ONLINE RETAIL STORE

Group 50

Samanyu Kamra 2021487

Shriya Verma 2021490

Project Scope:

The scope of this project is to create an end-to-end application to bridge the gap between our stakeholders - namely, customers, distributors, delivery agents, and administration, and allow the stakeholders to make suitable changes through our interface.

The customers will have attributes such as name, username, password, address, etc. The customer will be able to log in to the Customer Interface using username and password. The customer interface will allow customers to browse through the store inventory, look at prices, avail offers, add items to their cart, and check them out for delivery. The distributor has attributes such as name, address of the store, password, contact number, etc., and will be able to sell his/her products via our interface.

The admin will have attributes such as name, username, and password. Admin will be able to log in to the Admin interface using the username and password. The admin interface will allow the addition/removal of items to inventory, changing the prices of items, addition/removal of distributors, addition/removal of delivery agents, customer care, etc.

The items available in the store will have attributes including but not limited to name, measurements, rating, price, distributor, and stock.

All orders will have a price, and an order ID will be allotted to the order at check-out. A delivery agent will be assigned to the corresponding order ID. Delivery agents will have attributes such as; name, phone number, username, password, rating, etc. Order will have attributes such as; order ID, date, time, delivery location, and delivery agent allotted.

The final scope of this project lies in maintaining a smooth interaction between all the stakeholders.

Technical Requirements:

- **Backend**
  - MySQL Database
  - Python

Functional Requirements:

1) **Customers:**
    a) Customers add/remove items to/from the cart.
    b) Customers give feedback/ratings to the product/delivery agent.
    c) Customer places the order

2) **Administration:**
    a) Access data of all customers, orders, distributors, and delivery agents.
    b) Add/Update/Delete distributors, delivery agents, and products.
    c) Access customer care.
    d) Change the price of all products belonging to a certain category.
    e) Add offers for the final total.

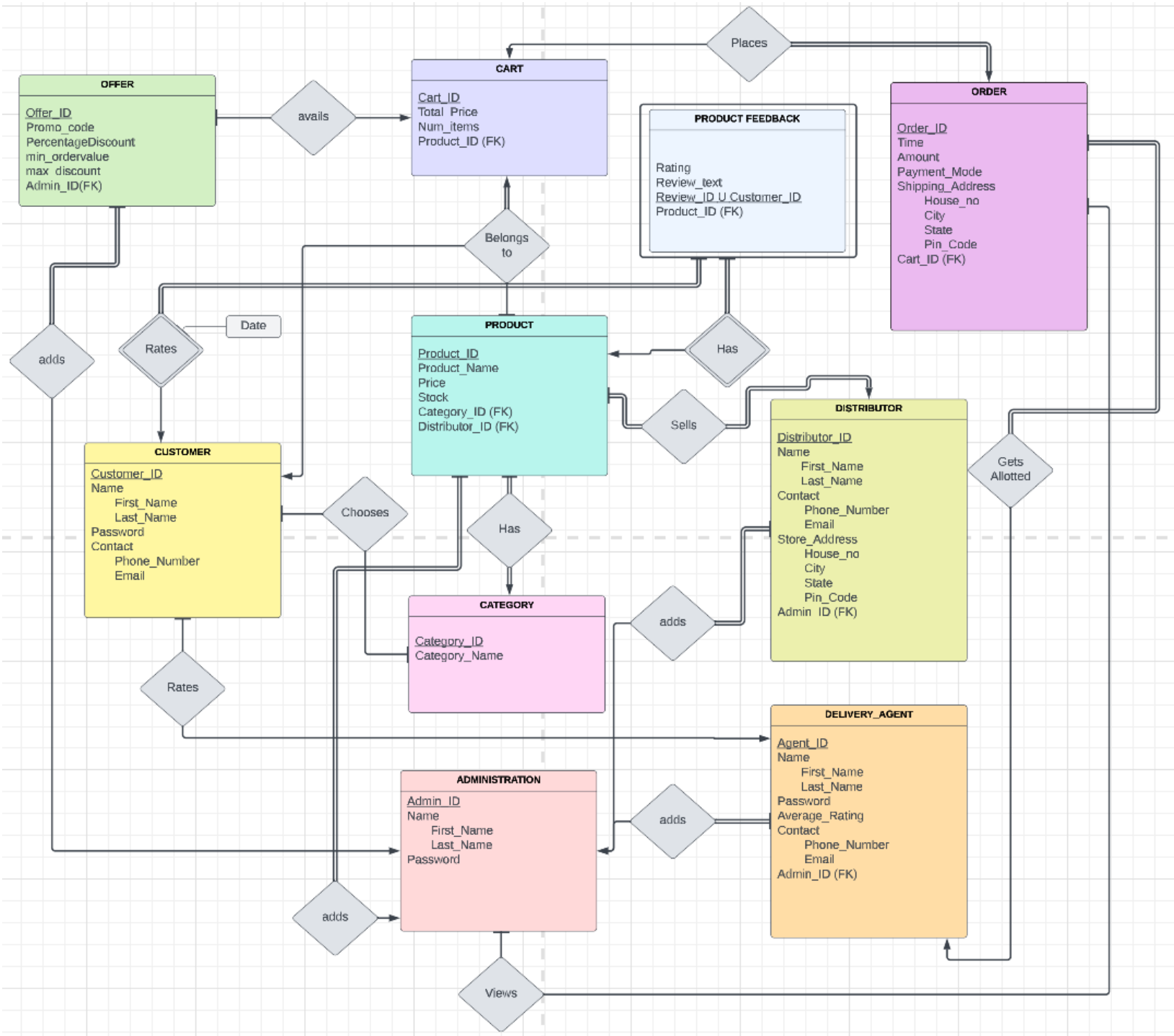
3) **Distributors:**
    a) Add/Update/Delete own products.

