CSE 4/560 Data Models and Query Language - Project

Car Rentals Data Management

Vignesh Venkatakumar
vvenkata

Nitin Dharmapal
nitindha

Mithil Jeevan Gaonkar
mithilje

***Abstract*—We chose to work on a car rental database project because it's a practical and relatable scenario. Car rentals are something most people have experience with, making it easier to understand and work on. This project helps us learn and practice important database skills.**

Our goal is to create a database for a car rental company. It will keep track of customers, cars, bookings, and more. Customers can sign up, book cars, and pay bills. The database will also help the company analyze data to make better business decisions.

This project is not only for learning but also has real-world applications in the car rental industry. It's a flexible and useful way to study and apply database concepts.

I. Problem Statement

Improving Car Rental Management with a centralized Database We're tackling the challenge of managing a car rental business. We want to answer these questions:

1) How can we keep track of customer information, car availability, and bookings effectively?
2) How can we make sure billing and reporting are accurate and automated in relation to the booking details?
3) How can we use data to make our business better, the experience of the employees and the customers and how to set prices?

Why use a database instead of an Excel file?

1) Better Organization: Databases keep our data tidy and prevent mistakes.
2) Faster and Reliable: Databases are quicker and handle lots of people using them at the same time.
3) Safety: Databases secure our customer details.
4) Scaling Up: As our business grows, databases can handle more data.
5) Smart Decisions: Databases help us understand our business and make it better. Excel can't do that effectively.

a. Discuss the background of the problem leading to your objectives. Why is it a significant problem?

Car rental companies provide a crucial service, but managing their operations is complex. They work with a lot of data, including information about cars and customers. It's difficult to maintain order and efficiency in everything.

Significance of the Problem: Due to its impact on the consumer experience, this issue is significant. Customers may not be satisfied if there is disorganization. Not to mention that data errors might be expensive. Efficient operations and the use of data to make sound decisions are critical to success.

Goals: Make things run smoothly by creating a good system for data. Avoid errors by keeping data accurate. Make data-driven decisions that make sense. Protect data from unwanted users.

b. Explain the potential of your project to contribute to your problem domain. Discuss why this contribution is crucial.

Potential Contribution: Our project has the potential to make a meaningful impact in the car rental industry by addressing several key aspects. These contributions are crucial for the industry's growth and improvement:

- Efficiency: Make car rental operations smoother.
- Accuracy: Ensure data is correct.
- Smart Decisions: Help companies make better choices using data.
- Data Protection: Keep customer information safe.
- Cost Savings: Manage vehicles effectively to save money.
- Competitiveness: Support companies in staying strong in their market.

It's crucial because it improves how car rental companies work. This means smoother operations, happier customers, and cost savings. It also helps protect customer information and make smart decisions based on data. All of these are vital for a car rental business to succeed and stay competitive in their market.

II. Target user

A. Users of the Database:

- Clients: Those looking to rent a car utilize the database to make reservations, review reservations from the past, and determine the total cost.
- Staff of the vehicle Rental Company: The database is used by the staff of the vehicle rental company to man-

age reservations, view available automobiles, and record client information.
- Management: The executives utilize the database to examine information and decide on things like auto rental rates and the most popular vehicles.
- Maintenance Team: The database is used by the individuals who look after the automobiles to find out when they need to be fixed and how far they have been driven.

B. Administration of the Database:

The database administrator (DBA) is like the guardian of the database. They are the experts who take care of the database, ensuring it runs smoothly and secure. They take care of the following:

- Data maintenance: They ensure that the database is always operational and ready. They take care of any issues that arise.
- Security: They control who has access to the database and safeguard consumer information.
- Database updates: They make the necessary modifications to the database if they are needed.
- Performance optimization: They ensure that, even with a large amount of data, the database operates quickly.
- Data backup and recovery: In the event that something goes wrong, they create duplicates of the data to keep it safe.

III. ER Diagram and Table Details

Table Details:

1. Customer Table: The Customer table holds important information about people who use the business's services, like their names, contact details, and more. It helps the business keep track of who their customers are and how to reach them. This information is used for things like sending bills and offering special deals to loyal customers.

Attributes/Columns: The "Customer" table includes several key attributes/columns, such as:

1) Customer_id(Primary Key): A unique identifier for each customer.
2) FirstName: The first name of the customer.
3) LastName: The last name of the customer.
4) Email: The customer's email address.
5) Phone: Contact phone number.
6) Address: Customer's physical or mailing address.
7) City: The city where the customer resides.
8) Zipcode: The postal code or ZIP code of the customer's location.
9) DateOfBirth: The customer's date of birth.
10) Membership_id: A reference to a membership or loyalty program if applicable.
11) LicenseNumber: The customer's driver's license number.

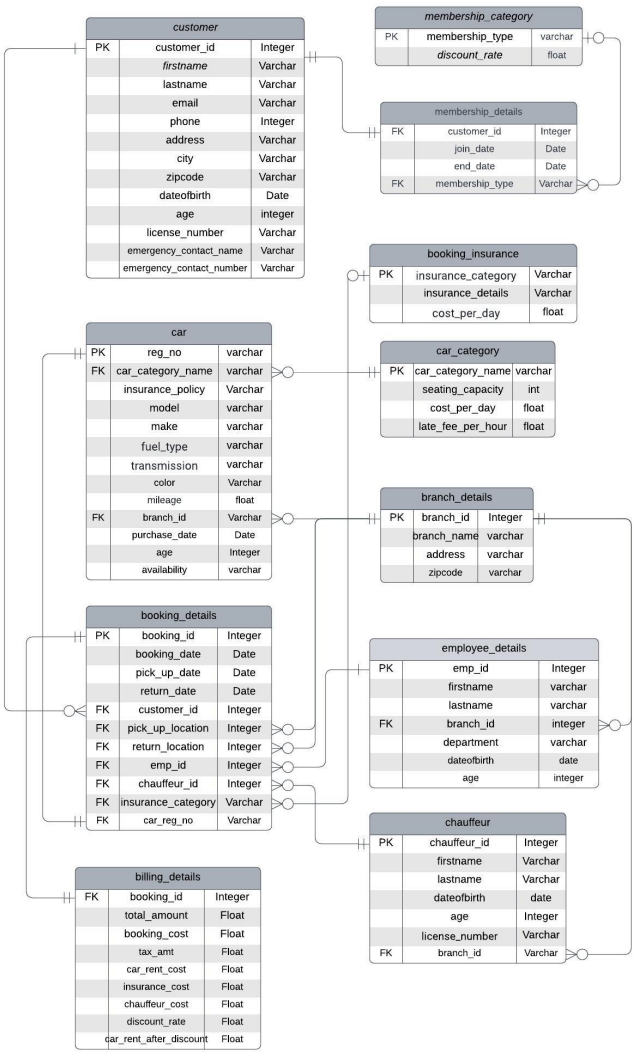


Fig. 1. Entity Relationship Diagram

12) EmergencyContactName: Name of the customer's emergency contact.
13) EmergencyContactPhone: Phone number of the emergency contact.

```
-- Constraint: billing_details_booking_id_fkey

-- ALTER TABLE IF EXISTS public.billing_details DROP CONSTRAINT IF EXISTS billing_details_booking_id_fkey;

ALTER TABLE IF EXISTS public.billing_details
    ADD CONSTRAINT billing_details_booking_id_fkey FOREIGN KEY (booking_id)
    REFERENCES public.booking_details (booking_id) MATCH SIMPLE
    ON UPDATE CASCADE
    ON DELETE CASCADE;
```

Fig. 2. SQL query to create customer table

This trigger is applied to the customer table. It calculates the customer's age from date of birth and populates the table when a new row is inserted.

2. Membership Details Table: The "Membership_details" table stores information about customers' membership or

```sql
-- Create a trigger to update the Age column
CREATE OR REPLACE FUNCTION update_customer_age()
RETURNS TRIGGER AS $$
BEGIN
    NEW.age := date_part('year', age(NEW.dateofbirth))::INT;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_customer_age_trigger
BEFORE INSERT OR UPDATE ON customer
FOR EACH ROW
EXECUTE FUNCTION update_customer_age();
```

Fig. 3. Trigger to populate age

loyalty program details. It helps the business keep track of who is a member, when they joined, when their membership expires, and what type of membership they have.

Attributes/Columns:

1) Customer_id (Foreign Key): This is a number that identifies which customer the membership details belong to.
2) Join_date: The date when the customer became a member.
3) End_date: The date when the customer's membership will expire.
4) Membership_type (Foreign Key): This links to a separate table that defines the different types of memberships available.

```sql
-- Create the membership_details table
CREATE TABLE membership_details (
    customer_id int references customer(customer_id),
    join_date DATE,
    end_date DATE,
    membership_type VARCHAR REFERENCES Membership_category(membership_type)
);
```

Fig. 4. SQL query to create membership details table

3. Membership Category Table: The "Membership_category" table serves as a catalog of various membership types that customers can choose from. Each membership type may offer different benefits or discounts. This table defines the characteristics of each membership type, such as its name and the associated discount rate.

Attributes/Columns:

1) Membership_type (Primary Key): This is a unique identifier for each membership type. It distinguishes one type of membership from another.
2) Discount_rate: This column specifies the discount rate or benefits associated with each membership type. It indicates how much of a discount or special treatment members receive.

4. Booking Insurance Table: The "Booking_insurance" table serves as a repository for details regarding insurance coverage options available to customers when booking services. It defines the types of insurance categories, provides descriptions

```sql
-- Create the Membership_category table
CREATE TABLE Membership_category (
    membership_type VARCHAR PRIMARY KEY,
    discount_rate FLOAT
);
```

Fig. 5. SQL query to create membership category table

of what each category covers, and specifies the cost associated with daily insurance coverage.

Attributes/Columns:

1) Insurance_category (Primary Key): This attribute is a unique identifier for each insurance category, distinguishing one type of coverage from another.
2) Insurance_details: This column contains a description of the specific coverage offered by each insurance category, outlining what is included or excluded from the insurance.
3) Cost_per_day: This attribute represents the cost associated with obtaining insurance coverage for a single day, helping customers understand the daily expense for added protection.

```sql
-- Create the booking_insurance
create table booking_insurance (
    insurance_category varchar primary key,
    insurance_details varchar,
    cost_per_day float
);
```

Fig. 6. SQL query to create booking insurance table

5. Car Table: The "Car" table serves as a repository for detailed information about individual cars in a business's fleet. It provides essential data about each car, including its unique registration number, model, make, characteristics like fuel type and color, mileage, and other features that are important for managing the fleet effectively.

Attributes/Columns:

1) Reg_no (Primary Key): This attribute serves as a unique identifier for each car and distinguishes one car from another.
2) Car_category_name (Foreign Key): This is a reference to a related table ("Car_category") and specifies the category or type to which the car belongs, defining its features and rental pricing.
3) Insurance_policy: Indicates the type of insurance policy that covers the car.
4) Model: The specific model name or number of the car.
5) Make: The manufacturer or brand of the car.
6) Fuel_type: Specifies the type of fuel the car uses, e.g., gasoline, diesel, electric, etc.

