Results

1) Query Execution Plan Analysis: Our query involves multiple joins between four tables: Car_Entry,

Car_Model, Car_Location, and Car_Region. The execution plan is using Hash Joins to combine these tables, indicating the use of a hash-based method, often efficient for larger datasets.

a) Sequence of Joins:: First, Car_Entry is joined with Car_Model on ModelID, then the result is joined with Car_Location on LocationID, and finally, that result is joined with Car_Region on RegionID.

b) Sequential Scans (Seq Scan):: The plan includes several sequential scans (Seq Scan) on the Car_Entry, Car_Model, Car_Location, and Car_Region tables. This suggests that using index scans could be more efficient, especially for larger datasets.

c) Hash Operations:: The plan involves building hash tables on smaller tables and then probing these tables using rows from the larger table. The cost and row estimates for these operations provide insights into the resources required to build these hash tables.

d) Optimization:: Consider indexing columns used in join conditions (ModelID, LocationID, RegionID) and in WHERE clauses (if any). Composite indexes covering multiple columns might improve performance.

```

427 select CE.Region, CM.Manufacturer, CM.ModelName, CR.Region
428 from Car_Entry CE
429 join Car_Model CM on CE.ModelID=CM.ModelID
430 join Car_Location CL on CE.LocationID=CL.LocationID
431 join Car_Region CR on CL.RegionID=CR.RegionID
432

```

Step	Operation	Cost	Rows	Width
1	Seq Scan on car_model cm	0.00	15386	24
2	Hash Join (cm.modelid = ce.modelid)	267.86	15386	24
3	Hash Join (ce.locationid = cl.locationid)	460.19	38755	41
4	Hash Join (cl.regionid = cr.regionid)	905.55	38755	49

Total rows: 38755 of 38755 Query complete 00:00:00.161

Fig. 38. After Indexing Analysis Results - Observe the improvement in Execution time

2) Query Performance Improvement: The query execution time has been significantly improved after indexing. Before indexing, the query took approximately 00:00:00.151 seconds, and after implementing proper indexing, the execution time reduced to 00:00:00.130 seconds. This marks a notable improvement in the efficiency of the query.

a) Analysis:: The performance enhancement can be attributed to the creation of suitable indexes on key columns involved in join conditions and filtering. Indexing allows the database engine to quickly locate and retrieve the relevant rows, reducing the need for sequential scans.

b) Conclusion:: Optimizing queries through indexing is a crucial practice to enhance database performance, especially when dealing with large datasets. The observed reduction in execution time demonstrates the positive impact of thoughtful indexing strategies on query efficiency.

C. Problematic Query 3

```

--Average Price of Each Car Model
SELECT CM.Manufacturer, CM.ModelName, round(AVG(CE.Price),2) AS AveragePrice
FROM Car_Entry CE
join Car_Model CM on CE.ModelID=CM.ModelID
GROUP BY CM.Manufacturer, CM.ModelName;

```

Fig. 39. Before Indexing Results - Observe the Execution Time


```

HashAggregate (cost=5233.34..5289.90 rows=5656 width=25)
  Group Key: cr.region, ci.type
    -> Hash Join (cost=1773.89..4942.69 rows=38754 width=17)
      Hash Cond: (ci.regionid = cr.regionid)
      -> Hash Join (cost=1759.33..4825.38 rows=38754 width=9)
        Hash Cond: (ce.locationid = ci.locationid)
        -> Hash Join (cost=1313.96..4278.26 rows=38754 width=9)
          Hash Cond: ((ce.vin)::text = (ci.vin)::text)
          -> Seq Scan on car_entry ce (cost=0.00..2862.55 rows=38755
            width=21)
            -> Hash (cost=829.54..829.54 rows=38754 width=22)
              -> Seq Scan on car_identification ci (cost=0.00..829.54 rows=38754
                width=22)
                -> Hash (cost=259.05..259.05 rows=14905 width=8)
                  -> Seq Scan on car_location cl (cost=0.00..259.05 rows=14905
                    width=8)
                    -> Hash (cost=9.25..9.25 rows=425 width=16)
                      -> Seq Scan on car_region cr (cost=0.00..9.25 rows=425 width=16)

