

PES UNIVERSITY

100 feet Ring Road, BSK 3rd Stage, Bengaluru 560085

Department of Computer Science and Engineering

Jan – May 2020

UE18CS252

Database Management Systems

Project Report

Vehicle Insurance Management System

PES1201800308 Bhargav SNV

4th Sem Sec. I Roll No. 15

|  |  |
| --- | --- |
|  |  |

|  |
| --- |
| **PROJECT SUMMARY**  This project aims to implement a system to record and maintain data about vehicle insurances and all related data by an insurance agency. An apt and descriptive data model has been described under the data model section on page 4. This project aims to replicate real world use cases for data queries and has been described as such in the below sections. Triggers have been created to not only perform audit-tails to log changes for users and insurance schemes, but also to help generation of registration details easier. Overall, the project aims to build a concise and effective database solution for use by insurance agencies. |

[Introduction 3](#_Toc41524772)

[Data Model 4](#_Toc41524773)

[FD and Normalization 5](#_Toc41524774)

[DDL 6](#_Toc41524775)

[Triggers 7](#_Toc41524776)

[SQL Queries 8](#_Toc41524777)

[Conclusion 10](#_Toc41524778)

# 

# **Introduction**

This project aims to implement a system to record and maintain data about vehicle insurances and all related data by an insurance agency. It efficiently stores data about the agency’s employees, customers and all related assets like credentials, vehicle details, insurances issued, validity of schemes, etc.

Thus, the mini-world chosen is that of an insurance agency which has employees who have job-roles and login credentials, customers who own vehicles, which have registration details and insurance. The entities in this mini world are as follows:

* Users
* User's login credentials
* User's Job roles
* Customers
* Vehicles
* Vehicle Insurance

The database has multiple relations between entities, these can be seen pictorially through the schema described in the later sections. In brief these relationships are:

* Users *have* login credentials and roles
* Users *manage* customers
* Customers *own* vehicles
* Vehicles *have* registration and insurance
* Insurance *needs* registration

# **Data Model**

The mini-world is represented by the below schema.

