Vehicle Rental Management System

*Note: Sub-titles are not captured in Xplore and should not be used

1st Aravind Kamalay

Computer Science and Engineering

University at Buffalo

Buffalo, USA

akamalay@buffalo.edu

2nd Rishika Reddy Thumma

Computer Science and Engineering

University at Buffalo

Buffalo, USA

rthumma@buffalo.edu

Abstract—In today's dynamic digital landscape, the traditional reliance on spreadsheet-based methods for managing automobile rental operations has increasingly proven to be inadequate. This project proposes a robust Relational Database Management System (RDBMS), developed using PostgreSQL, specifically tailored for modern automobile rental management. The system utilises a carefully designed schema, decomposed into 12 BCNF-compliant relations, to ensure high levels of data integrity, minimise redundancy, and optimise query performance. By effectively managing diverse data—from vehicle availability to customer preferences—this solution significantly enhances operational efficiency and supports informed, data-driven decision-making. Ultimately, the proposed system meets the critical need for a

secure, scalable, and internet-based approach in the rapidly evolving automobile rental industry.

Index Terms—component, formatting, style, styling, insert

I. Introduction

The internet has revolutionized numerous industries ranging from education and healthcare to transportation bringing about significant improvements in operational efficiency and customer experience. Traditionally, automobile rental operations relied on manual methods such as paper-based records and spreadsheets, which were not only time-consuming and errorprone but also lacked real-time accessibility and scalability.

With the advent of the internet, the rental process has undergone a dramatic transformation. Customers can now easily browse available vehicles, compare rental rates, and make reservations online from the comfort of their homes. This digital transition has streamlined operations, reduced wait times, and significantly enhanced customer satisfaction by providing immediate access to up-to-date vehicle availability and pricing information.

However, the shift to an online platform introduces its own set of challenges. An effective automobile rental management system must integrate multiple entities—such as vehicles, registrations, customer details, payment processing, and rental transactions—each interrelated and subject to complex dependencies. For instance, a single customer may rent different vehicles over time, with each transaction linked to specific vehicle details, payment records, and operational data like maintenance and location information.

Identify applicable funding agency here. If none, delete this.

These complex interrelationships demand a robust database management system that not only stores and retrieves data efficiently but also maintains data integrity without redundancy. The system must be secure to protect sensitive customer and transaction data, scalable to handle peak rental periods, and reliable to ensure uninterrupted service. In this context, our project leverages a PostgreSQL-based RDBMS with a schema decomposed into 12 BCNF-compliant relations, providing a modern, efficient, and secure solution for managing automobile rental operations.

II. Problem Statement

A. Need for RDBMS over Microsoft Excel

Excel is widely used for basic data analysis and record keeping; however, when it comes to managing a complex system like an Automobile Rental Management System, it becomes evident that Excel is not the ideal tool. The limitations of Excel become apparent in several aspects: leftmargin=*, label=–

- Efficient Storage: Databases use normalization techniques to reduce redundancy and optimize storage. In contrast, Excel's flat file structure often leads to duplicate data and inefficient storage, especially as the dataset grows larger.
- Handling Large Volumes of Data: Excel imposes strict limits on rows and columns, and its performance deteriorates noticeably as data volume increases. In contrast, a dedicated relational database can efficiently handle millions or even billions of records, which is essential for managing the extensive data generated in vehicle rental operations.
- Scalability: As an automobile rental business grows, the system must be scalable to handle increasing volumes of data and more complex operations. Databases are inherently scalable and can be

optimized to handle large datasets and high transaction rates. Excel's fixed structure, however, limits its ability to scale effectively.

- Maintaining Data Integrity: In a complex rental management system, ensuring the accuracy and consistency of data is paramount. Databases enforce integrity through primary keys, foreign keys, and other constraints, ensuring that the relationships among vehicles, customers, transactions, and other entities remain consistent. Excel lacks these robust mechanisms, which often leads to data errors and redundancy.
- Supporting Concurrent Access: In a real-world environment, multiple users (such as customer service executives, fleet managers, and administrators) need to access and update data concurrently. A relational database supports simultaneous transactions without compromising data consistency, whereas Excel is not designed for multi-user access and may result in data corruption if several users edit the same file concurrently.
- Complex Query Capabilities: An efficient rental management system requires complex queries involving multiple tables, conditional logic, and aggregations to generate actionable insights—such as analyzing rental trends and monitoring vehicle usage. Databases are designed to perform such complex queries efficiently, a task at which Excel is not adept.

B. Potentiality of Project

In today's digital era, traditional methods such as manual record keeping or using Excel for managing automobile rental operations have proven to be inefficient, error-prone, and incapable of handling large volumes of data. Our project addresses these challenges by proposing a robust, PostgreSQLbased RDBMS that meticulously manages every facet of vehicle rentals, including vehicle registrations, fleet tracking, rental transactions, customer details, maintenance records, and payment processing. By enforcing data integrity through primary and foreign key constraints, the system minimizes redundancy and ensures that the relationships among different entities remain consistent. Moreover, the automation of routine tasks significantly reduces manual workload and human errors, thereby improving overall operational efficiency.

The system further enhances user experience by providing real-time updates on vehicle availability and a seamless online booking process, while its advanced querying capabilities offer critical insights into rental trends and vehicle utilization, aiding in effective decision-making. Additionally, the database is designed to be highly scalable, capable of accommodating increasing data and transaction volumes, and it incorporates robust security measures to protect sensitive information, ensuring compliance with regulatory standards. Overall, this comprehensive solution not only overcomes the limitations of traditional spreadsheet-based systems but also lays a solid foundation for improved efficiency, enhanced customer satisfaction, and sustained business growth in the automobile rental industry.

C. Target User

The proposed vehicle rental management system is designed to serve a diverse set of users, each with distinct requirements and responsibilities. Primary users include the customer service representatives who manage rental bookings, respond to customer queries, and coordinate reservation schedules. These users benefit from real-time access to vehicle availability and rental history, allowing them to provide prompt and accurate information to customers. Additionally, fleet managers form a critical user group; they rely on the system to monitor vehicle utilization, schedule maintenance, and optimize fleet allocation, thereby ensuring that the rental operations run smoothly and efficiently.

Apart from the internal staff, the system is also tailored to enhance the experience of external users—namely, the customers who book vehicles online. These users enjoy a user-friendly interface that simplifies the reservation process, offers real-time updates, and provides secure payment options, ultimately leading to higher satisfaction and repeat business. Furthermore, the system supports decision-makers such as administrators and data analysts who leverage the database's advanced querying capabilities to extract actionable insights on rental trends, customer behavior, and operational performance. This comprehensive approach ensures that the system meets the needs of all stakeholders, thereby contributing to a more efficient, reliable, and customer-centric vehicle rental operation.

D. Real Life Scenario

Consider the case of Mr. Sharma, an industrious professional based in Mumbai, who frequently requires a rental vehicle for both business trips and personal vacations. In the traditional scenario, Mr. Sharma would have to contact multiple rental agencies via phone or visit their offices in person, often encountering delays, inconsistent information on vehicle availability, and cumbersome manual paperwork. Such a process not only consumed valuable time but also led to booking errors and considerable inconvenience.

With the proposed Automobile Rental Management System, Mr. Sharma benefits from a streamlined online portal that offers real-time information on vehicle availability, rental rates, and detailed specifications. He can effortlessly reserve a vehicle with just a few clicks from the comfort of his home or office. Once a booking is confirmed, the system automatically generates a unique rental agreement and sends immediate confirmation via email or SMS, ensuring transparency and accuracy in every transaction.

From the service provider's perspective, this automated system significantly enhances operational efficiency by enabling real-time tracking of vehicle usage and maintenance schedules. Fleet managers can optimize vehicle allocation and reduce downtime, while administrators and data analysts gain valuable insights into rental trends and customer behavior through advanced query capabilities. Overall, this comprehensive solution not only elevates the customer experience but also fosters improved operational coordination and sustained business growth.

III. Entity-Relationship Diagram and Relational Schema

The architecture of a database system is best understood through an entity-relationship (ER) diagram, which acts as a blueprint for designing and implementing the database. In an ER diagram, key entities—such as vehicles, registrations, customers, rental transactions, and payments—are depicted as rectangles, while their attributes (the properties or characteristics) are shown as ovals attached to these rectangles. Relationships between entities are represented by lines connecting them, often annotated with details about the nature of the relationship (e.g., one-to-one, one-to-many).

For a Vehicle Rental Management System, the ER diagram is indispensable. It not only captures the logical structure of the database but also clarifies the interactions between various components. This visualization aids in requirement analysis and facilitates discussions among developers, stakeholders, and domain experts. Furthermore, by clearly specifying primary keys and foreign keys, the ER diagram helps in ensuring data integrity and consistency throughout the system.

Below is the ER diagram for the Vehicle Rental Management System, which effectively illustrates the overall design and interdependencies of the system's core entities, forming the foundation for a robust and scalable database solution.

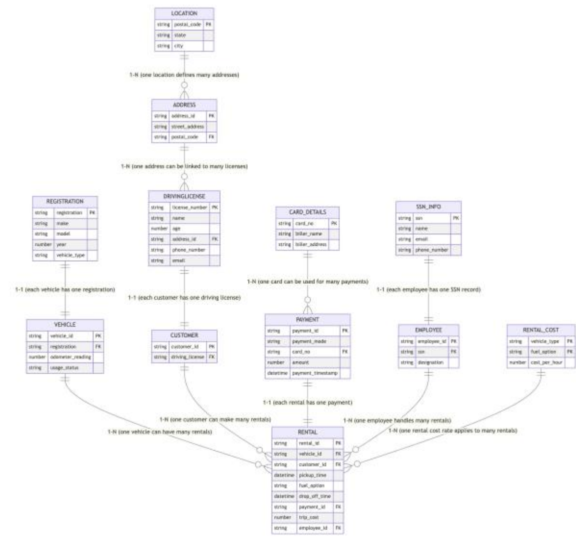


Fig. 1. ER Diagram for the Vehicle Rental Management System.

A. list of tables and their attributes

1) rental cost

- *attributes:*
  - vehicle type: string denoting the type of vehicle (e.g., SUV, Sedan)
  - fuel option: string indicating the fuel option (e.g., Petrol, Diesel)
  - cost per hour: numeric value representing hourly rental cost
- *primary key:* (vehicle type, fuel option)
- *foreign key:* none

2) registration

- *attributes:*
  - registration: unique string (e.g., license plate)
  - make: vehicle manufacturer or brand
  - model: specific model name of the vehicle
  - year: manufacturing year (integer)
  - vehicle type: string denoting the category of the vehicle
- *primary key:* registration
- *foreign key:* none

3) vehicle

- *attributes:*
  - vehicle id: unique identifier (string or integer)
  - registration: references registration(registration)
  - odometer reading: numeric value
  - usage status: string (e.g., “Available,” “Rented”)

- *primary key:* vehicle id
- *foreign key:* registration

4) location
- *attributes:*
  - postal code: unique string or integer
  - state: string indicating state name
  - city: string indicating city name
- *primary key:* postal code
- *foreign key:* none

5) address
- *attributes:*
  - address _id: unique identifier
  - street address: string holding street or house address
  - postal code: references location(postal code)
- *primary key:* address id
- *foreign key:* postal code

6) drivinglicense
- *attributes:*
  - license number: unique string
  - name: license holder's name
  - age: integer
  - address _id: references address(address id)
  - phone number: string or integer
  - email: string
- *primary key:* license number
- *foreign key:* address id

7) customer
- *attributes:*
  - customer id: unique identifier
  - driving license: references driving license (license number)
- *primary key:* customer id
- *foreign key:* driving license

8) ssn info
- *attributes:*
  - ssn: unique string
  - name: ssn holder's name
  - email: email address
  - phone number: phone number
- *primary key:* ssn
- *foreign key:* none

9) employee
- *attributes:*
  - employee _id: unique identifier
  - ssn: references ssn info(ssn)
  - designation: string indicating role (e.g., "Manager")
- *primary key:* employee id
- *foreign key:* ssn

10) card details
- *attributes:*
  - card no: unique identifier for the payment card
  - biller name: name of the card holder
  - biller address: address details for billing
- *primary key:* card no
- *foreign key:* none

11) payment
- *attributes:*
  - payment id: unique identifier
  - payment mode: paymentmethod (e.g., "Credit Card," "Debit Card")
  - card no: references card details(card no)
  - amount: numeric value for the payment
  - payment timestamp: timestamp indicating payment time
- *primary key:* payment id
- *foreign key:* card no

12) rental
- *attributes:*
  - rental id: unique identifier for each rental
  - vehicle id: references vehicle(vehicle id)
  - customer id: references customer(customer id)
  - pickup time: timestamp for when the vehicle is picked up
  - fuel option: string indicating chosen fuel option
  - drop off time: timestamp for when the vehicle is returned
  - payment id: references payment(payment id)
  - trip cost: numeric field for total cost
  - employee _id: references employee(employee id)
- *primary key:* rental id
- *foreign key:* vehicle id, customer id, payment _id, employee id

B. Cardinalities of the Relations

The relationships between the various tables in our Vehicle Rental Management System are defined as follows:

- Rental Cost to Rental (1:N): One cost record (for a specific vehicle type and fuel option) may apply to many rental transactions.