4) **Delivery agents:**
    a) Access delivery location of allotted order.
    b) Access payment method of allotted order.
    c) View rating/feedback received.

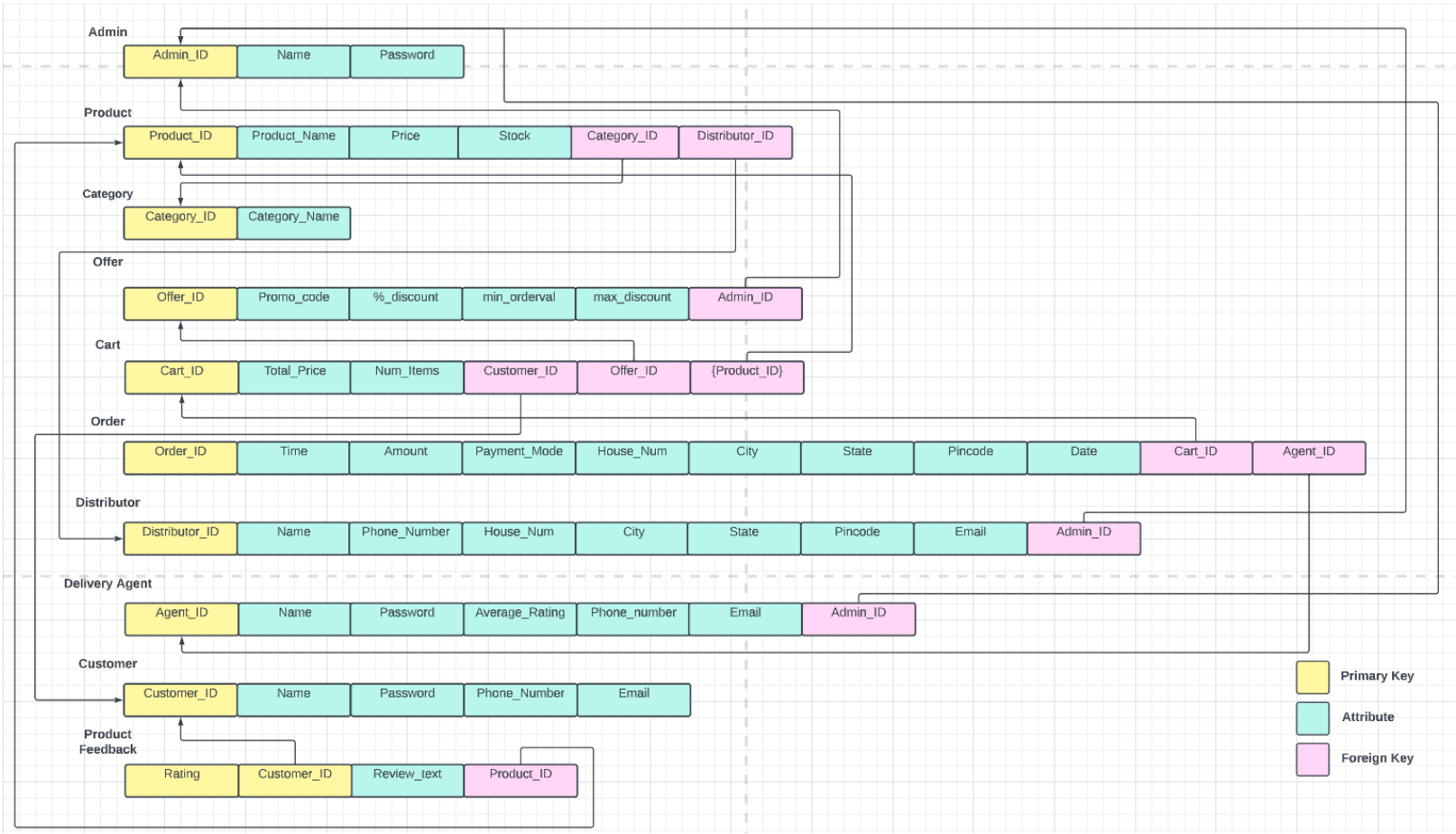
Our database has a total of 10 entities; (**Link to ER model and Relational model**)