7) Transmission: Describes the transmission type, such as automatic or manual.
8) Color: Indicates the exterior color of the car.
9) Mileage: Records the total mileage or distance the car has traveled.
10) Branch_id (Foreign Key): Links to a ”Branch_details” table, specifying the branch or location where the car is stationed.
11) Purchase_date: Reflects the date when the car was acquired or purchased.
12) Age: Provides the age of the car, often calculated based on the purchase date.
13) Availability: Indicates whether the car is available for rent or is currently in use.

```
-- Define the 'fuel_type' enum type
CREATE TYPE fuel_type_enum AS ENUM ('Gasoline', 'Diesel', 'Electric', 'Hybrid');

-- Define the 'transmission' enum type
CREATE TYPE transmission_enum AS ENUM ('Automatic', 'Manual', 'Semi-Automatic');

-- Define the availability enum type
create type availability_enum as enum('Yes','No');

-- Create the car table using the 'fuel_type_enum' and 'transmission_enum'
CREATE TABLE car (
    reg_no VARCHAR PRIMARY KEY,
    car_category_name VARCHAR REFERENCES car_category(car_category_name),
    insurance_policy VARCHAR,
    model VARCHAR,
    make VARCHAR,
    fuel_type fuel_type_enum, -- Use the 'fuel_type_enum' as the data type
    transmission transmission_enum, -- Use the 'transmission_enum' as the data type
    color VARCHAR,
    mileage FLOAT,
    branch_id int REFERENCES branch_details(branch_id),
    purchase_date date,
    age int,
    availability availability_enum
);
```

Fig. 7. SQL query to create car table

This trigger is applied to the car table. It calculates the car's age from date of birth and populates the table when a new row is inserted.

```
-- Create a trigger to update the Age column
CREATE OR REPLACE FUNCTION update_car_age()
RETURNS TRIGGER AS $$
BEGIN
    NEW.age := date_part('year', age(NEW.purchase_date))::INT;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_car_age_trigger
BEFORE INSERT OR UPDATE ON car
FOR EACH ROW
EXECUTE FUNCTION update_car_age();
```

Fig. 8. trigger to populate age of car

6. Car Category Table: The ”Car_category” table serves as a classification system for the cars in a business's fleet. It defines various categories or types of cars, each with unique characteristics and pricing. This helps customers select the type of car that suits their needs and budget.

Attributes/Columns:

1) Car_category_name (Primary Key): This attribute serves as a unique identifier for each car category and distinguishes one category from another.
2) Seating_capacity: Specifies the number of passengers that the car in this category can accommodate.
3) Cost_per_day: Indicates the rental cost per day for cars in this category.
4) Late_fee_per_hour: Specifies the additional cost incurred per hour if a rental exceeds its agreed-upon return time.

```
-- Create the car_category table
create table car_category (
    car_category_name varchar primary key,
    seating_capacity int,
    cost_per_day float,
    late_fee_per_hour float
)
```

Fig. 9. Enter Caption

7. Branch_details Table: The ”Branch_details” table stores essential information about the different branches or physical locations operated by a business. Each row in this table represents a specific branch, providing details about its name, address, and ZIP code.

Attributes/Columns:

1) Branch_id (Primary Key): This attribute is a unique identifier for each branch, distinguishing one branch from another.
2) Branch_name: Specifies the name or identifier of the branch, often providing context about its location or purpose.
3) Address: Records the physical address of the branch, including street name, number, and any additional location-specific details.
4) Zipcode: Indicates the postal code or ZIP code associated with the branch's location.

```
-- Create the branch_details table
create table branch_details (
    branch_id serial primary key,
    branch_name varchar,
    address varchar,
    zipcode varchar
);
```

Fig. 10. SQL query to create Branch details table

8. Employee Details Table: The ”Employee_details” table acts as a repository for comprehensive data about the employ-

ees within a business. It contains details about each employee, including their personal information, the branch or location where they work, their department, date of birth, and age. (This table has a trigger that populates age of employee using DOB)

Attributes/Columns:

1) Emp_id (Primary Key): This attribute serves as a unique identifier for each employee, distinguishing one employee from another.
2) Firstname: Records the first name of the employee.
3) Lastname: Specifies the last name of the employee.
4) Branch_id (Foreign Key): This links to the ”Branch_details” table, indicating the branch or location where the employee is stationed.
5) Department: Describes the department or role of the employee within the organization.
6) Dateofbirth: Represents the date of birth of the employee.
7) Age: Provides the calculated age of the employee, often determined based on the date of birth.

```
-- Create the updated 'department_enum' type
CREATE TYPE department_enum_updated AS ENUM ('HR', 'Finance', 'IT', 'Sales', 'Maintenance', 'Security');


-- Create the employee_details table using the 'department' enum
CREATE TABLE employee_details (
    emp_id serial PRIMARY KEY,
    first_name VARCHAR,
    last_name VARCHAR,
    branch_id INT references branch_details(branch_id),
    dateofbirth DATE,
    age int,
    department department_enum_updated
);
```

Fig. 11. SQL query to create employee details table

9. Chauffer Table : The ”Chauffer” table serves as a repository for data related to chauffeurs or drivers who operate vehicles for a business. It includes details about each chauffeur, such as their personal information, date of birth, age, driver’s license number, and the branch or location they are affiliated with.

Attributes/Columns:

1) Chauffer_id (Primary Key): This attribute serves as a unique identifier for each chauffeur, distinguishing one chauffeur from another.
2) Firstname: Records the first name of the chauffeur.
3) Lastname: Specifies the last name of the chauffeur.
4) Dateofbirth: Represents the date of birth of the chauffeur.
5) Age: Provides the calculated age of the chauffeur, often determined based on the date of birth.
6) License_number: Indicates the unique driver’s license number for the chauffeur.
7) Branch_id (Foreign Key): This links to the ”Branch_details” table, indicating the branch or location where the chauffeur is stationed.

```
-- Create the chauffeur table
CREATE TABLE chauffeur (
    chauffeur_id SERIAL PRIMARY KEY,
    firstname VARCHAR,
    lastname VARCHAR,
    dateofbirth DATE,
    age int,
    license_number VARCHAR,
    branch_id INTEGER REFERENCES branch_details(branch_id)
);
```

Fig. 12. SQL query to create chauffeur table

10. Booking Details Table: The ”Booking_details” table serves as a central repository for recording and tracking customer bookings. It captures crucial details related to the booking process, including booking and pickup dates, return date, customer information, pickup and return locations, employee handling the booking, assigned chauffeur (if any), chosen insurance category, and the registered car number.

Attributes/Columns:

1) Booking_id (Primary Key): This attribute provides a unique identifier for each booking, distinguishing one booking from another.
2) Booking_date: Records the date when the booking was made.
3) Pick_up_date: Indicates the date when the customer plans to pick up the rented vehicle.
4) Return_date: Specifies the date when the customer intends to return the rented vehicle.
5) Customer_id (Foreign Key): This links to the ”Customer” table, identifying the customer associated with the booking.
6) Pick_up_location (Foreign Key): Links to the ”Branch_details” table, specifying the branch or location where the customer plans to pick up the vehicle.
7) Return_location (Foreign Key): Links to the ”Branch_details” table, indicating the branch or location where the customer plans to return the vehicle.
8) Emp_id (Foreign Key): Connects to the ”Employee_details” table, identifying the employee responsible for handling the booking.
9) Chauffer_id (Foreign Key): Links to the ”Chauffer” table and identifies the assigned chauffeur (if any) for the booking.
10) Insurance_category (Foreign Key): Connects to the ”Booking_insurance” table, specifying the insurance category chosen by the customer.
11) Car_reg_no (Foreign Key): Links to the ”Car” table, indicating the registered number of the vehicle being rented.

Trigger checks: This trigger is responsible for checking various conditions when a new row is inserted into the `booking_details` table. Here’s a description of the checks performed by this trigger:

```sql
-- Create the booking_details table
CREATE TABLE booking_details (
    booking_id serial PRIMARY KEY,
    booking_date DATE,
    pick_up_date DATE,
    return_date DATE,
    customer_id INTEGER REFERENCES customer(customer_id),
    pick_up_location int references branch_details(branch_id),
    return_location int references branch_details(branch_id),
    emp_id INTEGER REFERENCES employee_details(emp_id),
    chauffeur_id INTEGER references chauffeur(chauffeur_id),
    insurance_category VARCHAR REFERENCES booking_insurance(insurance_category),
    car_reg_no varchar references car(reg_no)
);
```

Fig. 13. SQL query to create booking details table

1) Return Date Check: It ensures that the `return_date` cannot be less than the `pick_up_date`, which makes sense since a return date should be equal to or after the pick-up date.
2) Pick-up Date Check: It verifies that the `pick_up_date` is either today or in the future, ensuring that customers can only make bookings for the present or future dates.
3) Car Availability Check: It checks if the car is available. This check ensures that the selected car is not already booked during the specified pick-up and return dates.
4) Car's Branch Matching: It ensures that the branch of the selected car matches the specified pick-up location. This guarantees that the customer is picking up the car from the correct branch.
5) Employee's Branch Matching: It checks if the branch of the assigned employee matches the specified pick-up location. This confirms that the employee is available at the correct branch to assist with the booking.
6) Employee's Department Check: It verifies that the assigned employee's department is "sales." This check ensures that only employees from the sales department can be assigned to this type of booking. (Trigger images Fig. 14,15)

```sql
-- Create the PostgreSQL trigger
CREATE OR REPLACE FUNCTION validate_booking()
RETURNS TRIGGER AS $$
declare car_branch_id int;
declare car_availability varchar;
declare employee_branch_id int;
declare employee_department varchar;
BEGIN
    -- Check if the pick-up date is today or in the future
    IF NEW.pick_up_date < current_date THEN
        RAISE EXCEPTION 'Pick-up date must be today or in the future.';
    END IF;

    -- check if return_date is before pick_up date
    if new.return_date<new.pick_up_date then
        raise exception 'Return date cannot be before pick-up date';
    end if;

    -- Check car availability
    SELECT branch_id, availability INTO STRICT car_branch_id, car_availability
    FROM car
    WHERE reg_no = NEW.car_reg_no;

    IF car_branch_id != NEW.pick_up_location THEN
        RAISE EXCEPTION 'Car branch and pick-up location do not match.';
    END IF;

    IF car_availability != 'Yes' THEN
        RAISE EXCEPTION 'Car is not available.';
    END IF;

```

Fig. 14. Trigger checking for booking details part 1

```sql
    -- Check employee's branch and department
    SELECT branch_id,department INTO STRICT employee_branch_id,employee_department
    FROM employee_details
    WHERE emp_id = NEW.emp_id;

    IF employee_branch_id != NEW.pick_up_location THEN
        RAISE EXCEPTION 'Employee''s branch does not match pick-up location.';
    END IF;

    if employee_department != 'Sales' then
        raise exception 'Employee must be of a Sales Background';
    end if;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Create the trigger on the "booking_details" table
CREATE TRIGGER validate_booking_trigger
BEFORE INSERT ON booking_details
FOR EACH ROW
EXECUTE FUNCTION validate_booking();
```

Fig. 15. Trigger checking for booking details part 2

11. Billing Details Table: The "Billing_details" table serves as a repository for detailed billing information associated with customer bookings. It captures various cost components, including the total amount, booking cost, tax amount, car rental cost, insurance cost, chauffeur cost, discount rate applied to the booking, and the final car rental cost after applying the discount. We have included triggers that automatically populate the costs based on information from the "Booking_details" table. This automation streamlines the billing process and ensures accuracy in cost calculation

Attributes/Columns:

1) Booking_id (Foreign Key): This attribute links to the "Booking_details" table, associating each billing entry with a specific booking.
2) Total_amount: Represents the overall total cost of the booking, including all associated charges.
3) Booking_cost: Indicates the cost of the booking itself, reflecting the base charge before any additional costs.
4) Tax_amt: Records the tax amount applied to the booking, which contributes to the total cost.
5) Car_rent_cost: Specifies the cost of renting the car, which may include a base rate and any additional charges.
6) Insurance_cost: Captures the cost of insurance coverage chosen by the customer for the booking.
7) Chauffer_cost: Reflects the cost associated with hiring a chauffeur (if selected by the customer).
8) Discount_rate: Represents the discount rate applied to the booking, which can reduce the total cost.
9) Car_rent_after_discount: Indicates the final cost of renting the car after applying the discount, taking into account all costs and reductions.

