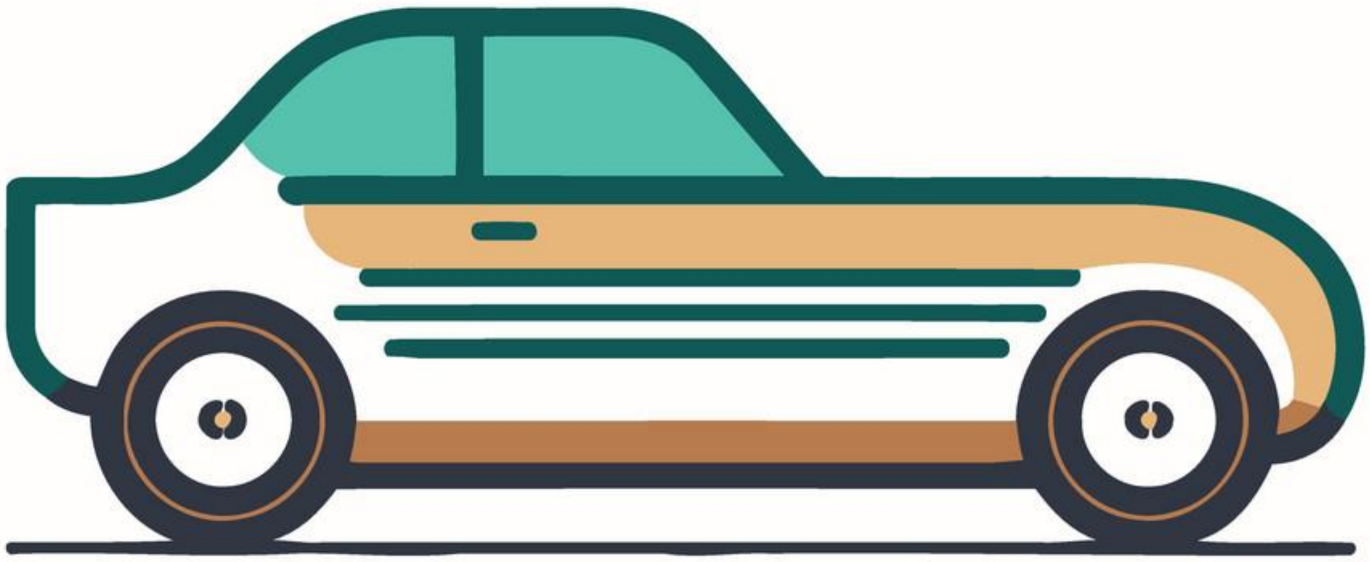
DATABASE MANAGEMENT SYSTEMS



PROJECT ON:
Car Dealership

Submitted By-
Advait Gourkhede 102267006
Drishti Sinha 102217188
Rajarshi Biswas 102217232

Submitted to:
Dr. Chinmaya Panigrahy

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

Index

Introduction

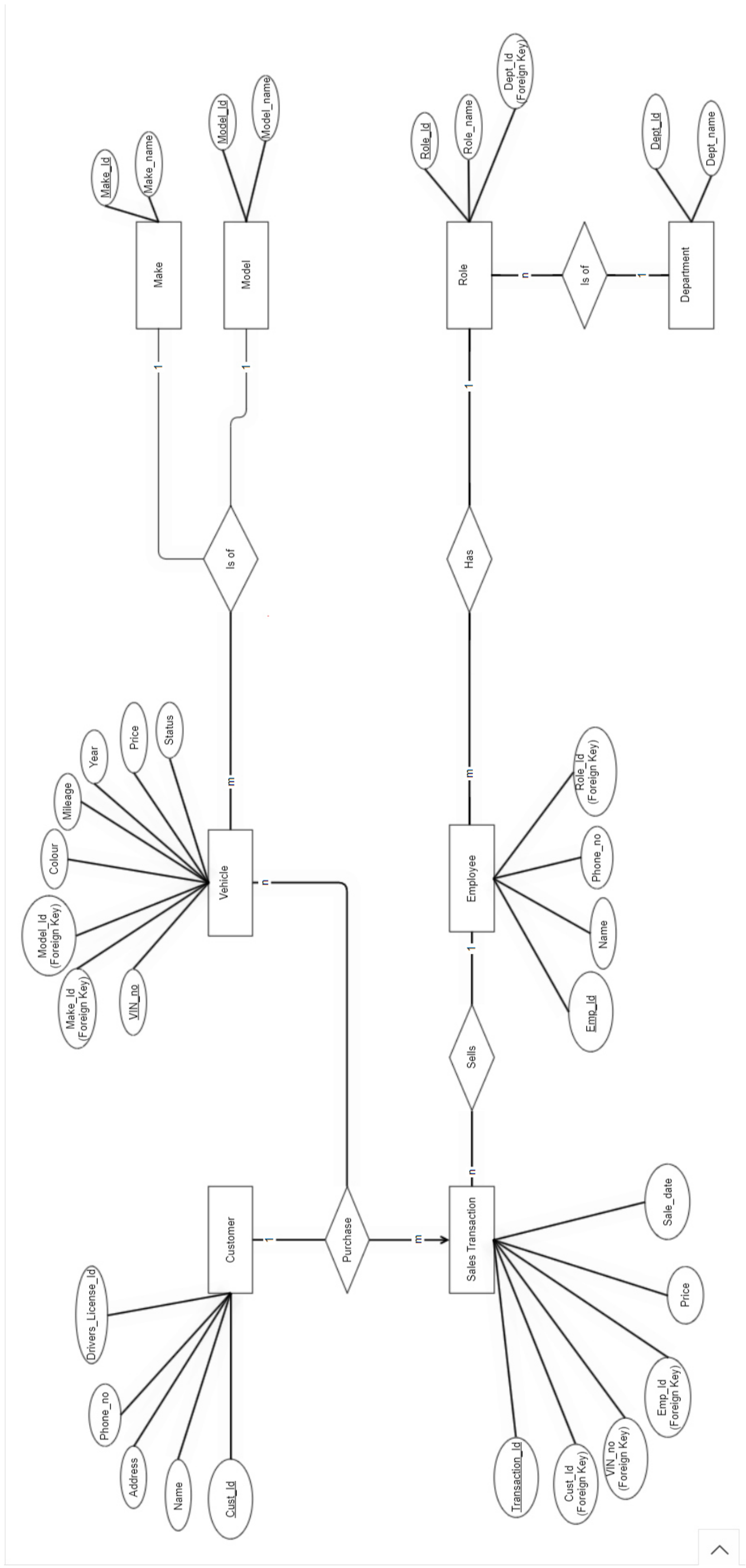
The modern car dealership thrives on efficiency and organization. In today's competitive market, managing inventory, customer relationships, and sales data effectively is crucial for success. This project aims to develop a robust Database Management System (DBMS) specifically designed to address the needs of a car dealership.

This DBMS will provide a centralized platform for storing, organizing, and retrieving critical information. By leveraging database technology, the system will offer functionalities to:

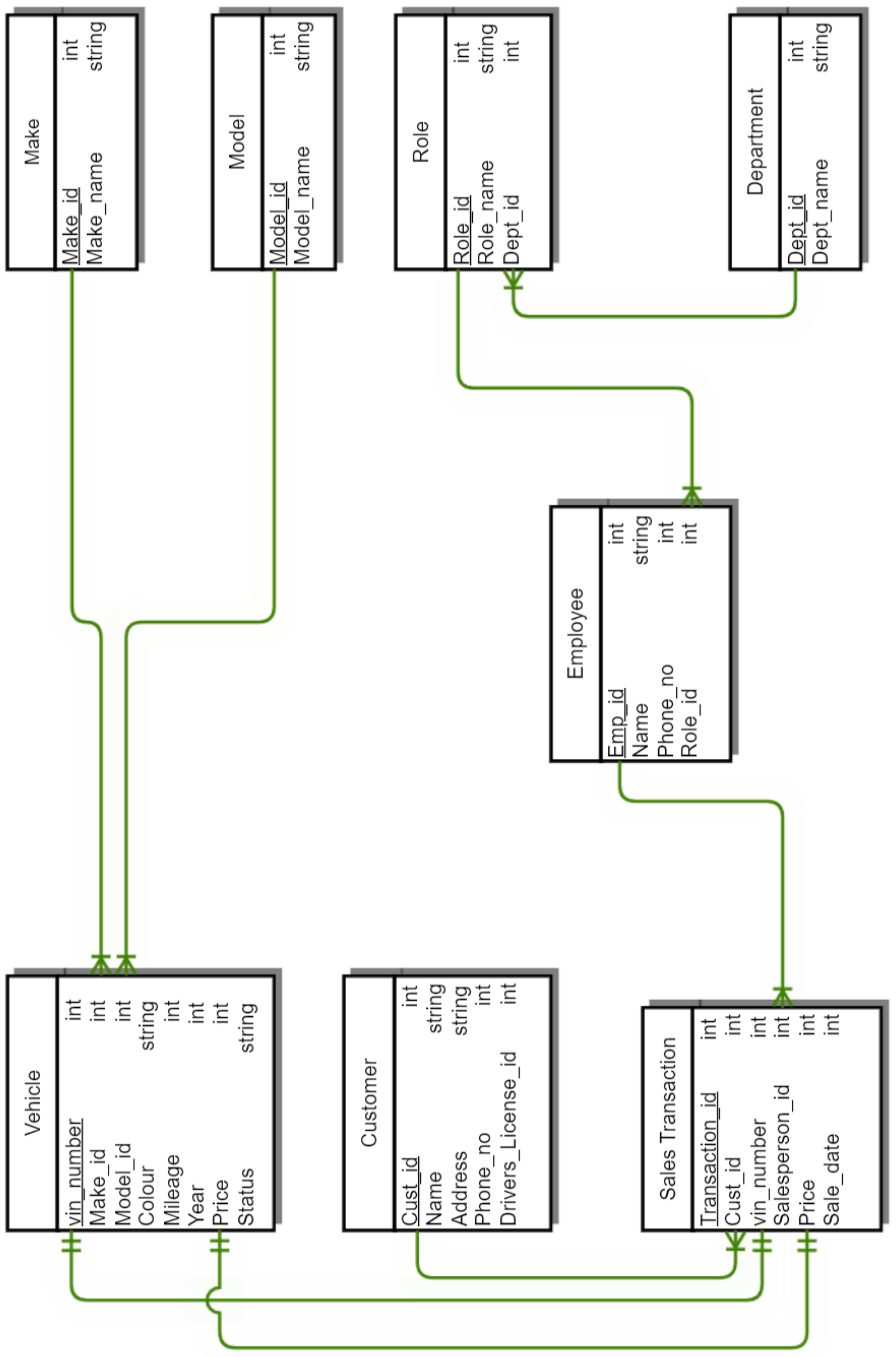
- **Manage Vehicle Inventory:** Detailed information on each vehicle, including make, model, year, features, and pricing will be readily available.
- **Track Customer Interactions:** Customer details, purchase history, and service records will be centralized, facilitating targeted marketing campaigns and improved customer service.
- **Streamline Sales and Finance:** The system will assist in managing sales transactions, financing options, and generating reports for informed decision-making.

