

Mobile Shop Management System Project Report

Introduction:

The mobile shop management system is a Database Management System that helps mobile shop owners manage their inventory, sales, and customer data. The system typically includes modules for managing the following:

- **Inventory:** The inventory module allows the shop owner to track the stock levels of all mobile phones in the store. This information can be used to ensure that the shop has enough stock to meet customer demand and to identify which phones are selling well and which are not.
- **Sales:** The sales module allows the shop owner to track the sale of mobile phones. This information can be used to generate reports on sales trends, identify top-selling phones, and track the performance of individual salespersons.
- **Customers:** The customer module allows the shop owner to store customer information, such as name, address, and phone number. This information can be used to target marketing campaigns and to provide personalized customer service.

Project Objectives:

The objectives of this project were to develop a mobile shop management system that would:

- Be easy to use and navigate
- Provide shop owners with the information they need to manage their inventory, sales, and customer data
- Be flexible enough to accommodate the needs of different types of mobile shops

Project Methodology:

The project was developed using the following methodology:

1. **Requirements gathering:** The first step was to gather requirements from potential users of the system. This involved interviewing mobile shop owners and salespersons to identify their needs and pain points.
2. **System design:** Based on the gathered requirements, a system design was created. This design outlined the architecture of the system, the functional requirements of each module, and the data that would be stored in the system.
3. **Development:** The system was developed using PostgreSQL. The development process involved creating the database schema, writing the code for each module, and testing the system for functionality and performance.

Functional Requirement

1. Supplier Management:

- The system should allow the addition, modification, and deletion of supplier records in the Supplier table.
- A supplier must have a unique supplier_id as the primary key.
- The system should store and manage supplier contact details such as email, phone number, and name (first name, middle name, last name).

2. Mobile Management:

- The system should support the addition, modification, and deletion of mobile device records in the Mobile table.
- Mobile devices have attributes like price, brand, model, RAM, ROM, battery, processor, screen size, operating system, and display quantity.

3. Stock Management:

- The system should manage stock records in the Stock table, including purchase date, quantity, and linking to a specific supplier and mobile device.
- The quantity field in the Stock table should be updated based on purchases.

4. Payment Management:

- The system should handle payment transactions in the Payment table, recording payment date, amount, payment mode, and linking to a specific supplier and stock.

- The total amount paid by a supplier for a particular stock should be calculated and updated in the Payment table.

5. Employee Management:

- The system should allow the addition, modification, and deletion of employee records in the Employee table.
- Each employee is identified by a unique employee_id and has attributes like name, salary, joining date, and role.

6. Warranty Management:

- The system should manage warranty details in the Warranty table, including type, duration, and starting date.
- Each warranty is identified by a unique warranty_id.

7. Customer Management:

- The system should handle customer records in the Customer table, including name, address, email, and phone number.
- Each customer is identified by a unique customer_id.

8. State Pincode Information:

- The system should store state and city information corresponding to pin codes in the StatePincodeInfo table.
- Each pin code should be unique.

9. Order Management:

- The system should manage customer orders in the OrderTable, including quantity, ordered date, and linking to a specific mobile device, customer, employee, and warranty.

- The quantity in the Stock table should be updated based on orders.

10. Bill Management:

- The system should generate bills in the Bill table, including total amount, date, and GST number.
- Each bill should be linked to a specific customer order.

These functional requirements provide an overview of the essential features and capabilities the database support. They serve as a foundation for developing and implementing the database in a project.

DDL Queries

-- Table 1: Supplier Table

```
CREATE TABLE Supplier (  
    supplier_id NUMERIC PRIMARY KEY,  
    email VARCHAR(255),  
    phoneno NUMERIC(15),  
    fname VARCHAR(50),  
    mname VARCHAR(50),  
    lname VARCHAR(50)  
);
```

-- Table 2: Mobile Table

```
CREATE TABLE Mobile (  
    mobile_id NUMERIC PRIMARY KEY,  
    price NUMERIC(10,2),  
    brand VARCHAR(50),  
    model VARCHAR(50),  
    RAM NUMERIC,  
    ROM NUMERIC,  
    battery VARCHAR(50),  
    processor VARCHAR(50),  
    screen_size NUMERIC(4,2),  
    operating_system VARCHAR(50),  
    display_quantity NUMERIC  
);
```

