ECOMMERCE DATABASE

DBMS Project

112001024 - MAYANK

112001038 - SHAILAB

112001003 - ADITYA

Contents

Problem Statement

In this modern era of online shopping, no one wants to live under the rock and miss out on all the opportunities it provides whether be a customer or seller and how can we bridge the gap between them. But this task is easier said than done.

What can be the potential solution to bring them closer?

Solution

In order to bring the customer and the seller closer to each other, we have developed one of the most essential parts of an online store, i.e., a **database management system (DBMS)**.

This will help the business owners to sell their products not only for offline sales but also for the online sales. This will also help the customers to order their products while sitting at home and chilling with a cool beer rather than being outside in the crowded market and finding the right product.

Advantage

The advantage of this eCommerce website over a direct transactional business is that it brings the business owners, especially the smaller ones and customers closer and. It helps both of them in the context that it saves time for both of them as the customer can just order the product from their house while the seller just have to process the order rather than negotiate over the pricing of their product, etc.

Entity Description

ENTITY	ATTRIBUTES	DESCRIPTION
Customer	Customer_id Name Email Contact DOB	This entity will be used to store the information about the customers that register.
Address	Apartment Street City State Pincode	This entity will store the address of each customer that register.
Orders	Order_id Amount Date Status	This entity will store all the necessary information about the order placed by the customer.
Payment	Payment_id Mode	This entity keeps track of all the payments made by the customer.

Seller	Seller_id Name Contact	This entity will store the information of all the seller from which a product is purchased.
Product	Product_id Price Name Stock	This entity stores the details about a product and how much of it is left in the inventory.

Relation Description

ENTITIES	RELATION (Attribute Added)	TYPE
Customer Address	Resides_at	Many to One
Customer Payment	Makes	One to Many
Customer Orders	Places	One to Many
Payment Orders	For	One to One
Orders Product	Contains (Count)	Many to Many
Seller Product	Sells (Stock)	Many to Many

Tables

There will be a total of **EIGHT** tables. Six of them will be of the entities that we have described and two of them will be of the relations that we have described above.

Entity Tables

- Customer Table

```
CREATE TABLE Customer(
    customer_id int PRIMARY KEY,
    name char(50) NOT NULL,
    email char(255),
    contact char(50),
    DOB char(30)
);
```

- Product Table

```
CREATE TABLE Product(
    product_id int NOT NULL,
    name char(50) NOT NULL,
    price int NOT NULL,
    stock int DEFAULT -1,
    PRIMARY KEY(product_id)
);
```

- Orders Table

```
CREATE TABLE Orders(
    order_id int NOT NULL,
    date char(50) NOT NULL,
    status char(20) NOT NULL DEFAULT "Ordered",
    amount int default -1,
    customer_id int,
    PRIMARY KEY(order_id),
    FOREIGN KEY (customer_id)
    REFERENCES Customer (customer_id)
);
```

- Address Table

```
CREATE TABLE Address(
    apartment char(50) NOT NULL,
    street char(30) NOT NULL,
    city char(20) NOT NULL,
    state char(30) NOT NULL,
    pincode int not null,
    customer_id int,
    PRIMARY KEY(customer_id, pincode, apartment),
    FOREIGN KEY (customer_id)
    REFERENCES Customer(customer_id)
);
```

- Payment Table

```
CREATE TABLE Payment(
    payment_id int NOT NULL,
    mode char(50) NOT NULL,
    customer_id int,
    order_id int,
    PRIMARY KEY(payment_id),
    FOREIGN KEY (customer_id)
    REFERENCES Customer(customer_id),
    FOREIGN KEY (order_id)
    REFERENCES Orders(order_id)
);
```

- Seller Table

```
CREATE TABLE Seller(
    seller_id int NOT NULL,
    name char(50) NOT NULL,
    Contact char(20) NOT NULL,
    PRIMARY KEY(seller_id)
);
```

Relationship Tables

---

1. Contains Table

```
CREATE TABLE contains(
    counts int NOT NULL,
    product_id int,
    order_id int,
    PRIMARY KEY(product_id, order_id),
    FOREIGN KEY (product_id)
    REFERENCES Product(product_id),
    FOREIGN KEY (order_id)
    REFERENCES Orders(order_id)
);
```

