2022

Ecommerce

TEAM MEMBERS
112001024: MAYANK RAWAT
112001038: SHAILAB CHAUHAN
112001003: ADITYA RAJ

Contents

Problem Statement 2

Solution 2

Advantage 2

Relational Diagram 3

Entity Description 4

Relation Description 5

Tables 6

- **Entity Tables** 6
- **Relationship Tables** 8

Role 9

Views 10

Function and Procedures 11

Trigger 15

Index 16

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

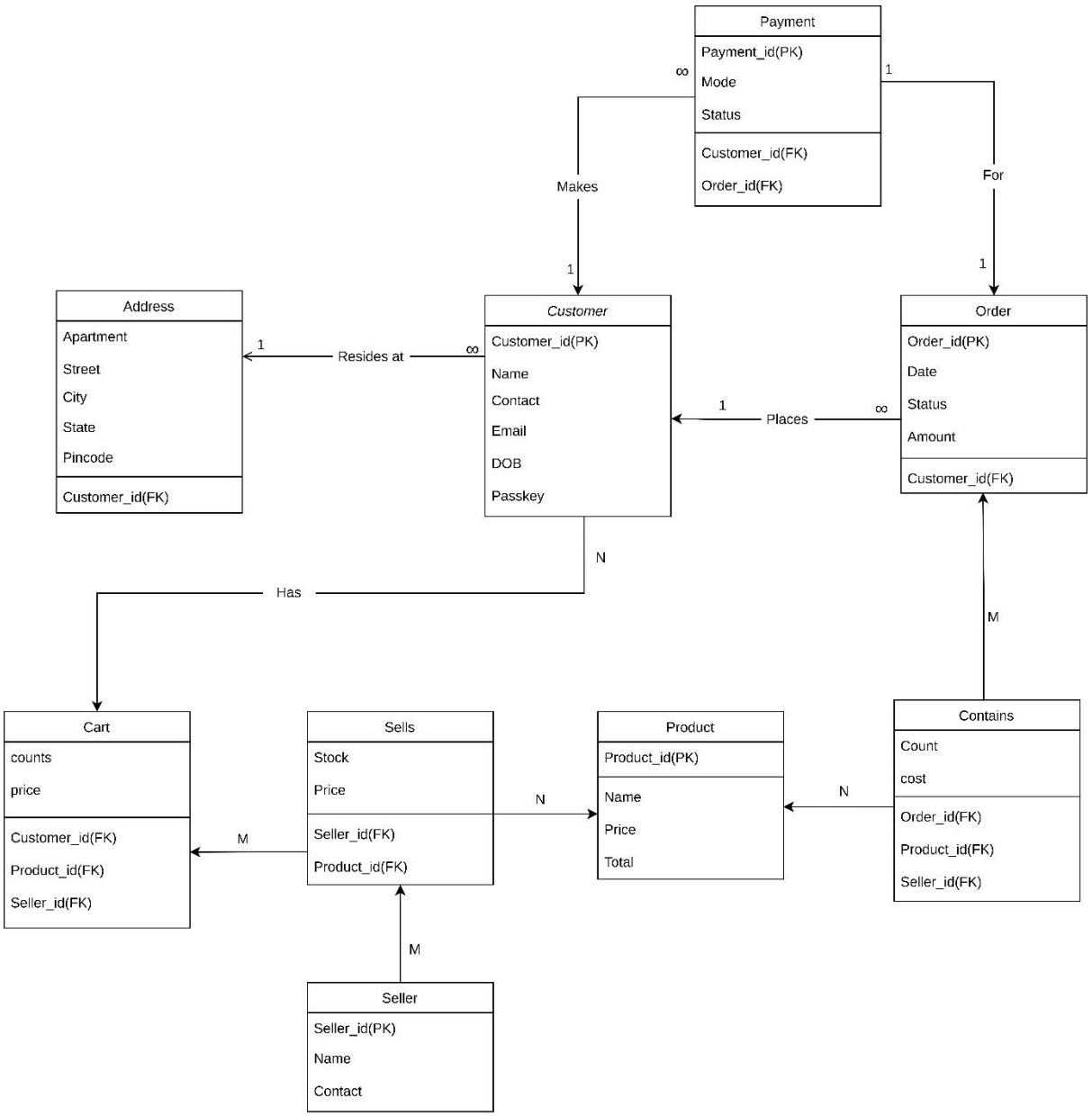
In order to bring the customer and the seller closer to each other, we have developed one of the most essential parts of an online store, i.e., a **database management system (DBMS)**.