-- Table 3: Stock Table

```
CREATE TABLE Stock (  
    stock_id NUMERIC PRIMARY KEY,  
    mobile_id NUMERIC,  
    purchase_date DATE,  
    quantity NUMERIC,  
    supplier_id NUMERIC,  
    FOREIGN KEY (supplier_id) REFERENCES stock(supplier_id),  
    FOREIGN KEY (mobile_id) REFERENCES Mobile(mobile_id)  
);
```

-- Table 4: Payment Table

```
CREATE TABLE Payment (  
    payment_id NUMERIC PRIMARY KEY,  
    payment_date DATE,  
    payment_amount DECIMAL(10,2),  
    payment_mode VARCHAR(50),  
    supplier_id NUMERIC,  
    stock_id NUMERIC,  
    FOREIGN KEY (supplier_id) REFERENCES Supplier,  
    FOREIGN KEY (stock_id) REFERENCES Stock  
);
```

-- Table 5: Employee Table

```
CREATE TABLE Employee (  
    employee_id NUMERIC PRIMARY KEY,  
    name VARCHAR(100),  
    salary NUMERIC(10,2),  
    joining_date DATE,  
    role VARCHAR(50)  
);
```

-- Table 6: Customer Table

```
CREATE TABLE Customer (  
    customer_id NUMERIC PRIMARY KEY,  
    fname VARCHAR(50),  
    mname VARCHAR(50),  
    lname VARCHAR(50),  
    street VARCHAR(255),  
    city VARCHAR(50),  
    email VARCHAR(255),  
    phone_no NUMERIC  
);
```

```
-- Table 7: Warranty Table
CREATE TABLE Warranty (
    warranty_id NUMERIC PRIMARY KEY,
    type VARCHAR(50),
    duration NUMERIC,
    starting_date DATE
);
```

```
-- Table 8: State Pincode Info Table
CREATE TABLE StatePincodeInfo (
    pincode NUMERIC PRIMARY KEY,
    city VARCHAR(50),
    state VARCHAR(50)
);
```

```
-- Table 9: Order Table
CREATE TABLE OrderTable (
    order_id NUMERIC PRIMARY KEY,
    quantity NUMERIC,
    mobile_id NUMERIC,
    customer_id NUMERIC,
    ordered_date DATE,
    employee_id NUMERIC,
    warranty_id NUMERIC,
    FOREIGN KEY (mobile_id) REFERENCES Mobile(mobile_id),
    FOREIGN KEY (customer_id) REFERENCES
Customer(customer_id),
    FOREIGN KEY (employee_id) REFERENCES
Employee(employee_id),
    FOREIGN KEY (warranty_id) REFERENCES Warranty(warranty_id)
);
```


-- Table 10: Bill Table

```
CREATE TABLE Bill (  
    bill_id NUMERIC PRIMARY KEY,  
    total_amount NUMERIC(10,2),  
    date DATE,  
    gst_no VARCHAR(50),  
    order_id NUMERIC,  
    customer_id NUMERIC,  
    FOREIGN KEY (order_id) REFERENCES OrderTable(order_id),  
    FOREIGN KEY (customer_id) REFERENCES Customer(customer_id)  
);
```

SQL Queries

-- 1 : Retrieve the details of the most expensive mobile:

```
SELECT *  
FROM MOBILE  
ORDER BY PRICE DESC  
LIMIT 1;
```

	mobile_id [PK] numeric	price numeric (10,2)	brand character varying (50)	model character varying (50)	ram numeric	rom numeric	battery character varying (50)	processor character varying (50)	screen_size numeric (4,2)	operating_system character varying (50)	display_quantity numeric
1	102	80000.00	Apple	iPhone 13	6	128	3110 mAh	A15 Bionic	6.10	iOS	150

-- 2 : Retrieve the order details with the corresponding employee names.

```
SELECT O.ORDER_ID, O.QUANTITY, E.NAME AS EMPLOYEE_NAME  
FROM ORDERTABLE O  
JOIN EMPLOYEE E ON O.EMPLOYEE_ID = E.EMPLOYEE_ID;
```

	order_id numeric	quantity numeric	employee_name character varying (100)
1	1	2	Magan
2	2	1	Dhruvil
3	3	3	Malhar
4	4	2	Magan
5	5	1	Varun