---

2. Sells Table

```
CREATE TABLE sells(
    stock int NOT NULL,
    seller_id int,
    product_id int,
    PRIMARY KEY(seller_id, product_id),
    FOREIGN KEY (seller_id)
    REFERENCES Seller (seller_id),
    FOREIGN KEY (product_id)
    REFERENCES Product (product_id)
);
```

Queries

We've created a total of **FIVE** queries with of one of them performing a specific task. The queries are:

1. **How much worth of products are sold and how many orders are placed on a certain date.**

```
SELECT Orders.date,
COUNT(Order.order_id), SUM(Orders.amount)
FROM Orders
GROUP BY Orders.date
HAVING  Orders.date=<Orders.date>
```

Here <Orders.date> is the input parameter, where we'll give a specific date and the query will return the total orders placed on that date and the total worth of that order.

2. **How many products are ordered by a customer.**

```
SELECT Orders.Order_id SUM(Contains.counts)
FROM Orders NATURAL JOIN
Contains NATURAL JOIN
Product
GROUP BY Orders.Order_id
HAVING  Orders.order_id=<Orders.order_id>
```

Here <Orders.order_id> is the input parameter, where we'll give a specific order_id and the query will return the total number of products ordered by that customer_id.

3. Total amount each spent by each customer.

```
SELECT Customer.customer_id, SUM(Orders.amount)
FROM Orders
NATURAL JOIN Customer
GROUP BY Customer.customer_id
```

4. Calculate how many products each seller has.

```
SELECT s.seller_id, COUNT(p.product_id)
FROM sells as t
JOIN Seller as s ON
JOIN Product as p ON
GROUP BY s.seller_id
```

5. Different number of products each order has.

```
SELECT o.order_id COUNT(p.product_id)
FROM Orders as o,
Product as p,
Contains as c
WHERE o.order_id = c.order_id
AND c.product_id = p.product_id
GROUP BY o.order_id
```

Views

There are a total of **TWO** views that we've created:

1. *total_amount* : this view shows the total amount that has been spend by each customer.

```
CREATE VIEW total_amount AS
    SELECT customer_id, SUM(amount)
    FROM Orders
    NATURAL JOIN Customer
    GROUP BY customer_id
```

2. *total_product* : this view shows how many different products each seller has.

```
CREATE VIEW total_product AS
    SELECT COUNT
    FROM sells as t
    JOIN Seller as s ON
    t.seller_id = s.seller_id
    JOIN Product as p ON
    t.product_id = p.product_id
    GROUP BY s.seller_id
```

Roles and Privileges

We have created **THREE** different roles with 3 different privileges:

1. Mayank: No privileges.

```
CREATE ROLE Mayank
LOGIN
PASSWORD "password";
```

2. Shailab: Have privileges of creating new databases.

```
CREATE ROLE Shailab
CREATEDB
LOGIN
PASSWORD "password";
```

3. Aditya: Have all the privileges.

```
CREATE ROLE Aditya
SUPERUSER
LOGIN
PASSWORD "password";
```

Function

The function ***get_price*** will take *product_id* as an input and returns its price.

```
CREATE FUNCTION get_price(p_id int)
RETURNS int
LANGUAGE plpgsql
AS
$$
DECLARE
p_price int;
BEGIN
SELECT price INTO p_price
FROM Product
WHERE product_id = p_id;
RETURN p_price;
END;
$$;
```

We can call this function by the following method:

```
SELECT get_price(<product_id>)
```

Here <product_id> is the argument from the Product table whose price will be returned when calling this function.

Procedure

The procedure ***update_price*** will take an *amount* and a *product_id* as an input and will update the price of that product wrt that amount.