Trigger: This trigger is activated when the booking details table is filled. This trigger is responsible for calculating and inserting values into the `billing_details` table based on

```sql
-- Create the billing_details table
CREATE TABLE billing_details(
    booking_id int references booking_details(booking_id),
    total_amount float,
    booking_cost float,
    insurance_cost float,
    chauffeur_cost float,
    car_rent_cost float,
    discount_rate float,
    car_rent_after_discount float,
    tax_amt float
);
```

Fig. 16. SQL query to create billing details table

various factors. Here's a description of what this trigger does:

1) Car Rent Cost Calculation: It calculates the car rent cost as the number of days the car is booked multiplied by the car's daily rate.
2) Insurance Cost Calculation: It computes the insurance cost as the insurance category's daily rate multiplied by the number of days the car is booked.
3) Chauffeur Cost Calculation: If a chauffeur is assigned, it calculates the chauffeur cost as the chauffeur's daily rate (which is $75 for all chauffeurs) multiplied by the number of days the car is booked. If no chauffeur is assigned, the chauffeur cost is set to 0.
4) Membership Discount Application: If the customer has a membership, it applies a membership discount to the car rent cost. The discount is determined based on the type of membership the customer holds.
5) Booking Cost Calculation: It calculates the total booking cost, which is the sum of the car rent cost after applying the membership discount, the insurance cost, and the chauffeur cost.
6) Tax Amount Calculation: It calculates the tax amount, which is 10% of the booking cost.
7) Total Cost Calculation: It computes the total cost, which is the booking cost plus the tax amount. (Trigger images Fig. 17,18)

```sql
-- Create the trigger function
CREATE OR REPLACE FUNCTION fill_billing_details()
RETURNS TRIGGER AS $$
declare car_rent_cost float;
declare insurance_cost float;
declare chauffeur_cost float;
declare booking_cost float;
declare tax_amt float;
declare total_amount float;
declare discount_rate float;
declare car_rent_after_discount float;
BEGIN
    -- Calculate car_rent_cost
    SELECT (NEW.return_date - NEW.pick_up_date) * cc.cost_per_day
    INTO STRICT car_rent_cost
    FROM car_category cc
    WHERE cc.car_category_name = (SELECT car_category_name FROM car WHERE reg_no = NEW.car_reg_no);

    -- Calculate insurance cost
    SELECT (NEW.return_date - NEW.pick_up_date) * booking_insurance.cost_per_day
    into strict insurance_cost
    from booking_insurance
    where booking_insurance.insurance_category = new.insurance_category;

    --calculate chauffeur cost
    if new.chauffeur_id is NULL then
        chauffeur_cost:=0;
    else
        chauffeur_cost := (NEW.return_date - NEW.pick_up_date) * 75;
    end if;

    -- Check customer membership for discount
    SELECT mc.discount_rate
    INTO STRICT discount_rate
    FROM customer c
    LEFT JOIN membership_details md ON c.customer_id = md.customer_id
    LEFT JOIN Membership_category mc ON md.membership_type = mc.membership_type
    WHERE c.customer_id = NEW.customer_id;

```

Fig. 17. Trigger that populates billing details table part 1

```sql
    -- calculate car_rent_after_discount
    car_rent_after_discount := car_rent_cost - (discount_rate * 0.01 * car_rent_cost);

    -- Calculate booking cost
    booking_cost :=car_rent_after_discount + insurance_cost + chauffeur_cost;

    -- Calculate tax_amt
    tax_amt := 0.1 * booking_cost;

    -- Calculate total_amount
    total_amount := booking_cost + tax_amt;


    -- Insert data into the "billing_details" table
    INSERT INTO billing_details (booking_id, total_amount, booking_cost, tax_amt,car_rent_cost,
                                 chauffeur_cost,insurance_cost,car_rent_after_discount,discount_rate)
    VALUES (NEW.booking_id, total_amount, booking_cost, tax_amt,car_rent_cost,
                                chauffeur_cost,insurance_cost,car_rent_after_discount,discount_rate);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Create the trigger on the "booking_details" table
CREATE TRIGGER fill_billing_details_trigger
AFTER INSERT ON booking_details
FOR EACH ROW
EXECUTE FUNCTION fill_billing_details();
```

Fig. 18. Trigger that populates billing details table part 2

IV. Table Relationships

1. Car to car category:

Every car is associated with one car category, which will give details about the car seating capacity, per day renting costs, etc.

2. Car to branch details:

Each car has a branch id. Customers can pick up cars only from this location. Branch id of the car is updated when the car is dropped off at a different location.

3. Customer to membership category:

Each customer has a row in membership details table along with the customers membership type which will determine the discount to be applied for each customer when billing.

4. Employee details to branch details: Each employee belongs to a branch. An sales employee from one branch can

help a customer book a car that is located in employee's branch only.

5. Chauffeur to branch details:

Each chauffeur is based in one branch. The customers pick up location and chauffer's branch id must be same.

6. Booking details to (customer, car, branch details, chauffeur, booking insurance, employee details):

Each booking is associated with one car (car reg no) , one employee from sales who is responsible for the booking, one customer who has booked the car, pick up and return location (branch id's), chauffeur may or may not be booked, one type of booking insurance

7. Billing details to booking details:

Billing table is a week entity, which uses booking id, the primary key from booking details table as its foreign key. When a new row is inserted to booking details, billing details also gets inserted with the booking id and calculated costs.

V. Functional Dependency and BCNF Analysis

BCNF (Boyce-Codd Normal Form) Analysis:

- BCNF is a higher level of normalization for relational database tables. A table is in BCNF if, for every non-trivial functional dependency X→Y, X is a superkey. Here's what that means:
  - A superkey is a set of attributes that uniquely identifies each row in a table.
  - Non-trivial means that X and Y are not the same (i.e., Y is not a subset of X), and they are not both empty.
- In simpler terms, a table is in BCNF if, for every functional dependency in the table, the left-hand side (X) is a superkey. This ensures that there is no redundancy in the data and that updates, insertions, and deletions can be performed without anomalies.

a) Membership_category Table::

- BCNF Status: Meets BCNF.
- Dependencies: membership_type→discount_rate
- Explanation: The functional dependency "membership_type→discount_rate" is valid, and "membership_type" can be considered a superkey. Therefore, this table is in BCNF.

b) Booking_insurance Table::

- BCNF Status: Meets BCNF.
- Dependencies:
  - insurance_category→insurance_details
  - insurance_category→cost_per_day
- Explanation: The functional dependencies "insurance_category→insurance_details" and "insurance_category→cost_per_day" are valid, with "insurance_category" as a superkey. Thus, the table is in BCNF.

c) Car_category Table::

- BCNF Status: Meets BCNF.
- Dependencies:
  - car_category_name→Seating capacity
  - car_category_name→Cost_per_day
  - car_category_name→Late_fee_per_hour
- Explanation: The functional dependencies "car_category_name→Seating capacity," "car_category_name→Cost_per_day," and "car_category_name→Late_fee_per_hour" are valid, with "car_category_name" as a superkey. This table satisfies BCNF.

d) Branch_details Table::

- BCNF Status: Meets BCNF.
- Dependencies:
  - branch_id→branch_name
  - branch_id→address
  - branch_id→zipcode
- Explanation: As "branch_id" is assumed to be unique, the functional dependencies "branch_id -¿ branch_name," "branch_id→address," and "branch_id→zipcode" are valid. Consequently, the table is in BCNF.

e) Customer Table::

- BCNF Status: Meets BCNF.
- Dependencies:
  - customer_id→first_name
  - customer_id→last_name
  - customer_id→email
  - customer_id→phone_no
  - customer_id→zipcode
  - customer_id→dateofbirth
  - customer_id→membership_id
  - customer_id→license_number
  - customer_id→emergency_contact_number
- Explanation: The functional dependencies involving "customer_id" are all valid, with "customer_id" as a superkey. Therefore, the table complies with BCNF.

f) Billing_details Table (Weak Entity)::

- BCNF Status: Meets BCNF.
- Dependency: billing_id (FK)→(Booking_details)
- Explanation: Since the "Billing_details" table is a weak entity related to the strong entity "Booking_details," and

the ”Booking_details” table already meets BCNF, the ”Billing_details” table also adheres to BCNF.

g) Booking_details Table::

- BCNF Status: Meets BCNF.
- Dependencies:
  - booking_id→booking_date
  - booking_id→pickup_date
  - booking_id→return_date
  - booking_id→insurance_company
  - booking_id→pickup_location
  - booking_id→return_location
  - booking_id→billing_id (FK)
  - booking_id→chauffeur_id
- Explanation: All the functional dependencies that include ”booking_id” are valid, making it a superkey. Hence, the ”Booking_details” table is in BCNF.

h) Car Table::

- BCNF Status: Meets BCNF.
- Dependencies:
  - reg_no (PK)→car_category_name
  - reg_no→insurance_policy
  - reg_no→model
  - reg_no→fuel_type
  - reg_no→transmission
  - reg_no→color
  - reg_no→mileage
  - reg_no→branch_id
  - reg_no→purchase_date
  - reg_no→age
  - reg_no→availability
- Explanation: The functional dependencies involving ”reg_no” are valid, and ”reg_no” can be considered a super key. Therefore, this table satisfies BCNF.

i) Chauffeur Table::

- BCNF Status: Meets BCNF.
- Dependencies:
  - chauffeur_id→first_name
  - chauffeur_id→last_name
  - chauffeur_id→dateofbirth
  - chauffeur_id→age
  - chauffeur_id→license_number
  - chauffeur_id→branch_id (FK)
- Explanation: The functional dependencies related to ”chauffeur_id” are valid, with ”chauffeur_id” as a superkey. Thus, the table adheres to BCNF.

j) Employee_details Table::

- BCNF Status: Meets BCNF.
- Dependencies:
  - emp_id→firstname
  - emp_id→lastname
  - emp_id→branch_id
  - emp_id→department
  - emp_id→dateofbirth
  - emp_id→age
- Explanation: All the functional dependencies that involve ”emp_id” are valid, and ”emp_id” can be considered a superkey. Therefore, this table is in BCNF.

k) Membership_details Table (Weak Entity)::

- BCNF Status: Meets BCNF.
- Dependencies:
  - customer_id (FK)→(Customer)
  - membership_type (FK)→(Membership_category)
- Explanation: Since the ”Membership_details” table is a weak entity related to the strong entities ”Customer” and ”Membership_category,” and both of these tables are in BCNF, the ”Membership_details” table also complies with BCNF.