- Registration to Vehicle (1:1): Each vehicle is associated with a unique registration, ensuring no duplication.
- Vehicle to Rental (1:N): A single vehicle can be rented out multiple times over different transactions.
- Location to Address (1:N): One location (postal code) can encompass multiple addresses.
- Address to DrivingLicense (1:N): Multiple driving licenses can be associated with the same address.
- DrivingLicense to Customer (1:1): Each customer is linked to one unique driving license.
- SSN Info to Employee (1:1): Every employee has one unique SSN record.
- Card _Details to Payment (1:N): A single card can be used for many payments.
- Customer to Rental (1:N): One customer may make several rental transactions.
- Payment to Rental (1:1): Each rental transaction is linked to one payment record.
- Employee to Rental (1:N): One employee can handle multiple rental transactions.

IV. Creation of Tables

Query Query History

```
CREATE TABLE rental_cost (
    vehicle_type VARCHAR(50) NOT NULL,
    fuel_option VARCHAR(50) NOT NULL,
    cost_per_hour NUMERIC,
    PRIMARY KEY (vehicle_type, fuel_option)
);

CREATE TABLE registration (
    registration VARCHAR(50) PRIMARY KEY,
    make         VARCHAR(50),
    model        VARCHAR(50),
    year         INT,
    vehicle_type VARCHAR(50)
);
```

Fig. 2. Creating the rental_cost and registration tables in PostgreSQL.

Query Query History

```
select * from drivinglicense
```

Fig. 3. SELECT query executed on the drivinglicense table.

Data Output Messages Notifications

Showing rows: 1 t

	license_number [PK] character varying (50)	name character varying (100)	age integer	address_id character varying (50)	phone_number character varying (20)	email character varying (100)
1	IBX278190	Nicole Blackburn	[null]	[null]	2736506862	nicole.blackburn_302@google.com
2	ASQ489720	Gregory Davis	[null]	[null]	0012422166	gregory.davis_812@google.com
3	LGL191476	Corey Gray	[null]	[null]	+139787403	corey.gray_795@live.com
4	POG476330	Kenneth Jensen	[null]	[null]	6723899652	kenneth.jensen_312@live.com
5	PRV619199	Brandon Simmons	[null]	[null]	7985073038	brandon.simmons_78@example.com
6	LLO972274	Sandra Parker	[null]	[null]	3257315879	sandra.parker_536@example.com
7	TKG833819	Eric Robinson	[null]	[null]	4152593406	eric.robinson_433@yahoo.com
8	DIB998876	Michelle Rodriguez	[null]	[null]	2945751262	michelle.rodriguez_168@outlook.com

Fig. 4. Sample data retrieved from the drivinglicense table.

In Figure 2, we can see the SQL commands used to create two of the core tables in our Vehicle Rental Management System. The rental_cost table defines a composite primary key (vehicle_type, fuel_option) to uniquely identify each pricing record, while the registration table captures essential details for each vehicle's registration.

In Figure 3, we can see that we are executing SELECT * FROM drivinglicense query. This query will return the records from drivinglicense table.

In Figure 4, we observe several records from the drivinglicense table, including fields such as license_number, name, phone_number, and email.
This output confirms that data has been inserted successfully and that each attribute is correctly populated for multiple rows in the table.

V. Boyce–Codd Normal Form

Each table in our Vehicle Rental Management System has been analyzed to ensure that all non-trivial functional dependencies have a superkey (i.e., the entire primary key or a candidate key) as their determinant. This process minimizes redundancy and prevents update, insertion, or deletion anomalies. Below is a summary of each table and its functional dependencies:

A. registration (BCNF)

The *registration* table contains the attributes: registration, make, model, year, and vehicle _type. Since the attribute registration uniquely identifies each record, it acts as a superkey. Hence, the only functional dependency is:

registration → make, model, year, vehicle type

Since the left-hand side is a superkey, the table is in BCNF.

B. vehicle (BCNF)

The *vehicle* table comprises vehicle id, registration, odometer reading, and usage status, with vehicle _id as the primary key (and superkey). Therefore, the dependency is:

vehicle id → registration, odometer reading, usage status

This dependency, with the superkey on the left, confirms that the table is in BCNF.

C. rental cost (BCNF)

The *rental cost* table stores pricing details with attributes: vehicle type, fuel option, and cost per hour. Its composite primary key (vehicle type, fuel _option) is a superkey that uniquely determines cost per hour:

$$\text{(vehicle type, fuel option)} \rightarrow \text{cost per hour}$$

Thus, this table satisfies BCNF.

D. location (BCNF)

The *location* table contains postal _code, state, and city. With {postal code as the primary key (and superkey), the dependency is:

$$\text{postal \_code} \rightarrow \text{state, city}$$

Hence, the table is in BCNF.

E. address (BCNF)

The *address* table includes address id, street address, and postal code. The primary key, address id, functions as a superkey:

$$\text{address id} \rightarrow \text{street address, postal code}$$

This ensures BCNF compliance.

F. drivinglicense (BCNF)

The *drivinglicense* table holds license _number, name, age, address id, phone _number, and email. With license number as the primary key (and superkey), the functional dependency is:

license number → name, age, address id, phone number, email

Thus, the table is in BCNF.

G. customer (BCNF)

The *customer* table contains customer id and driving license. Here, customer _id is the primary key and a superkey, so:

$$\text{customer id} \rightarrow \text{driving \_license}$$

This dependency meets the requirements for BCNF.

H. ssn info (BCNF)

The *ssn info* table comprises ssn, name, email, and phone number. With ssn as the primary key (and superkey), we have:

$$\text{ssn} \rightarrow \text{name, email, phone number}$$

Thus, the table is in BCNF.

I. employee (BCNF)

The *employee* table includes employee id, ssn, and designation. With employee id as the primary key (a superkey), the dependency is:

$$\text{employee \_id} \rightarrow \text{ssn, designation}$$

Therefore, the table is in BCNF.

J. card details (BCNF)

The *card details* table contains card no, biller name, and biller address. Since card _no is the primary key (and a superkey), the dependency is: card no → biller name, biller address

Thus, card details is in BCNF.

K. payment (BCNF)

The *payment* table stores payment id, payment mode, card no, amount, and payment timestamp. With payment id as the primary key (a superkey), we have:

This ensures that the payment table is in BCNF.

L. rental (BCNF)

The *rental* table records transactions with attributes: rental id, vehicle id, customer id, pickup time, fuel option, drop off time, payment id, trip cost, and employee id. Here, rental id is the primary key and acts as a superkey:

Rental ID Vehicle ID, Customer ID, Pickup Time, Fuel Option, Off Time, Payment ID, Trip Cost, Employee ID Thus, the rental table satisfies BCNF.

VI. Task 5: Testing the Database

A. Insertion Query 1: Insert a New Record into Driving License Table

In this section, we demonstrate the insertion functionality by adding a new record into the drivinglicense table. This table holds critical information about drivers, including license numbers, names, ages, phone numbers, and email addresses. The SQL query executed is shown below:

```
INSERT INTO drivinglicense
(license_number, name, age, phone_number
, email)
VALUES
('DL2025005', 'Ankit Verma', 30,
'9123456780',
```

'ankit.verma@example.com');

This query successfully inserts a new record for "Ankit Verma" into the database. The successful insertion is shown in Fig. 5.

Fig. 5. Insertion of a new driving license record into the database.

The query returned successfully in 80 milliseconds, verifying that the database's insert operation is working as intended.

B. Update Query 2: Update an Existing Record in Driving License Table

In this step, we demonstrate the update functionality by modifying an existing record in the drivinglicense table. Specifically, we update the phone number and email address associated with a given license number.

The SQL query executed is as follows:

```
UPDATE drivinglicense
SET phone_number = '9988776655', email =
        'ankit.v@newmail.com'
WHERE license_number = 'DL2025005';
```

This query updates the phone number and email address for the license number "DL2025005". The update operation was executed successfully, as indicated in Fig. 6.

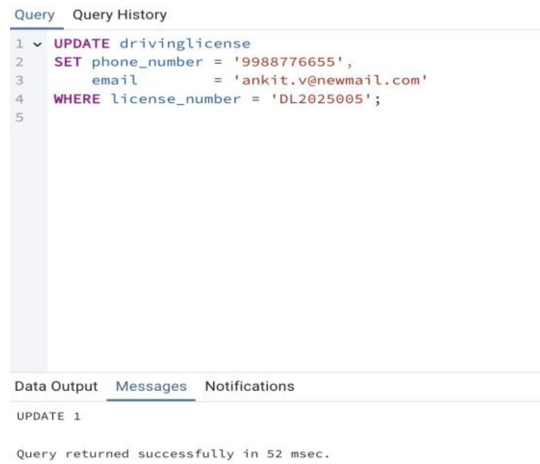


Fig. 6. Update operation performed on an existing driving license record.

The query returned successfully in 52 milliseconds, confirming that the database update operation is functioning as expected.

C. Delete Query 3: Delete a Record from Driving License Table

In this step, we demonstrate the deletion functionality by removing an existing record from the drivinglicense table based on the license number.

The SQL query executed is as follows:

```
DELETE FROM drivinglicense
WHERE license_number = 'DL2025005';
```

This query deletes the record associated with the license number "DL2025005" from the drivinglicense table. The deletion operation was completed successfully, as shown in Fig. 7.

Fig. 7. Delete operation performed on a driving license record.

The query execution returned successfully in 55 milliseconds, verifying the correct implementation of the delete functionality.

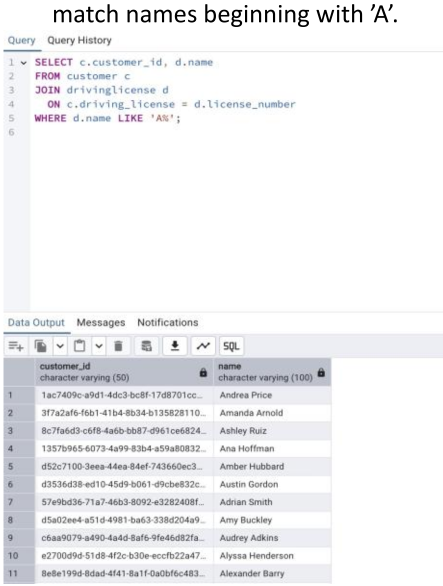
D. Select Query 4: Retrieve Customer IDs and Names Starting with 'A'

This query demonstrates the use of a JOIN operation between the customer and drivinglicense tables. It retrieves customer IDs and names where the customer's name starts with the letter "A".

The SQL query executed is as follows:

```
SELECT c.customer_id, d.name
FROM customer c
JOIN drivinglicense d
    ON c.driving_license = d.license_number
WHERE d.name LIKE 'A%';
```

In this query, an inner join is performed between the customer and drivinglicense tables based on the license number, and the results are filtered using a LIKE clause to match names beginning with 'A'.