1. Customer
2. Distributor
3. Product
4. Category
5. Administration
6. Delivery Agent
7. Cart
8. Order
9. Product Feedback
10. Offers

ER Model :


Places
CART
Cart_ID
Total Price
Num_items
Product_ID (FK)
ORDER
Order_ID
Time
Amount
Payment_Mode
Shipping_Address
House_no
City
State
Pin_Code
Cart_ID (FK)
OFFER
Offer_ID
Promo_code
PercentageDiscount
min_ordervalue
max discount
Admin_ID(FK)
avails
PRODUCT FEEDBACK
Rating
Review_text
Review_ID U Customer_ID
Product_ID (FK)
Belongs to
Date
Rates
adds
PRODUCT
Product_ID
Product_Name
Price
Stock
Category_ID (FK)
Distributor_ID (FK)
Has
Sells
DISTRIBUTOR
Distributor_ID
Name
First_Name
Last_Name
Contact
Phone_Number
Email
Store_Address
House_no
City
State
Pin_Code
Admin_ID (FK)
Gets Allotted
CUSTOMER
Customer_ID
Name
First_Name
Last_Name
Password
Contact
Phone_Number
Email
Chooses
Has
CATEGORY
Category_ID
Category_Name
adds
Rates
DELIVERY_AGENT
Agent_ID
Name
First_Name
Last_Name
Password
Average_Rating
Contact
Phone_Number
Email
Admin_ID (FK)
ADMINISTRATION
Admin_ID
Name
First_Name
Last_Name
Password
adds
adds
Views

Relational Model :