VI. Effects of Primary Key Deletion

1) Billing Details Table: If you delete a record in the `booking_details` table (for instance, if a booking is canceled and removed from the system), the corresponding record in the `billing_details` table, linked to the same `booking_id`, will also be deleted. This ensures that no orphaned or incorrect data remains in the `billing_details` table. (Fig. 19)

```
-- Constraint: billing_details_booking_id_fkey

-- ALTER TABLE IF EXISTS public.billing_details DROP CONSTRAINT IF EXISTS billing_details_booking_id_fkey;

ALTER TABLE IF EXISTS public.billing_details
    ADD CONSTRAINT billing_details_booking_id_fkey FOREIGN KEY (booking_id)
    REFERENCES public.booking_details (booking_id) MATCH SIMPLE
    ON UPDATE CASCADE
    ON DELETE CASCADE;
```

Fig. 19. Delete Cascade on Billing details

2) Booking Details Table: A trigger is set up to create a copy of a row in the ”booking details” table whenever a row in that table is deleted. This copy is then stored in a new table called ”deleted_booking_details. The trigger is configured to activate when a row in the ”booking details” table is deleted. It copies all the information from the deleted row and inserts it into a new table named ”deleted_booking_details.” This effectively creates a backup or archive of the deleted data. This can be valuable for auditing, troubleshooting, or any other reference purposes in the future. If there is ever a need to look back and see what was deleted and when, the ”deleted_booking_details” table provides that historical data. (Fig. 20,21)

```
-- FUNCTION: public.copy_to_deleted_booking()

-- DROP FUNCTION IF EXISTS public.copy_to_deleted_booking();

CREATE OR REPLACE FUNCTION public.copy_to_deleted_booking()
    RETURNS trigger
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE NOT LEAKPROOF
AS $BODY$
BEGIN
    -- Copy the row to deleted_booking_details
    INSERT INTO deleted_booking_details (booking_id, booking_date, pick_up_date, return_date, customer_id,
        pick_up_location, return_location, emp_id, chauffeur_id, insurance_category, car_reg_no)
    SELECT OLD.booking_id, OLD.booking_date, OLD.pick_up_date, OLD.return_date, OLD.customer_id,
        OLD.pick_up_location, OLD.return_location, OLD.emp_id, OLD.chauffeur_id, OLD.insurance_category, OLD.car_reg_no;

    -- Delete the row from booking_details
    DELETE FROM booking_details WHERE booking_id = OLD.booking_id;

    RETURN OLD;
END;
$BODY$;

ALTER FUNCTION public.copy_to_deleted_booking()
    OWNER TO postgres;
```

Fig. 20. Function to copy rows on delete

```
-- Trigger: copy_to_deleted_booking_trigger

-- DROP TRIGGER IF EXISTS copy_to_deleted_booking_trigger ON public.booking_details;

CREATE OR REPLACE TRIGGER copy_to_deleted_booking_trigger
    AFTER DELETE
    ON public.booking_details
    FOR EACH ROW
    EXECUTE FUNCTION public.copy_to_deleted_booking();
```

Fig. 21. Trigger to copy rows to deleted booking details table

3) Delete Cascade on Booking Table: When a change occurs in the `chauffeur` table, it is specified to cascade the effect of that change to the `booking_details` table. This means that if, for example, a `chauffeur_id` is updated or deleted in the `chauffeur` table, the same update or deletion will automatically be applied to the related records in the `booking_details` table. (Fig. 22)

```
-- Constraint: booking_details_chauffeur_id_fkey

-- ALTER TABLE IF EXISTS public.booking_details DROP CONSTRAINT IF EXISTS booking_details_chauffeur_id_fkey;

ALTER TABLE IF EXISTS public.booking_details
    ADD CONSTRAINT booking_details_chauffeur_id_fkey FOREIGN KEY (chauffeur_id)
    REFERENCES public.chauffeur (chauffeur_id) MATCH SIMPLE
    ON UPDATE CASCADE
    ON DELETE CASCADE;
```

Fig. 22. Delete/Update cascade for Chauffer ID

The cascading actions are applied to all foreign keys in the `booking_details` table, including `car_reg_no`, `chauffeur_id`, `customer_id`, `insurance_category`, `pick_up_location`, and `return_location` .

4) Employee Deletion: When an employee is deleted from the employee records their `emp_id` is removed from the database. The `booking_details` table contains information about bookings and a reference to the employee who handled the booking. This reference is typically stored as an `emp_id`. Instead of deleting the entire booking when an employee is removed, the strategy is to set the `emp_id` in the `booking_details` table to null. This means that the reference to the employee who managed the booking becomes null or empty. This approach is designed to preserve customer bookings. (Fig. 23)

Query Query History

```
-- Constraint: booking_details_emp_id_fkey

-- ALTER TABLE IF EXISTS public.booking_details DROP CONSTRAINT IF EXISTS booking_details_emp_id_fkey

ALTER TABLE IF EXISTS public.booking_details
    ADD CONSTRAINT booking_details_emp_id_fkey FOREIGN KEY (emp_id)
    REFERENCES public.employee_details (emp_id) MATCH SIMPLE
    ON UPDATE CASCADE
    ON DELETE SET NULL;
```

Fig. 23. Emp id Null on Deletion

VII. Data In Tables

Due to the specific requirements of the project and the unavailability of a suitable external data source, the data was manually generated and inserted into the tables. The data has been populated using a custom Python script. The Python script was designed to generate realistic and meaningful data for each table, ensuring that the database accurately represents the car rental operations.

1) Membership_category table:

	membership_type [PK] character varying	discount_rate double precision
1	bronze	3
2	gold	10
3	no_membership	0
4	platinum	15
5	silver	6

Fig. 24. Membership Category Table

2) Branch_details table:

	branch_id [PK] integer	branch_name character varying	address character varying	zipcode character varying
1	41	Walker, Fisher and Owens	2619 Sarah Valley	68116
2	42	Garcia-Gutierrez	43673 Zachary Fields	86451
3	43	Spencer, Thomas and Montoya	39705 Hill Landing Apt. 330	74821
4	44	Garcia and Sons	3709 Bradley Parks Apt. 946	64805
5	45	Ross Ltd	5340 Hunt Knolls	17923
6	46	Norton-Kelley	1602 Bailey Flats Apt. 528	63793
7	47	Fowler, Stuart and Brock	2792 Ronald Loaf Suite 573	10377
8	48	Brewer, Fox and Sanchez	634 Kyle Overpass	17724
9	49	Williams-Gonzalez	1154 Villa Grove Suite 722	60184
10	50	Green Group	278 Kimberly Mountain	97992

Fig. 25. Branch details Table

3) Car_category table:

	car_category_name [PK] character varying	seating_capacity integer	cost_per_day double precision	late_fee_per_hour double precision
1	Convertible	2	150	30
2	Coupe	2	110	22
3	Hatchback	4	70	14
4	Pickup	4	95	19
5	Sedan	5	80	16
6	SUV	7	100	20
7	Van/Minivan	8	120	24
8	Wagon	6	90	18

Fig. 26. Car category Table

4) Car Table: Count of cars from each Branch

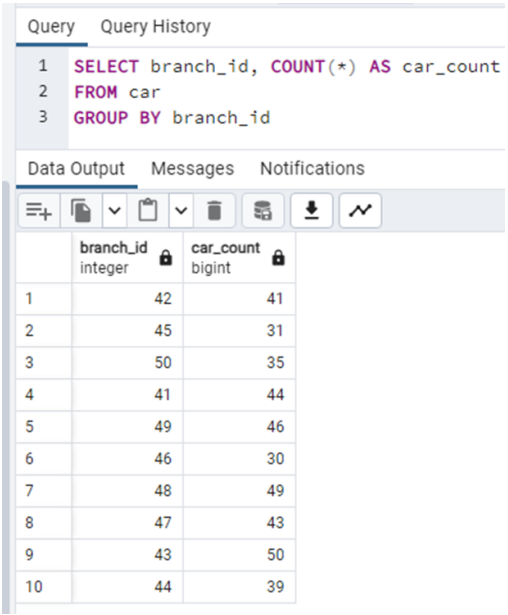


Fig. 27. Car Table

5) Car Category table: Count of cars in each car category

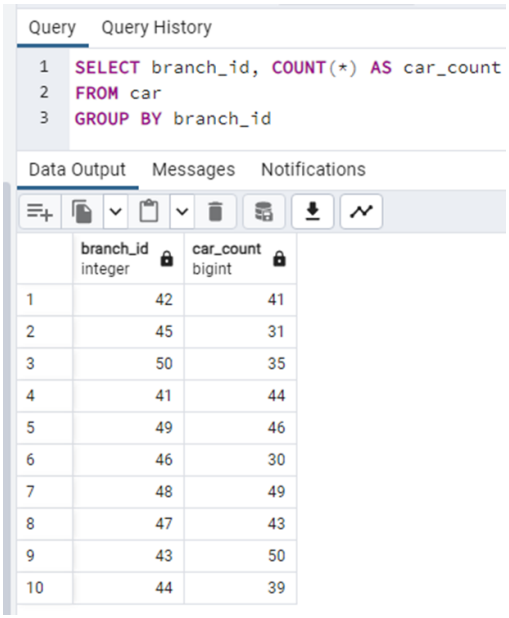


Fig. 28. Number of Cars in each category

6) Chauffer Table: Count of Chauffers from each Branch

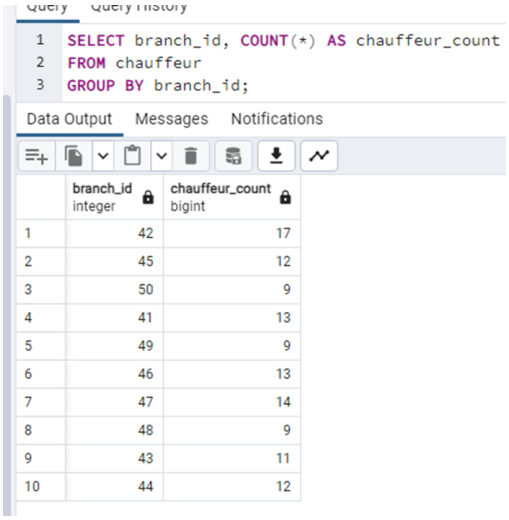


Fig. 29. Count of Chauffer from each Branch

7) Booking insurance Table:

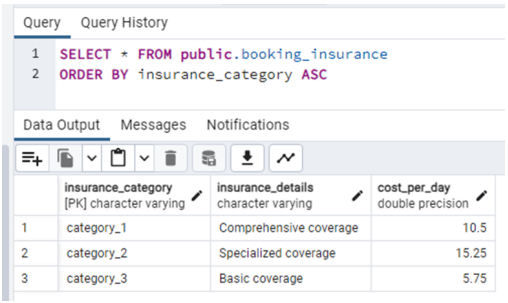


Fig. 30. Booking Insurance Table

8) Customer Table Total Number of Customers from Customers Table

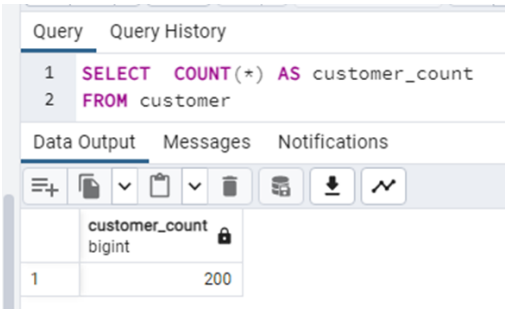


Fig. 31. Customers Table

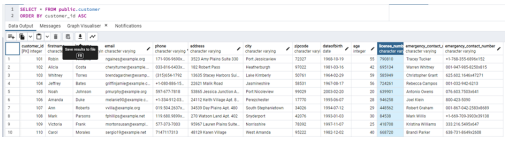


Fig. 32. Customers Table

9) Membership_details table: Number of Customers for Each Membership Type

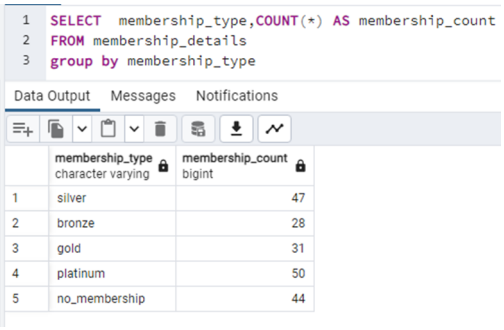


Fig. 33. Membership Details Table

10) Employee Details Table: Number of Employees in each Branch

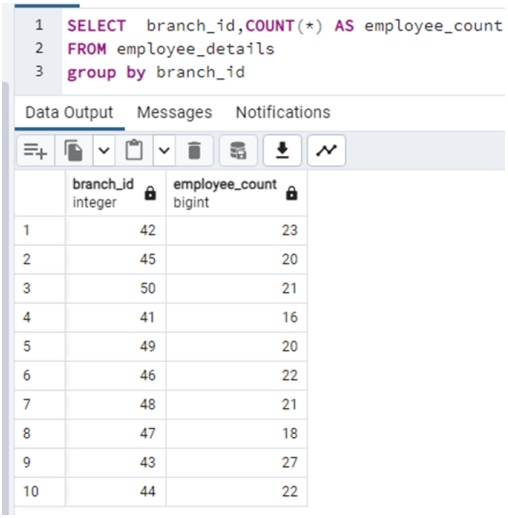


Fig. 34. Employee Details Table

Number of Employees in each Department

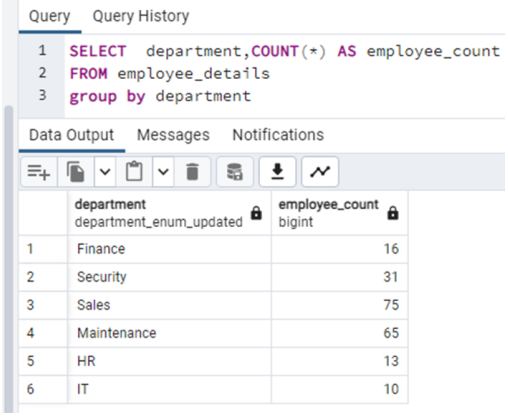


Fig. 35. Employee Details by Department

VIII. Challenges with Large Datasets and Indexing Strategies

Handling larger datasets posed problems like slower query execution times, these could be mitigated by implementing efficient indexing strategies.