-- 3 : Retrieve the number of orders placed by each customer.

```
SELECT C.FNAME, C.LNAME, COUNT(O.ORDER_ID) AS  
ORDER_COUNT  
FROM CUSTOMER C  
JOIN ORDERTABLE O ON C.CUSTOMER_ID = O.CUSTOMER_ID  
GROUP BY C.CUSTOMER_ID;
```

	fname character varying (50) 🔒	lname character varying (50) 🔒	order_count bigint 🔒
1	Chirag	Rathod	1
2	Dev	Mali	1
3	Nisarg	Soni	1
4	Dakshil	Gorasiya	1
5	Dhruv	Patel	1

-- 4 : List the mobiles with their brand and model having a price greater than 30000.

```
SELECT BRAND, MODEL  
FROM MOBILE  
WHERE PRICE > 30000;
```

	brand character varying (50) 🔒	model character varying (50) 🔒
1	Apple	iPhone 13

-- 5 : Show the customers who have placed orders with a total amount greater than 50000.

```
SELECT DISTINCT C.CUSTOMER_ID, C.FNAME, C.LNAME
FROM CUSTOMER C
JOIN ORDERTABLE O ON C.CUSTOMER_ID = O.CUSTOMER_ID
JOIN BILL B ON O.ORDER_ID = B.ORDER_ID
WHERE B.TOTAL_AMOUNT > 50000;
```

	customer_id [PK] numeric	fname character varying (50)	lname character varying (50)
1	2	Nisarg	Soni

-- 6 : List the customers who have ordered a mobile with a warranty duration of 24 months.

```
SELECT C.FNAME, C.LNAME
FROM CUSTOMER C
JOIN ORDERTABLE O ON C.CUSTOMER_ID = O.CUSTOMER_ID
JOIN WARRANTY W ON O.WARRANTY_ID = W.WARRANTY_ID
WHERE W.DURATION = 24;
```

	fname character varying (50)	lname character varying (50)
1	Dakshil	Gorasiya
2	Chirag	Rathod
3	Dev	Mali

-- 7 : Retrieve the orders placed in May 2023.

```
SELECT *  
FROM ORDERTABLE  
WHERE ORDERED_DATE BETWEEN '2023-05-01' AND '2023-05-31';
```

	order_id [PK] numeric	quantity numeric	mobile_id numeric	customer_id numeric	ordered_date date	employee_id numeric	warranty_id numeric
1	5	1	105	5	2023-05-20	5	5

-- 8 : Display the customers who live in Gujarat.

```
SELECT C.FNAME, C.LNAME  
FROM CUSTOMER C  
JOIN STATEPINCODEINFO SPI ON C.CITY = SPI.CITY  
WHERE SPI.STATE = 'Gujarat';
```

	fname character varying (50)	lname character varying (50)
1	Dakshil	Gorasiya
2	Nisarg	Soni
3	Chirag	Rathod
4	Dhruv	Patel
5	Dev	Mali

-- 9 : List the customers who have ordered mobiles with a screen size greater than 6.0 inches.

```
SELECT C.FNAME, C.LNAME
FROM CUSTOMER C
JOIN ORDERTABLE O ON C.CUSTOMER_ID = O.CUSTOMER_ID
JOIN MOBILE M ON O.MOBILE_ID = M.MOBILE_ID
WHERE M.SCREEN_SIZE > 6.0;
```

	fname character varying (50) 🔒	lname character varying (50) 🔒
1	Dakshil	Gorasiya
2	Nisarg	Soni
3	Chirag	Rathod
4	Dhruv	Patel

-- 10 : Display the total number of orders placed by each employee.

```
SELECT E.NAME, COUNT(O.ORDER_ID) AS TOTAL_ORDERS
FROM EMPLOYEE E
JOIN ORDERTABLE O ON E.EMPLOYEE_ID = O.EMPLOYEE_ID
GROUP BY E.EMPLOYEE_ID;
```

	name character varying (100) 🔒	total_orders bigint 🔒
1	Malhar	1
2	Magan	2
3	Dhruvil	1
4	Varun	1