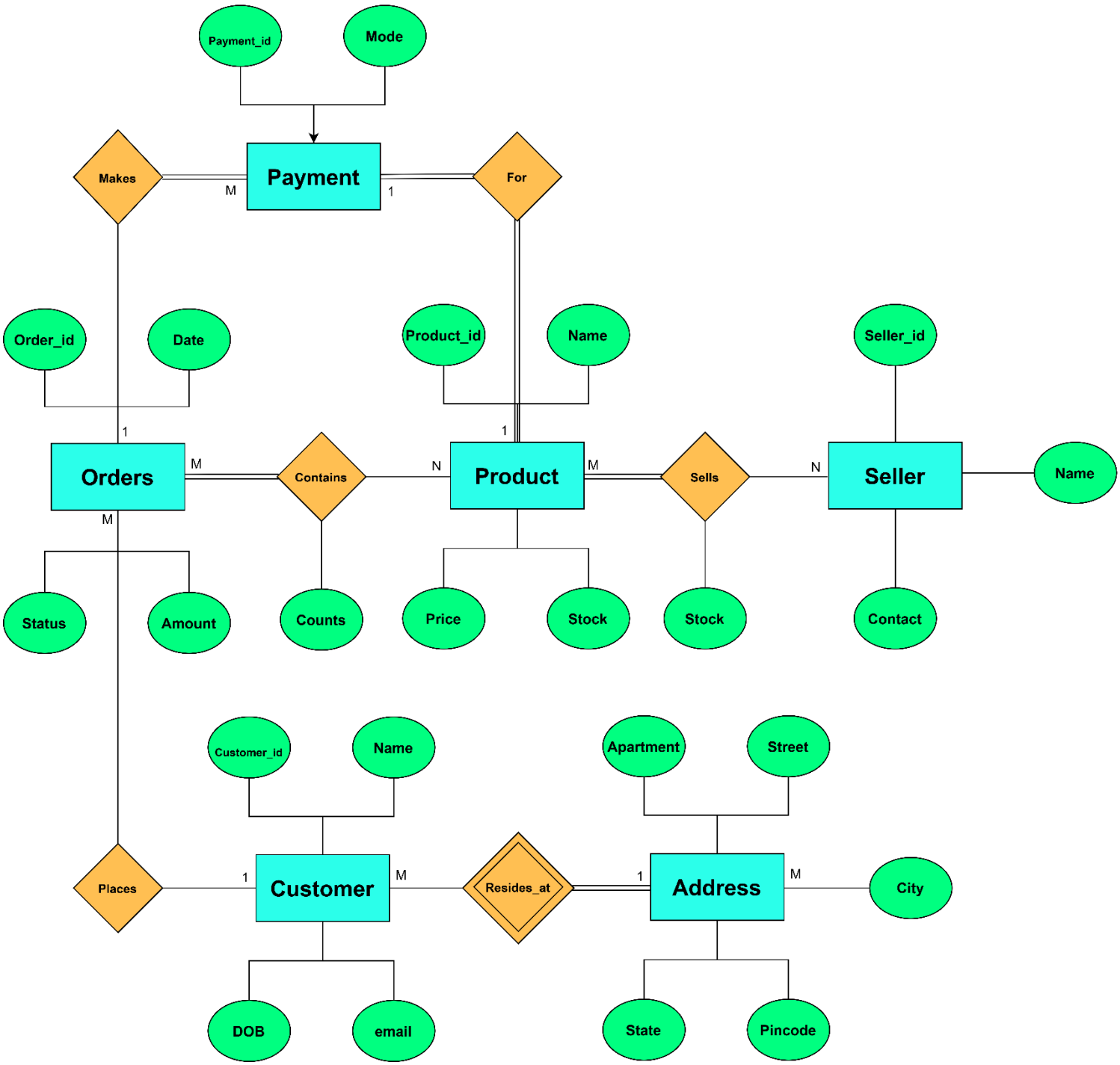
```
CREATE PROCEDURE update_price(change int, p_id int)
LANGUAGE plpgsql
AS
$$
DECLARE
p_price int;
BEGIN
UPDATE Product
SET price = price + change
WHERE product_id = p_id;
COMMIT;
END;
$$;
```

We can call this procedure by the following method:

```
CALL update_price(<change>, <product_id>)
```

Here <change> is the amount by which we've to change the price of the product and <product_id> is the argument from the Product table for which we've to change the price.

Entity-Relationship Model



Relational Model