This project will explore the design and implementation of the DBMS, outlining the data model, relationships between entities, and functionalities offered to users. The ultimate goal is to create a user-friendly and efficient system that empowers dealerships to operate at peak performance.

E-R Diagram


Make
Make_Id
Make_name
Model
Model_Id
Model_name
Role
Role_Id
Role_name
Dept_Id (Foreign Key)
Is of
Department
Dept_Id
Dept_name
Is of
Has
Vehicle
VIN_no
Make_Id (Foreign Key)
Model_Id (Foreign Key)
Colour
Mileage
Year
Price
Status
Employee
Role_Id (Foreign Key)
Phone_no
Name
Emp_Id
Sells
Customer
Drivers_License_Id
Phone_no
Address
Name
Cust_Id
Purchase
Sales Transaction
Sale_date
Price
Emp_Id (Foreign Key)
VIN_no (Foreign Key)
Cust_Id (Foreign Key)
Transaction_Id
1
n
m

E-R To Tables



Make
Make_id int
Make_name string
Model
Model_id int
Model_name string
Role
Role_id int
Role_name string
Dept_id int
Department
Dept_id int
Dept_name string
Employee
Emp_id int
Name string
Phone_no int
Role_id int
Vehicle
vin_number int
Make_id int
Model_id int
Colour string
Mileage int
Year int
Price int
Status string
Customer
Cust_id int
Name string
Address string
Phone_no int
Drivers_License_id int
Sales Transaction
Transaction_id int
Cust_id int
vin_number int
Salesperson_id int
Price int
Sale_date int

Normalization

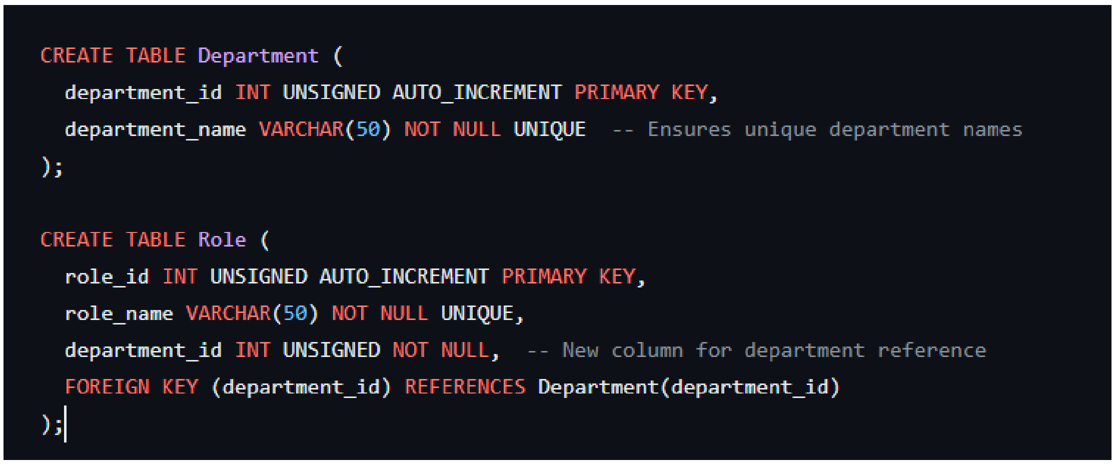
Table Name	Normalization Level	Notes
Customer	1NF	Phone number assumed to be single-valued. Address remains a single string.
Vehicle	1NF	Colour assumed to be single-valued.
Employee	1NF	Phone number assumed to be single-valued.
Sales_Transaction	1NF	All attributes assumed single-valued.
Make	3NF	Separate ID and name attributes suggest compliance.
Model	3NF	Separate ID and name attributes suggest compliance.
Dept	3NF	Separate ID and name attributes suggest compliance.
Role	3NF	Separate ID, role name, and department ID suggest compliance.