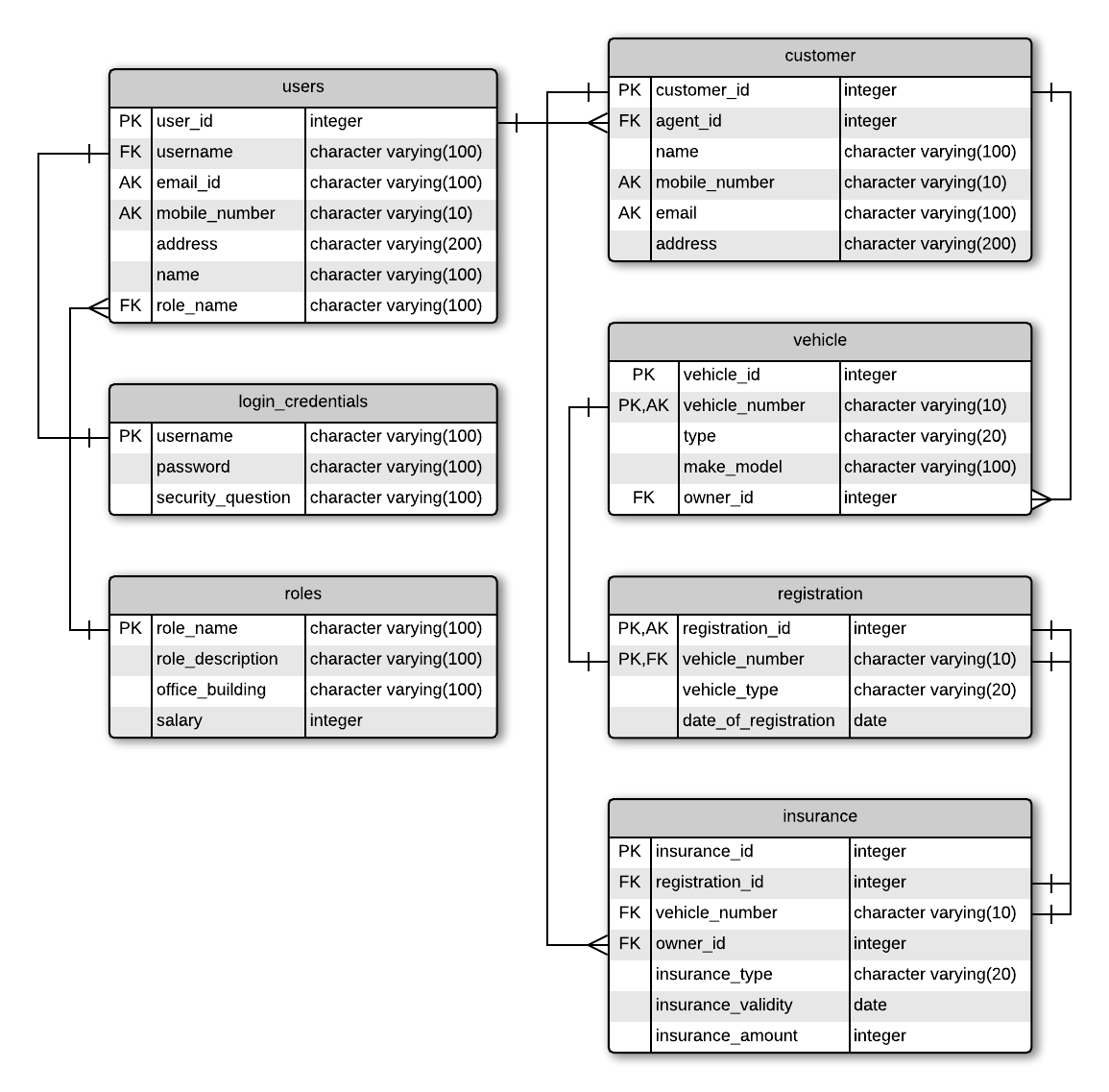


Table Keys:

* **Users** 
  + Primary Key: *user\_id*
  + Alternate Keys: *email\_id, mobile\_number*
* **Login\_Credentials** 
  + Primary Key: *username*
* **Roles** 
  + Primary Key: *role\_name*
* **Customer** 
  + Primary Key: *customer\_id*
  + Alternate Keys: *mobile\_number, email*
* **Vehicle** 
  + Primary Key: *vehicle\_id, vehicle\_number*
* **Registration** 
  + Primary Key: *registration\_id, vehicle\_number*
  + Here, *vehicle\_number* is chosen as primary and foreign key so that no mismatch can occur for a given pair *registration\_id* and *vehicle\_number*.
* **Insurance** 
  + Primary Key: *insurance\_id*

# **FD and Normalization**

The database has been normalised to 3 NF. This is violated if in the schema, **Users, Login\_Credentials, Roles** were stored as a single table instead of 3. This is because of transitive dependency. The primary key of **Users** can determine *roles* and *username*, which in turn can determine *role\_description, office\_building, salary* and *password, security\_question* respectively.

The functional dependencies are now only between primary keys and other attributes of a table. These functional dependencies are:

* **Users**:
  + *user\_id -> username, email\_id, mobile\_number, address, name, role\_name*
* **Login\_Credentials:** 
  + *username -> password, security\_question*
* **Roles:** 
  + *r*o*le\_name -> role\_description, office\_building, salary*
* **Customer**:
  + c*ustomer\_id -> agent\_id, name, mobile\_number, email, address*
* **Vehicle**:
  + *vehicle\_id, vehicle\_number -> type, make\_model, owner\_id*
* **Registration**:
  + *registration\_id, vehicle\_number -> vehicle\_type, date\_of\_registration*
* **Insurance**:
  + *insurance\_id -> registration\_id, vehicle\_number, owner\_id, insurance\_type, insurance\_validity, insurance\_amount*

# **DDL**

CREATE TABLE Login\_Credentials (

username varchar(100),

password varchar(100) NOT NULL CHECK (char\_length(password) > 7 AND char\_length(password) <= 100),

security\_question varchar(100),

PRIMARY KEY (username)