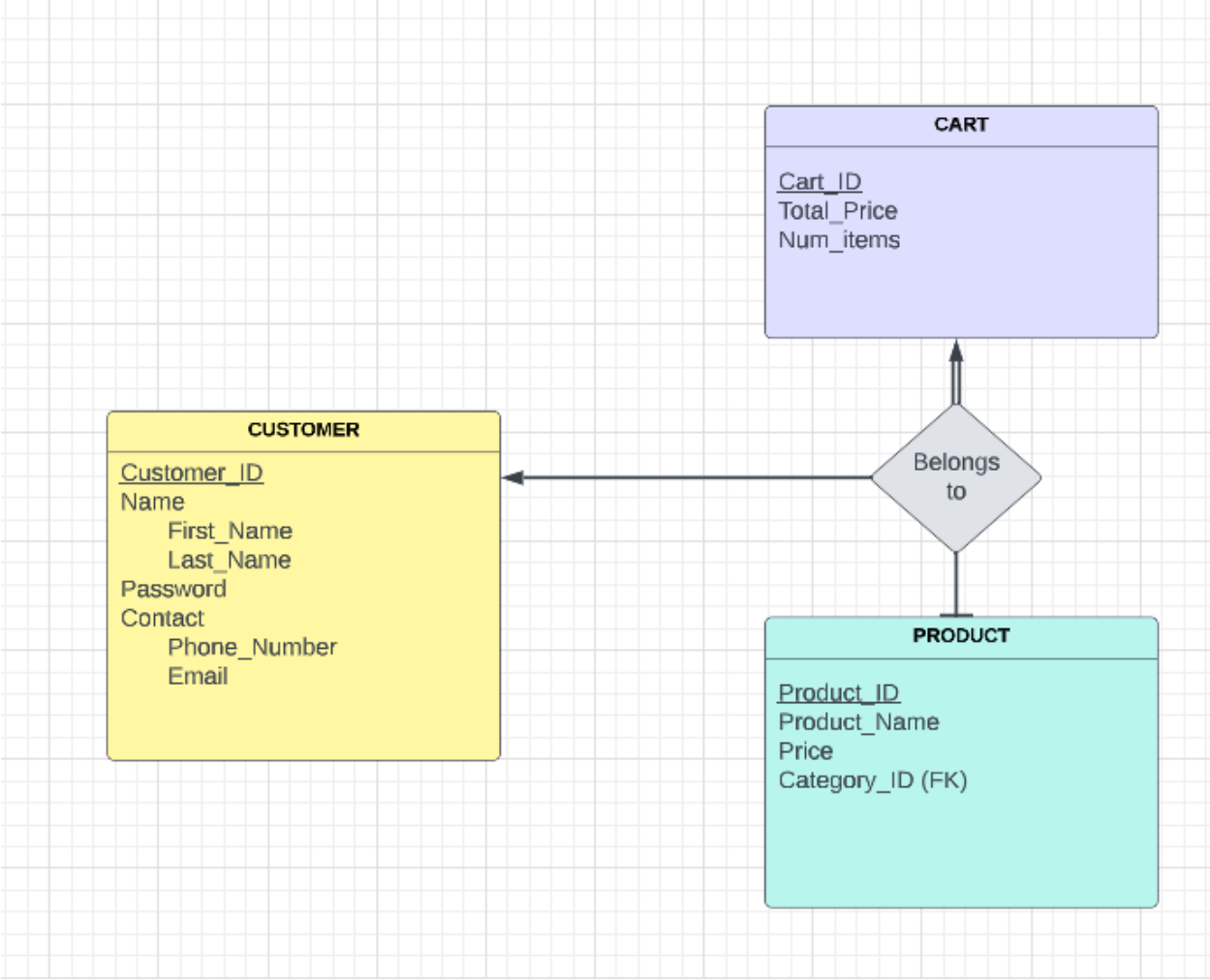


Conceptual Model :

1) **Customers**(customer_id: string , full_name: string , password: string , phone_number: varchar , email: string)
2) **Admin**(admin_id: string , password: string , full_name: string)
3) **Delivery Agent**(agent_id: string , full_name: string , password: string , phone_number: varchar , email: string , average_rating: real, admin_id: string)
4) **Distributor**(distributor_id: string , full_name: string , house_no: int , city: string , state: string , pincode: int, Phone_Number: varchar , email: string, admin_id : string)
5) **Order**(order_id: string , payment_mode: string , amount: int , time: datetime , house_no: int , city: string , state: string , pincode: int, agent_id : string, cart_id : string)