SQL and PL/SQL

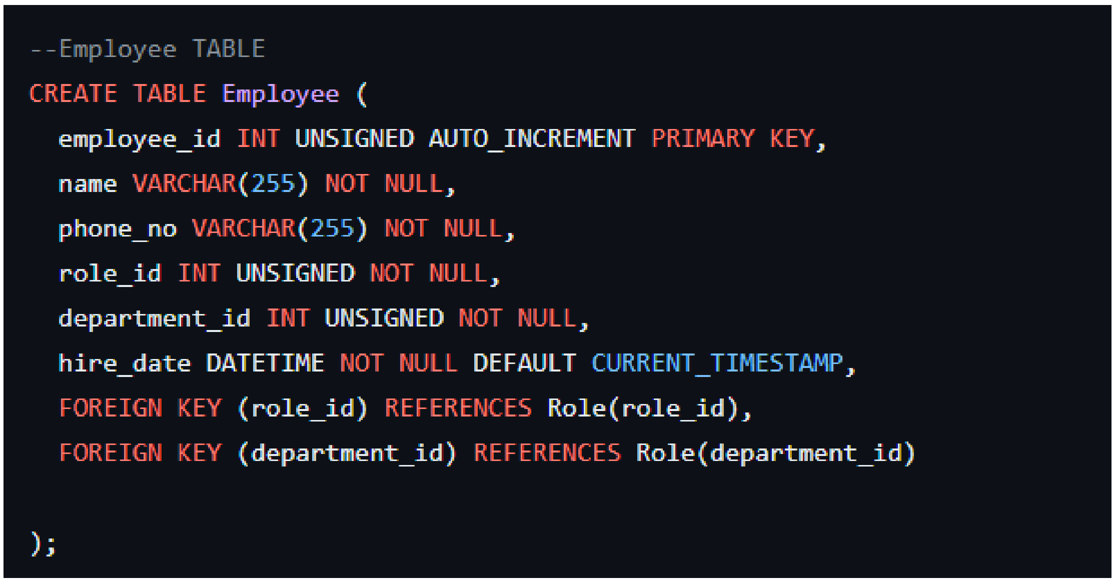
DATABASE OBJECTS USED IN OUR PROJECT

Objects	**Count**
TABLES	8
PROCEDURES	1
FUNCTIONS	4
TRIGGERS	3
CURSORS	6

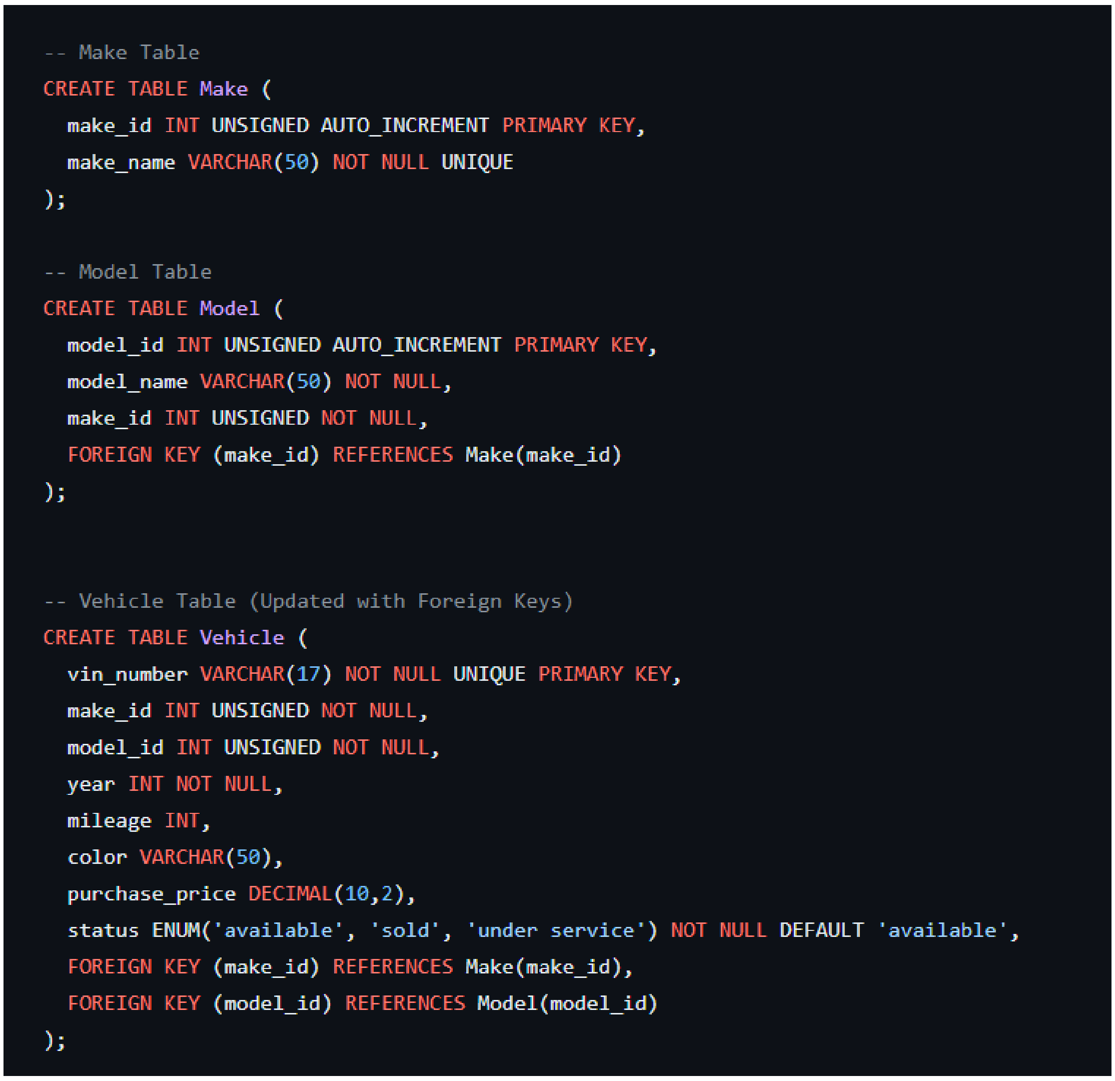
TABLES :



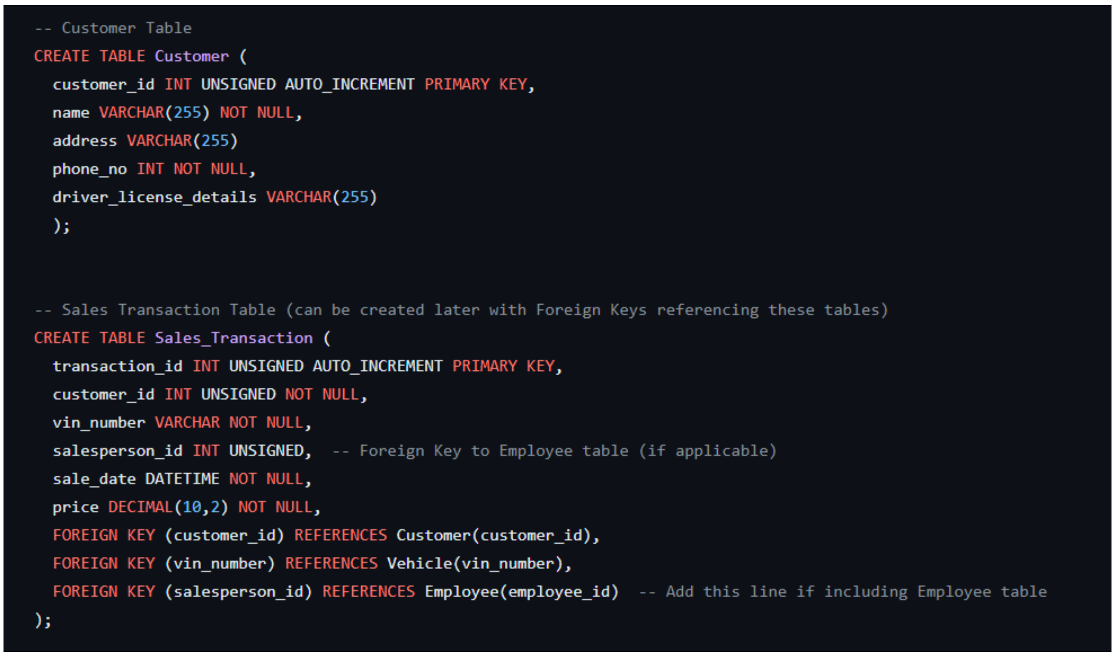
```sql
CREATE TABLE Department (
  department_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  department_name VARCHAR(50) NOT NULL UNIQUE  -- Ensures unique department names
);

CREATE TABLE Role (
  role_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  role_name VARCHAR(50) NOT NULL UNIQUE,
  department_id INT UNSIGNED NOT NULL,  -- New column for department reference
  FOREIGN KEY (department_id) REFERENCES Department(department_id)
);
```



```sql
--Employee TABLE
CREATE TABLE Employee (
  employee_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  phone_no VARCHAR(255) NOT NULL,
  role_id INT UNSIGNED NOT NULL,
  department_id INT UNSIGNED NOT NULL,
  hire_date DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (role_id) REFERENCES Role(role_id),
  FOREIGN KEY (department_id) REFERENCES Role(department_id)

);
```

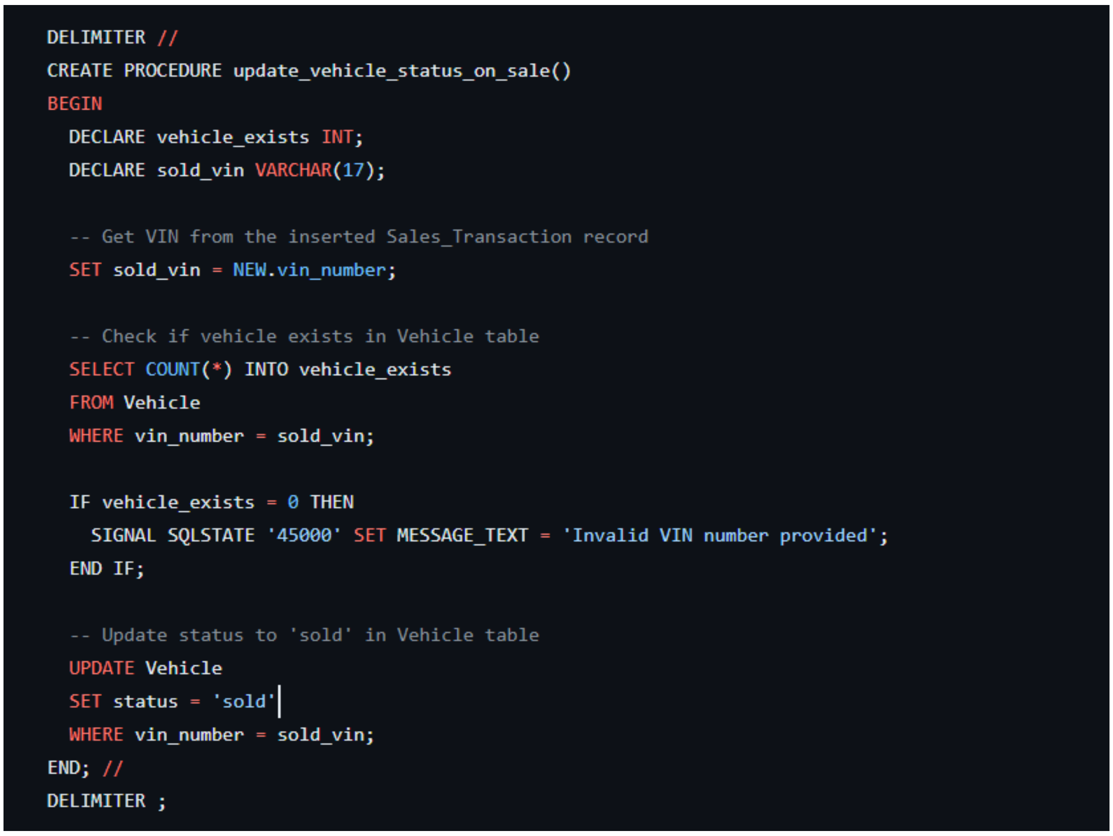


```sql
-- Make Table
CREATE TABLE Make (
  make_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  make_name VARCHAR(50) NOT NULL UNIQUE
);

-- Model Table
CREATE TABLE Model (
  model_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  model_name VARCHAR(50) NOT NULL,
  make_id INT UNSIGNED NOT NULL,
  FOREIGN KEY (make_id) REFERENCES Make(make_id)
);


-- Vehicle Table (Updated with Foreign Keys)
CREATE TABLE Vehicle (
  vin_number VARCHAR(17) NOT NULL UNIQUE PRIMARY KEY,
  make_id INT UNSIGNED NOT NULL,
  model_id INT UNSIGNED NOT NULL,
  year INT NOT NULL,
  mileage INT,
  color VARCHAR(50),
  purchase_price DECIMAL(10,2),
  status ENUM('available', 'sold', 'under service') NOT NULL DEFAULT 'available',
  FOREIGN KEY (make_id) REFERENCES Make(make_id),
  FOREIGN KEY (model_id) REFERENCES Model(model_id)
);
```



```sql
-- Customer Table
CREATE TABLE Customer (
  customer_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  address VARCHAR(255)
  phone_no INT NOT NULL,
  driver_license_details VARCHAR(255)
  );


-- Sales Transaction Table (can be created later with Foreign Keys referencing these tables)
CREATE TABLE Sales_Transaction (
  transaction_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  customer_id INT UNSIGNED NOT NULL,
  vin_number VARCHAR NOT NULL,
  salesperson_id INT UNSIGNED,  -- Foreign Key to Employee table (if applicable)
  sale_date DATETIME NOT NULL,
  price DECIMAL(10,2) NOT NULL,
  FOREIGN KEY (customer_id) REFERENCES Customer(customer_id),
  FOREIGN KEY (vin_number) REFERENCES Vehicle(vin_number),
  FOREIGN KEY (salesperson_id) REFERENCES Employee(employee_id)  -- Add this line if including Employee table
);
```