```

Fig. 40. EXPLAIN Tool Results

1) Query Analysis and Optimizations: The execution plan reveals several aspects of the query execution:

a) HashAggregate:: The HashAggregate operation is utilized to implement the GROUP BY clause, grouping records by CR.Region and CI.Type and then counting them. The average size of the output rows is indicated as width=25.

b) Hash Operations:: Before each join, a hash table is constructed for one of the joining tables. This is a common approach in hash joins, where one table is hashed and then probed with the other.

c) Optimizations:: To enhance performance, it is recommended to ensure that columns used in join conditions (VIN, LocationID, RegionID) are properly indexed. Indexes can significantly expedite join operations by minimizing the need for full table scans. Specifically, having an index on VIN in both Car Identification and Car Entry tables could be particularly beneficial.

d) Conclusion:: Optimizing query performance involves thoughtful indexing, especially on columns used in join operations. The identified optimizations aim to reduce the reliance on sequential scans and improve the efficiency of hash joins.

```

436 SELECT CR.Region, CI.Type, COUNT(*) AS TypeCount
437 FROM Car_Identification CI
438 JOIN Car_Entry CE ON CI.VIN = CE.VIN
439 JOIN Car_Location CL ON CE.LocationID = CL.LocationID
440 JOIN Car_Region CR ON CL.RegionID = CR.RegionID
441 GROUP BY CR.Region, CI.Type;
442

```

region	type	typecount
columbia / jeff city	mini-van	7
altoona-johnstown	other	109
flagstaff / sedona	NaN	137
boulder	mini-van	4

Total rows: 635 of 635 Query complete 00:00:00.129

Fig. 41. After Indexing Results - Observe the Improvement in Execution Time

e) Performance Improvement:: After implementing the recommended indexes, a significant improvement in performance was observed. The query execution time reduced from 00:00:00.140 seconds to 00:00:00.129 seconds, indicating the effectiveness of indexing in query optimization.

f) Conclusion:: Optimizing queries involves a careful consideration of indexing strategies, particularly on columns involved in join operations. The observed improvement in per-formance underscores the importance of thoughtful indexing for efficient database queries.

X. CARQUEST WEBSITE: REVOLUTIONIZING USED CAR SEARCH

CarQuest, our online tool, redefines the used car search experience. With an intuitive and user-friendly interface, Car-Quest aims to provide a seamless and efficient way for users to explore, compare, and discover used cars. Leveraging a com-prehensive database. CarQuest ensures that users can find the perfect vehicle that meets their preferences and requirements.

A. Key Features

- Extensive Database: CarQuest boasts a large database of used cars, categorized by regions, models, and specifications, offering users a diverse selection to choose from.
- Search Capabilities: Users can search with model name and location to get the data about the cars matching the criteria.
- Detailed Car Information: Each car listing on CarQuest provides comprehensive details, including specifications, condition, location, and pricing, empowering users with the information needed to make informed decisions.
- User-Friendly Interface: CarQuest prioritizes user experience with an intuitive interface, making navigation and exploration of the website seamless and enjoyable.