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.

Relational Diagram



Entity Description

Entities	Attributes	Description
Customer	Customer_id Name Email Contact Passkey 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.
Cart	Counts Price	This entity will store the products that the customer wants to buy.
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 Status	This entity keeps track of all the payments made by the customer.
Seller	Seller_id sname Contact	This entity will store the information of all the seller from which a product is purchased.
Product	Product_id Pname Total	This entity stores the details about a product and how much of it is left in the inventory.

Relation Description

Entities	Relation (Attributes 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) (Cost)	Many to Many
Seller Product	Sells (Stock) (Price)	Many to Many
Cart Customer	Has	Many to One

Tables

There will be a total of **NINE** tables. Seven of which will be of the entities that we have described and two of them will be of the relations as described above.

Entity Tables

- Customer Table

```
CREATE TABLE Customer (
customer_id int PRIMARY KEY,
name char (30) NOT NULL,
email char (50) NOT NULL,
contact char (15) NOT NULL,
passkey text,
DOB char (12)
);
```

- Product Table

```
CREATE TABLE Product (
product_id int NOT NULL,
pname char (50) NOT NULL,
total 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) 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 (5) NOT NULL,
street char (15) NOT NULL,
city char (20) NOT NULL,
state char (20) NOT NULL,
pincode int NOT NULL,
customer_id int,
PRIMARY KEY (customer_id, pincode, apartment, street),
FOREIGN KEY (customer_id) REFERENCES Customer (customer_id)
);
```

- Payment Table

```
CREATE TABLE Payment (
payment_id int NOT NULL,
mode char (20) NOT NULL,
status char (10) DEFAULT 'Pending',
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,
sname char (25) NOT NULL,
Contact char (10) NOT NULL,
PRIMARY KEY (seller_id)
);
```

- Cart Table

```
CREATE TABLE Cart (
customer_id int,
product_id int,
seller_id int,
counts int DEFAULT 1,
price int NOT NULL,
PRIMARY KEY (customer_id, product_id, seller_id),
FOREIGN KEY (customer_id) REFERENCES Customer (customer_id),
FOREIGN KEY (seller_id, product_id) REFERENCES sells (seller_id,
product_id)
);
```

Relationship Tables

- Contains Table

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

- Sells Table

```
CREATE TABLE sells (
stock int DEFAULT 0,
price decimal,
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)
);
```

Role

There are **THREE** roles in our database.

- Customer: This role is for the buyers, with each customer having their own role. Each customer will have some privileges and policies on them.

- Seller: This role is for the business owners who wants to sells their products on our website.

- Admin: The most important role with all the privileges on the whole database and necessary views to keep track of the website.

```
CREATE ROLE <customer_username> LOGIN PASSWORD 'passkey';
GRANT SELECT ON Customer, Address, Payment, Orders, sells, contains
to <customer_username>;

CREATE ROLE seller_role LOGIN PASSWORD 'new_seller';
GRANT SELECT, INSERT, UPDATE ON sells TO seller_role;
GRANT SELECT, INSERT, UPDATE ON Product TO seller_role;
GRANT SELECT, INSERT, UPDATE ON Seller TO seller_role;
GRANT SELECT ON product_price_avg TO seller_role;

CREATE ROLE admin LOGIN PASSWORD 'admindatabase';
GRANT ALL ON ALL TABLES IN SCHEMA "public" To admin;
```

NOTE: The necessary privileges and policies for customer are done in function *role_creation()*. Please refer to that function for more details.