6) **Cart**(cart_id: string , total_price: int , num_items: int, customer_id: string, offer_id: string, product_id : string)
7) **Category**(category_id: string , category_name: string)
8) **Products**(product_id: string , price: int ,product_name: string, category_id : string, distributor_id : string)
9) **Offers**(offer_id: string , Discount_percentage: int , promo_code: string , min_orderValue: int , max_DiscountValue: int, admin_id : string)

Explanation of Ternary Relationship:

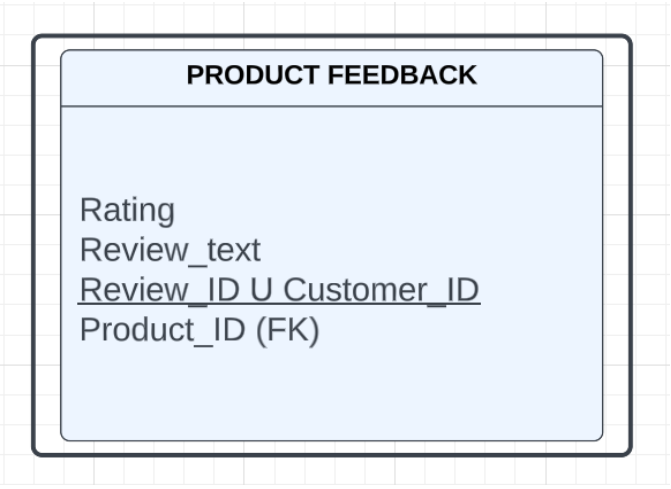


- **Cart and Product:**
  Relationship Type: 1 to N
  Participation: Partial participation for Product, total participation for Cart

- **Customer and Cart:**
  Relationship Type: 1 to 1
  Participation: Partial participation for Customer, total participation for Cart

- **Customer and Product:**
  Relationship Type: 1 to N
  Participation: Partial participation for Product and Customer.

An explanation for Weak Entity:



We have taken product feedback to be a weak entity as it is entirely **dependent on the product entity**. We can never have product feedback for a product that has not been added to the database.
This entity has a composite key - formed by the union of the Review ID and the Customer ID.
The product Feedback entity will have total participation with the product entity where each feedback is associated with a product.

SCHEMA CREATION
We first created a database on MySQL using the command:

CREATE DATABASE [database_name].

Then we inserted tables into the database for each of the above-stated entities; we did this using the syntax:

CREATE TABLE [database_name].[table_name] **(**
[col_name] [datatype]
[col_name] [datatype]
…**);**

- **Primary Keys:**
  A primary key is an attribute that acts as a unique Identifier.
  We defined the primary keys using: **PRIMARY KEY** (col_name)

Primary keys use **AUTO_INCREMENT** and are **NOT NULL**

	ENTITY NAME	PRIMARY KEY
1.	Customer	Customer_ID
2.	Distributor	Distributor_ID
3.	Product	Product_ID
4.	Category	Category_ID
5.	Administration	Admin_ID
6.	Delivery Agent	Agent_ID
7.	Cart	Cart_ID
8.	Order	Order_ID
9.	Product Feedback	Review_ID U Customer_ID
10.	Offers	Offer_ID

- **Foreign Keys:** A foreign key is an attribute that links two tables together.

We defined foreign keys using: **FOREIGN KEY** (col_name) **REFERENCES** table**(**col_name**)**

	ENTITY NAME	FOREIGN KEY	REASON
1.	Customer	-	-
2.	Distributor	Admin_ID	To view which admin added distributor
3.	Product	Category_ID Distributor_ID	To group all products of each category To group all products by a single distributor
4.	Category	-	-
5.	Administration	-	-
6.	Delivery Agent	Admin_ID	To view which admin added agent
7.	Cart	Product_ID	To store all products added to cart

8.	Order	Cart_ID	To view which cart placed order
9.	Product Feedback	Product_ID	To group all reviews of each product
10.	Offers	Admin_ID	To view which admin added offer

- **Integrity Constraints:**

1. Auto_Increment
2. Not Null
3. Primary Key
4. Foreign Key

- **Indexes:** Indexes are used to retrieve data from the database very fast.
- **Indexes** help to speed up queries.