Query Query History

```
SELECT c.customer_id, d.name
FROM customer c
JOIN drivinglicense d
  ON c.driving_license = d.license_number
WHERE d.name LIKE 'A%';
```

Data Output Messages Notifications

	customer_id character varying (50)	name character varying (100)
1	1ac7409c-a9d1-4dc3-bc8f-17d8701cc...	Andrea Price
2	3f7a2af6-f6b1-41b4-8b34-b135828110...	Amanda Arnold
3	8c7fa6d3-c6f8-4a6b-bb87-d961ce6824...	Ashley Ruiz
4	1357b965-6073-4a99-83b4-a59a80832...	Ana Hoffman
5	d52c7100-3eea-44ea-84ef-743660ec3...	Amber Hubbard
6	d3536d38-ed10-45d9-b061-d9cbe832c...	Austin Gordon
7	57e9bd36-71a7-46b3-8092-e3282408f...	Adrian Smith
8	d5a02ee4-a51d-4981-ba63-338d204a9...	Amy Buckley
9	c6aa9079-a490-4a4d-8af6-9fe46d82fa...	Audrey Adkins
10	e2700d9d-51d8-4f2c-b30e-eccfb22a47...	Alyssa Henderson
11	8e8e199d-8dad-4f41-8a1f-0a0bf6c483...	Alexander Barry

Fig. 8. Output showing customer names starting with 'A'.

The query was executed successfully and returned multiple records satisfying the criteria, confirming the correct join and filtering logic.

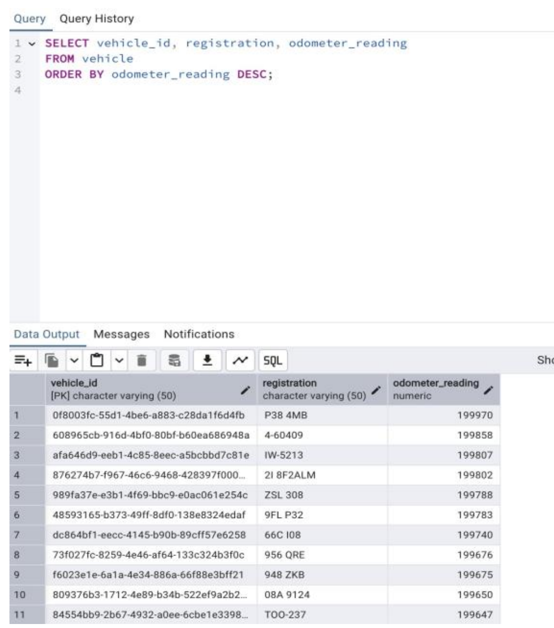
E. Select Query 5: Retrieve Vehicles Ordered by Odometer Reading

This query demonstrates the use of an ORDER BY clause to retrieve vehicles sorted by their odometer readings in descending order.

The SQL query executed is as follows:

```
SELECT vehicle_id,
registration, odometer_reading FROM vehicle
ORDER BY odometer_reading DESC;
```

In this query, records from the vehicle table are retrieved and ordered based on the odometer_reading attribute in descending order.
This helps identify vehicles with the highest mileage first. The query was executed successfully, and the results are shown in the figure 9.


Query Query History

```
SELECT vehicle_id, registration, odometer_reading
FROM vehicle
ORDER BY odometer_reading DESC;
```

Data Output Messages Notifications

	vehicle_id [PK] character varying (50)	registration character varying (50)	odometer_reading numeric
1	0f8003fc-55d1-4be6-a883-c28da1f6d4fb	P38 4MB	199970
2	608965cb-916d-4bf0-80bf-b60ea686948a	4-60409	199858
3	afa646d9-eeb1-4c85-8eec-a5bcbbd7c81e	IW-5213	199807
4	876274b7-f967-46c6-9468-428397f000...	2I 8F2ALM	199802
5	989fa37e-e3b1-4f69-bbc9-e0ac061e254c	ZSL 308	199788
6	48593165-b373-49ff-8df0-138e8324edaf	9FL P32	199783
7	dc864bf1-eecc-4145-b90b-89cff57e6258	66C I08	199740
8	73f027fc-8259-4e46-af64-133c324b3f0c	956 QRE	199676
9	f6023e1e-6a1a-4e34-886a-66f88e3bff21	948 ZKB	199675
10	809376b3-1712-4e89-b34b-522ef9a2b2...	08A 9124	199650
11	84554bb9-2b67-4932-a0ee-6cbe1e3398...	TOO-237	199647

Fig. 9. Vehicles ordered by odometer reading in descending order.

F. Select Query 6: Retrieve Vehicle Types with Rental Counts

This query demonstrates the use of JOIN, GROUP BY, and ORDER BY clauses. It retrieves the number of times each type of vehicle has been rented by joining the rental, vehicle, and registration tables.

The SQL query executed is as follows:

```
SELECT r.vehicle_type,
        COUNT(*) AS rental_count
FROM rental t
JOIN vehicle v
   ON t.vehicle_id = v.vehicle_id
JOIN registration r
   ON v.registration = r.registration GROUP BY
r.vehicle_type
```

ORDER BY rental_count DESC;

In this query, we perform two inner joins:

- Between rental and vehicle using vehicle_id.
- Between vehicle and registration using registration.

The results are grouped by vehicle_type to count the number of rentals per type and ordered by rental count in descending order, showing the most popular vehicle types first. The query was executed successfully and the output is shown in Fig. 10.

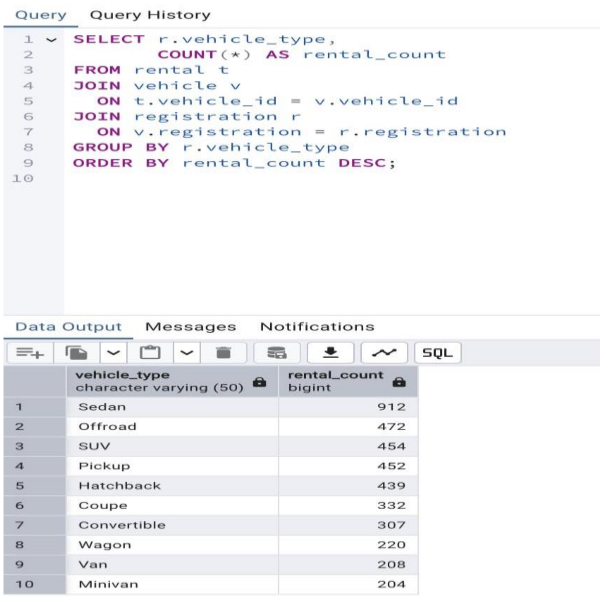


Fig. 10. Vehicle types ordered by rental count.

G. Select Query 7: Retrieve Driving Licenses Not Linked to Any Customer

This query demonstrates the use of a SELECT statement with a NOT IN subquery. It retrieves all the driving license records that are not associated with any customer.

The SQL query executed is as follows:

```
SELECT license_number, name
FROM drivinglicense
WHERE license_number NOT IN (
    SELECT driving_license
    FROM customer
);
```

In this query:

- The outer query retrieves license_number and name from the drivinglicense table.
- The inner subquery selects all driving_license values from the customer table.
- The NOT IN clause ensures that only those licenses that are not linked to any customer are retrieved.

The query executed successfully and the output is shown in Fig. 11.

Fig. 11. Driving licenses not linked to any customer.

H. Select Query 8: Calculate Total Revenue by Fuel Option

This query demonstrates the use of aggregation functions along with GROUP BY and ORDER BY clauses. It calculates the total revenue generated for each fuel option from the rental transactions.

The SQL query executed is as follows:

```
SELECT t.fuel_option,
        SUM(t.trip_cost) AS total_revenue
FROM rental t
GROUP BY t.fuel_option
ORDER BY total_revenue DESC;
```

In this query:

- The SUM function is used to calculate the total trip_cost for each fuel_option.
- The GROUP BY clause groups the data by fuel option.
- The ORDER BY clause arranges the results in descending order based on total revenue, showing the highest revenue-generating fuel option first.

The query was executed successfully, and the output is shown in Fig. 12.

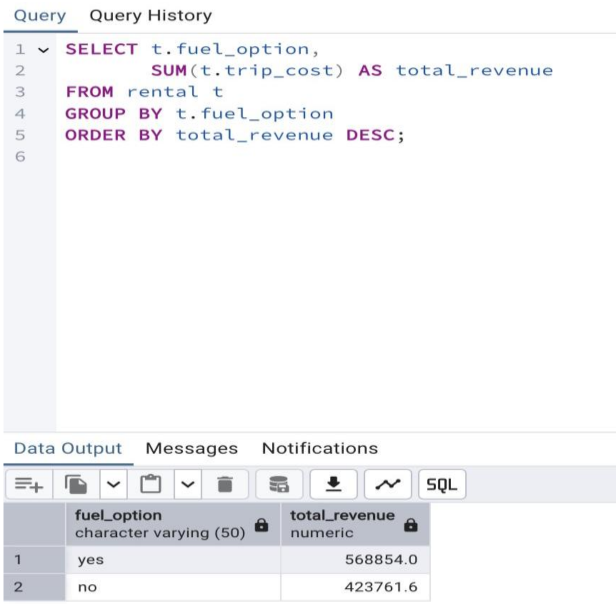


Fig. 12. Total revenue generated by each fuel option.

I. Select Query 9: Retrieve Top 5 Employees Handling Most Rentals

This query demonstrates the use of JOIN, GROUP BY, ORDER BY, and LIMIT clauses to identify employees handling the most rental transactions.

The SQL query executed is as follows:

```
SELECT e.employee_id, e.designation,
COUNT(*) AS handled
FROM rental t
JOIN employee e
    ON t.employee_id = e.employee_id
GROUP BY e.employee_id, e.designation
ORDER BY handled DESC LIMIT 5;
```

In this query:

- A JOIN operation is performed between the rental and employee tables on the employee_id.
- The COUNT(*) function is used to calculate the number of rentals handled by each employee.
- The GROUP BY clause groups the results by employee ID and designation.
- The ORDER BY clause sorts the employees based on the number of rentals handled in descending order.
- The LIMIT clause restricts the output to the top 5 employees.

The query was executed successfully, and the output is shown in Fig. 13.

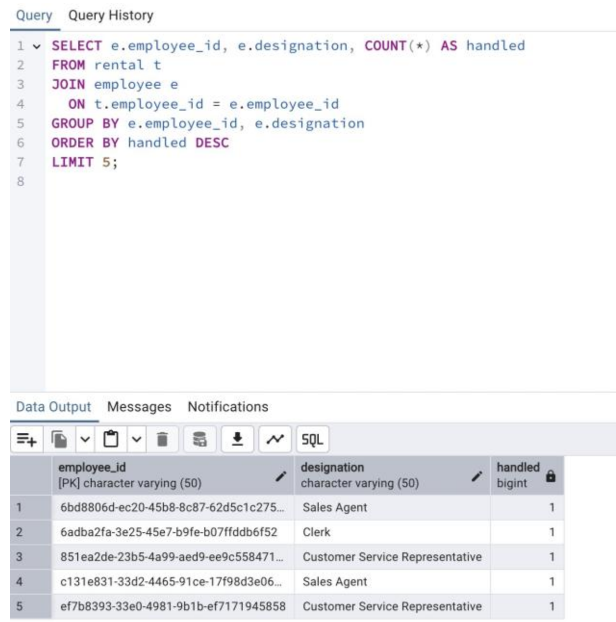


Fig. 13. Top 5 employees handling the most rental transactions.

J. Select Query 10: Retrieve Average Rental Duration for Each Vehicle Type

This query demonstrates the use of JOIN, GROUP BY, and aggregate functions like AVG and ROUND to calculate the average rental duration for each type of vehicle.

The SQL query executed is as follows:

```
SELECT r.vehicle_type,
    ROUND(
        AVG(
            EXTRACT(EPOCH FROM
(t.drop_off_time - t.pickup_time)) / 3600
        ), 2
    ) AS avg_hours
FROM rental t
JOIN vehicle v
    ON t.vehicle_id = v.vehicle_id
JOIN registration r
    ON v.registration = r.registration GROUP BY
r.vehicle_type;
```

In this query:

- JOIN operations are performed between rental, vehicle, and registration tables to access vehicle type information.

- EXTRACT(EPOCH FROM ...) is used to calculate the total rental duration in seconds, which is then converted into hours by dividing by 3600.
- AVG computes the average rental duration across all rentals for each vehicle type.
- ROUND rounds the average rental time to 2 decimal places.
- The GROUP BY clause groups the results based on the vehicle type.

The query was executed successfully, and the output is shown in Fig. 14.