Functions

-- 1 : Function to Calculate Total Stock Value by Mobile Brand

```
CREATE OR REPLACE FUNCTION
calculate_stock_value_by_brand(brand_name VARCHAR(50))
RETURNS NUMERIC(10, 2)
LANGUAGE plpgsql
AS $$
DECLARE
    total_value NUMERIC(10, 2);
BEGIN
    SELECT SUM(price * quantity) INTO total_value
    FROM Stock
    JOIN Mobile ON Stock.mobile_id = Mobile.mobile_id
    WHERE brand = brand_name;

    RETURN total_value;
END;
$$
```

```
select calculate_stock_value_by_brand('Apple');
```

-- 2 : Function to Calculate Total Salary Expense by Role

```
CREATE OR REPLACE FUNCTION
calculate_salary_expense_by_role(role_name VARCHAR(50))
RETURNS NUMERIC(10, 2)
LANGUAGE plpgsql
AS $$
DECLARE
    total_salary NUMERIC(10, 2);
BEGIN
    SELECT SUM(salary) INTO total_salary
    FROM Employee
    WHERE role = role_name;

    RETURN total_salary;
END;
$$
```

```
select calculate_sal
```


-- 3 : Function to Retrieve Order Details by ID

```
CREATE OR REPLACE FUNCTION get_order_details(orderid
NUMERIC)
RETURNS TABLE (
    quantity NUMERIC,
    mobile_brand VARCHAR(50),
    f_name VARCHAR(150),
    l_name VARCHAR(150),
    ordered_date DATE
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY SELECT o.quantity, m.brand, c.fname, c.lname,
o.ordered_date
    FROM OrderTable o
    JOIN Mobile m ON o.mobile_id = m.mobile_id
    JOIN Customer c ON o.customer_id = c.customer_id
    WHERE order_id = orderid;
END;
$$

select * from get_order_details(1);
```

-- 4 : Function to Calculate Total Revenue by Date Range

```
CREATE OR REPLACE FUNCTION
calculate_total_revenue_by_date_range(start_date DATE, end_date
DATE)
RETURNS NUMERIC(10, 2)
LANGUAGE plpgsql
AS $$
DECLARE
    total_revenue NUMERIC(10, 2);
BEGIN
    SELECT SUM(total_amount) INTO total_revenue
    FROM Bill
    WHERE date BETWEEN start_date AND end_date;

    RETURN total_revenue;
END;
$$

select calculate_total_revenue_by_date_range('2023-03-01','2023-03-
28');
```

-- 5 : Function to Retrieve Stock Quantity by Mobile ID

```
CREATE OR REPLACE FUNCTION
get_stock_quantity_by_mobile_id(mobileid NUMERIC)
RETURNS NUMERIC
LANGUAGE plpgsql
AS $$
DECLARE
    stock_quantity NUMERIC;
BEGIN
    SELECT SUM(quantity) INTO stock_quantity
    FROM Stock
    WHERE Stock.mobile_id = mobileid;

    RETURN stock_quantity;
END;
$$
```

```
select get_stock_quantity_by_mobile_id(101);
```

-- 6 : Function to Calculate Total Payment Amount by Payment Mode

```
CREATE OR REPLACE FUNCTION  
calculate_total_payment_amount_by_mode(paymentMode  
VARCHAR(50))
```

```
RETURNS DECIMAL(10, 2)
```

```
LANGUAGE plpgsql
```

```
AS $$
```

```
DECLARE
```

```
    total_amount DECIMAL(10, 2);
```

```
BEGIN
```

```
    SELECT SUM(payment_amount) INTO total_amount
```

```
    FROM Payment
```

```
    WHERE Payment.payment_mode = paymentMode;
```

```
    RETURN total_amount;
```

```
END;
```

```
$$
```

```
select calculate_total_payment_amount_by_mode('Cash');
```

-- 7 : Function to Retrieve State for a Given Pincode

```
CREATE OR REPLACE FUNCTION get_state_by_pincode(pc  
NUMERIC)
```

```
RETURNS VARCHAR(50)
```

```
LANGUAGE plpgsql
```

```
AS $$
```

```
DECLARE
```

```
    state_name VARCHAR(50);
```

```
BEGIN
```

```
    SELECT state INTO state_name
```

```
    FROM StatePincodeInfo
```

```
    WHERE pincode = pc;
```

```
    RETURN state_name;
```

```
END;
```

```
$$
```

```
select get_state_by_pincode(380001);
```