Indexing Techniques Used:

1) Primary Key Indexing: Each table like booking details, customer, car, etc., has primary keys (booking id, customer id, reg no, etc.) which are automatically indexed. This is essential for quick lookups and joins on these keys.

2) Foreign Key Indexing: Indexing foreign keys, such as those in the booking details table referencing customer, car, and others, helped speed up join operations.

3) Index on branch id in car and employee details: This index speeds up queries that join these tables with branch details, as it allows quick lookups of cars and employees based on their associated branch.

4) Index on pick up location in booking details: This optimizes searches and joins where the pickup location is a filter criterion, reducing query time for location-based booking queries.

5) Index on booking id in booking details and billing details: As a primary key in booking details and a foreign key in billing details, this index ensures efficient joins and data retrieval for billing information related to specific bookings.

6) Index on car reg no in booking details: This index is crucial for queries that filter or report based on specific cars, enhancing the performance of such searches.

7) Index on lastname in customer: It significantly speeds up queries searching for bookings by customer last names, making these operations more efficient.

Example of where indexing helps:

SELECT pick_up_date, return_date FROM booking_details WHERE car_reg_no = '326 KAS';

This query is used everytime we try to insert a new row in booking_details table to make sure that the new booking will not collide with the previous booking date.

Here there is a seq scan to get all the booking dates. This is the simplest form of scanning. It reads every row in a table sequentially. It's generally used when a large portion of the table needs to be scanned, as it's faster in such cases than repeatedly accessing the index and then fetching data from the table. CREATE INDEX idx_booking_details_car_reg_no

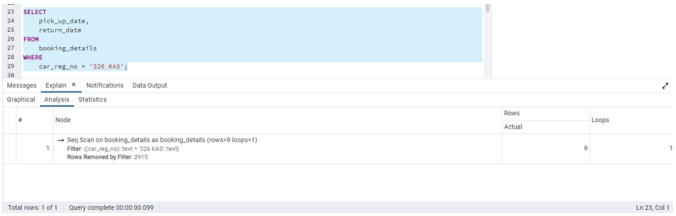


Fig. 36. Before Indexing

ON booking_details (car_reg_no);

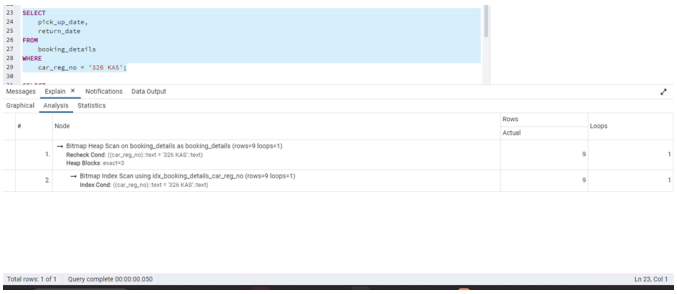


Fig. 37. After Indexing

After indexing the execution time reduces on avg by 20 ms, as it performs a bitmap heap scan. This method is used when an index can be used to quickly identify the rows to be retrieved. First, a Bitmap Index Scan is performed to create a bitmap of row locations in the table. Then, the Bitmap Heap Scan uses this bitmap to efficiently retrieve only the necessary rows. This method is faster than a Seq Scan when only a small subset of rows needs to be retrieved, especially when these rows are spread out across the table.

IX. Database Testing and Query Executions

A. Insert Queries

- Inserting into Booking Details:

  INSERT INTO booking details (booking date, pick up date, return date, customer id, pick up location, return location, emp id, chauffeur id, insurance category, car reg no) VALUES ('2023-09-01', '2023-09-05', '2023-09-10', 1111, 47, 47, 137, 10, 'category 1', '1-I8679');

```
-- Constraint: billing_details_booking_id_fkey

-- ALTER TABLE IF EXISTS public.billing_details DROP CONSTRAINT IF EXISTS billing_details_booking_id_fkey;

ALTER TABLE IF EXISTS public.billing_details
    ADD CONSTRAINT billing_details_booking_id_fkey FOREIGN KEY (booking_id)
    REFERENCES public.booking_details (booking_id) MATCH SIMPLE
    ON UPDATE CASCADE
    ON DELETE CASCADE;
```

Fig. 38. Insert into Booking Details

```
-- Constraint: billing_details_booking_id_fkey

-- ALTER TABLE IF EXISTS public.billing_details DROP CONSTRAINT IF EXISTS billing_details_booking_id_fkey;

ALTER TABLE IF EXISTS public.billing_details
    ADD CONSTRAINT billing_details_booking_id_fkey FOREIGN KEY (booking_id)
    REFERENCES public.booking_details (booking_id) MATCH SIMPLE
    ON UPDATE CASCADE
    ON DELETE CASCADE;
```

Fig. 39. Checking the inserted row

Inserting into Booking Details: This query adds a new booking record to the 'booking details' table with specific details like booking date, pick-up and return dates, customer ID, locations, employee ID, chauffeur ID, insurance category, and car registration number.

- Inserting into Car:

  INSERT INTO car (reg no, car category name, insurance policy, model, make, fuel type, transmission, color, mileage, branch id, purchase date, availability) VALUES ('8VA UFO', 'Sedan', 'POL-0246', 'Zonda R', 'Pagani', 'Gasoline', 'Automatic', 'Red', 50000, 48, '2023-01-15', 'Yes');

Fig. 40. Insert into Car table

Fig. 41. Checking the inserted row in car table

Inserting into Car: This query adds a new car to the 'car' table, specifying details such as registration number, category, insurance policy, model, make, fuel type, transmission, color, mileage, branch ID, purchase date, and availability..

B. Delete Queries

- Deleting from Booking Details:

  DELETE FROM booking details WHERE booking id = 201;

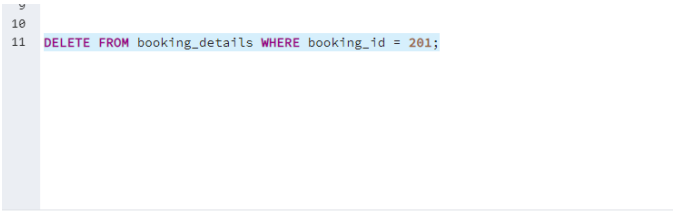


Fig. 42. deleted from booking details

Fig. 43. Checking the deleted row in booking details table

Fig. 44. Delete on cascade deletes the booking from billing details table as well

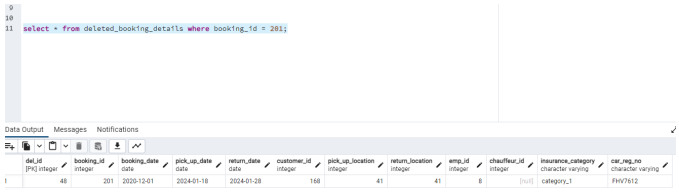


Fig. 45. The deleted booking gets stored in the deleted booking details table and preserves the history of deletion

Deleting from Booking Details: This query removes a specific booking (identified by booking ID) from the 'booking details' table. Related records in the 'billing details' table are also deleted due to the cascade delete setup, and a copy of the deleted booking is stored in the 'deleted booking details' table.

- Deleting from customer:

  DELETE FROM customer WHERE customer id = 102;

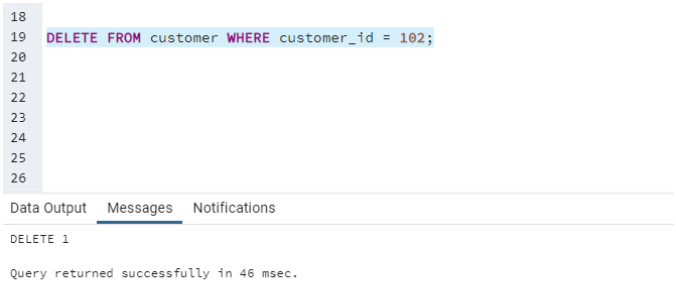


Fig. 46. Deleted from customer table

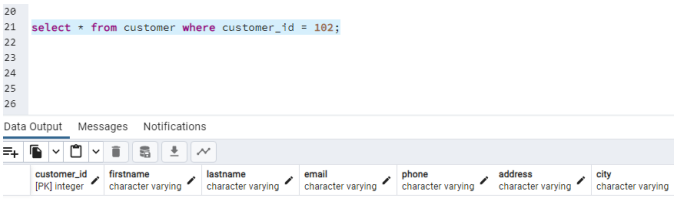


Fig. 47. Checking the deleted row in customer table

Fig. 48. The deleted customers booking details also gets deleted

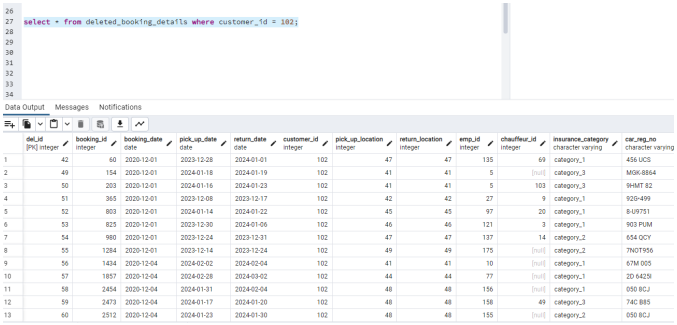


Fig. 49. After deletion, the triggers stores the details in the deleted booking details table

Deleting from Customer: This query deletes a customer (identified by customer ID) from the 'customer' table. Bookings associated with this customer are also deleted, and these deleted bookings are then stored in the 'deleted booking details' table

C. Update Queries

- Updating in car table:

  UPDATE car SET reg no = '029 AAA' WHERE insurance policy = 'POL-7471';

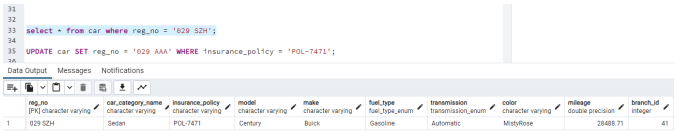


Fig. 50. Before updating the reg no in the Car table

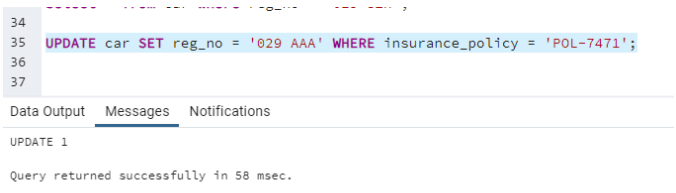


Fig. 51. Update query run for changing Reg number

Updating in Car Table: This query updates the registration number of cars in the 'car' table that are under a specific insurance policy. The update also cascades to the 'booking details' table, reflecting the new registration number.