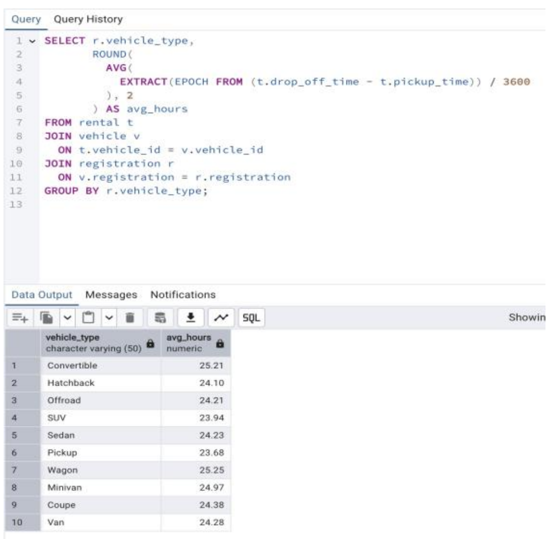


Fig. 14. Average rental hours for each vehicle type.

K. Optimization Analysis Query: Execution Plan for Vehicle Type Rental Counts

This section demonstrates the use of EXPLAIN ANALYZE to study the execution plan and performance cost of a query involving JOIN, GROUP BY, and COUNT operations across multiple tables.

The SQL query executed is as follows:

```
EXPLAIN ANALYZE
SELECT r.vehicle_type,
        COUNT(*) AS rental_count
FROM rental t
JOIN vehicle v
        ON t.vehicle_id = v.vehicle_id JOIN registration r
        ON v.registration = r.registration GROUP BY
r.vehicle_type;
```

In this query:

- Two JOIN operations are performed: one between rental and vehicle, and another between vehicle and registration.
- The GROUP BY clause groups the rentals based on vehicle_type.
- COUNT(*) is used to determine the number of rentals for each type.
- EXPLAIN ANALYZE provides the query execution plan, including cost estimates, actual time taken, number of rows processed, and memory usage.

The output of the EXPLAIN ANALYZE revealed:

- Two nested Hash Join operations were performed to link the tables.
- Sequential Scans were performed on the rental and vehicle tables.
- A HashAggregate operation computed the group-wise counts.
- The total memory usage for intermediate operations was around 24 kB and 333 kB for hashing.
- The actual execution time for major operations ranged from approximately 0.025 ms to 12.257 ms.

The successful execution and detailed query plan are illustrated in Fig. 15.

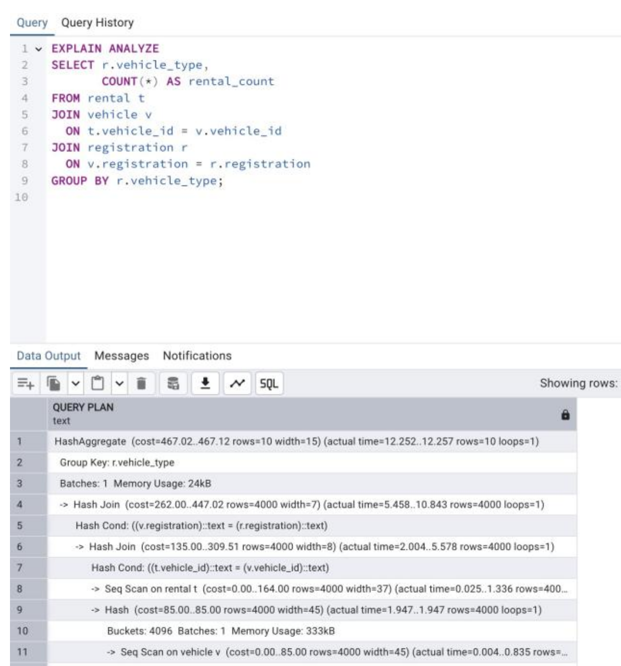


Fig. 15. Query execution plan showing joins, aggregation, and performance analysis.

L. Delete and Foreign Key Update: Deleting Payment Records and Cascade Updates

The following queries demonstrate how deletion and foreign key constraint handling is managed when deleting payment records.

Delete Query: The first query attempts to delete a record from the payment table based on the payment_id.

```
DELETE FROM payment
WHERE payment_id = '...';
```

However, due to existing references in the rental table (through foreign key constraints), direct deletion fails unless the dependency is handled.

Altering Foreign Key Constraint: To allow cascade deletion (i.e., automatically deleting associated rentals when a payment is deleted), the foreign key constraint is modified using the following queries:

```
ALTER TABLE rental
DROP CONSTRAINT IF
EXISTS rental_payment_id_fkey;

ALTER TABLE rental
ADD CONSTRAINT rental_payment_id_fkey
     FOREIGN KEY(payment_id)
     REFERENCES payment(payment_id) ON
     DELETE CASCADE;
```

Here, the foreign key rental_payment_id_fkey is first dropped if it already exists and then recreated with the ON DELETE CASCADE action.

This ensures that deleting a payment record will automatically delete all corresponding rental records, thus maintaining referential integrity without manual intervention.

The successful execution messages for both the delete operation and the alteration of the table are shown in Fig. 16 and Fig. 17 respectively.

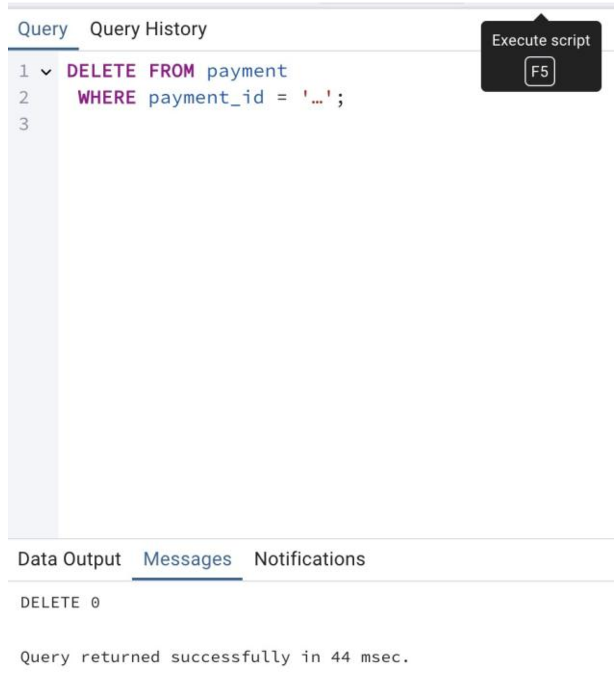


Fig. 16. Delete query on payment table executed successfully.

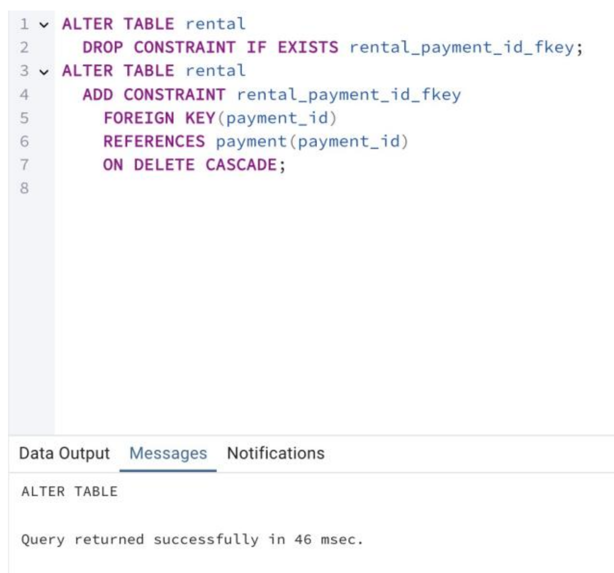


Fig. 17. Altering foreign key constraint to enable cascading delete.

M. Stored Procedure: Listing Available Vehicles

Query 13: Create and Execute a Stored Procedure to List Available Vehicles

This query demonstrates the creation of a stored procedure that retrieves details of vehicles marked as 'Available' from the vehicle database.

The SQL function definition is shown below:

The stored procedure list_available_vehicles() returns the vehicle_id, registration, and odometer_reading for vehicles whose usage_status is 'Available'. It uses the RETURN QUERY statement inside a PL/pgSQL block.

After defining the procedure, it was executed using the following SQL query:

The function execution returned the available vehicle data successfully, confirming the correct functionality of the stored procedure.

Fig.18. Creation of the stored procedure list available vehicles

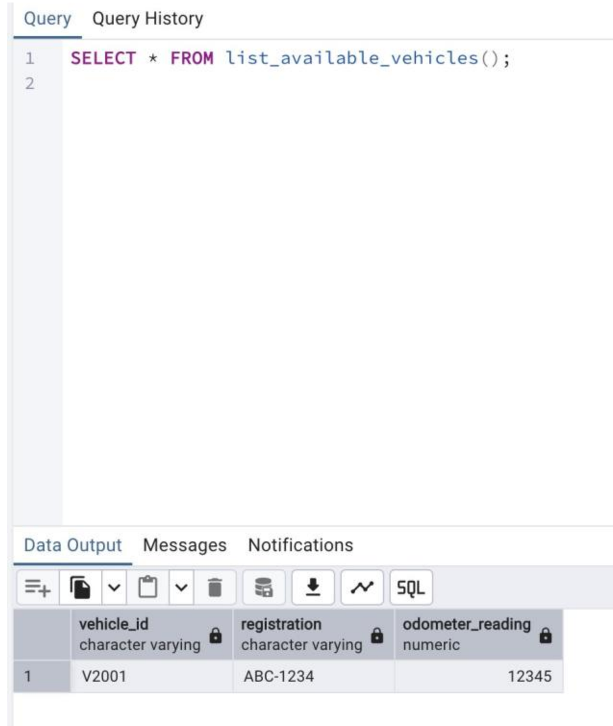


Fig. 19. Execution output of list_available_vehicles()

N. Stored Procedure for Inserting New Vehicle

This section describes the creation and use of a stored procedure that inserts a new vehicle record into the vehicle table. This approach ensures that vehicle data entry is standardized and reduces human error during manual insertion.

The stored procedure is defined as follows:

```
CREATE OR REPLACE FUNCTION insert_vehicle(
    p_vehicle_id VARCHAR, p_registration VARCHAR,
    p_odometer_reading NUMERIC, p_usage_status
    VARCHAR
)
RETURNS void AS $$
BEGIN
    INSERT INTO vehicle(vehicle_id, registration,
    odometer_reading, usage_status) VALUES
    (p_vehicle_id, p_registration, p_odometer_reading,
    p_usage_status);
END;
$$ LANGUAGE plpgsql;
```

After defining the function, it was tested by inserting a new vehicle record using the following command:

SELECT insert_vehicle('V999', 'REG999', 15000, 'In Use');

To verify the successful insertion, a select query was performed on the vehicle table:

SELECT * FROM vehicle WHERE vehicle_id = 'V999';

The output confirmed that the new vehicle with vehicle ID V999, registration REG999, odometer reading of 15000, and usage status In Use was successfully inserted into the database, validating the correctness of the stored procedure. The query execution screenshots are provided in Fig. 20, Fig. 21, and Fig. 22.