-- 8 : Function to Caculate Total Revenue by Employee

```
CREATE OR REPLACE FUNCTION
calculate_total_revenue_by_employee(employeeid NUMERIC)
RETURNS NUMERIC(10, 2)
LANGUAGE plpgsql
AS $$
DECLARE
    total_revenue NUMERIC(10, 2);
BEGIN
    SELECT SUM(total_amount) INTO total_revenue
    FROM Bill
    JOIN OrderTable ON Bill.order_id = OrderTable.order_id
    WHERE OrderTable.employee_id = employeeid;

    RETURN total_revenue;
END;
$$
```

```
select calculate_total_revenue_by_employee(1);
```

-- 9 : Function to Retrieve Mobiles with Low Stock

```
CREATE OR REPLACE FUNCTION
get_low_stock_mobiles(threshold_quantity NUMERIC)
RETURNS TABLE (
    mobile_id NUMERIC,
    brand VARCHAR(50),
    model VARCHAR(50),
    current_stock NUMERIC
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY SELECT M.mobile_id, M.brand, M.model,
S.quantity
    FROM Mobile M
    JOIN Stock S ON M.mobile_id = S.mobile_id
    WHERE S.quantity < threshold_quantity;
END;
$$

select * from get_low_stock_mobiles(30);
```

-- 10 : Function to Calculate Total Revenue by City

```
CREATE OR REPLACE FUNCTION calculate_total_revenue_by_city(ct
VARCHAR(50))
```

```
RETURNS NUMERIC(10, 2)
```

```
LANGUAGE plpgsql
```

```
AS $$
```

```
DECLARE
```

```
    total_revenue NUMERIC(10, 2);
```

```
BEGIN
```

```
    SELECT SUM(total_amount) INTO total_revenue
```

```
    FROM Bill
```

```
    JOIN OrderTable ON Bill.order_id = OrderTable.order_id
```

```
    JOIN Customer ON OrderTable.customer_id = Customer.customer_id
```

```
    WHERE Customer.city = ct;
```

```
    RETURN total_revenue;
```

```
END;
```

```
$$
```

```
select calculate_total_revenue_by_city('Anand');
```


Procedure

-- 1 : Procedure to Update Employee Salary

```
CREATE OR REPLACE PROCEDURE update_employee_salary(  
    IN employeeid NUMERIC,  
    IN new_salary NUMERIC(10, 2)  
)  
AS $$  
BEGIN  
    UPDATE Employee  
    SET salary = new_salary  
    WHERE Employee.employee_id = employeeid;  
END;  
$$ LANGUAGE plpgsql;  
  
CALL update_employee_salary(1,50000);  
  
select salary from employee where employee_id = 1;
```

-- 2 : Procedure to Update Stock Quantity for a Mobile

```
CREATE OR REPLACE PROCEDURE update_stock_quantity(  
    IN mobileid NUMERIC,  
    IN new_quantity NUMERIC  
)  
LANGUAGE plpgsql  
AS $$  
BEGIN  
    UPDATE Stock  
    SET quantity = quantity + new_quantity  
    WHERE Stock.mobile_id = mobileid;  
END;  
$$
```

```
CALL update_stock_quantity(102, 20);
```

```
select quantity from stock where mobile_id = 102;
```

-- 3 : Procedure to Insert Employee

```
CREATE OR REPLACE PROCEDURE insert_employee(  
    IN in_employee_id NUMERIC,  
    IN in_name VARCHAR(100),  
    IN in_salary NUMERIC(10,2),  
    IN in_joining_date DATE,  
    IN in_role VARCHAR(20)  
)  
LANGUAGE plpgsql  
AS $$  
BEGIN  
    INSERT INTO employee (employee_id, name, salary, joining_date,  
role)  
        VALUES (in_employee_id, in_name, in_salary, in_joining_date,  
in_role);  
END;  
$$  
  
CALL insert_employee(6, 'Tirth', 15000, '2023-11-23', 'Salesperson');  
  
select * from employee;
```

-- 4 : Procedure to Delete Expired Warranties

```
CREATE OR REPLACE PROCEDURE delete_expired_warranties()  
LANGUAGE plpgsql  
AS $$  
BEGIN  
    DELETE FROM Warranty  
    WHERE starting_date + interval '1 month' * duration <  
    CURRENT_DATE;  
END $$;  
  
CALL delete_expired_warranties();
```