PROCEDURES :



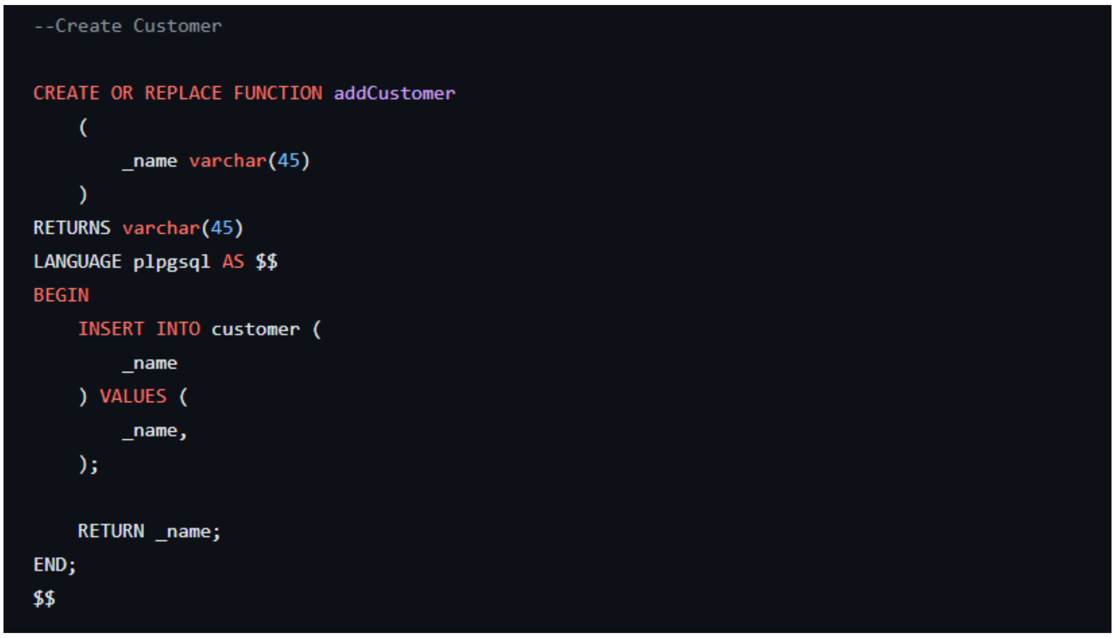
```
DELIMITER //
CREATE PROCEDURE update_vehicle_status_on_sale()
BEGIN
  DECLARE vehicle_exists INT;
  DECLARE sold_vin VARCHAR(17);

  -- Get VIN from the inserted Sales_Transaction record
  SET sold_vin = NEW.vin_number;

  -- Check if vehicle exists in Vehicle table
  SELECT COUNT(*) INTO vehicle_exists
  FROM Vehicle
  WHERE vin_number = sold_vin;

  IF vehicle_exists = 0 THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Invalid VIN number provided';
  END IF;

  -- Update status to 'sold' in Vehicle table
  UPDATE Vehicle
  SET status = 'sold'
  WHERE vin_number = sold_vin;
END; //
DELIMITER ;
```

This procedure is designed to update the status of a vehicle to 'sold' in the Vehicle table when a new sales transaction is recorded in the Sales_Transaction table. The procedure begins by retrieving the VIN of the sold vehicle from the newly inserted record in the Sales_Transaction table.

It then checks if a vehicle with the provided VIN exists in the Vehicle table by counting the occurrences of the VIN.

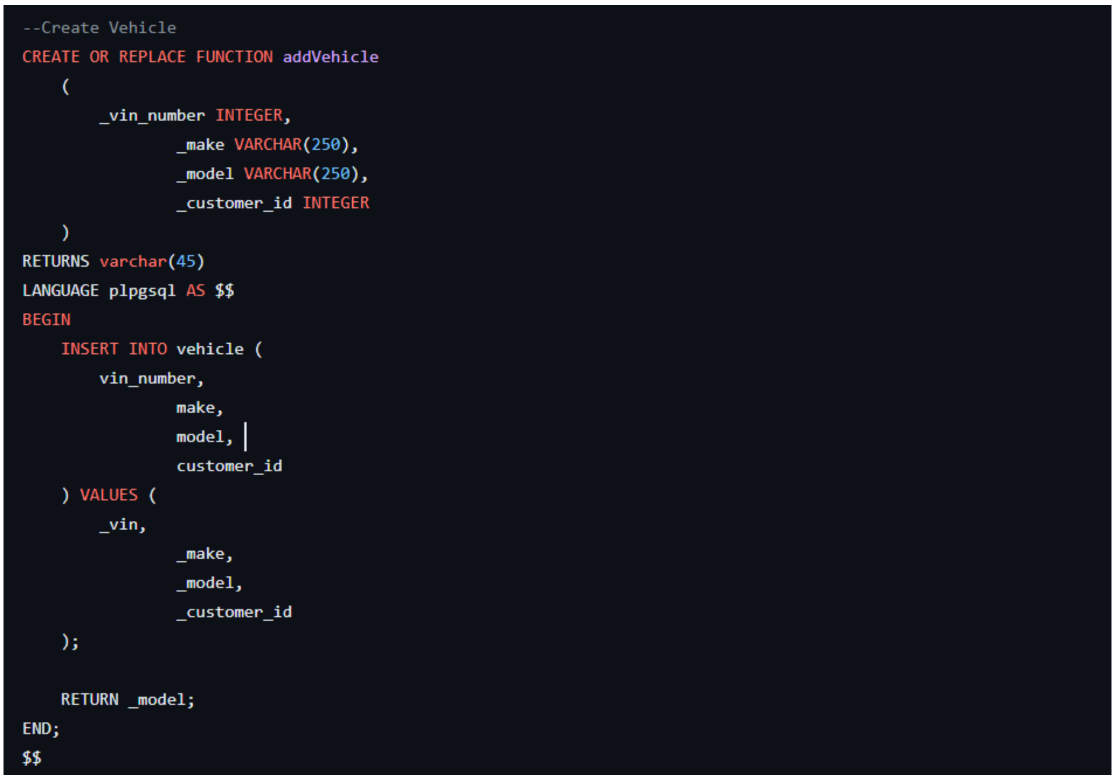
If the VIN is found in the Vehicle table (i.e., the count is not zero), the procedure proceeds to update the status of the corresponding vehicle to 'sold'.

FUNCTIONS :



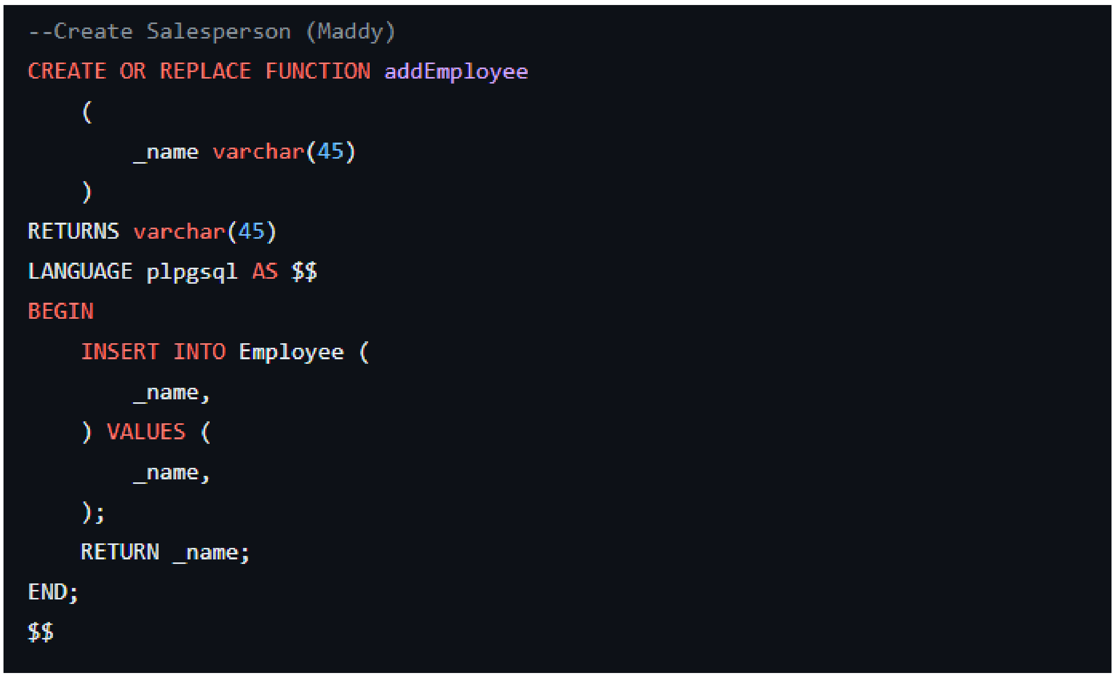
```
--Create Customer

CREATE OR REPLACE FUNCTION addCustomer
    (
        _name varchar(45)
    )
RETURNS varchar(45)
LANGUAGE plpgsql AS $$
BEGIN
    INSERT INTO customer (
        _name
    ) VALUES (
        _name,
    );

    RETURN _name;
END;
$$
```

addCustomer : This function is used to add a new customer to the database. It takes one parameter _name, which represents the name of the customer. Upon invocation, the function inserts a new record into the Customer table with the provided name. It then returns the name of the newly added customer.