Views

We have created **FIVE** views with each view with some specific role:

- *Customer_product_details:* The customer will have the access on this view. It shows all the products available on the website with the seller's name, product name, price for each seller and stock available for each seller.

```
CREATE OR REPLACE VIEW customer_product_details AS (
SELECT product.pname AS Product_Name,
seller.sname AS Seller_Name,
sells.price, sells.stock
FROM sells
NATURAL JOIN product
NATURAL JOIN seller);
```

- *Product_price_avg:* The seller will have the access on this view. It shows all the product available with the average price of each product.

```
CREATE OR REPLACE VIEW product_price_avg AS (
SELECT pname AS Product_Name,
cast(AVG(sells.price) as integer) AS Average_Price
FROM Product
LEFT JOIN sells
ON Product.product_id = sells.product_id
GROUP BY Product.product_id, pname);
```

- *Sales_per_day:* The admin will have access on this view. It shows the total amount of sales made per day.

```
CREATE OR REPLACE VIEW sales_per_day AS (
SELECT date, SUM(amount) AS total_sales
FROM orders
GROUP BY date);
```

- *Revenue_generated:* The admin will have access on this view. It lists all the seller based on the revenue generated by them for the platform.

```
CREATE OR REPLACE VIEW revenue_generated AS (
SELECT s.seller_id, s.sname AS seller_name,
SUM(c.counts * c.cost) AS total_sales
FROM seller s JOIN sells t
ON s.seller_id = t.seller_id JOIN contains c
ON t.seller_id = c.seller_id
AND t.product_id = c.product_id
GROUP BY s.seller_id, s.sname
ORDER BY SUM(c.counts * c.cost) DESC);
```

Function and Procedures

- *Get_amount:* This function will update the total attribute of orders table and get the order total in orders table.

```
CREATE OR REPLACE procedure get_amount(id int)
LANGUAGE plpgsql
AS
$$
DECLARE
amt float;
BEGIN
SELECT SUM(counts * cost) INTO amt
FROM contains
WHERE order_id = id;
IF amt IS NOT NULL THEN
UPDATE Orders
SET amount = amt WHERE order_id = id;
ELSE
UPDATE Orders
SET amount = 0 WHERE order_id = id;
END IF;
END;
$$;
```

- *Place_order:* This procedure will place the order and make all the necessary changes to the entire database.

```
CREATE OR REPLACE PROCEDURE place_order(cid int)
LANGUAGE plpgsql
As $$
DECLARE
new_id int;
BEGIN
SELECT order_id INTO new_id FROM Orders ORDER BY order_id DESC
LIMIT 1;
new_id = new_id + 1;
INSERT INTO Orders (order_id, date, customer_id) values
(new_id,to_char (CURRENT_DATE, 'DD-MM-YYYY'),cid);
INSERT INTO contains SELECT counts, price, seller_id, product_id,
new_id FROM Cart WHERE customer_id = cid;
UPDATE sells
SET stock = stock - counts
FROM cart
WHERE sells.seller_id = cart.seller_id
AND sells.product_id = cart.product_id;
DELETE FROM cart
WHERE customer_id = cid;
call get_amount(new_id);
END;
$$;
```

- *Procedure_failed_payment:* This procedure is called by trigger and update necessary changes such as cancelling the order and changing the stock of seller and product.

```
CREATE OR REPLACE FUNCTION procedure_failed_payment()
RETURNS TRIGGER
LANGUAGE plpgsql
AS
$$
DECLARE
change int;
BEGIN
IF NEW.status ILIKE '%Failed%' THEN
   UPDATE Orders
   SET status = 'Cancelled'
   WHERE order_id = NEW.order_id;

   UPDATE sells
   SET stock = stock + contains.counts
   FROM contains
   WHERE sells.seller_id = contains.seller_id
   AND sells.product_id = contains.product_id
   AND order_id = NEW.order_id;
END IF;
RETURN NEW;
END;
$$;
```