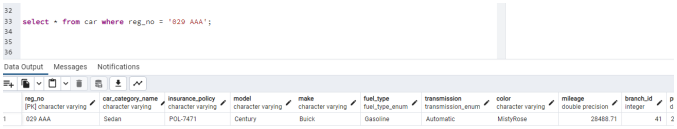


Fig. 52. Checking the record after update is performed

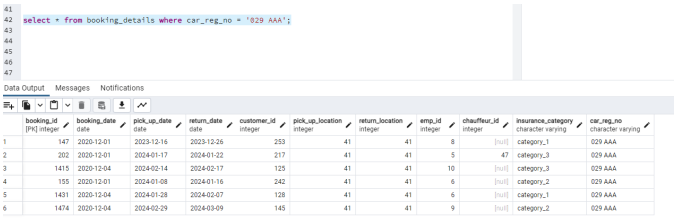


Fig. 53. Car reg number changes cascaded to the booking details table as well

- Updating in Chauffeur table:

  UPDATE chauffeur SET chauffeur id = 200 WHERE firstname='Donna' and lastname='Adams';

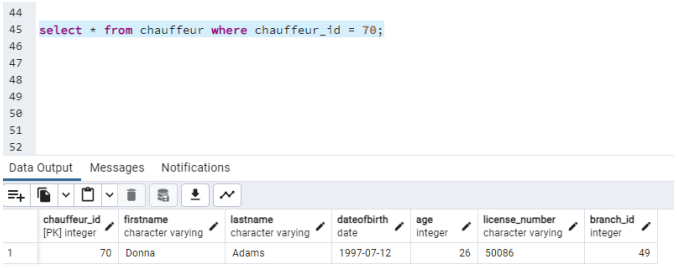


Fig. 54. Before updating chauffeur id 70

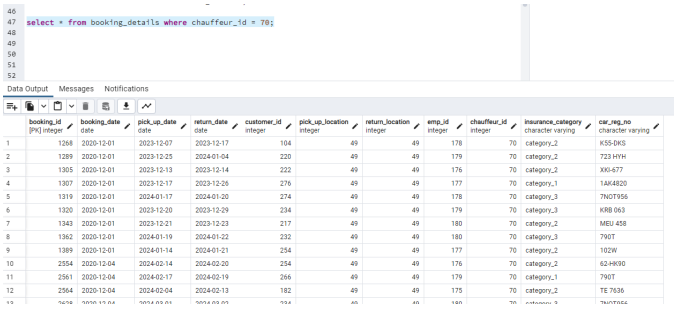


Fig. 55. Booking details table before updating chauffeur id 70

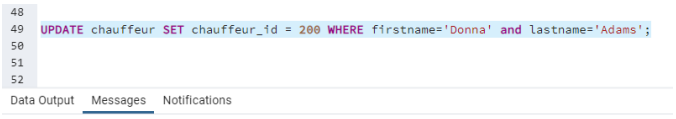


Fig. 56. Updating the chauffeur id from 70 to 200

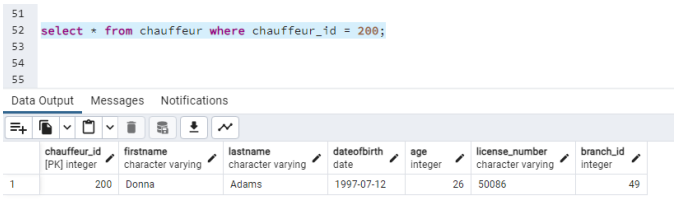


Fig. 57. Chauffeur table after updating

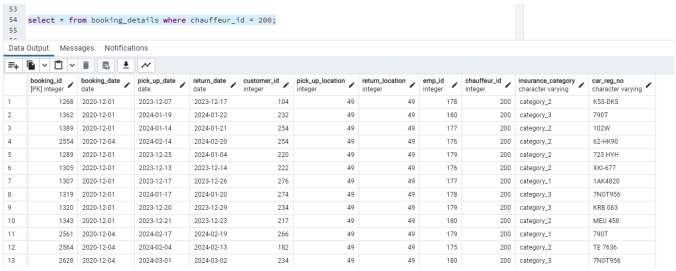


Fig. 58. Update cascaded to Booking details table showing that chauffeur ID changed in all existing booking

Updating in Chauffeur Table: This query changes the ID of a chauffeur, identified by first and last name, to a new ID. The change is also reflected in the 'booking details' table, updating the chauffeur ID for all relevant bookings.

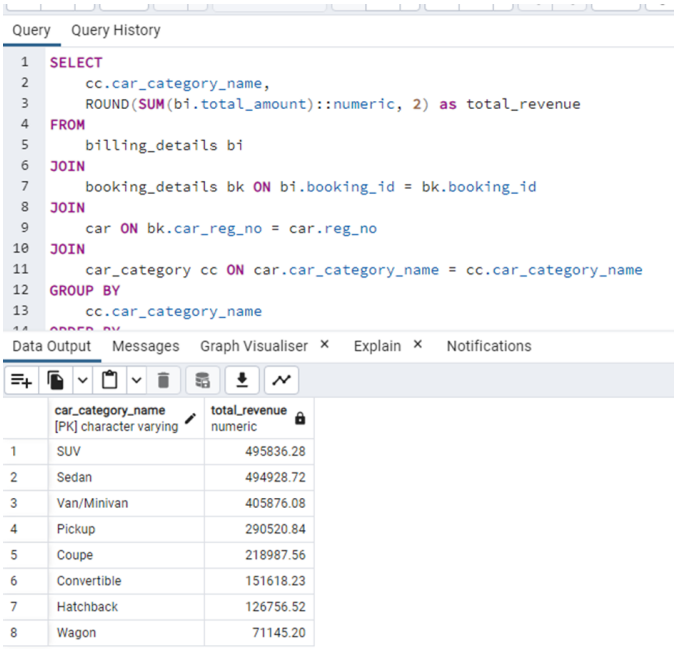


Fig. 59. Select Query analyzing revenue by car category

Aggregate Select Query for Revenue Analysis: This query calculates the total revenue generated by each car category, aggregating data from billing and booking details and grouping by car category.

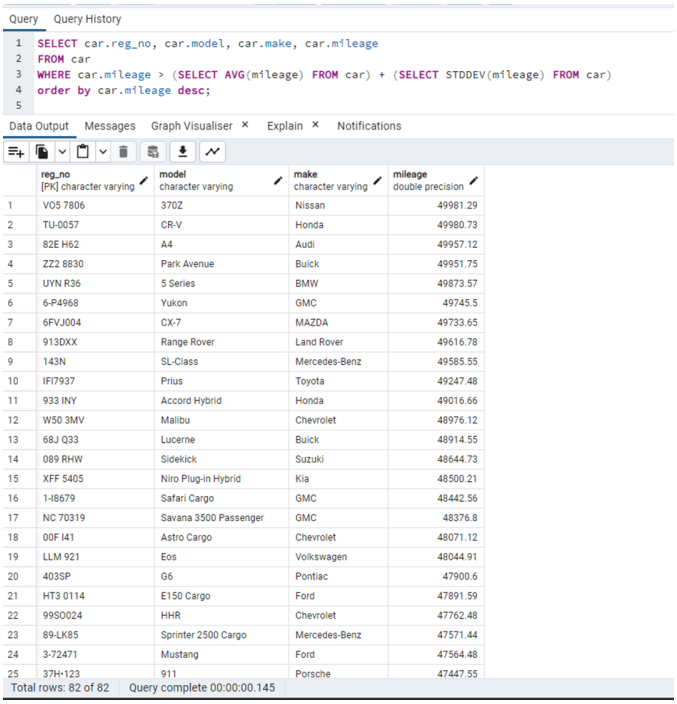
D. Select Queries

- Aggregate Select query to Analyze Revenue by Car Category:

  SELECT cc.car category name, ROUND(SUM(bi.total amount)::numeric, 2) as total revenue FROM billing details bi JOIN booking details bk ON bi.booking id = bk.booking id JOIN car ON bk.car reg no = car.reg no JOIN car category cc ON car.car category name = cc.car category name GROUP BY cc.car category name ORDER BY total revenue DESC;

- Query for Finding Cars with High Mileage for Maintenance Check:

  SELECT car.reg no, car.model, car.make, car.mileage FROM car WHERE car.mileage ¿ (SELECT AVG(mileage) FROM car) + (SELECT STDDEV(mileage) FROM car) order by car.mileage desc;



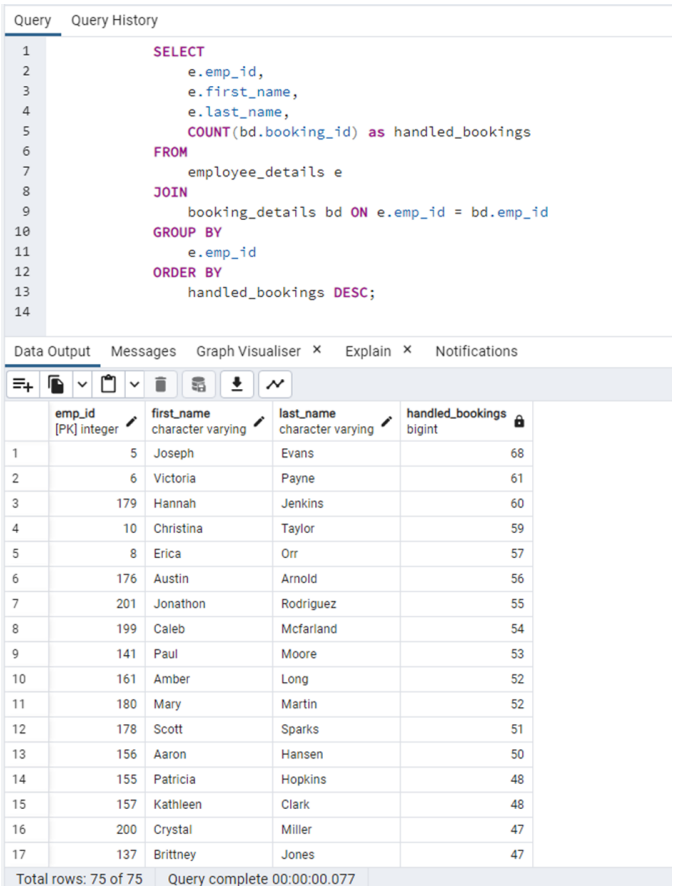
```
SELECT car.reg_no, car.model, car.make, car.mileage
FROM car
WHERE car.mileage > (SELECT AVG(mileage) FROM car) + (SELECT STDDEV(mileage) FROM car)
order by car.mileage desc;
```

	reg_no [PK] character varying	model character varying	make character varying	mileage double precision
1	VO5 7806	370Z	Nissan	49981.29
2	TU-0057	CR-V	Honda	49980.73
3	82E H62	A4	Audi	49957.12
4	ZZ2 8830	Park Avenue	Buick	49951.75
5	UYN R36	5 Series	BMW	49873.57
6	6-P4968	Yukon	GMC	49745.5
7	6FVJ004	CX-7	MAZDA	49733.65
8	913DXX	Range Rover	Land Rover	49616.78
9	143N	SL-Class	Mercedes-Benz	49585.55
10	IFI7937	Prius	Toyota	49247.48
11	933 INY	Accord Hybrid	Honda	49016.66
12	W50 3MV	Malibu	Chevrolet	48976.12
13	68J Q33	Lucerne	Buick	48914.55
14	089 RHW	Sidekick	Suzuki	48644.73
15	XFF 5405	Niro Plug-in Hybrid	Kia	48500.21
16	1-I8679	Safari Cargo	GMC	48442.56
17	NC 70319	Savana 3500 Passenger	GMC	48376.8
18	00F I41	Astro Cargo	Chevrolet	48071.12
19	LLM 921	Eos	Volkswagen	48044.91
20	403SP	G6	Pontiac	47900.6
21	HT3 0114	E150 Cargo	Ford	47891.59
22	99S0024	HHR	Chevrolet	47762.48
23	89-LK85	Sprinter 2500 Cargo	Mercedes-Benz	47571.44
24	3-72471	Mustang	Ford	47564.48
25	37H•123	911	Porsche	47447.55

Total rows: 82 of 82 Query complete 00:00:00.145

Fig. 60. Cars with High Mileage

Query for High Mileage Cars: This query selects cars with mileage higher than the average plus the standard deviation of mileage, indicating cars that might need maintenance due to high usage.



```
SELECT
    e.emp_id,
    e.first_name,
    e.last_name,
    COUNT(bd.booking_id) as handled_bookings
FROM
    employee_details e
JOIN
    booking_details bd ON e.emp_id = bd.emp_id
GROUP BY
    e.emp_id
ORDER BY
    handled_bookings DESC;
```

	emp_id [PK] integer	first_name character varying	last_name character varying	handled_bookings bigint
1	5	Joseph	Evans	68
2	6	Victoria	Payne	61
3	179	Hannah	Jenkins	60
4	10	Christina	Taylor	59
5	8	Erica	Orr	57
6	176	Austin	Arnold	56
7	201	Jonathon	Rodriguez	55
8	199	Caleb	Mcfarland	54
9	141	Paul	Moore	53
10	161	Amber	Long	52
11	180	Mary	Martin	52
12	178	Scott	Sparks	51
13	156	Aaron	Hansen	50
14	155	Patricia	Hopkins	48
15	157	Kathleen	Clark	48
16	200	Crystal	Miller	47
17	137	Brittney	Jones	47

Total rows: 75 of 75 Query complete 00:00:00.077

Fig. 61. Employee Performance on number of bookings

Employee Performance Based on Bookings: This query analyzes employees' performance by counting the number of bookings each employee has managed.