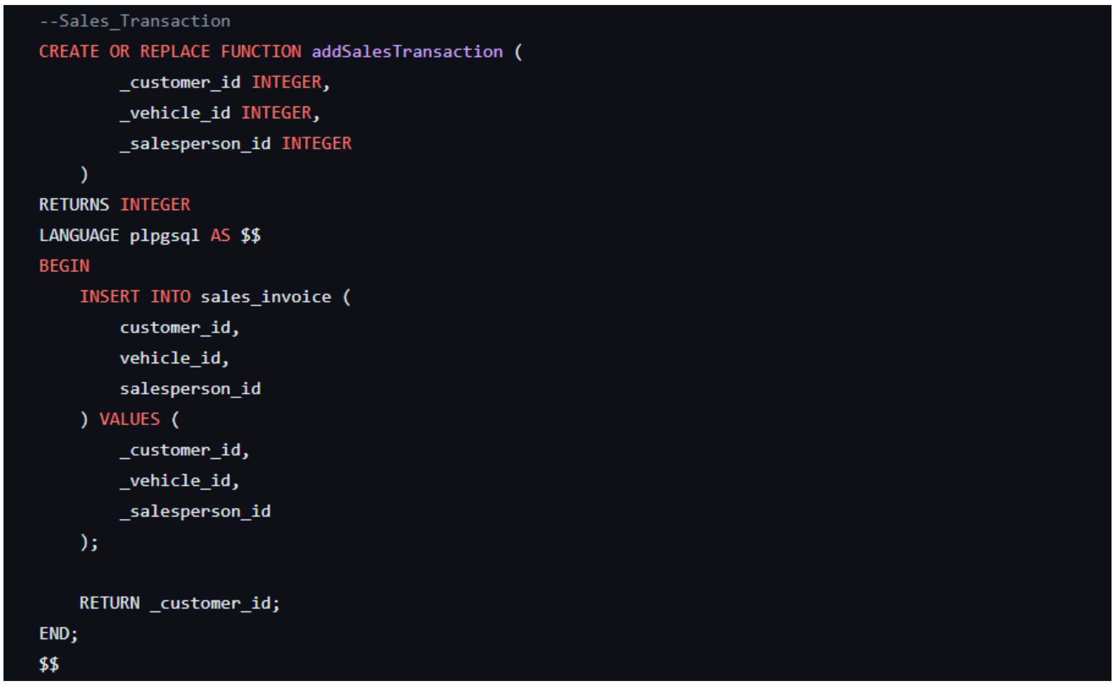
```
--Create Vehicle
CREATE OR REPLACE FUNCTION addVehicle
    (
        _vin_number INTEGER,
                _make VARCHAR(250),
                _model VARCHAR(250),
                _customer_id INTEGER
    )
RETURNS varchar(45)
LANGUAGE plpgsql AS $$
BEGIN
    INSERT INTO vehicle (
        vin_number,
                make,
                model,
                customer_id
    ) VALUES (
        _vin,
                _make,
                _model,
                _customer_id
    );

    RETURN _model;
END;
$$
```

addVehicle : This function is used to add a new vehicle to the database. It takes four parameters _vin_number, _make, _model, and _customer_id, representing the VIN number of the vehicle, its make, model, and the ID of the customer who owns the vehicle. When called, the function inserts a new record into the Vehicle table with the provided VIN number, make, model, and customer ID. It then returns the model of the newly added vehicle.



```
--Create Salesperson (Maddy)
CREATE OR REPLACE FUNCTION addEmployee
    (
        _name varchar(45)
    )
RETURNS varchar(45)
LANGUAGE plpgsql AS $$
BEGIN
    INSERT INTO Employee (
        _name,
    ) VALUES (
        _name,
    );
    RETURN _name;
END;
$$
```

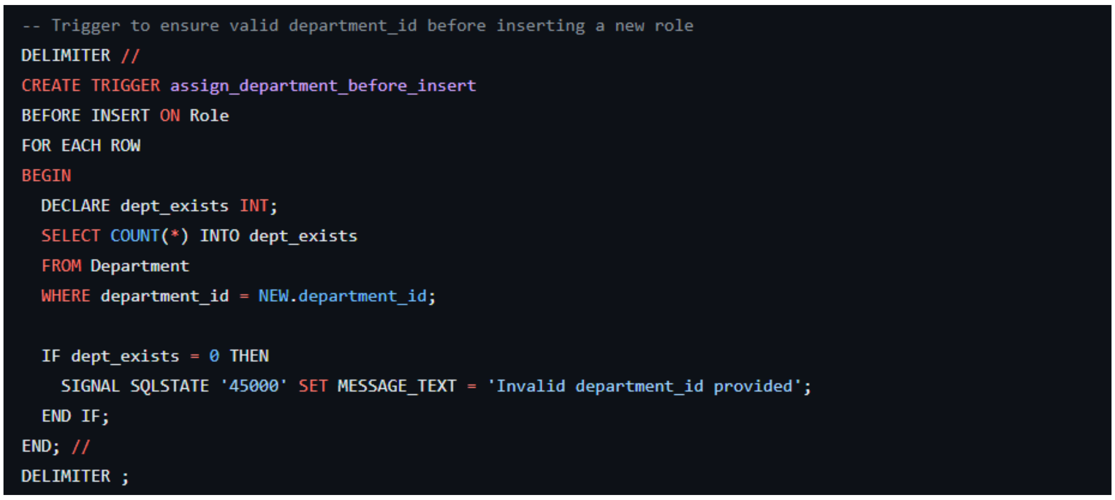
addEmployee : This function is used to add a new employee (such as a salesperson) to the database. It takes one parameter _name, representing the name of the employee. Upon execution, the function inserts a new record into the Employee table with the provided name. It then returns the name of the newly added employee.



```
--Sales_Transaction
CREATE OR REPLACE FUNCTION addSalesTransaction (
        _customer_id INTEGER,
        _vehicle_id INTEGER,
        _salesperson_id INTEGER
    )
RETURNS INTEGER
LANGUAGE plpgsql AS $$
BEGIN
    INSERT INTO sales_invoice (
        customer_id,
        vehicle_id,
        salesperson_id
    ) VALUES (
        _customer_id,
        _vehicle_id,
        _salesperson_id
    );

    RETURN _customer_id;
END;
$$
```

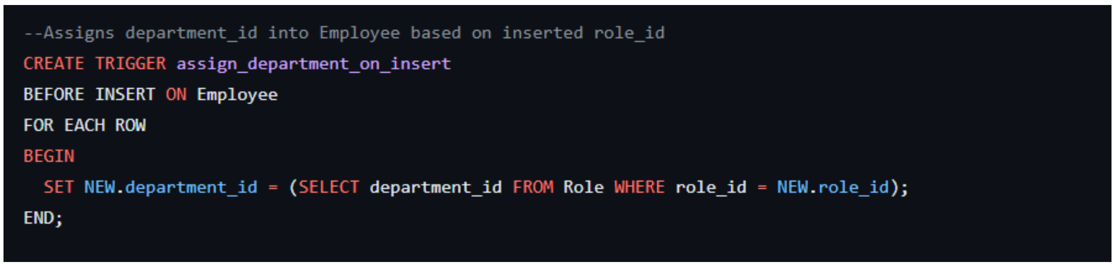
addSalesTransaction **:** This function is used to add a new sales transaction to the database. It takes three parameters _customer_id, _vehicle_id, and _salesperson_id, representing the IDs of the customer, vehicle, and salesperson involved in the transaction. When invoked, the function inserts a new record into the Sales_Transaction table with the provided customer ID, vehicle ID, and salesperson ID. It then returns the ID of the customer associated with the transaction.