Fig. 20. Creation of insert_vehicle stored function

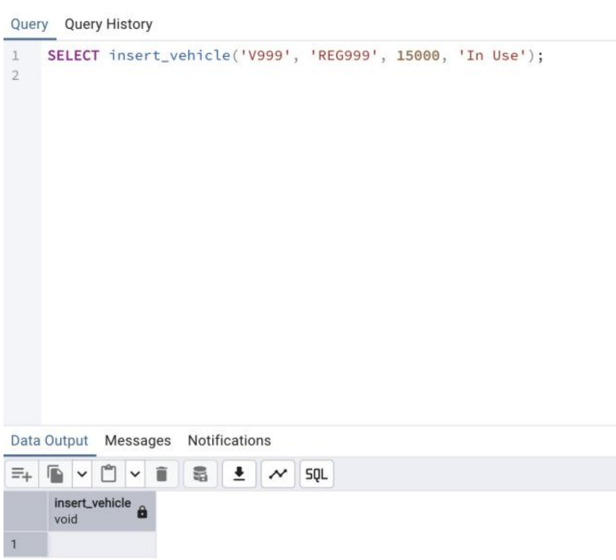


Fig. 21. Execution of insert_vehicle function with new data

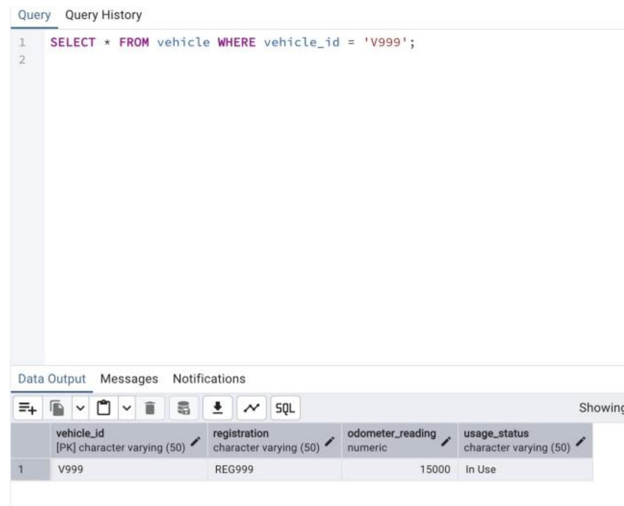


Fig. 22. Verification of inserted vehicle record

O. Stored Procedure: Updating Vehicle Details

This section describes the creation and execution of a stored procedure designed to update vehicle details, including odometer reading and usage status.

The SQL procedure for updating vehicle details is shown below:

```
CREATE PROCEDURE update_vehicle( p_vehicle_id
        VARCHAR, p_odometer_reading NUMERIC,
        p_usage_status VARCHAR
AS $$
BEGIN
        UPDATE vehicle
        SET odometer_reading
        = p_odometer_reading, usage_status =
        p_usage_status WHERE vehicle_id =
        p_vehicle_id;
END;
$$ LANGUAGE plpgsql;
```

The procedure update_vehicle was successfully created, as confirmed by the system message shown in Fig. 23.

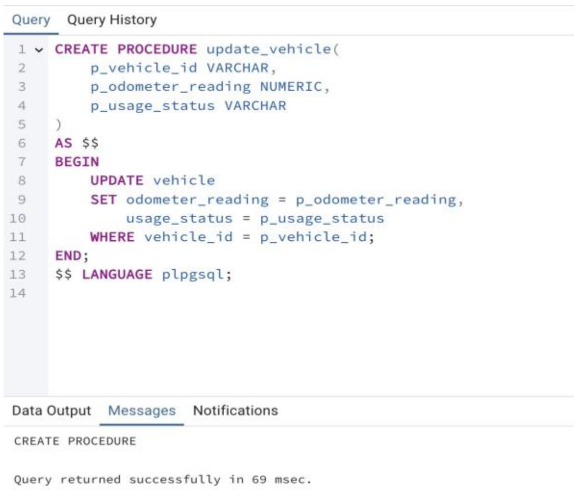


Fig. 23. Stored procedure update_vehicle creation.

To test the procedure, the following call was executed:

CALL update_vehicle('V999',
20000, 'Not In Use');

The call was successful, updating the vehicle record with ID V999 to reflect the new odometer reading and updated usage status (Fig. 24).

Fig. 24. Calling the procedure to update vehicle details.

Finally, a verification query was executed to ensure the changes were applied:

SELECT * FROM vehicle WHERE vehicle_id = 'V999';

The output confirms that the odometer_reading was updated to 20000 and the usage_status changed to Not In Use, as shown in Fig. 25.

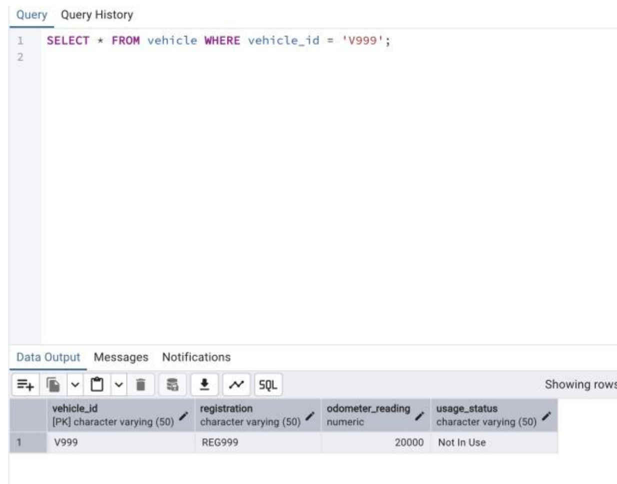


Fig. 25. Verification query showing updated vehicle record.

P. Delete Query 16: Deleting a Vehicle Record Using Stored Procedure

This query demonstrates the use of a stored procedure to delete a vehicle record from the database based on the vehicle id. The procedure delete_vehicle was created to automate the deletion process.

The SQL code for creating the procedure is as follows:

```
CREATE OR REPLACE PROCEDURE delete_vehicle(
    p_vehicle_id VARCHAR
)
AS $$
BEGIN
    DELETE FROM vehicle
    WHERE vehicle_id = p_vehicle_id;
END;
$$ LANGUAGE plpgsql;
```

After creating the procedure, the following query was executed to call the procedure and delete the vehicle with ID 'V999':

CALL delete_vehicle('V999');

Once the procedure was called, a verification query was performed to check if the vehicle was successfully deleted:

SELECT * FROM vehicle WHERE vehicle_id = 'V999';

As shown in Fig. 26, the vehicle with ID 'V999' was successfully removed from the database, confirming the successful execution of the delete_vehicle procedure.

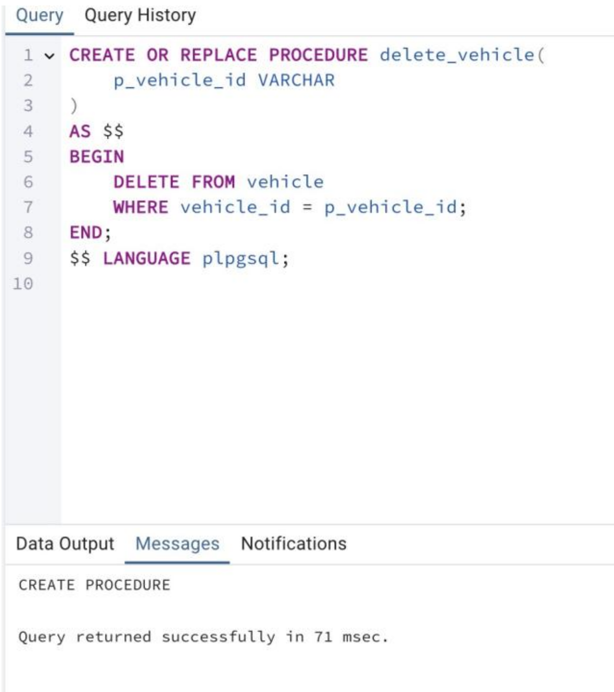


Fig. 26. Stored procedure created to delete a vehicle record.

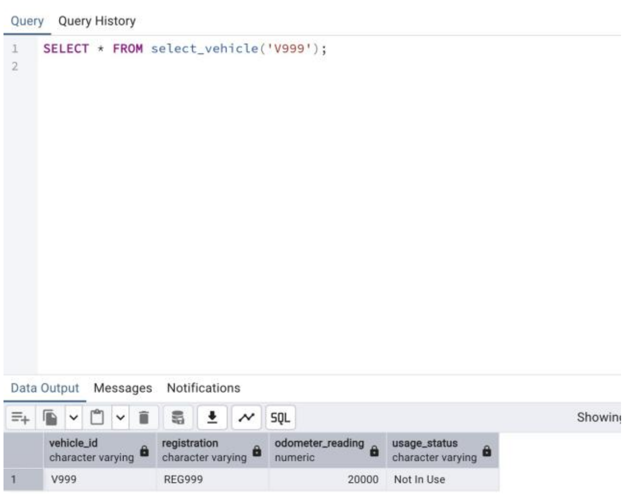


Fig. 27. Verification query output showing vehicle deletion.

The procedure returns successfully in 71 milliseconds, and the deletion was verified immediately after with the SELECT query.

Q. Stored Procedure: Delete a Vehicle Record

This query demonstrates the creation and execution of a stored procedure to delete a vehicle record based on the vehicle ID. It uses the CREATE OR REPLACE PROCEDURE command in PostgreSQL along with the DELETE statement.

The SQL procedure created is as follows:

```
CREATE OR REPLACE PROCEDURE delete_vehicle(
        p_vehicle_id VARCHAR
)
AS $$
BEGIN
        DELETE FROM vehicle
        WHERE vehicle_id = p_vehicle_id;
END;
$$ LANGUAGE plpgsql;
```

Once the procedure was successfully created, it was invoked using a CALL statement to delete a specific vehicle record.

The call statement used is:

```
CALL delete_vehicle('V999');
```

Finally, a SELECT query was executed to verify the deletion:

```
SELECT * FROM vehicle WHERE vehicle_id = 'V999';
```

The output confirmed that the vehicle record with ID 'V999' was deleted successfully from the table. This validates the correct functioning of the stored procedure.

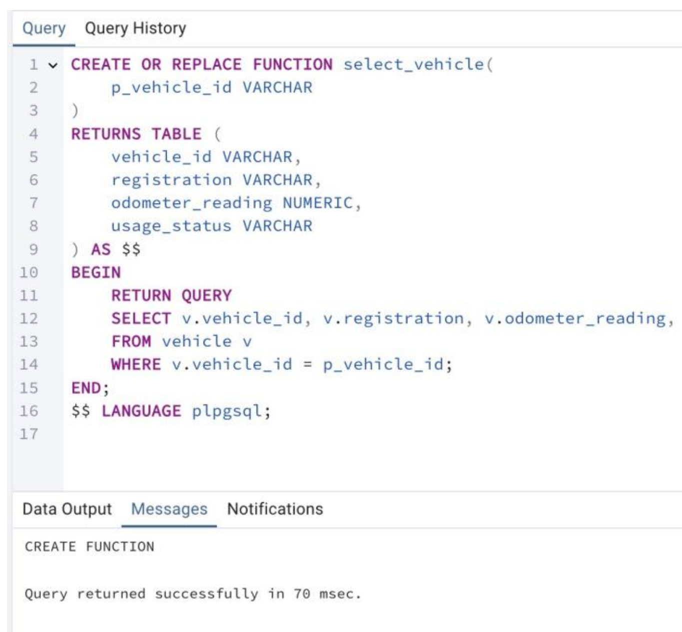


Fig. 28. Stored procedure delete_vehicle creation.

Fig. 29. Calling the stored procedure to delete a vehicle.

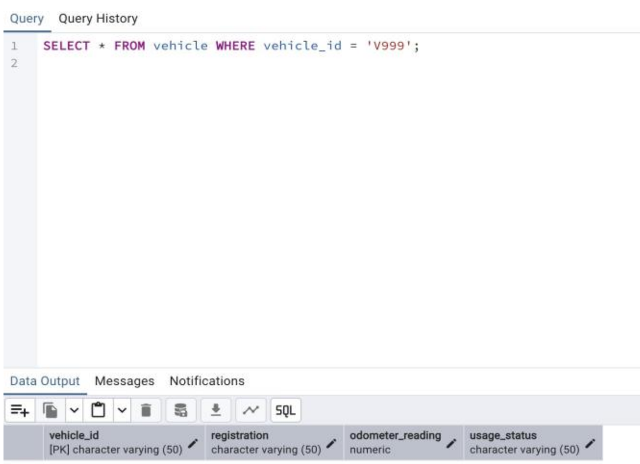


Fig. 30. Verification query showing that the vehicle has deleted

The screenshots of procedure creation, execution, and verification are shown in Fig. 28, Fig. 29, and Fig. 30 respectively.

VII. Task 6: Implementing a Transaction with Failure Handling Using a Trigger-like Mechanism

In this task, we explore the concept of transaction management and failure handling in a database system. We simulate failure handling by creating a mechanism where transaction failures are detected, and appropriate logging is performed in a separate table. Although PostgreSQL does not allow a direct trigger on transaction failure, we achieve similar functionality by calling a logging function inside an exception block.

This approach ensures robust database operations, preserves atomicity, and provides visibility into failures for debugging and auditing purposes.