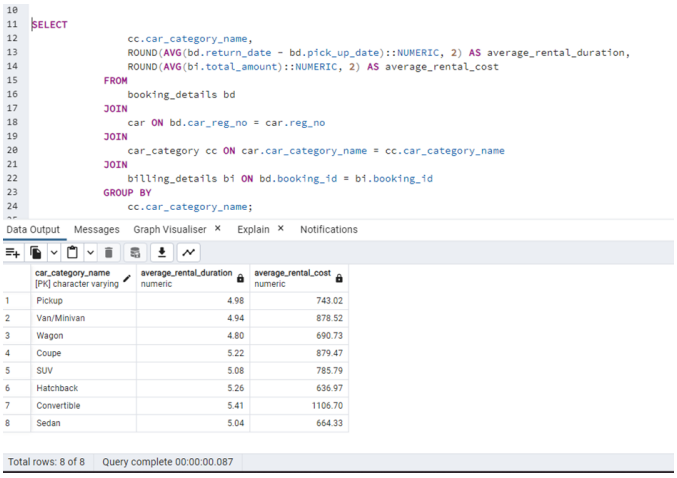
- Query for Employee Performance Analysis Based on Number of Handled Bookings:

  SELECT e.emp id, e.first name, e.last name, COUNT(bd.booking id) as handled bookings FROM employee details e JOIN booking details bd ON e.emp id = bd.emp id GROUP BY e.emp id ORDER BY handled bookings DESC;

- Query for Average Rental Duration and Cost by Car Category:

  SELECT cc.car_category_name, ROUND(AVG(bd.return_date - bd.pick_up_date)::NUMERIC, 2) AS average_rental_duration, ROUND(AVG(bi.total_amount)::NUMERIC, 2) AS average_rental_cost FROM booking_details bd JOIN car

ON bd.ca ° _reg_no = car.reg_no JOIN car_category cc ON car.car_category_name = cc.car_category_name JOIN billing_details bi ON bd.booking_id = bi.booking_id GROUP BY cc.car_category_name;



```
SELECT
    cc.car_category_name,
    ROUND(AVG(bd.return_date - bd.pick_up_date)::NUMERIC, 2) AS average_rental_duration,
    ROUND(AVG(bi.total_amount)::NUMERIC, 2) AS average_rental_cost
FROM
    booking_details bd
JOIN
    car ON bd.car_reg_no = car.reg_no
JOIN
    car_category cc ON car.car_category_name = cc.car_category_name
JOIN
    billing_details bi ON bd.booking_id = bi.booking_id
GROUP BY
    cc.car_category_name;
```

Data Output | Messages | Graph Visualiser | Explain | Notifications

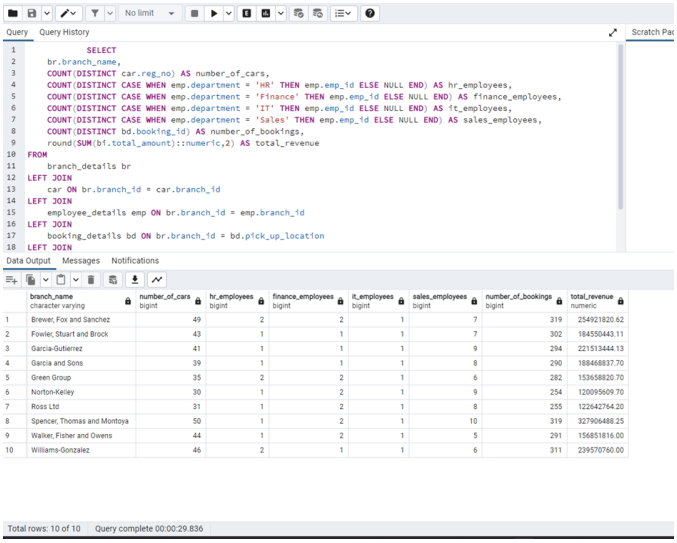
	car_category_name [PK] character varying	average_rental_duration numeric	average_rental_cost numeric
1	Pickup	4.98	743.02
2	Van/Minivan	4.94	878.52
3	Wagon	4.80	690.73
4	Coupe	5.22	879.47
5	SUV	5.08	785.79
6	Hatchback	5.26	636.97
7	Convertible	5.41	1106.70
8	Sedan	5.04	664.33

Total rows: 8 of 8 | Query complete 00:00:00.087

Fig. 62. Average Rental Duration and Cost by Car

Query for Average Rental Duration and Cost: This query calculates the average rental duration and cost for each car category, using data from bookings and billing details.

COUNT(DISTINCT bd.booking id) AS number of bookings, round(SUM(bi.total amount)::numeric,2) AS total revenue FROM branch details br LEFT JOIN car ON br.branch id = car.branch id LEFT JOIN employee details emp ON br.branch id = emp.branch id LEFT JOIN booking details bd ON br.branch id = bd.pick up location LEFT JOIN billing details bi ON bd.booking id = bi.booking id GROUP BY br.branch name;



```
SELECT
    br.branch_name,
    COUNT(DISTINCT car.reg_no) AS number_of_cars,
    COUNT(DISTINCT CASE WHEN emp.department = 'HR' THEN emp.emp_id ELSE NULL END) AS hr_employees,
    COUNT(DISTINCT CASE WHEN emp.department = 'Finance' THEN emp.emp_id ELSE NULL END) AS finance_employees,
    COUNT(DISTINCT CASE WHEN emp.department = 'IT' THEN emp.emp_id ELSE NULL END) AS it_employees,
    COUNT(DISTINCT CASE WHEN emp.department = 'Sales' THEN emp.emp_id ELSE NULL END) AS sales_employees,
    COUNT(DISTINCT bd.booking_id) AS number_of_bookings,
    round(SUM(bi.total_amount)::numeric,2) AS total_revenue
FROM
    branch_details br
LEFT JOIN
    car ON br.branch_id = car.branch_id
LEFT JOIN
    employee_details emp ON br.branch_id = emp.branch_id
LEFT JOIN
    booking_details bd ON br.branch_id = bd.pick_up_location
LEFT JOIN
```

Data Output | Messages | Notifications

	branch_name character varying	number_of_cars bigint	hr_employees bigint	finance_employees bigint	it_employees bigint	sales_employees bigint	number_of_bookings bigint	total_revenue numeric
1	Brewer, Fox and Sanchez	49	2	2	1	7	319	254921820.62
2	Fowler, Stuart and Brock	43	1	1	1	7	302	184550443.11
3	Garcia-Gutierrez	41	1	1	1	9	294	221513444.13
4	Garcia and Sons	39	1	1	1	8	290	188468837.70
5	Green Group	35	2	2	1	6	282	153658820.70
6	Norton-Kelley	30	1	2	1	9	254	120095609.70
7	Ross Ltd	31	1	2	1	8	255	122642764.20
8	Spencer, Thomas and Montoya	50	1	2	1	10	319	327906488.25
9	Walker, Fisher and Owens	44	1	2	1	5	291	156851816.00
10	Williams-Gonzalez	46	2	1	1	6	311	239570760.00

Total rows: 10 of 10 | Query complete 00:00:29.836

Fig. 63. Branch Performance analysis

- Query for Branch Performance Analysis:

SELECT br.branch name, COUNT(DISTINCT car.reg no) AS number of cars, COUNT(DISTINCT CASE WHEN emp.department = 'HR' THEN emp.emp id ELSE NULL END) AS hr employees, COUNT(DISTINCT CASE WHEN emp.department = 'Finance' THEN emp.emp id ELSE NULL END) AS finance employees, COUNT(DISTINCT CASE WHEN emp.department = 'IT' THEN emp.emp id ELSE NULL END) AS it employees, COUNT(DISTINCT CASE WHEN emp.department = 'Sales' THEN emp.emp id ELSE NULL END) AS sales employees,

Query for Branch Performance Analysis: This query analyzes the performance of each branch, counting the number of cars, employees in different departments, number of bookings, and total revenue generated, grouped by branch name.

X. Query Execution Analysis

A. Analyzing First Query

- Before Indexing

Query | Query History

```
1  explain analyze SELECT
2      bd.booking_id,
3      bd.booking_date,
4      bd.pick_up_date,
5      bd.return_date,
6      bd.customer_id,
7      bd.pick_up_location,
8      bd.return_location,
9      bd.emp_id,
10     bd.chauffeur_id,
11     bd.insurance_category,
12     bd.car_reg_no
13 FROM
14     booking_details bd
15 JOIN
16     customer c ON bd.customer_id = c.customer_id
17 WHERE
18     c.lastname = 'Holder';
19
```

Fig. 64. Query 1 Analysis

Messages | Explain × | Notifications | Data Output

	QUERY PLAN text
1	Hash Join (cost=7.51..76.59 rows=15 width=55) (actual time=0.164..0.413 rows=9 loops=1)
2	Hash Cond: (bd.customer_id = c.customer_id)
3	-> Seq Scan on booking_details bd (cost=0.00..61.24 rows=2924 width=55) (actual time=0.006..0.123 rows=2924 loop...
4	-> Hash (cost=7.50..7.50 rows=1 width=4) (actual time=0.029..0.030 rows=1 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> Seq Scan on customer c (cost=0.00..7.50 rows=1 width=4) (actual time=0.015..0.025 rows=1 loops=1)
7	Filter: ((lastname)::text = 'Holder'::text)
8	Rows Removed by Filter: 199
9	Planning Time: 0.199 ms
10	Execution Time: 0.430 ms

Fig. 65. Query 1 Plan

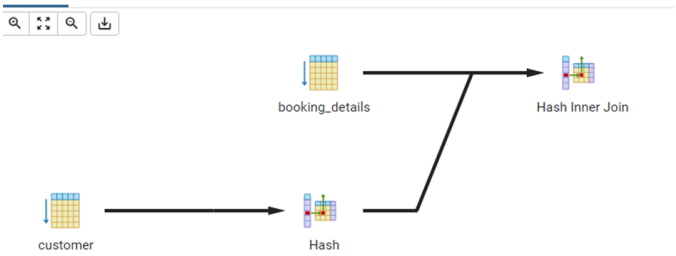


Fig. 66. Query 1 Flow

- After Indexing
  CREATE INDEX idx_booking_details_customer_id ON booking_details (customer_id);