TRIGGERS :



```
-- Trigger to ensure valid department_id before inserting a new role
DELIMITER //
CREATE TRIGGER assign_department_before_insert
BEFORE INSERT ON Role
FOR EACH ROW
BEGIN
  DECLARE dept_exists INT;
  SELECT COUNT(*) INTO dept_exists
  FROM Department
  WHERE department_id = NEW.department_id;

  IF dept_exists = 0 THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Invalid department_id provided';
  END IF;
END; //
DELIMITER ;
```

assign_department_before_insert : This trigger ensures that a valid department_id is provided before inserting a new role into the Role table. Before each insertion into the Role table, this trigger is activated. It checks if the department_id provided in the new role (referenced by NEW.department_id) exists in the Department table. If the department_id does not exist, it raises an error indicating that an invalid department_id was provided.



```
--Assigns department_id into Employee based on inserted role_id
CREATE TRIGGER assign_department_on_insert
BEFORE INSERT ON Employee
FOR EACH ROW
BEGIN
  SET NEW.department_id = (SELECT department_id FROM Role WHERE role_id = NEW.role_id);
END;
```

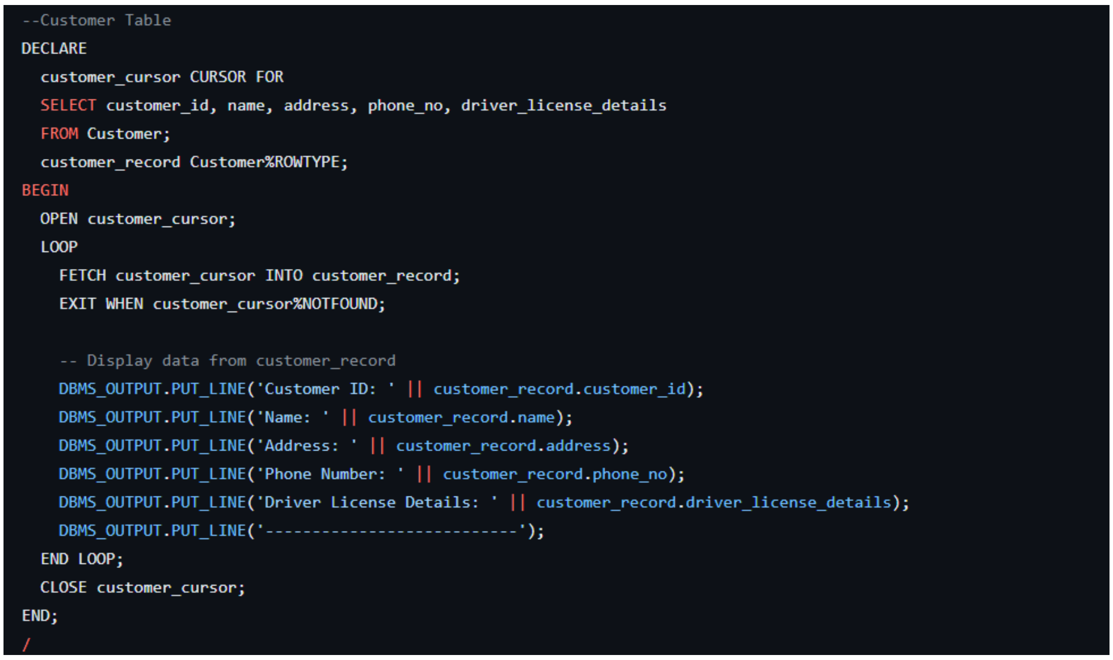
assign_department_on_insert : This trigger automatically assigns the department_id to an employee based on the inserted role_id. Before each insertion into the Employee table, this trigger is activated. It sets the department_id of the new employee (referenced by NEW.department_id) by querying the Role table to find the department_id associated with the inserted role_id (referenced by NEW.role_id).

```
CREATE TRIGGER update_vehicle_on_sale_after_insert
AFTER INSERT ON Sales_Transaction
FOR EACH ROW
BEGIN
  CALL update_vehicle_status_on_sale();
END;
```

update_vehicle_on_sale_after_insert : This trigger automatically updates the status of a vehicle to 'sold' after a new sales transaction is inserted into the Sales_Transaction table. After each insertion into the Sales_Transaction table, this trigger is activated. It calls the "update_vehicle_status_on_sale" procedure to update the status of the vehicle associated with the transaction to 'sold'. This ensures that the inventory status is accurately reflected after a vehicle sale.

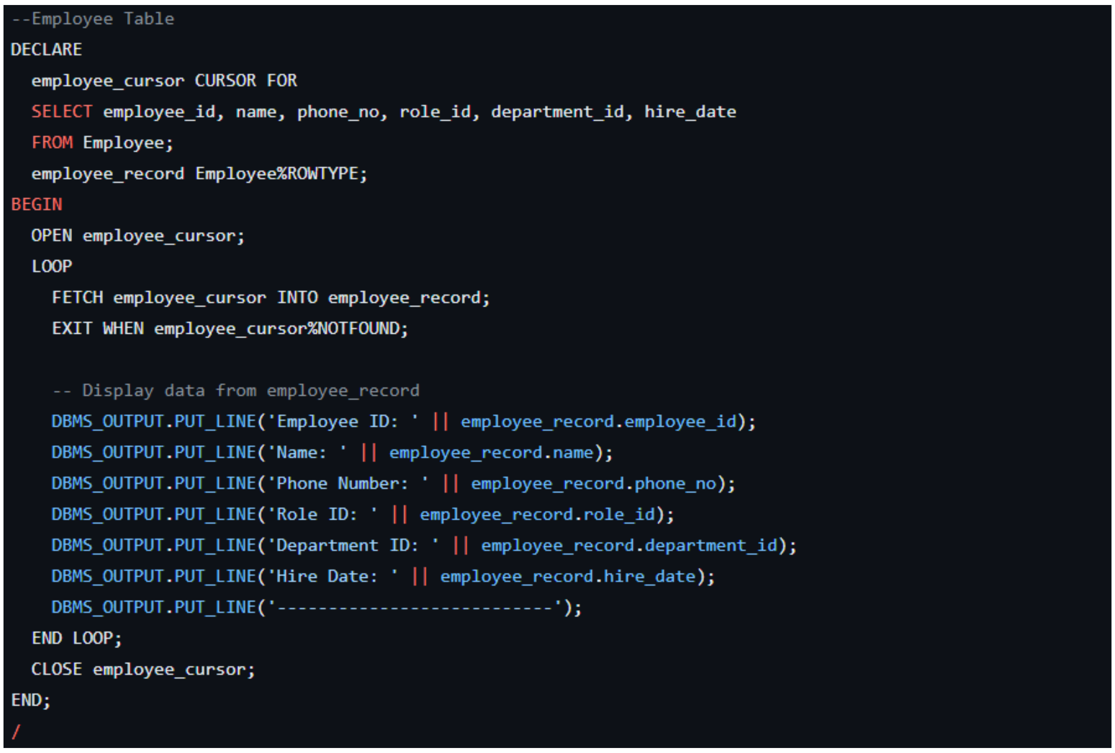
CURSORS :

These cursor declarations are used to fetch and display data from individual tables in the database.



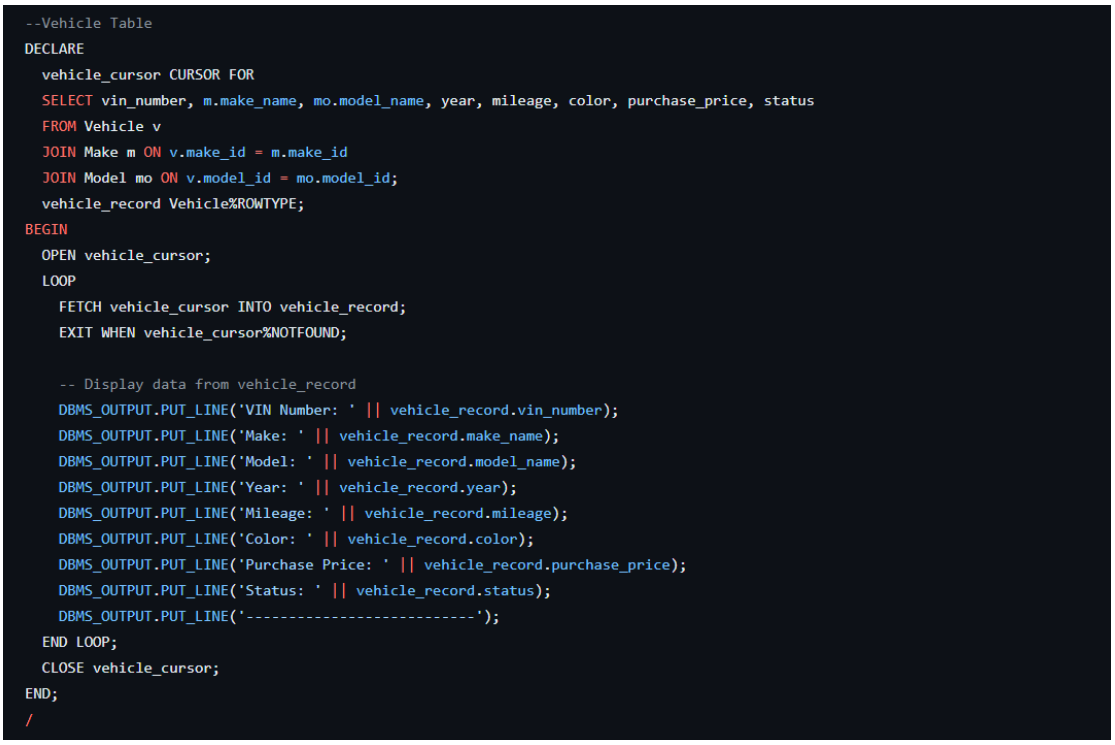
```
--Customer Table
DECLARE
  customer_cursor CURSOR FOR
  SELECT customer_id, name, address, phone_no, driver_license_details
  FROM Customer;
  customer_record Customer%ROWTYPE;
BEGIN
  OPEN customer_cursor;
  LOOP
    FETCH customer_cursor INTO customer_record;
    EXIT WHEN customer_cursor%NOTFOUND;

    -- Display data from customer_record
    DBMS_OUTPUT.PUT_LINE('Customer ID: ' || customer_record.customer_id);
    DBMS_OUTPUT.PUT_LINE('Name: ' || customer_record.name);
    DBMS_OUTPUT.PUT_LINE('Address: ' || customer_record.address);
    DBMS_OUTPUT.PUT_LINE('Phone Number: ' || customer_record.phone_no);
    DBMS_OUTPUT.PUT_LINE('Driver License Details: ' || customer_record.driver_license_details);
    DBMS_OUTPUT.PUT_LINE('--------------------------');
  END LOOP;
  CLOSE customer_cursor;
END;
/
```

Customer Table Cursor : This cursor fetches and displays data from the Customer table, such as customer ID, name, address, phone number, and driver license details. It declares a cursor named "customer_cursor" to select data from the Customer table. It opens the cursor and enters a loop to fetch each record from the cursor. Inside the loop, it fetches the data into the "customer_record" variable of type Customer%ROWTYPE. It then displays each field of the fetched record using DBMS_OUTPUT.PUT_LINE. Finally, it closes the cursor after processing all records.