We created Indices using:

CREATE INDEX idx_name **ON** table_name(col_name);

	ENTITY NAME	ATTRIBUTE INDEXED
1.	Customer	full_name
2.	Distributor	full_name
3.	Product	Product_name price
4.	Category	category_name
5.	Administration	Admin_ID
6.	Delivery Agent	Full_name Avg_rating
7.	Cart	Total_price num_items
8.	Order	Date Amount payment_mode
9.	Product Feedback	product_id
10.	Offers	Percentage_discount minimum order

DATA POPULATION:

In order to produce bulk data (as in the case of customers, distributors, offers, etc.), we have used an online bulk data generator.

QUERIES

A database query is a request for data from a database.

Query 1

Query1.sql

```sql

  SELECT customer.customer_id , customer.full_name
  FROM customer
  WHERE customer.customer_id IN (SELECT cart.customer_id
                                 FROM cart
                                 WHERE cart.cart_id IN (SELECT _order.cart_id
                                                        FROM _order
                                                        WHERE _order.amount>=500 ));
```

Explanation 1:

Return Customer ID and Names of customers who placed orders worth 500 or more.
This nested query works on multiple tables (namely, customer, cart, and order) and returns the customer id and full name of the customer(s) who have placed an order worth more than or equal to Rs. 500.

Query 2

Query2.sql

```sql
-- Retrieve the number of products in each category:

SELECT category.category_name,(SELECT COUNT(*)
                               FROM product WHERE category_id = category.category_id) as Product_Count
FROM category;

```

Explanation 2:

This nested query calculates and returns the product count for each category.

Query 3

Query3.sql

```sql
-- Retrieve the sellers who have products in multiple categories:

SELECT distributor.full_name , COUNT(DISTINCT product.category_id) as Category_Count
FROM distributor
INNER JOIN product ON distributor.distributor_id = product.distributor_id
GROUP BY distributor.distributor_id
HAVING Category_Count>1
```

Explanation 3:

This query returns distributors who have products in more than one category.

This query works on two tables, namely, distributor and product. We have used an *inner-join* on the above-stated two tables, which combined records from both tables whenever there are matching values in distributor_id, a field common to both tables, distributor, and product.

Query 4

Query4.sql

```sql
-- Top 10 products.

SELECT product.product_name , COUNT(*) AS FEEDBACK_COUNT FROM product
JOIN product_feedback
ON product.product_id = product_feedback.product_id
GROUP BY product.product_id
ORDER BY FEEDBACK_COUNT DESC
LIMIT 10
```

Explanation 4:

This query returns the top 10 products with the most reviews.

This query returns the top 10 popular products by sorting them based on the number of reviews received. The product with the most number of feedbacks/reviews is assumed to be the most popular product. We have utilized *JOIN* and *ORDER BY (DESC) in this query*. We have set the limit to 10 as we want to view only the top 10 products.

Query 5

Query5.sql

```sql
-- Details of all orders at a particular city and payment mode

SELECT order_id, payment_mode
FROM _order
WHERE city = 'Delhi'
UNION
SELECT order_id, 'COD' AS payment_mode
FROM _order
WHERE _order.cart_id IN (
    SELECT cart.cart_id
    FROM cart , _order
    WHERE cart.total_price>0
)
```

Explanation 5:

This query filters orders by city and payment mode and returns the order_id and Payment mode.

In this query, we are selecting and displaying the order_id and payment mode of all orders placed in a particular city (Delhi) and having a specific mode of payment (Cash On Delivery). We have used *UNION* in this query.

Query 6

Query6.sql

```sql
-- Change delivery agents contact info

UPDATE Delivery
SET phone = '+91-XXXXX-XXXXX'
WHERE Delivery.avg_rating < 3.5 AND Delivery.agent_id IN (
    SELECT _order.date_
    FROM _order
    WHERE _order.date_ > '2023-01-01'
)
```

Explanation 6:

This query allows us to update the phone number of a delivery agent.