	QUERY PLAN text
1	Nested Loop (cost=4.40..39.38 rows=15 width=55) (actual time=0.021..0.034 rows=9 loops=1)
2	-> Seq Scan on customer c (cost=0.00..7.50 rows=1 width=4) (actual time=0.012..0.018 rows=1 loops=1)
3	Filter: ((lastname)::text = 'Holder'::text)
4	Rows Removed by Filter: 199
5	-> Bitmap Heap Scan on booking_details bd (cost=4.40..31.73 rows=15 width=55) (actual time=0.006..0.011 rows=9 loops=1)
6	Recheck Cond: (customer_id = c.customer_id)
7	Heap Blocks: exact=9
8	-> Bitmap Index Scan on idx_booking_details_customer_id (cost=0.00..4.39 rows=15 width=0) (actual time=0.004..0.004 rows=9 loop...
9	Index Cond: (customer_id = c.customer_id)
10	Planning Time: 1.169 ms
11	Execution Time: 0.050 ms

Fig. 67. Query Plan After Indexing

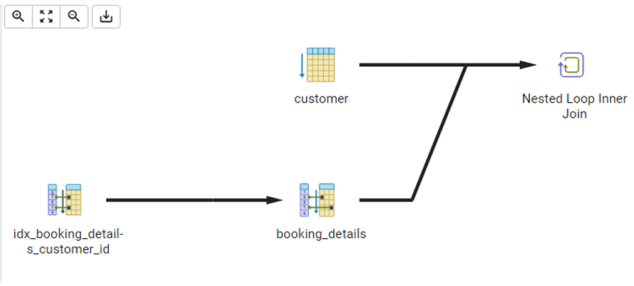


Fig. 68. Query Flow After Indexing

The execution time has decreased after indexing due to the optimization of data retrieval methods used by the database. In the first image, without indexing, the database performed a Sequential Scan on booking_details and a Hash Join operation, which are more time-consuming as they involve scanning entire tables and computing hash tables for joins. After indexing, as shown in the second image, the database used a Bitmap Heap Scan on booking_details and a Bitmap Index Scan on the index created for idx_booking_details_customer_id. These operations are faster because they use a pre-computed index to directly pinpoint the rows needed for the join, avoiding a full table scan. The Bitmap Heap and Index Scans are generally much faster, especially when working with larger datasets, which is why you see a significant reduction in execution time.

B. Analyzing Second Query

- Before Indexing

```
explain analyze SELECT COUNT(*) AS number_of_bookings
FROM booking_details bd
JOIN car ON bd.car_reg_no = car.reg_no
WHERE car.make = 'BMW';
```

Fig. 69. Second Query

	QUERY PLAN text
1	Aggregate (cost=80.61..80.62 rows=1 width=8) (actual time=0.677..0.677 rows=1 loops=1)
2	-> Hash Join (cost=11.31..80.31 rows=122 width=0) (actual time=0.075..0.671 rows=117 loops=1)
3	Hash Cond: ((bd.car_reg_no)::text = (car.reg_no)::text)
4	-> Seq Scan on booking_details bd (cost=0.00..61.24 rows=2924 width=8) (actual time=0.008..0.129 rows=2924 loop...
5	-> Hash (cost=11.10..11.10 rows=17 width=8) (actual time=0.058..0.058 rows=17 loops=1)
6	Buckets: 1024 Batches: 1 Memory Usage: 9kB
7	-> Seq Scan on car (cost=0.00..11.10 rows=17 width=8) (actual time=0.009..0.053 rows=17 loops=1)
8	Filter: ((make)::text = 'BMW'::text)
9	Rows Removed by Filter: 391
10	Planning Time: 0.626 ms
11	Execution Time: 0.703 ms

Fig. 70. Query 2 Plan

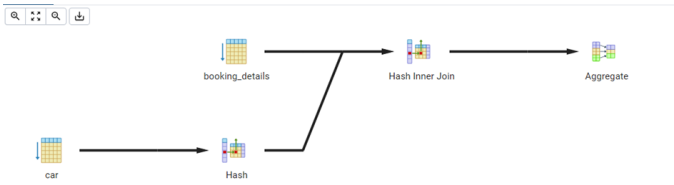


Fig. 71. Query 2 Flow

- After Indexing

CREATE INDEX idx_car_make ON car (make); CREATE INDEX idx_booking_details_car_reg_no ON booking_details (car_reg_no);

	QUERY PLAN text
1	Aggregate (cost=50.83..50.84 rows=1 width=8) (actual time=0.120..0.120 rows=1 loops=1)
2	-> Nested Loop (cost=4.56..50.52 rows=122 width=0) (actual time=0.033..0.114 rows=117 loops=1)
3	-> Bitmap Heap Scan on car (cost=4.28..10.49 rows=17 width=8) (actual time=0.021..0.027 rows=17 loops=1)
4	Recheck Cond: ((make)::text = 'BMW'::text)
5	Heap Blocks: exact=6
6	-> Bitmap Index Scan on idx_car_make (cost=0.00..4.28 rows=17 width=0) (actual time=0.017..0.017 rows=17 loops=1)
7	Index Cond: ((make)::text = 'BMW'::text)
8	-> Index Only Scan using idx_booking_details_car_reg_no on booking_details bd (cost=0.28..2.28 rows=7 width=8) (actual time=0.004..0.005 rows=7 loop...
9	Index Cond: (car_reg_no = (car.reg_no)::text)
10	Heap Fetches: 3
11	Planning Time: 0.998 ms
12	Execution Time: 0.145 ms

Fig. 72. Query 2 Plan After Indexing

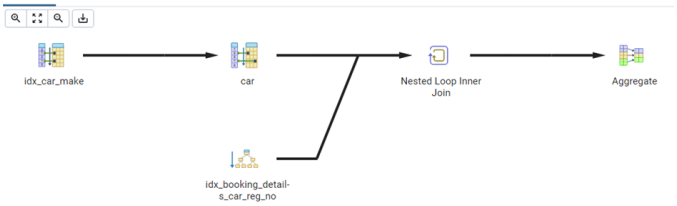


Fig. 73. Query 2 Flow After Indexing

Explanation: The execution time has decreased after indexing because the database is now able to utilize the indexes to locate records more efficiently. The first query plan shows a Sequential Scan on the car table, which is a slower operation as it scans each row of the table. After indexing (as shown in the second query plan), the database performs an Index Only Scan on idx_car_make for the car table and a Bitmap Index Scan on idx_booking_details_car_reg_no for the booking_details table. These index scans are much faster because they allow the database to quickly locate the rows that meet the query conditions without scanning the entire table.

C. Analyzing Third Query

- Before Indexing

Query | Query History

```
1  explain analyze SELECT
2      e.emp_id,
3      e.first_name,
4      e.last_name,
5      COUNT(bd.booking_id) AS total_bookings
6  FROM
7      employee_details e
8  JOIN
9      booking_details bd ON e.emp_id = bd.emp_id
10 WHERE
11     e.emp_id = 5
12 GROUP BY
13     e.emp_id
```

Fig. 74. Third Query

	QUERY PLAN text
1	GroupAggregate (cost=0.00..74.33 rows=1 width=26) (actual time=0.339..0.340 rows=1 loops=1)
2	Group Key: e.emp_id
3	-> Nested Loop (cost=0.00..73.94 rows=76 width=22) (actual time=0.017..0.333 rows=68 loops=1)
4	-> Seq Scan on employee_details e (cost=0.00..4.63 rows=1 width=18) (actual time=0.012..0.019 rows=1 loops=1)
5	Filter: (emp_id = 5)
6	Rows Removed by Filter: 209
7	-> Seq Scan on booking_details bd (cost=0.00..68.55 rows=76 width=8) (actual time=0.003..0.308 rows=68 loop...
8	Filter: (emp_id = 5)
9	Rows Removed by Filter: 2856
10	Planning Time: 0.173 ms
11	Execution Time: 0.360 ms

Fig. 75. Query 3 Plan

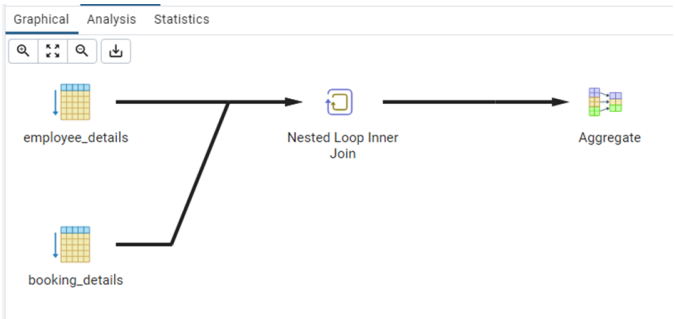


Fig. 76. Query 3 flow

- After Indexing
  CREATE INDEX idx_booking_details_emp_id ON booking_details (emp_id);

	QUERY PLAN text
1	GroupAggregate (cost=4.87..43.59 rows=1 width=26) (actual time=0.101..0.101 rows=1 loops=1)
2	Group Key: e.emp_id
3	-> Nested Loop (cost=4.87..43.20 rows=76 width=22) (actual time=0.057..0.091 rows=68 loops=1)
4	-> Seq Scan on employee_details e (cost=0.00..4.63 rows=1 width=18) (actual time=0.013..0.028 rows=1 loops=1)
5	Filter: (emp_id = 5)
6	Rows Removed by Filter: 209
7	-> Bitmap Heap Scan on booking_details bd (cost=4.87..37.82 rows=76 width=8) (actual time=0.040..0.051 rows=68 loops=1)
8	Recheck Cond: (emp_id = 5)
9	Heap Blocks: exact=4
10	-> Bitmap Index Scan on idx_booking_details_emp_id (cost=0.00..4.85 rows=76 width=0) (actual time=0.033..0.033 rows=68 loop...
11	Index Cond: (emp_id = 5)
12	Planning Time: 0.408 ms
13	Execution Time: 0.141 ms

Fig. 77. Query 3 Plan After Indexing

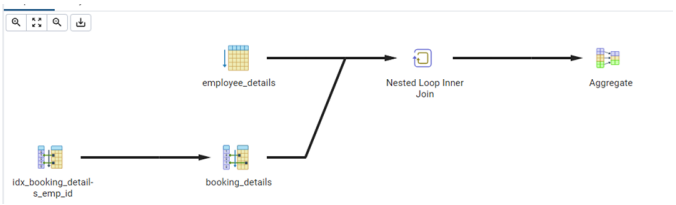


Fig. 78. Query 3 Flow After Indexing

The reduction in execution time between the first and second query plans can be attributed to the use of a Bitmap Index Scan in the second plan. This indicates that an index on emp_id in the booking_details table was utilized, allowing for a more efficient retrieval of rows. Bitmap Index Scans are generally faster than Sequential Scans, especially when filtering on indexed columns, leading to a lower execution time.

NOTE: Indexing columns other than booking_id in the booking_details table can cause delays in insert operations due to the nature of how indexes work. When a new record is inserted, the database needs to update each index associated with the table. This means the database has to insert entries into each index for the new row. If there are multiple indexes (especially on frequently changing columns), this process becomes more time-consuming. Each index update requires additional write operations, potentially slowing down the overall insert process. So these indexing are done just to show how select operations can be improved with indexing, but in real DB we will not index these columns as inserting data into booking_details will get delayed.

XI. Roles and Responsibilities

- Nitin: Database Tables Design: Lead the creation and structuring of database tables, ensuring they meet project requirements.
  ER Diagrams: Develop and maintain Entity-Relationship diagrams, mapping out the database schema.
  Integration of Front End and Back End: Ensure seamless integration between the front-end and back-end systems.

- Mithil: Triggers: Design and implement database triggers to automate processes and maintain data integrity.
  Data Insertions into Tables: Manage the insertion of data into the tables, ensuring accuracy and efficiency.
  Backend to Support Front End: Develop and maintain the backend logic, especially focusing on API creation and management.

- Vignesh: Flask Code for Front End: Develop the front-end interface using Flask, focusing on user experience and interface design.
  Report and Documentation: Handle the preparation of detailed project reports and documentation of all processes and structures.
  Testing and Quality Assurance: Lead testing efforts to identify and fix bugs, ensuring the overall quality of the software.