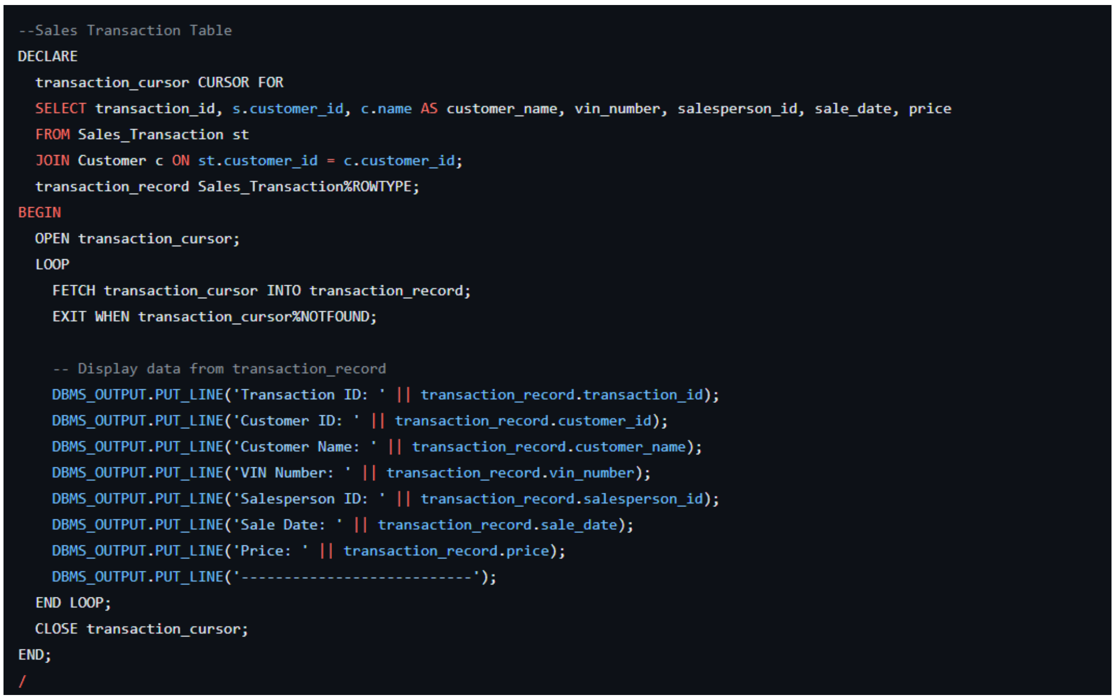
```
--Employee Table
DECLARE
  employee_cursor CURSOR FOR
  SELECT employee_id, name, phone_no, role_id, department_id, hire_date
  FROM Employee;
  employee_record Employee%ROWTYPE;
BEGIN
  OPEN employee_cursor;
  LOOP
    FETCH employee_cursor INTO employee_record;
    EXIT WHEN employee_cursor%NOTFOUND;

    -- Display data from employee_record
    DBMS_OUTPUT.PUT_LINE('Employee ID: ' || employee_record.employee_id);
    DBMS_OUTPUT.PUT_LINE('Name: ' || employee_record.name);
    DBMS_OUTPUT.PUT_LINE('Phone Number: ' || employee_record.phone_no);
    DBMS_OUTPUT.PUT_LINE('Role ID: ' || employee_record.role_id);
    DBMS_OUTPUT.PUT_LINE('Department ID: ' || employee_record.department_id);
    DBMS_OUTPUT.PUT_LINE('Hire Date: ' || employee_record.hire_date);
    DBMS_OUTPUT.PUT_LINE('---------------------------');
  END LOOP;
  CLOSE employee_cursor;
END;
/
```

Employee Table Cursor : This cursor fetches and displays data from the Employee table, including employee ID, name, phone number, role ID, department ID, and hire date. Similar to the customer cursor, it selects data from the Employee table, fetches records into the "employee_record" variable, and displays the fields.



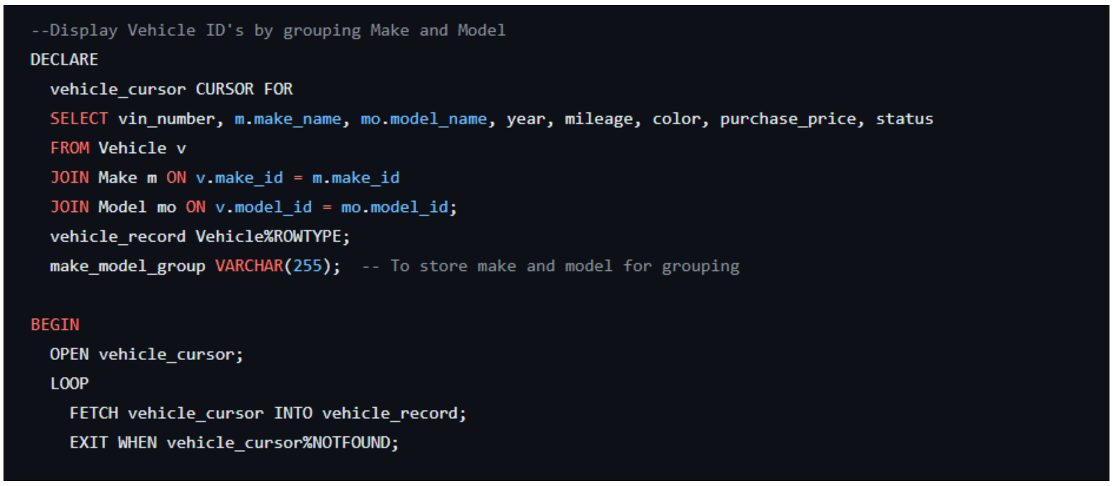
```
--Vehicle Table
DECLARE
  vehicle_cursor CURSOR FOR
  SELECT vin_number, m.make_name, mo.model_name, year, mileage, color, purchase_price, status
  FROM Vehicle v
  JOIN Make m ON v.make_id = m.make_id
  JOIN Model mo ON v.model_id = mo.model_id;
  vehicle_record Vehicle%ROWTYPE;
BEGIN
  OPEN vehicle_cursor;
  LOOP
    FETCH vehicle_cursor INTO vehicle_record;
    EXIT WHEN vehicle_cursor%NOTFOUND;

    -- Display data from vehicle_record
    DBMS_OUTPUT.PUT_LINE('VIN Number: ' || vehicle_record.vin_number);
    DBMS_OUTPUT.PUT_LINE('Make: ' || vehicle_record.make_name);
    DBMS_OUTPUT.PUT_LINE('Model: ' || vehicle_record.model_name);
    DBMS_OUTPUT.PUT_LINE('Year: ' || vehicle_record.year);
    DBMS_OUTPUT.PUT_LINE('Mileage: ' || vehicle_record.mileage);
    DBMS_OUTPUT.PUT_LINE('Color: ' || vehicle_record.color);
    DBMS_OUTPUT.PUT_LINE('Purchase Price: ' || vehicle_record.purchase_price);
    DBMS_OUTPUT.PUT_LINE('Status: ' || vehicle_record.status);
    DBMS_OUTPUT.PUT_LINE('---------------------------');
  END LOOP;
  CLOSE vehicle_cursor;
END;
/
```

<u>Vehicle Table Cursor</u> : This cursor fetches and displays data from the Vehicle table, along with related information from the Make and Model tables (such as make name and model name). It joins the Vehicle table with the Make and Model tables to retrieve additional information about each vehicle. The fetched records are displayed similarly to the previous cursors.

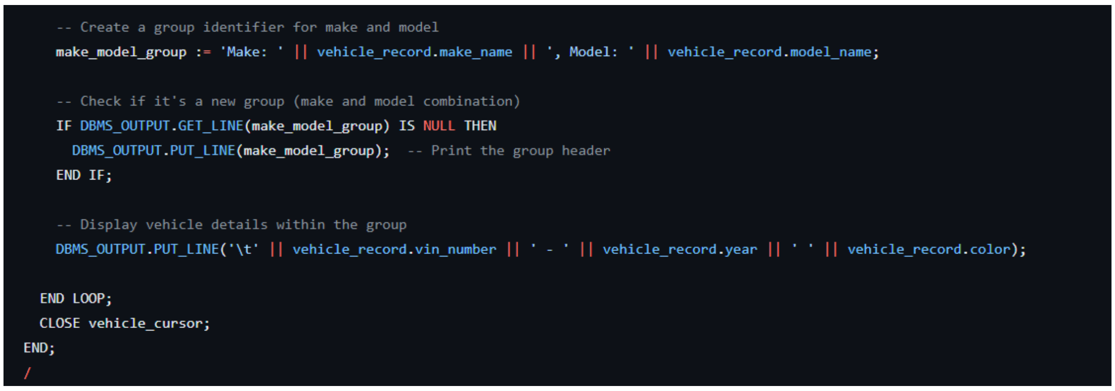


```
--Sales Transaction Table
DECLARE
  transaction_cursor CURSOR FOR
  SELECT transaction_id, s.customer_id, c.name AS customer_name, vin_number, salesperson_id, sale_date, price
  FROM Sales_Transaction st
  JOIN Customer c ON st.customer_id = c.customer_id;
  transaction_record Sales_Transaction%ROWTYPE;
BEGIN
  OPEN transaction_cursor;
  LOOP
    FETCH transaction_cursor INTO transaction_record;
    EXIT WHEN transaction_cursor%NOTFOUND;

    -- Display data from transaction_record
    DBMS_OUTPUT.PUT_LINE('Transaction ID: ' || transaction_record.transaction_id);
    DBMS_OUTPUT.PUT_LINE('Customer ID: ' || transaction_record.customer_id);
    DBMS_OUTPUT.PUT_LINE('Customer Name: ' || transaction_record.customer_name);
    DBMS_OUTPUT.PUT_LINE('VIN Number: ' || transaction_record.vin_number);
    DBMS_OUTPUT.PUT_LINE('Salesperson ID: ' || transaction_record.salesperson_id);
    DBMS_OUTPUT.PUT_LINE('Sale Date: ' || transaction_record.sale_date);
    DBMS_OUTPUT.PUT_LINE('Price: ' || transaction_record.price);
    DBMS_OUTPUT.PUT_LINE('---------------------------');
  END LOOP;
  CLOSE transaction_cursor;
END;
/
```