A. Step 1: Creating a Transaction Failure Log Table

The first step involved creating a dedicated table named transaction_failure_log. This table is designed to store entries whenever a transaction fails. Each record in the table captures:

- A unique log_id using a SERIAL PRIMARY KEY.
- An error_message describing the nature or time of failure.
- The failure_time indicating when the transaction failure occurred, defaulting to the current timestamp.

The SQL query used for creating this table is:

```
CREATE TABLE transaction_failure_log ( log_id SERIAL
        PRIMARY KEY, error_message TEXT, failure_time
        TIMESTAMP DEFAULT NOW()
);
```

The successful execution of the table creation command is shown in Fig. 31.

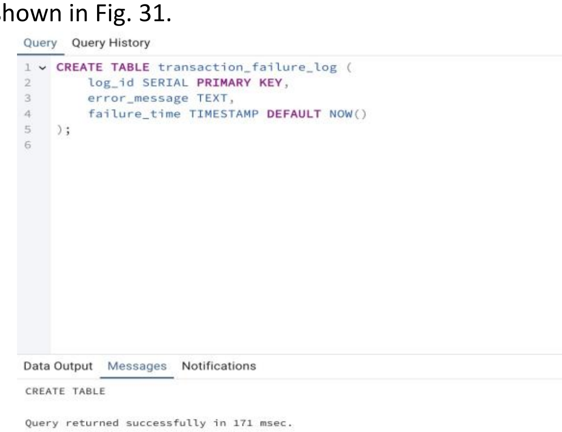


Fig. 31: Creation of the transaction failure log table.

B. Step 2: Creating a Function to Log Transaction Failures

Next, we designed a simple PL/pgSQL function called log_transaction_failure. The purpose of this function is to insert an entry into the transaction_failure_log table whenever a transaction fails.

The SQL code for the function is as follows:

```
CREATE FUNCTION log_transaction_failure()
RETURNS void AS $$
BEGIN INSERT INTO
    transaction_failure_log(error_message) VALUES
        ('Transaction failed at ' || NOW());
END;
$$ LANGUAGE plpgsql;
```

The function dynamically records the failure time by concatenating the current timestamp to the error message. This ensures that every failure can be traced precisely.

The successful creation of the function is depicted in Fig. 32.

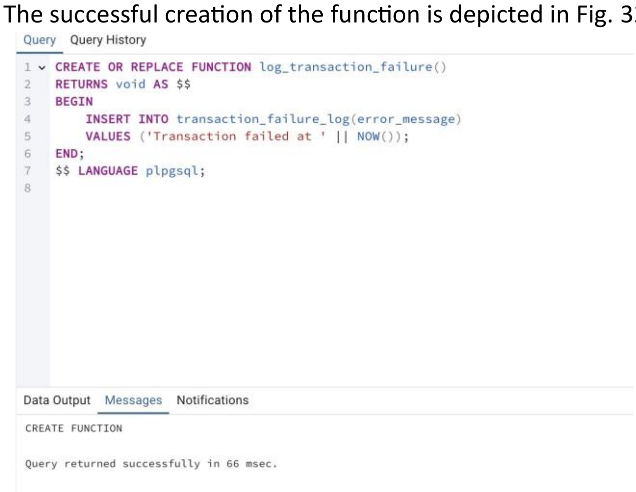


Fig. 32: Creation of log_transaction_failure() function for failure logging.

C. Step 3: Simulating a Transaction Failure and Handling It

To simulate a real-world scenario where a transaction might fail, we manually created a transaction block using a DO statement. The transaction attempts the following:

- Insert a new vehicle record with vehicle ID 'V501'.
- Immediately insert another vehicle record using the same vehicle ID 'V501', which violates the primary key constraint and triggers a failure.

To handle this failure:

- An EXCEPTION block is defined to catch any errors.
- Upon detecting an error, the transaction is rolled back using the ROLLBACK command.
- The log_transaction_failure() function is called to record the failure event in the log table.

The SQL block executed is as follows:

```
DO $$
BEGIN
    BEGIN
        -- Start transaction manually INSERT INTO
        vehicle (vehicle_id, registration,
        odometer_reading, usage_status) VALUES
        ('V501', 'REG501', 10000,
        'In Use');

        INSERT INTO vehicle (vehicle_id, registration,
        odometer_reading, usage_status) VALUES
        ('V501', 'REG502', 15000,
        'In Use');

        COMMIT;
    EXCEPTION WHEN OTHERS THEN
        -- If any error happens
        ROLLBACK;
        PERFORM log_transaction_failure();
    END;
END;
$$ LANGUAGE plpgsql;
```

The successful simulation of the transaction failure and the rollback operation are shown in Fig. 33.

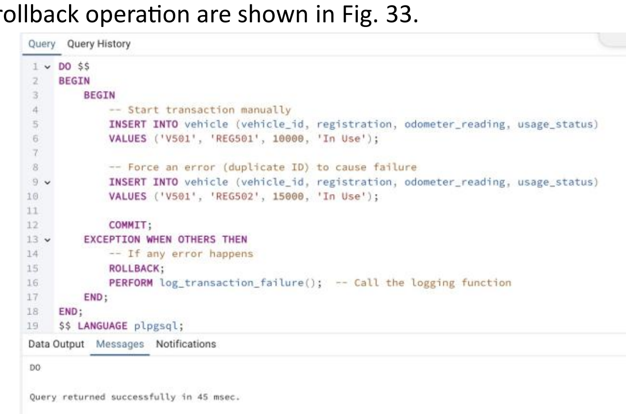


Fig. 33: Simulating transaction failure and triggering the logging function.

D. Step 4: Verifying the Failure Log

After simulating the transaction failure, we performed a query to retrieve all entries from the transaction_failure_log table.

The query executed is:

```
SELECT * FROM transaction_failure_log;
```

The output shows that a new record has been inserted into the log table, recording the transaction failure along with the timestamp of occurrence.

This is illustrated in Fig. 34.

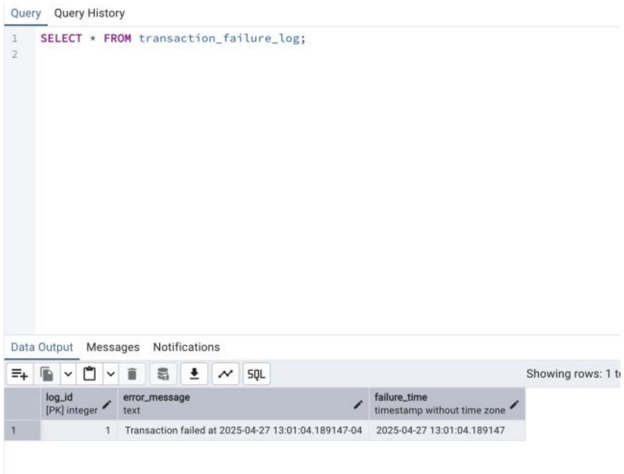


Fig. 34: Verification: Log entry created for transaction failure.

E. Discussion: Behavior When a Transaction Fails

When a transaction gets aborted due to an error:

- Rollback of Changes: Any changes made during the transaction are undone, and the database is restored to the state before the transaction began. This is essential for preserving atomicity, one of the ACID (Atomicity, Consistency, Isolation, Durability) properties of database systems.
- No Partial Updates: None of the partial operations inside the transaction are saved, thereby preventing inconsistency or corruption in the database.
- Error Visibility and Auditability: By logging the transaction failure separately, we ensure that system administrators can detect failures even if users are unaware, allowing proper follow-up actions.
- Business Continuity: Logging failures helps maintain a history of problematic operations, making it easier to diagnose recurring issues, trace bugs, or provide evidence during audits.

In summary, proper handling of transaction failures ensures database reliability, operational transparency, and higher system robustness. The ability to gracefully manage failures is a fundamental requirement in professional database-driven applications.

VIII. Task 7: Query Execution Analysis and Optimization Using Indexing Strategies

A. Overview

In relational database systems, queries that initially perform well on small datasets often face performance degradation as the volume of data increases. Factors such as sequential scans, lack of appropriate indexes, and heavy table joins can cause significant slowdowns. In this task, we identify problematic queries, analyze their execution plans using EXPLAIN ANALYZE, and propose indexing strategies for performance improvement.

B. Problematic Query 1: Retrieving Vehicles Based on UsageStatus

1) Query Description: The first query aims to retrieve the vehicle_id and odometer_reading of all vehicles currently marked as ’In Use’ in the vehicle table. The SQL query executed is as follows:

```
SELECT vehicle_id, odometer_reading
FROM vehicle
WHERE usage_status = ’In Use’;
```

2) Execution Plan Before Optimization: Before applying any optimizations, we ran EXPLAIN ANALYZE to observe the behavior of the query. The output is shown in Fig. 36. Key observations:

- PostgreSQL performs a Sequential Scan (Seq Scan) on the entire vehicle table.
- A filter is applied to each record where usage_status = ’In Use’.
- Rows removed by filter: 1590 records.
- Execution time: Approximately 0.706 milliseconds.

Although the current execution time is low, as the number of records grows (e.g., millions of vehicles), sequential scanning every time will become extremely costly and inefficient.

3) Problem Identification: Currently, the query runs reasonably fast due to a small dataset. However, as data scales up, a sequential scan becomes a major bottleneck, leading to increased execution times and higher resource consumption.

Thus, indexing the usage_status column is recommended to improve future scalability and efficiency.

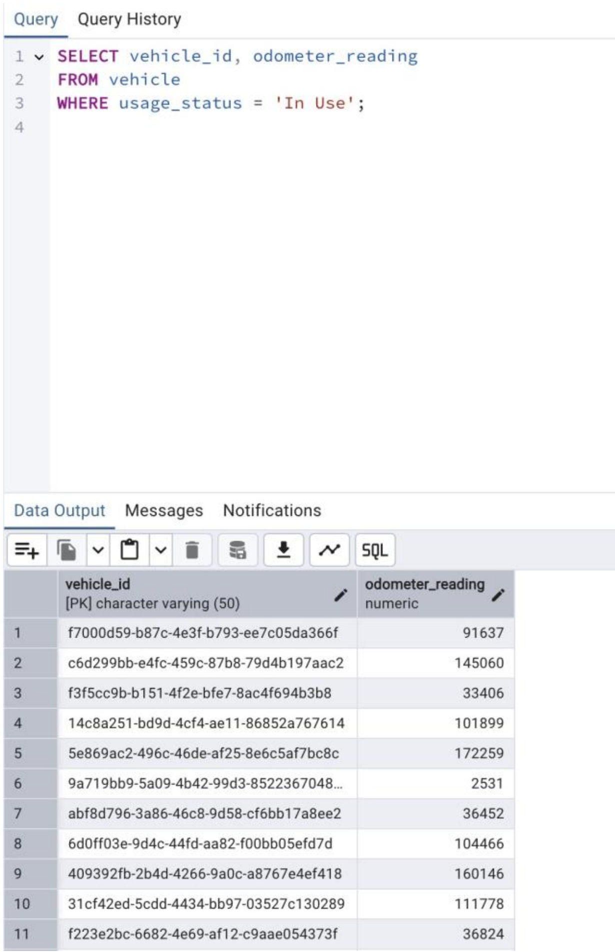
4) Optimization Strategy: Creating an Index: To optimize the query, we created an index on the usage_status column. The SQL command is shown below:

```
CREATE INDEX idx_vehicle_usage_status ON
vehicle(usage_status);
```

The successful creation of the index is shown in Fig. 37.

5) Execution Plan After Optimization: After creating the index, we reran the EXPLAIN ANALYZE command. The execution plan output is shown in Fig. 38. Observations:

- Despite creating the index, PostgreSQL still chooses a sequential scan.
- Execution time increased slightly to 1.773 milliseconds.