This is an *UPDATE* query, which allows us to change the delivery agent's phone number, we have applied the conditions of the avg_rating being greater than 3.5 and the order delivered by the said delivery agent being delivered on 2023-01-01 to select the agent whose phone number is to be updated.

Query 7

Query7.sql

```sql
-- Update price of a product in a particular category

UPDATE product
SET price = price * 1.1
WHERE product.category_id = (
    SELECT category.category_id
    FROM category
    WHERE category.category_name = 'Dairy'
)
```

Explanation 7:

This query allows us to update the prices of all products in a category at once.

This is an *UPDATE* query, which allows us to change the price of all products in a particular category. In this case, we have increased the prices of all dairy products by 1.1 times their original cost.

Query 8

Query8.sql

```sql
-- Work done by a particular admin

SELECT full_name,phone
FROM distributor
WHERE distributor.admin_id = 3
UNION
SELECT promocode,min_orderval
FROM offer
WHERE offer.admin_id = 3
UNION
SELECT full_name,avg_rating
FROM Delivery
WHERE Delivery.admin_id = 3

```

Explanation 8:

This Query shows all entities (distributors/offers/agents) added by a particular admin.
This query allows us to check all the additions (distributors, offers, and the delivery agents added) made by a particular admin (in this case, admin no.3)

Query 9

Query9.sql

```sql
-- Create a view named 'CustomerOrders' that shows the order_id, payment_mode, and amount for all
-- orders made by a cart with cart_id 'Cart001':

CREATE VIEW CustomerOrders AS
SELECT order_id, payment_mode, amount
FROM _order
WHERE _order.cart_id = 242;
```

Explanation 9:

Returns details of an order made by a particular cart_id.

This query creates a *VIEW* named Customer0rders which checks what orders are placed by a particular cart (cart_id = 242) and returns the order_id, payment_mode, and amount of the order.

Query 10

Query10.sql

```sql
-- Delete all orders with COD as payment mode
DELETE FROM _order
WHERE payment_mode = 'COD';
```

Explanation 10:

This is a DELETE query. It checks for all orders with payment_mode = 'COD' and deletes them.

Query 11

Query11.sql

```sql
-- Find details of a all carts by a customer.
SELECT *
FROM Customer
WHERE customer.customer_id = 20095
UNION
SELECT *
FROM cart
WHERE cart.customer_id = 20095
```

Explanation 11:

Displays all the details of the customer associated with a particular Cart (Cart_ID = 20095 in this case)

Query 12

Query12.sql

```sql
-- Retrieve the names of all customers who have ordered from the same delivery agent
-- more than once, and the delivery agent's average rating
SELECT customer.full_name, AVG(Delivery.avg_rating) AS avg_rating
FROM Customer
JOIN cart ON customer.customer_id = cart.customer_id
JOIN _order ON cart.cart_id = _order.cart_id
JOIN Delivery ON Delivery.agent_id = _order.agent_id
```

Explanation 12:

This query returns the names of all customers who have been allotted the same delivery agent twice or more times.

This query uses *JOINS.* We group the data by delivery.agent_id. We have joined four tables: Customer, Cart, _Order, and Delivery.

Query 13

Query13.sql

```sql
-- Customers who have not placed an order

SELECT customer.customer_id, customer.full_name
FROM Customer
WHERE customer_id NOT IN (
    SELECT DISTINCT customer_id
    FROM Cart
    JOIN _order ON cart.cart_id = _order.cart_id
)
```

Explanation 13:

Find all customers who have not placed an order.

This query uses NOT IN to find the carts that have not placed an order and, in turn, find the customer_id associated with the said carts to see all the customers who haven't placed an order yet.

Query 14

Query14.sql

```sql
-- Displays order placed by customer and the delivery address

SELECT
    CONCAT(_order.house_no, ', ', _order.city, ', ', _order.State, ' ', _order.pin_code) AS full_address,
    _order.order_id,
    CONCAT(customer.full_name, ' ') AS customer_name
FROM _order
JOIN cart ON _order.cart_id = cart.cart_id
JOIN customer ON cart.customer_id = customer.customer_id;
```

Explanation 14:

Display all orders, the names of the customers who placed them, and the delivery address.