Sales Transaction Table Cursor : This cursor fetches and displays data from the Sales_Transaction table, including transaction ID, customer ID, customer name, VIN number, salesperson ID, sale date, and price. It joins the Sales_Transaction table with the Customer table to retrieve the customer name associated with each transaction. The fetched records are then displayed as before.



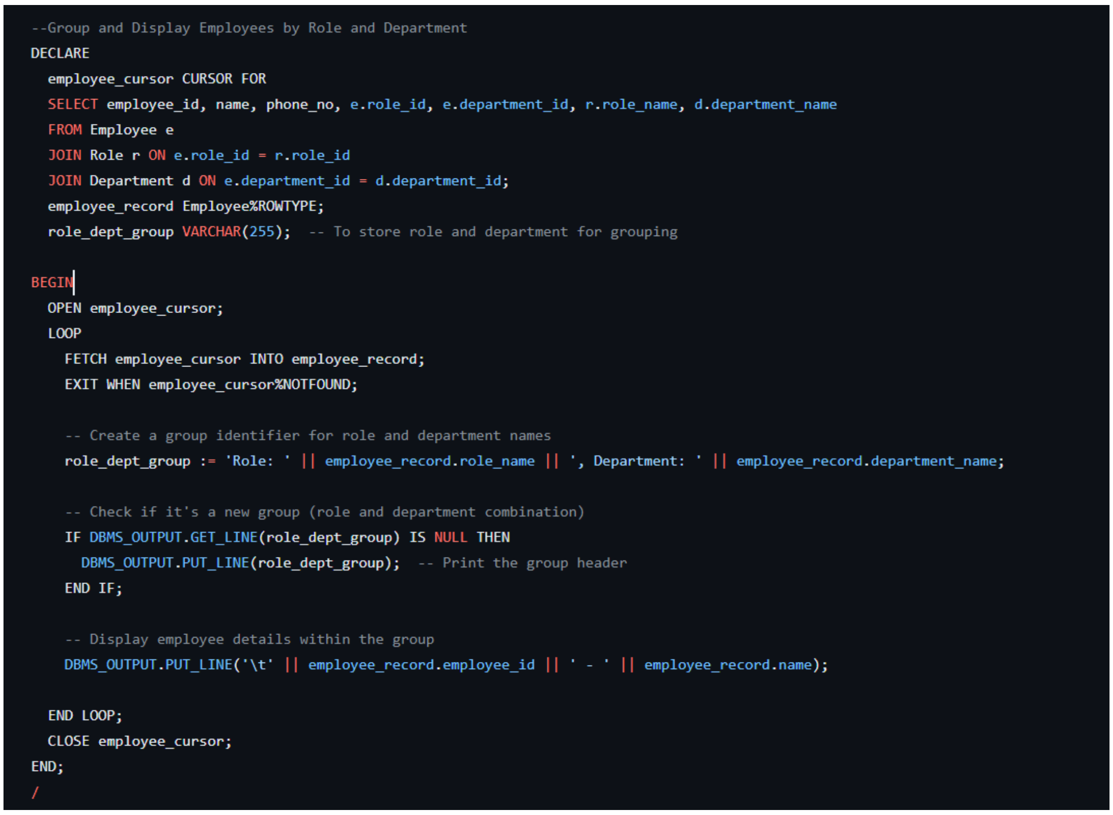
```
--Display Vehicle ID's by grouping Make and Model
DECLARE
  vehicle_cursor CURSOR FOR
  SELECT vin_number, m.make_name, mo.model_name, year, mileage, color, purchase_price, status
  FROM Vehicle v
  JOIN Make m ON v.make_id = m.make_id
  JOIN Model mo ON v.model_id = mo.model_id;
  vehicle_record Vehicle%ROWTYPE;
  make_model_group VARCHAR(255);  -- To store make and model for grouping

BEGIN
  OPEN vehicle_cursor;
  LOOP
    FETCH vehicle_cursor INTO vehicle_record;
    EXIT WHEN vehicle_cursor%NOTFOUND;
```



```
    -- Create a group identifier for make and model
    make_model_group := 'Make: ' || vehicle_record.make_name || ', Model: ' || vehicle_record.model_name;

    -- Check if it's a new group (make and model combination)
    IF DBMS_OUTPUT.GET_LINE(make_model_group) IS NULL THEN
      DBMS_OUTPUT.PUT_LINE(make_model_group);  -- Print the group header
    END IF;

    -- Display vehicle details within the group
    DBMS_OUTPUT.PUT_LINE('\t' || vehicle_record.vin_number || ' - ' || vehicle_record.year || ' ' || vehicle_record.color);

  END LOOP;
  CLOSE vehicle_cursor;
END;
/
```

Grouping and displaying data in a structured manner is crucial for several real-world purposes. Grouping vehicles by make and model allows dealerships to efficiently manage their inventory. It provides a clear overview of the available vehicles, making it easier to track sales, monitor stock levels, and identify popular models.

Vehicle Table Cursor : This cursor fetches and displays data from the Vehicle table, grouped by the make and model of the vehicles. It joins the Vehicle table with the Make and Model tables to retrieve additional information about each vehicle, such as make name and model name. Then, it creates a group identifier based on the combination of make and model. For each vehicle record, it checks if it belongs to a new group and prints the group header if necessary. Finally, it displays the details of each vehicle within its respective group.



```
--Group and Display Employees by Role and Department
DECLARE
  employee_cursor CURSOR FOR
  SELECT employee_id, name, phone_no, e.role_id, e.department_id, r.role_name, d.department_name
  FROM Employee e
  JOIN Role r ON e.role_id = r.role_id
  JOIN Department d ON e.department_id = d.department_id;
  employee_record Employee%ROWTYPE;
  role_dept_group VARCHAR(255);  -- To store role and department for grouping

BEGIN
  OPEN employee_cursor;
  LOOP
    FETCH employee_cursor INTO employee_record;
    EXIT WHEN employee_cursor%NOTFOUND;

    -- Create a group identifier for role and department names
    role_dept_group := 'Role: ' || employee_record.role_name || ', Department: ' || employee_record.department_name;

    -- Check if it's a new group (role and department combination)
    IF DBMS_OUTPUT.GET_LINE(role_dept_group) IS NULL THEN
      DBMS_OUTPUT.PUT_LINE(role_dept_group);  -- Print the group header
    END IF;

    -- Display employee details within the group
    DBMS_OUTPUT.PUT_LINE('\t' || employee_record.employee_id || ' - ' || employee_record.name);

  END LOOP;
  CLOSE employee_cursor;
END;
/
```

Employee Table Cursor (Grouped by Role and Department) : This cursor fetches and displays data from the Employee table, grouped by the role and department of the employees. Similar to the vehicle cursor, it joins the Employee table with the Role and Department tables to retrieve additional information about each employee, such as role name and department name. Then, it creates a group identifier based on the combination of role and department. For each employee record, it checks if it belongs to a new group and prints the group header if necessary. Finally, it displays the details of each employee within its respective group.

Conclusion

This project has successfully developed and implemented a Database Management System (DBMS) tailored for the needs of a modern car dealership. The system provides a centralized platform for managing vehicle inventory, customer interactions, and sales data, promoting efficiency and improved decision-making.

The key achievements of this project include:

Enhanced Data Organization: Critical information is now organized and readily accessible, eliminating the need for manual record-keeping and streamlining daily operations.

Improved Customer Service: Centralized customer data allows for targeted marketing campaigns and personalized service, fostering stronger customer relationships.

Informed Decision-Making: The ability to analyse and generate reports through sales data empowers dealerships to make informed decisions regarding inventory management, pricing strategies, and marketing efforts.

Looking forward, this DBMS can be further enhanced by integrating functionalities such as:

Online Inventory Management: Allowing customers to browse vehicle listings and schedule test drives directly through the system.

Service Appointment Scheduling: Enabling customers to schedule service appointments online, improving convenience and efficiency.

Data Analytics Integration: Providing deeper insights into customer behaviour and sales trends, enabling data-driven marketing strategies.

By continuing to develop and expand the capabilities of this DBMS, car dealerships can further optimize their operations and achieve a competitive advantage in the marketplace. This project serves as a foundation for a data-driven approach to car dealership management, paving the way for a more efficient and customer-centric future.