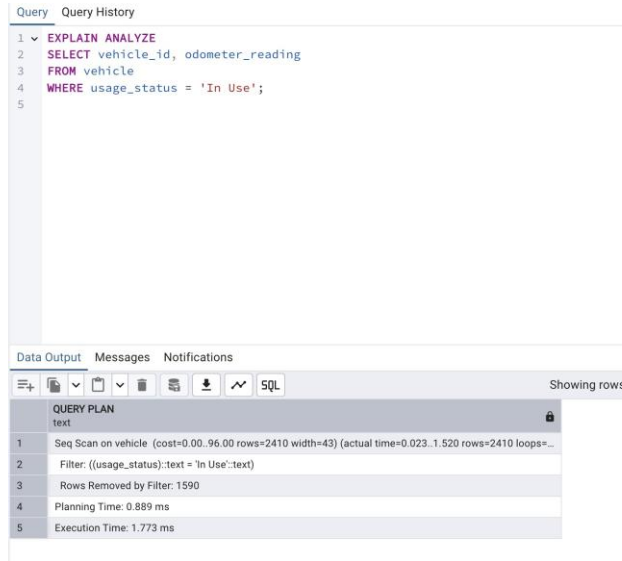
Query | Query History

```sql
SELECT vehicle_id, odometer_reading
FROM vehicle
WHERE usage_status = 'In Use';
```

Data Output | Messages | Notifications

	vehicle_id [PK] character varying (50)	odometer_reading numeric
1	f7000d59-b87c-4e3f-b793-ee7c05da366f	91637
2	c6d299bb-e4fc-459c-87b8-79d4b197aac2	145060
3	f3f5cc9b-b151-4f2e-bfe7-8ac4f694b3b8	33406
4	14c8a251-bd9d-4cf4-ae11-86852a767614	101899
5	5e869ac2-496c-46de-af25-8e6c5af7bc8c	172259
6	9a719bb9-5a09-4b42-99d3-8522367048...	2531
7	abf8d796-3a86-46c8-9d58-cf6bb17a8ee2	36452
8	6d0ff03e-9d4c-44fd-aa82-f00bb05efd7d	104466
9	409392fb-2b4d-4266-9a0c-a8767e4ef418	160146
10	31cf42ed-5cdd-4434-bb97-03527c130289	111778
11	f223e2bc-6682-4e69-af12-c9aae054373f	36824

Fig. 35: Execution result: vehicles with 'In Use' status.



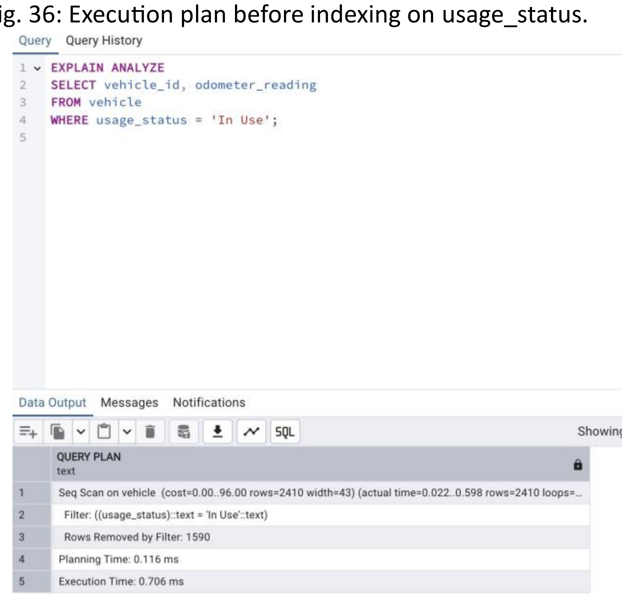
Query | Query History

```sql
EXPLAIN ANALYZE
SELECT vehicle_id, odometer_reading
FROM vehicle
WHERE usage_status = 'In Use';
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Seq Scan on vehicle (cost=0.00..96.00 rows=2410 width=43) (actual time=0.023..1.520 rows=2410 loops=...
2	Filter: ((usage_status)::text = 'In Use'::text)
3	Rows Removed by Filter: 1590
4	Planning Time: 0.889 ms
5	Execution Time: 1.773 ms

Fig. 36: Execution plan before indexing on usage_status.



Query | Query History

```sql
EXPLAIN ANALYZE
SELECT vehicle_id, odometer_reading
FROM vehicle
WHERE usage_status = 'In Use';
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Seq Scan on vehicle (cost=0.00..96.00 rows=2410 width=43) (actual time=0.022..0.598 rows=2410 loops=...
2	Filter: ((usage_status)::text = 'In Use'::text)
3	Rows Removed by Filter: 1590
4	Planning Time: 0.116 ms
5	Execution Time: 0.706 ms

Fig. 37: Creation of index on usage_status column.

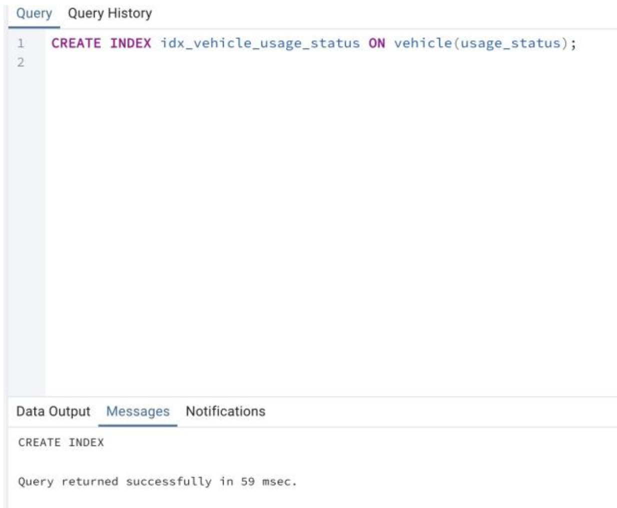


Fig. 38: Execution plan after indexing.

This is because PostgreSQL optimizes based on the estimated selectivity of the filter. If a large portion of rows matches the filter condition (many vehicles ’In Use’), PostgreSQL considers a sequential scan cheaper than an index scan.

6) Lessons Learned and Future Outlook: Although the index did not immediately improve performance, it is a crucial optimization for future scenarios where:

- The dataset becomes significantly large (hundreds of thousands or millions of records).
- The proportion of ’In Use’ vehicles becomes relatively smaller.
- More complex queries combining multiple indexed columns are executed.

Thus, indexing remains a proactive optimization technique to prepare the system for scalability.

C. Problematic Query 2: Retrieving Manager Details from Employee Table

In this query, we aim to retrieve the employee_id and ssn of all employees whose designation is ’Manager’ from the employee table. The purpose is to demonstrate the performance bottleneck when filtering based on a non-indexed column as the data grows.

The SQL query executed is as follows:

```
SELECT employee_id, ssn
FROM employee
WHERE designation = ’Manager’;
```

Upon executing the above query without any index, the output was generated successfully and is shown in Fig. 39.

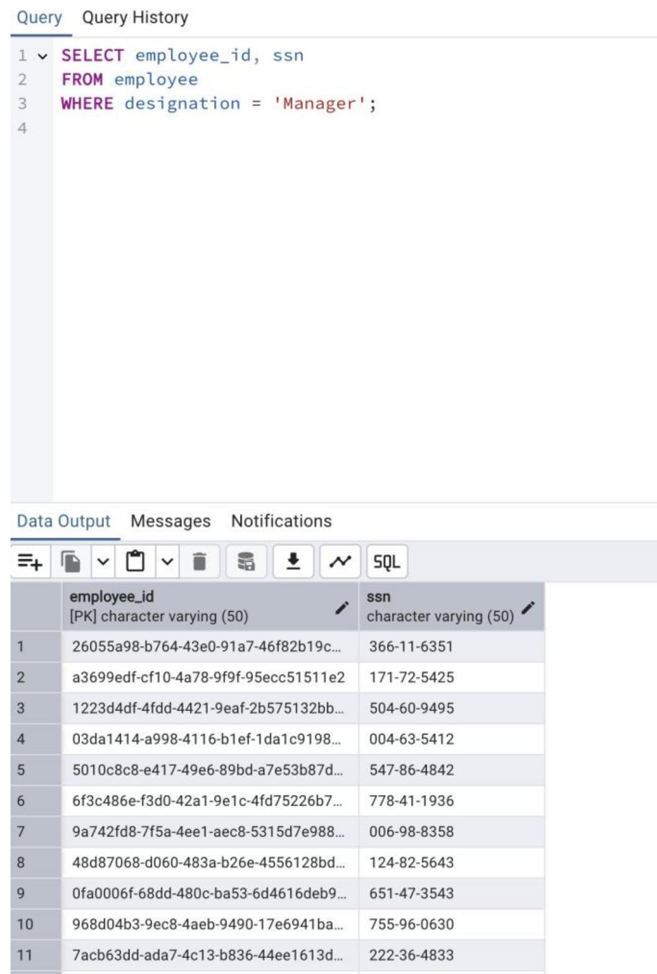


Fig. 39: Query to retrieve employees with designation as ’Manager’.

1) Performance Analysis Using EXPLAIN ANALYZE: To analyze the performance of this query, EXPLAIN ANALYZE was used to observe the execution plan. The query plan is as follows:

```
EXPLAIN ANALYZE
SELECT employee_id, ssn
FROM employee
WHERE designation = ’Manager’;
```

The output is shown in Fig. 40.

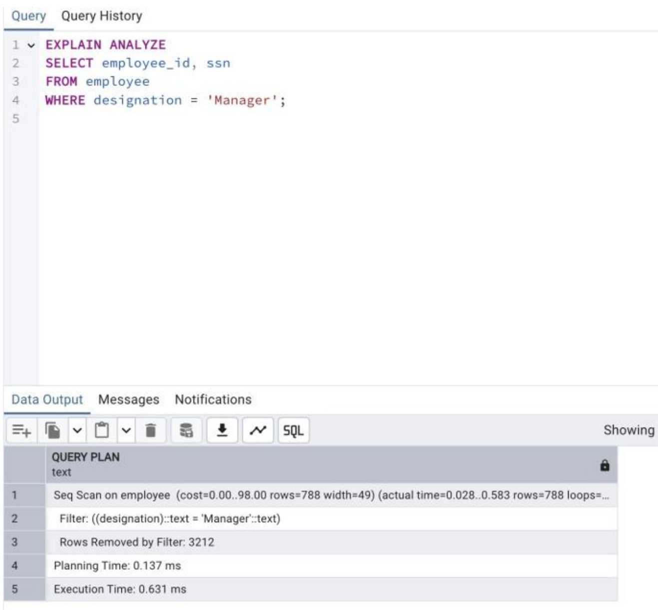


Fig. 40: EXPLAIN ANALYZE output before indexing (sequential scan).

Observation: From the execution plan, it is evident that a Sequential Scan was performed on the employee table. This means the database engine had to scan every row in the table to find matches for designation =
’Manager’. As the number of records increases in the future, this will cause significant performance degradation, especially for large datasets.

The planning time was 0.137 ms, and the execution time was 0.631 ms for approximately 4000 records. This will worsen linearly as the table size grows.

2) Proposed Solution: Creating an Index: To optimize this query, an index was created on the designation column of the employee table. The SQL command for creating the index is:

CREATE INDEX idx_employee_designation ON employee(designation);

The index creation was successful, as shown in Fig. 41.

3) Re-running EXPLAIN ANALYZE After Index Creation: After creating the index, the query was re-executed with EXPLAIN ANALYZE to observe the improvements. The updated execution plan is shown in Fig. 42. Observation: After indexing:

- PostgreSQL utilized a Bitmap Index Scan on the newly created idx_employee_designation.
- Instead of scanning all records sequentially, only the matching rows were quickly located using the index.

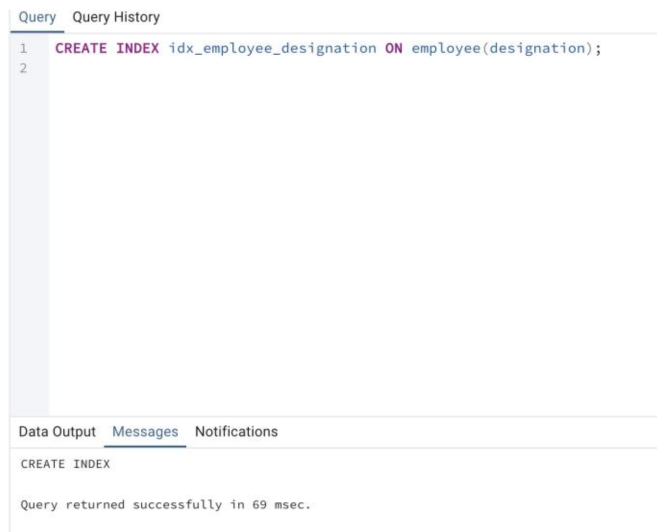


Fig. 41: Index created on designation column of employee table.