);

CREATE TABLE Roles (

role\_name varchar(100),

role\_description varchar(100),

office\_building varchar(100),

salary integer,

PRIMARY KEY (role\_name)

);

CREATE TABLE Users (

user\_id integer,

username varchar(100),

email\_id varchar(100) NOT NULL UNIQUE,

mobile\_number varchar(10) NOT NULL UNIQUE CHECK (char\_length(mobile\_number) = 10),

address varchar(200) NOT NULL,

name varchar(100) NOT NULL,

role\_name varchar(100),

PRIMARY KEY (user\_id),

FOREIGN KEY (role\_name) REFERENCES Roles (role\_name) ON DELETE SET NULL ON UPDATE CASCADE,

FOREIGN KEY (username) REFERENCES Login\_Credentials (username) ON DELETE SET NULL ON UPDATE CASCADE

);

CREATE TABLE Customer (

customer\_id integer,

agent\_id integer,

name varchar(100) NOT NULL,

mobile\_number varchar(10) NOT NULL UNIQUE CHECK (char\_length(mobile\_number) = 10),

email varchar(100) NOT NULL UNIQUE,

address varchar(200) NOT NULL,

PRIMARY KEY (customer\_id),

FOREIGN KEY(agent\_id) REFERENCES Users(user\_id) ON DELETE SET NULL ON UPDATE CASCADE

);

CREATE TABLE Vehicle (

vehicle\_id integer,

vehicle\_number varchar(15) UNIQUE CHECK ( vehicle\_number ~ $$^[A-Z]{2}\s[0-9]{2}\s[A-Z]{2}\s[0-9]{4}$$), -- check with regex

type varchar(20) NOT NULL,

make\_model varchar(100),

owner\_id integer,

PRIMARY KEY (vehicle\_id),

FOREIGN KEY(owner\_id) REFERENCES Customer(customer\_id) ON DELETE RESTRICT ON UPDATE CASCADE

);

CREATE TABLE Registration (

registration\_id SERIAL,

vehicle\_number varchar(15),

vehicle\_type varchar(20),

date\_of\_registration date NOT NULL,

PRIMARY KEY (registration\_id, vehicle\_number),

FOREIGN KEY(vehicle\_number) REFERENCES Vehicle(vehicle\_number) ON DELETE RESTRICT ON UPDATE CASCADE

);

CREATE TABLE Insurance (

insurance\_id integer,

registration\_id integer,

vehicle\_number varchar(15),

owner\_id integer,

insurance\_type varchar(20) NOT NULL,

insurance\_validity date NOT NULL,

insurance\_amount integer NOT NULL,

PRIMARY KEY (insurance\_id),

FOREIGN KEY(registration\_id, vehicle\_number) REFERENCES Registration(registration\_id, vehicle\_number) ON DELETE SET NULL ON UPDATE CASCADE,

FOREIGN KEY(owner\_id) REFERENCES Customer(customer\_id) ON DELETE SET NULL ON UPDATE CASCADE

);

# **Triggers**

3 triggers were made. They are:

* *create\_registration\_trigger*: This is triggered whenever data is inserted into the **vehicle** table. As data is inserted into the **vehicle** table, registration details are automatically inferred and inserted into the **Registration** table. This helps maintain the proper details of vehicles as and when they are inserted.
* *audit\_user\_trigger*: This is triggered whenever changes are made to the **Users** table. The *user id* and *time stamp* are logged in the table named **audit\_users**.
* *audit\_insurance\_trigger*: This is triggered whenever changes are made to the **Insurance** table. The *customer id, agent id, insurance id* and *time stamp* are logged into the table named **audit\_insurance**.