-- 5 : Procedure to Update Mobile Prices with Discount

```
CREATE OR REPLACE PROCEDURE  
update_mobile_prices_with_discount(  
    IN discount_percentage NUMERIC  
)  
LANGUAGE plpgsql  
AS $$  
BEGIN  
    UPDATE mobile  
    SET price = price - (price * discount_percentage / 100);  
END $$;  
  
CALL update_mobile_prices_with_discount(10);
```

Cursor

-- 1 : Cursor to Retrieve Customer Information

```
CREATE OR REPLACE FUNCTION Cursor_to_get_customer_info()
RETURNS SETOF Customer
LANGUAGE plpgsql
AS $$
DECLARE
    customer_cursor CURSOR FOR SELECT * FROM Customer;
    customer_record Customer;
BEGIN
    OPEN customer_cursor;
    LOOP
        FETCH customer_cursor INTO customer_record;
        EXIT WHEN NOT FOUND;
        RETURN NEXT customer_record;
    END LOOP;
    CLOSE customer_cursor;
    RETURN;
END;
$$
```

```
SELECT * FROM Cursor_to_get_customer_info();
```

-- 2 : Cursor to Fetch Employees with Role 'Salesperson'

CREATE OR REPLACE FUNCTION Cursor_to_process_salespersons()

RETURNS SETOF Employee

LANGUAGE plpgsql

AS \$\$

DECLARE

 salespersons_cursor CURSOR FOR SELECT * FROM Employee
WHERE role = 'Salesperson';

 employee_record Employee%ROWTYPE;

BEGIN

 OPEN salespersons_cursor;

 LOOP

 FETCH salespersons_cursor INTO employee_record;

 EXIT WHEN NOT FOUND;

 RETURN NEXT employee_record;

 END LOOP;

 CLOSE salespersons_cursor;

 RETURN;

END;

\$\$;

SELECT * FROM Cursor_to_process_salespersons();

-- 3 : Cursor to Fetch Customers in Rajkot

CREATE OR REPLACE FUNCTION

Cursor_to_process_rajkot_customers()

RETURNS SETOF Customer

LANGUAGE plpgsql

AS \$\$

DECLARE

rajkot_customers_cursor CURSOR FOR SELECT * FROM Customer
WHERE city = 'Rajkot';

customer_record Customer%ROWTYPE;

BEGIN

OPEN rajkot_customers_cursor;

LOOP

FETCH rajkot_customers_cursor INTO customer_record;

EXIT WHEN NOT FOUND;

RETURN NEXT customer_record;

END LOOP;

CLOSE rajkot_customers_cursor;

RETURN;

END;

\$\$;

SELECT * FROM Cursor_to_process_rajkot_customers();

-- 4 : Cursor to Fetch Stock Purchased in 2023

```
CREATE OR REPLACE FUNCTION Cursor_to_process_stock_2023()
RETURNS SETOF Stock
LANGUAGE plpgsql
AS $$
DECLARE
    stock_2023_cursor CURSOR FOR SELECT * FROM Stock WHERE
EXTRACT(YEAR FROM purchase_date) = 2023;
    stock_record Stock%ROWTYPE;
BEGIN
    OPEN stock_2023_cursor;
    LOOP
        FETCH stock_2023_cursor INTO stock_record;
        EXIT WHEN NOT FOUND;

        RETURN NEXT stock_record;
    END LOOP;
    CLOSE stock_2023_cursor;
    RETURN;
END;
$;
```

```
SELECT * FROM Cursor_to_process_stock_2023();
```


-- 5 : Cursor to Fetch Orders Placed in May 2023:

```
CREATE OR REPLACE FUNCTION
Cursor_to_process_may_2023_orders()
RETURNS SETOF OrderTable
LANGUAGE plpgsql
AS $$
DECLARE
    may_orders_cursor CURSOR FOR SELECT * FROM OrderTable
    WHERE ordered_date BETWEEN '2023-05-01' AND '2023-05-31';
    order_record OrderTable%ROWTYPE;
BEGIN
    OPEN may_orders_cursor;
    LOOP
        FETCH may_orders_cursor INTO order_record;
        EXIT WHEN NOT FOUND;

        RETURN NEXT order_record;
    END LOOP;
    CLOSE may_orders_cursor;
    RETURN;
END;
$;
```

```
SELECT * FROM Cursor_to_process_may_2023_orders();
```