- *Role_creation:* This function will create the customer role with necessary policies for each customer.

```
CREATE OR REPLACE FUNCTION role_creation()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
id int;
idname text;
pass text;
BEGIN
SELECT NEW.customer_id INTO id;
SELECT id::varchar (5) INTO idname;
SELECT NEW.passkey INTO pass;
EXECUTE FORMAT('CREATE ROLE %I LOGIN PASSWORD %L', idname, pass);
EXECUTE FORMAT('CREATE POLICY customer_view ON customer FOR
SELECT TO %I USING (customer_id = %s::integer)', idname, id);
EXECUTE FORMAT('CREATE POLICY address_view ON address FOR SELECT
TO %I USING (customer_id = %s::integer)', idname, id);
EXECUTE FORMAT('CREATE POLICY orders_view ON orders FOR SELECT TO
%I USING (customer_id = %s::integer)', idname, id);
EXECUTE FORMAT('CREATE POLICY payment_view ON payment FOR SELECT
TO %I USING (customer_id = %s::integer)', idname, id);
EXECUTE FORMAT('CREATE POLICY cart_view ON cart FOR SELECT TO %I
USING (customer_id = %s::integer)', idname, id);
EXECUTE FORMAT('CREATE POLICY cart_insert ON cart FOR INSERT TO
%I WITH CHECK (customer_id = %s::integer)', idname, id);
```

```
EXECUTE FORMAT('CREATE POLICY cart_update ON cart FOR UPDATE TO
%I USING (customer_id = %s::integer)', idname, id);
EXECUTE FORMAT('CREATE POLICY cart_delete ON cart FOR DELETE TO
%I WITH CHECK (customer_id = = %s::integer)', idname, id);
EXECUTE FORMAT('GRANT SELECT ON customer TO %I', idname);
EXECUTE FORMAT('GRANT SELECT ON address TO %I', idname);
EXECUTE FORMAT('GRANT SELECT ON orders TO %I', idname);
EXECUTE FORMAT('GRANT SELECT ON payment TO %I', idname);
EXECUTE FORMAT('GRANT SELECT, UPDATE(counts), INSERT, DELETE ON
Cart TO %I', idname);
EXECUTE FORMAT('GRANT SELECT ON customer_product_details TO %I',
idname);
RETURN NEW;
END;
$$;
```

- *Get_stock:* this trigger function will update the stock for each product.

```
CREATE OR REPLACE FUNCTION get_stock()
RETURNS trigger
LANGUAGE plpgsql
AS $$
DECLARE
p int;
pid int;
BEGIN
SELECT NEW.product_id INTO pid;
SELECT SUM (stock) INTO p FROM sells WHERE product_id = pid;
IF p IS NOT NULL THEN
UPDATE Product SET total = p WHERE product_id = pid;
ELSE
UPDATE Product SET total = 0 WHERE product_id = pid;
END IF;
RETURN NULL;
END;
$$;
```

- *Check_stock*:

```
CREATE OR REPLACE FUNCTION check_stock()
RETURNS TRIGGER AS $$
DECLARE
stock_count int;
BEGIN
SELECT stock INTO stock_count FROM sells WHERE seller_id =
NEW.seller_id AND product_id = NEW.product_id;
IF EXISTS (SELECT 1 FROM sells
WHERE seller_id = NEW.seller_id AND product_id = NEW.product_id
AND stock < NEW.counts)
THEN RAISE NOTICE 'Available stock is %, Therefore cannot add %
items in your cart', stock_count, NEW.counts;
RETURN NULL;
END IF;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

- *Procedure_cancelled_order:*