-- Audit Logs and triggers

--------------------------

CREATE TABLE audit\_users (

user\_id integer NOT NULL,

entry\_date TIMESTAMP NOT NULL

);

CREATE OR REPLACE FUNCTION auditlogusers() RETURNS TRIGGER AS $table$

BEGIN

INSERT INTO audit\_users

VALUES (new.user\_id, current\_timestamp);

RETURN NEW;

END;

$table$ LANGUAGE plpgsql;

CREATE TRIGGER audit\_user\_trigger

AFTER INSERT OR UPDATE OR DELETE ON Users

FOR EACH ROW EXECUTE PROCEDURE auditlogusers();

CREATE TABLE audit\_insurance (

owner\_id integer NOT NULL,

agent\_id integer NOT NULL,

insurance\_id integer NOT NULL,

entry\_date TIMESTAMP NOT NULL

);

CREATE OR REPLACE FUNCTION auditloginsurance() RETURNS TRIGGER AS $table$

BEGIN

INSERT INTO audit\_insurance

VALUES (new.owner\_id, (

SELECT agent\_id

FROM customer

WHERE customer\_id = new.owner\_id), new.insurance\_id, current\_timestamp);

RETURN NEW;

END;

$table$ LANGUAGE plpgsql;

CREATE TRIGGER audit\_insurance\_trigger

AFTER INSERT OR UPDATE OR DELETE ON Insurance

FOR EACH ROW EXECUTE PROCEDURE auditloginsurance();

CREATE OR REPLACE FUNCTION create\_registration() RETURNS TRIGGER AS $table$

BEGIN

INSERT INTO Registration(vehicle\_number, vehicle\_type, date\_of\_registration)

VALUES (new.vehicle\_number, new.type, NOW());

RETURN NEW;

END;

$table$ LANGUAGE plpgsql;

CREATE TRIGGER create\_registration\_trigger

AFTER INSERT ON Vehicle

FOR EACH ROW EXECUTE PROCEDURE create\_registration();

# **SQL** **Queries**

Aggregate Queries

1. Find the count of customers each agent manages

SELECT U.name, C.agent\_id, COUNT(C.agent\_id)

FROM customer C, users U

WHERE U.user\_id = C.agent\_id

GROUP BY (U.name, C.agent\_id);

1. Find agents with salary higher than average salary

SELECT name, salary

FROM Users NATURAL JOIN Roles

WHERE salary > (

SELECT AVG(salary)

FROM Users NATURAL JOIN Roles

);

Nested Queries

1. Find customer name, id, vehicle number and type where vehicles have insurance lasting beyond 2023.

SELECT name, customer\_id, vehicle\_number, type

FROM Customer C INNER JOIN Vehicle V on V.owner\_id = C.customer\_id

WHERE C.customer\_id in (

SELECT owner\_id

FROM insurance

WHERE insurance\_validity > '2023-02-08'

);

1. Find agents with customers having cars registered before 2020.

SELECT U.name, U.user\_id

FROM Users U

WHERE user\_id in (

SELECT C.agent\_id

FROM Customer C

WHERE C.customer\_id in (

SELECT V.owner\_id

FROM vehicle V

WHERE V.vehicle\_number in (

SELECT vehicle\_number

FROM Registration

WHERE (date\_of\_registration < '2022-01-01')

)

)

);

Correlated Queries

1. Find insurance details for all non-geared motor cycles

SELECT I.vehicle\_number, I.insurance\_amount, I.insurance\_validity, I.insurance\_type

FROM insurance I

WHERE I.registration\_id = (

SELECT R.registration\_id

FROM Registration R

WHERE R.vehicle\_type = 'MCWOG'

);

1. Update vehicle make and model based off type mentioned in Registration

UPDATE Vehicle V

SET make\_model = CONCAT((

SELECT R.vehicle\_type

FROM Registration R

WHERE V.vehicle\_number = R.vehicle\_number), ': ', make\_model

);

Outer Join Queries

SELECT \* FROM Users FULL OUTER JOIN Roles on Users.role\_name = Roles.role\_name;

SELECT \* FROM Users FULL OUTER JOIN Insurance on Users.user\_id = Insurance.owner\_id;

# **Conclusion**

This system can efficiently manage vehicle insurances. If equipped with a front-end, it can prove to be a powerful application which insurance companies can use to track and maintain records. The only limitations are there are no alerting methods to inform the end users of events like expiry of insurance. This however can be tackled quite easily and implemented as future scope.