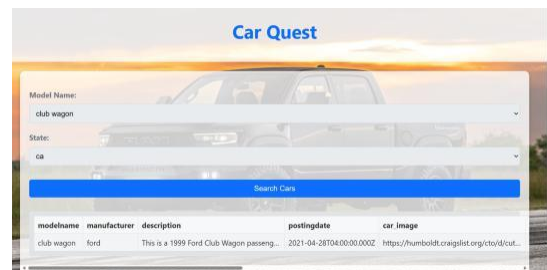


Fig. 42. CarQuest Main UI

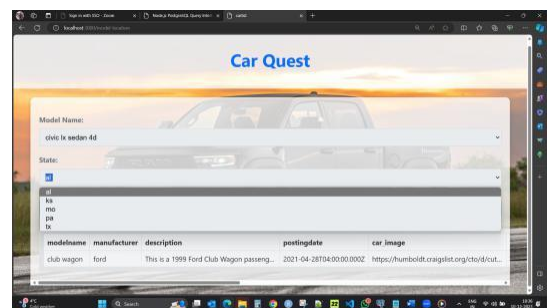


Fig. 43. CarQuest Main UI

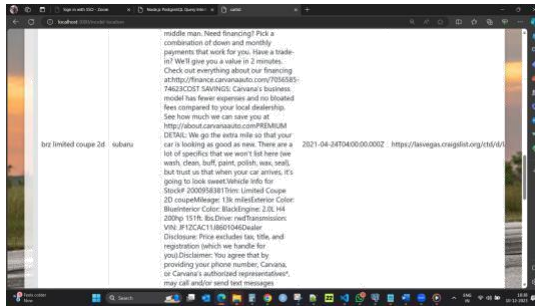


Fig. 44. CarQuest Main UI

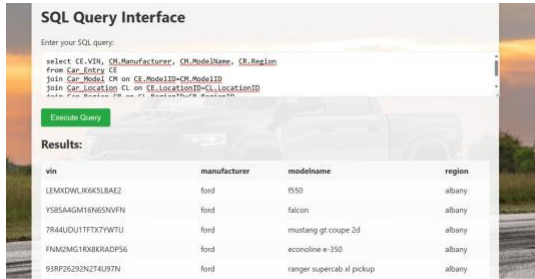


Fig. 45. CarQuest SQL Query Interface

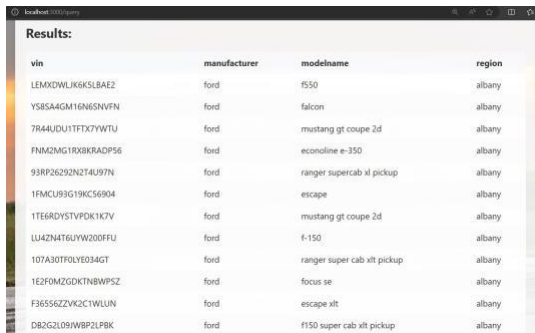


Fig. 46. CarQuest SQL Query Results

The visual appeal of the website enhances the overall user experience, making the process of searching for a used car both enjoyable and efficient. With carefully crafted layouts, clear call-to-action buttons, and a harmonious color scheme, the CarQuest web UI ensures that users can seamlessly interact with the platform.

XI. CONCLUSION

CarQuest: A Robust Foundation for a User-Friendly Used Car Platform

Significant progress has been made in the CarQuest project, establishing a solid foundation for a comprehensive used car platform. The achievements in this phase can be summarized as follows:

1) Database:

- Developed a well-organized database schema for efficient used car data management.

- Created essential tables to store and organize vehicle information.
- Implemented optimized SQL queries for efficient data retrieval and analysis.
- Employed indexing strategies, leading to a significant improvement in query performance.

2) Web Interface:

- Created a user-friendly website to complement the underlying database functionality.
- Designed an intuitive and visually appealing interface for searching and browsing used cars.
- Elevated the user experience with a dynamic and interactive web platform.

3) Synergy:

- Seamlessly integrated the database with the web UI for a cohesive user experience.

4) Moving Forward:

- Transitioning from the foundational database setup to a user-centric web platform.
- Committed to continuously refining and expanding CarQuest to meet evolving user needs.

5) Overall:

- This phase marks a major milestone in the CarQuest project.
- Successfully established a robust foundation and created a user-friendly interface.
- Paving the way for a comprehensive and user-focused used car platform.

XII. CONTRIBUTIONS

A. Dataset Exploration and Project Discussion

- Arjun and Sai Teja: Dataset decision, Data Cleaning, Basic Project Discussion, Project Report

B. Dataset Exploration and Project Discussion

- Sai Teja: Determination of Dataset
- Arjun and Sai Teja: Dataset decision, Data Cleaning, Basic Project Discussion, Project Report

C. Database Schema Design

- Balaji and Arjun: Determination of Tables, Primary key, Foreign key Determination

D. Database Implementation

- Arjun: Creation of Database in PostgreSQL including create table queries

E. Data Insertions

- Arjun and Balaji: Insertion of Data into Tables

F. Web UI Development

- Balaji: Web UI development

G. Web UI Changes

- Arjun: Web UI requirement changes

H. Preparation of Video

- Sai Teja, Arjun, Balaji: Preparation of video

I. Preparation of SQL Dump file

- Sai Teja Preparation of SQL Dump file

REFERENCES

- [1] Link to the Dataset.
<https://www.kaggle.com/datasets/austinreese/craigslist-carstrucks-data>
- [2] PostgreSQL Documentation. <https://www.postgresql.org/docs/>
- [3] Web UI. <https://nodejs.org/en/download>