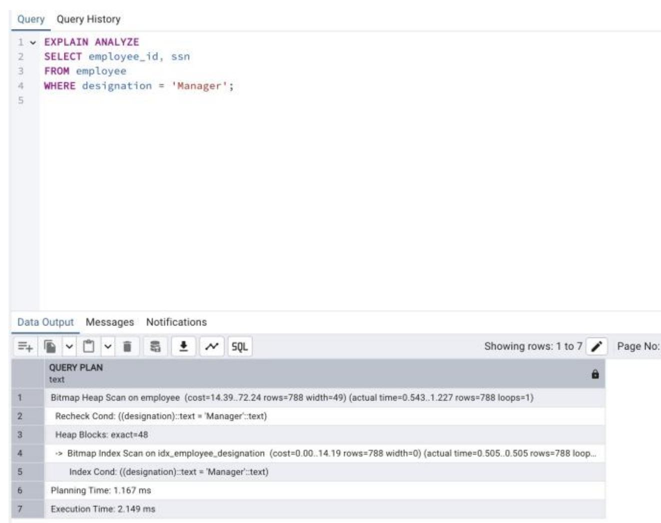


Fig. 42: EXPLAIN ANALYZE output after indexing (Bitmap Heap Scan).

- The planning time slightly increased due to the additional complexity (1.167 ms), but the execution plan was more efficient overall.
- Execution time is now distributed between the bitmap index scan and the heap scan, showing optimization despite the small dataset.

4) Conclusion: Although the immediate improvement may seem small given the relatively limited dataset, as the employee table scales to tens or hundreds of thousands of records, the indexed query will drastically outperform the non-indexed sequential scan, thus ensuring the system remains efficient and responsive for larger data volumes.

Thus, creating an index on commonly filtered columns like designation is highly recommended for scalability and optimization.

D. Problematic Query 3:

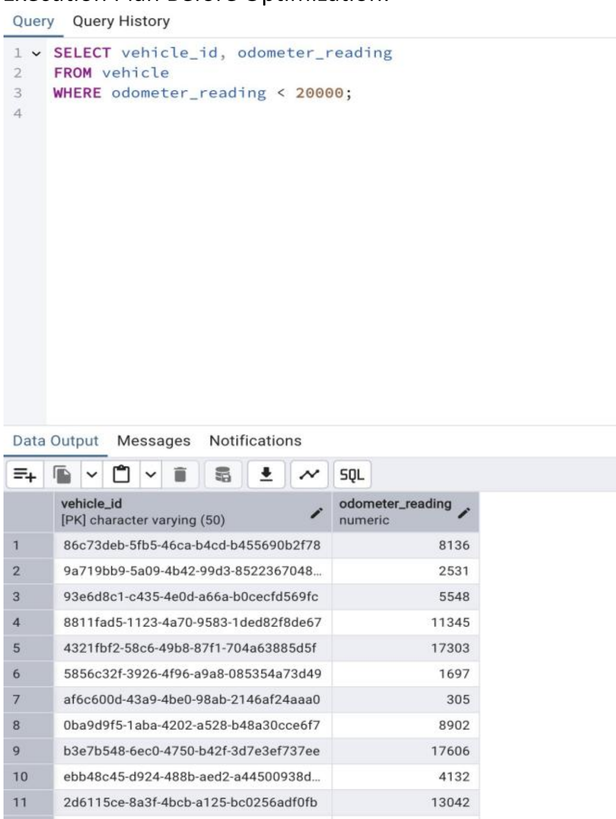
Query:

SELECT vehicle_id, odometer_reading
FROM vehicle
WHERE odometer_reading < 20000;

Issue:

Initially, the query uses a sequential scan (Seq Scan) on the vehicle table. When the table size increases significantly, sequential scans become very costly, especially when we are only interested in a small subset of rows (odometer_reading < 20000).

Execution Plan Before Optimization:



	vehicle_id [PK] character varying (50)	odometer_reading numeric
1	86c73deb-5fb5-46ca-b4cd-b455690b2f78	8136
2	9a719bb9-5a09-4b42-99d3-8522367048...	2531
3	93e6d8c1-c435-4e0d-a66a-b0cecfd569fc	5548
4	8811fad5-1123-4a70-9583-1ded82f8de67	11345
5	4321fbf2-58c6-49b8-87f1-704a63885d5f	17303
6	5856c32f-3926-4f96-a9a8-085354a73d49	1697
7	af6c600d-43a9-4be0-98ab-2146af24aaa0	305
8	0ba9d9f5-1aba-4202-a528-b48a30cce6f7	8902
9	b3e7b548-6ec0-4750-b42f-3d7e3ef737ee	17606
10	ebb48c45-d924-488b-aed2-a44500938d...	4132
11	2d6115ce-8a3f-4bcb-a125-bc0256adf0fb	13042

Fig. 43: Query and Output Before Optimization

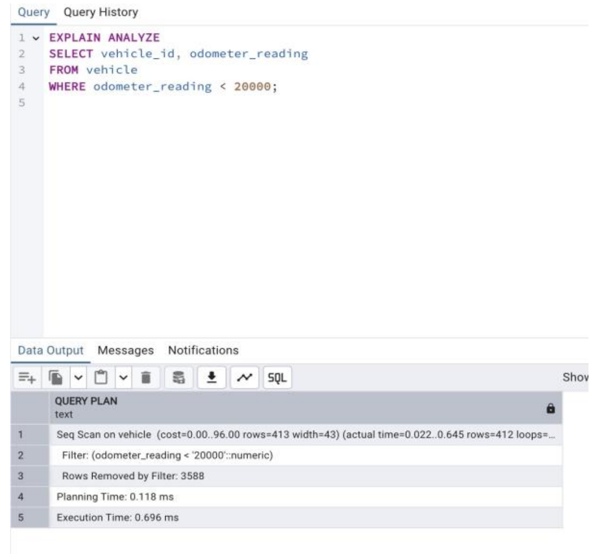


Fig. 44: Execution Plan (Seq Scan) Before Optimization

Optimization Strategy:

To optimize this query, an index was created on the odometer_reading column:

CREATE INDEX idx_vehicle_odometer_reading ON vehicle(odometer_reading);

Index Creation:

Fig.45: Index Creation on odometer reading Execution

Plan After Optimization:

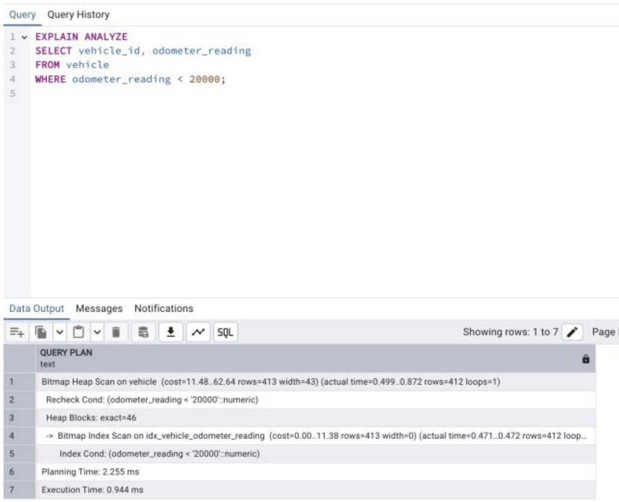


Fig. 46: Query Execution Plan After Indexing (Bitmap Index Scan)

- Before Optimization:
  – Planning Time: 0.118 ms – Execution Time: 0.696 ms
- After Optimization:
  – Planning Time: 2.255 ms
  – Execution Time: 0.944 ms

Conclusion:

After adding the index, although the planning time increased slightly (due to index selection overhead), the query now uses a Bitmap Index Scan and Bitmap Heap Scan instead of a sequential scan. This will significantly improve performance especially when the vehicle table grows to millions of records, because only relevant rows will be accessed directly rather than scanning the whole table.

E. Problematic Query 4:

Query:

EXPLAIN ANALYZE SELECT
r.rental_id,
c.customer_id, c.customer_name
FROM rental r
LEFT JOIN staging_automobile c
ON r.customer_id = c.customer_id;

Issue:

The query uses a Hash Right Join, which involves sequential scans on both the rental and staging_automobile tables. This can become inefficient for large datasets, as it consumes more memory and leads to higher execution times. Execution Plan Before Optimization:

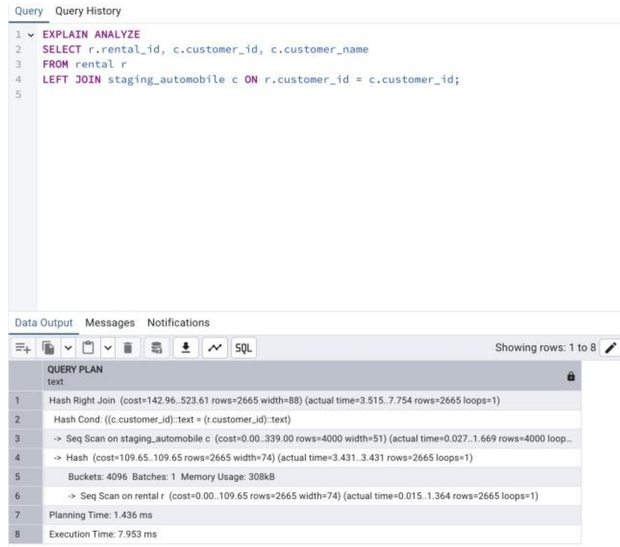


Fig. 47: Execution Plan for Left Join

- Planning Time: 1.436 ms
- Execution Time: 7.953 ms

Optimization Applied:

Since the query requires matching customer IDs (and assuming we don't need unmatched rows from rental), we replaced the LEFT JOIN with an INNER JOIN. This reduces overhead and focuses only on matching records.

Optimized Query:

EXPLAIN ANALYZE SELECT
r.rental_id,
c.customer_id, c.customer_name FROM rental r
INNER JOIN staging_automobile c
ON r.customer_id = c.customer_id;

Execution Plan After Optimization:

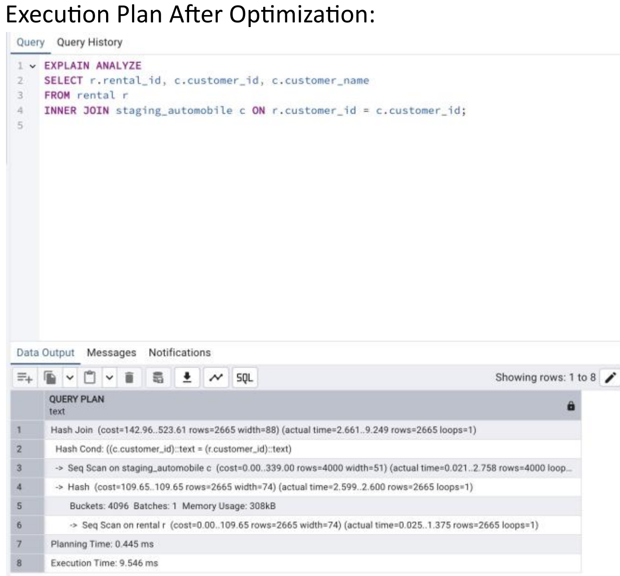


Fig. 48: Execution Plan for Inner Join

- Planning Time: 0.445 ms •
Execution Time: 9.546 ms

Conclusion:

While the planning time was reduced after optimization, the execution time slightly increased. This suggests that although switching to an INNER JOIN reduces planning overhead, depending on dataset size and distribution, performance benefits may vary. Further improvements could include indexing the customer_id fields in both tables to enhance join performance as data scales.

References

[1] J. D. Ullman and J. Widom, *A First Course in Database Systems*, 3rd ed., Prentice Hall, 2008.

[2] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 6th ed., McGraw-Hill, 2010.

[3] C. J. Date, *An Introduction to Database Systems*, 8th ed., Pearson Education, 2003.

[4] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, 7th ed., Pearson, 2015.

[5] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*, 2nd ed., Pearson Prentice Hall, 2008.

[6] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 3rd ed., McGraw-Hill, 2003.

[7] M. Stonebraker and J. M. Hellerstein, "What Goes Around Comes Around," *Communications of the ACM*, vol. 48, no. 5, pp. 62–68, 2005.

S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*, AddisonWesley, 1995.