```
CREATE OR REPLACE FUNCTION procedure_cancelled_order()
RETURNS TRIGGER
LANGUAGE plpgsql
AS
$$
BEGIN
IF NEW.status ILIKE '%Cancelled%' THEN
   UPDATE sells
   SET stock = stock + contains.counts
   FROM contains
   WHERE sells.seller_id = contains.seller_id
   AND sells.product_id = contains.product_id
   AND order_id = NEW.order_id;
END IF;
RETURN NEW;
END;
$$;
```

- *Add_new_product:*

```
CREATE OR REPLACE FUNCTION add_new_product (pro text, id int,
cost int, stk int)
RETURNS VOID
LANGUAGE plpgsql
AS
$$
DECLARE
new_id int;
BEGIN
SELECT product_id INTO new_id FROM Product ORDER BY product_id
DESC;
new_id = new_id + 1;
INSERT INTO Product (product_id, pname) VALUES (new_id, pro);
INSERT INTO sells (product_id, seller_id, price, stock) VALUES
(new_id, id, cost, stk);
END;
$$;
```

- *Update_price:*

```
CREATE OR REPLACE PROCEDURE update_price (New_Price INT, p_id
INT, s_id INT)
LANGUAGE plpgsql
AS $$
BEGIN
   UPDATE sells
   SET price = New_Price
   WHERE product_id = p_id AND seller_id = s_id;
END;
$$;
```

Trigger

- *failed_payment:* In case a payment is failed for any order, we need to cancel the order and add the stock of each seller for that order.

```
CREATE TRIGGER failed_payment
AFTER UPDATE OR INSERT ON payment
FOR EACH ROW
EXECUTE PROCEDURE procedure_failed_payment();
```

- *add_new_user:* Upon insertion of a new user in the table, this trigger creates the role for that user and grant privileges and policies for that role.

```
CREATE TRIGGER add_new_user
AFTER INSERT ON customer
FOR EACH ROW
EXECUTE FUNCTION role_creation();
```

- *Update_product_total_stock_trigger:* Whenever there is any change in the stock of the seller, this trigger will be activated to update the total stock of that product.

```
CREATE TRIGGER update_product_total_stock_trigger
AFTER UPDATE OR INSERT ON sells
FOR EACH ROW
EXECUTE FUNCTION get_stock();
```

- *check_stock_trigger:* Incase the customer puts the count of the product more than the available stock, the trigger will raise a notice, thereby preventing the scenario.

```
CREATE TRIGGER check_stock_trigger
BEFORE INSERT ON Cart
FOR EACH ROW
EXECUTE FUNCTION check_stock();
```

- *Cancel_order:* When a customer cancels the order, the trigger activates and performs all the necessary changes to the database.

```
CREATE TRIGGER cancel_order
AFTER UPDATE ON Orders
FOR EACH ROW
EXECUTE FUNCTION procedure_cancelled_order()
```

Index

- *BTree Index:* If anyone wants to see the product with certain price range, for that we are using Btree Index.

  CREATE INDEX price_range ON sells(price);

  *Query: SELECT * FROM customer_product_details WHERE price BETWEEN 100 AND 200*

- *GIN Index:* In order to match the patterns for searching products under any role in the table, we use Gin Index as the index will match the Pattern for the product name.

  CREATE EXTENSION pg_trgm;
  CREATE INDEX trgm_idx ON Product USING GIN (pname gin_trgm_ops);

  *Query: SELECT * FROM Product WHERE pname = '%micro%'*

- *Hash Index:* For checking the status of the Order and Payment such as whether the Order is placed, shipped, cancelled, delivered and whether the payment is Accepted, Failed, pending we are using the Hash Index.

  CREATE INDEX status_pay ON Payment USING HASH (status);
  CREATE INDEX status_order ON Orders USING HASH (status);

  *Query: SELECT * FROM Payment WHERE status = 'Failed';*
  *Query: SELECT * FROM Orders WHERE status = 'Cancelled';*