Trigger

-- 1 : Trigger to Update Stock Quantity After an Order

```
CREATE OR REPLACE FUNCTION  
update_stock_quantity_after_order()
```

```
RETURNS TRIGGER
```

```
LANGUAGE plpgsql
```

```
AS $$
```

```
BEGIN
```

```
    UPDATE Stock
```

```
    SET quantity = quantity - NEW.quantity
```

```
    WHERE mobile_id = NEW.mobile_id;
```

```
    RETURN NEW;
```

```
END;
```

```
$$
```

```
CREATE TRIGGER update_stock_quantity
```

```
AFTER INSERT ON ordertable
```

```
FOR EACH ROW
```

```
EXECUTE FUNCTION update_stock_quantity_after_order();
```

```
insert into ordertable values(100,40,102,1,'2023-01-01',1,1);
```

```
select * from ordertable;
```

```
select * from stock;
```

-- 2 : Trigger to Check Email Format in Customer Table

```
CREATE OR REPLACE FUNCTION check_customer_email_format()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    IF NEW.email !~ '^[a-zA-Z0-9._%~]+@[a-zA-Z0-9._%~]+\.[a-zA-Z]{2,4}$' THEN
        RAISE EXCEPTION 'Invalid email format.';
    END IF;
    RETURN NEW;
END;
$$
```

```
CREATE TRIGGER check_customer_email_format_trigger
BEFORE INSERT OR UPDATE ON Customer
FOR EACH ROW
EXECUTE FUNCTION check_customer_email_format();
```

```
insert into customer(customer_id, email) values(100,'error');
```

-- 3 : Trigger to Prevent Negative Stock Quantity

```
CREATE OR REPLACE FUNCTION prevent_negative_stock_quantity()  
RETURNS TRIGGER AS $$  
BEGIN  
    IF NEW.quantity < 0 THEN  
        RAISE EXCEPTION 'Stock quantity cannot be negative.';  
    END IF;  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER check_negative_stock_quantity  
BEFORE INSERT OR UPDATE ON Stock  
FOR EACH ROW  
EXECUTE FUNCTION prevent_negative_stock_quantity();
```

```
insert into stock values(999,101,'2020-11-12',-8);
```

-- 4 : Trigger to Prevent Deleting Suppliers with Active Stock

```
CREATE OR REPLACE FUNCTION prevent_delete_supplier()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    IF TG_OP = 'DELETE' THEN
        -- Check if there are active stocks for the supplier
        IF EXISTS (
            SELECT 1
            FROM Stock
            WHERE OLD.supplier_id = Stock.supplier_id
        ) THEN
            RAISE EXCEPTION 'Cannot delete supplier with active stock';
        END IF;
    END IF;

    RETURN OLD;
END $$;
```

```
CREATE TRIGGER prevent_delete_supplier_trigger
BEFORE DELETE ON Supplier
FOR EACH ROW
EXECUTE FUNCTION prevent_delete_supplier();

DELETE FROM Supplier WHERE supplier_id = 1;
```

-- 5 : Trigger to Enforce Unique Email

```
CREATE OR REPLACE FUNCTION enforce_unique_email()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    IF EXISTS (SELECT 1 FROM customer WHERE email = NEW.email)
    THEN
        RAISE EXCEPTION 'Email must be unique.';
    END IF;
    RETURN NEW;
END;
$$
```

```
CREATE TRIGGER enforce_unique_email_trigger
BEFORE INSERT ON customer
FOR EACH ROW
EXECUTE FUNCTION enforce_unique_email();
```

```
INSERT INTO customer(customer_id, email)
VALUES (100, 'dakshilgorasiya@gmail.com');
```

Project Results:

- The mobile shop management system was successfully developed and meets all of the project objectives.
- The mobile shop management system has the potential to significantly impact the operations and profitability of mobile shops.
- By streamlining inventory management, enhancing sales tracking, and fostering customer relationships, the system can contribute to increased sales, reduced costs, and improved customer satisfaction.

Conclusion:

The mobile shop management system is a valuable tool for mobile shop owners. The system can help shop owners to improve their efficiency, productivity, and profitability.