This query uses CONCATENATE Function to combine all the columns corresponding to the address and displays the customer associated with the said order.

Query 15

Query15.sql

```sql
-- Apply discount to price of cart and update order price

UPDATE _order
JOIN cart ON _order.cart_id = cart.cart_id
JOIN offer ON cart.offer_id = offer.offer_id
SET _order.amount = cart.total_price * (1 - offer.percentagediscount/100)
WHERE _order.order_id = 15091;
```

Explanation 15:

Apply a discount on the cart price and reflect the updated price in the order price.

This query is an update query that uses multiplication and subtraction operations to update the price of an order after a discount.

Embedded Queries:

1. **Changes prices of all products of the chosen category.**

```
def change_price():
    n = float(input("Enter the price multiplier: "))
    s = str(input("Enter Category Name: "))
    sql = "UPDATE product SET price = price * %s WHERE product.category_id =
(SELECT category.category_id FROM category WHERE
category.category_name = %s)"
    val = (n,s)

    mycursor.execute(sql,val)
    for i in mycursor:
        print(i)

    mydb.commit()
    print(mycursor.rowcount, "record(s) affected")

    print("Price Updated")
```

2. **Add A delivery Agent:**

```
def add_agent():
    n = int(input("Enter your Admin ID: "))
```

```
    m = int(input("Assign an ID to the new Delivery Agent: "))
    s = str(input("Enter delivery agent name: "))
    s1 = str(input("Create a Default Password: "))
    k = int(input("Enter Phone no: "))
    s2 = str(input("Enter E-mail: "))
    sql = "INSERT INTO Delivery(admin_id,agent_id ,full_name ,pass_word
,avg_rating  ,phone ,email) VALUES(%s,%s,%s,%s,%s,%s,%s)"
    val = (n,m,s,s1,2,k,s2)
    mycursor.execute(sql,val)
    for i in mycursor:
        print(i)

    mydb.commit()
    print(mycursor.rowcount, "record(s) affected")

    print("Delivery Agent added")
```

3. **Allows customer to view all their carts**

```
def my_cart():
    n = int(input("Enter the Customer ID: "))
    sql = "SELECT * FROM Customer WHERE customer.customer_id = %s
UNION SELECT * FROM cart WHERE cart.customer_id = %s"
    val = (n,n)
    mycursor.execute(sql,val)
    for i in mycursor:
        print(i)

    mydb.commit()
    print("Here are the details.")
    print(mycursor.rowcount, "record(s) affected")
```

OLAP Queries:

1. **What is the total revenue the online retail store generates in a given time period?**

```
SELECT SUM(amount) AS total_revenue
FROM Order
WHERE date_ BETWEEN [start_date] AND [end_date];
```

2. **What is the average price of products in each category?**

```
SELECT
   c.category_name AS Category,
   AVG(p.price) AS AvgPrice
FROM
   Product p
   JOIN Category c ON p.category_id = c.category_id
GROUP BY
   c.category_name

ORDER BY avg_price DESC;
```

3. **What are the Top 10 products?**

```
SELECT product.product_name , COUNT(*) AS FEEDBACK_COUNT
FROM product
JOIN product_feedback
ON product.product_id = product_feedback.product_id
GROUP BY ROLL UP( product.product_id)
ORDER BY FEEDBACK_COUNT DESC
LIMIT 10
```

4. **What are the top three delivery agents with the highest average rating?**

```
SELECT Delivery_Agent.first_name, Delivery_Agent.last_name,
Delivery_Agent.average_rating
FROM Delivery_Agent
ORDER BY Delivery_Agent.average_rating DESC
```

```
LIMIT 3;
```

Triggers:

1. **<u>If a customer is removed from the database, all feedback associated with that customer is also deleted :</u>**

```
DELIMITER $$
CREATE
TRIGGER remove_customer_corresponding_product_feedback BEFORE
DELETE
ON Customer FOR EACH ROW

BEGIN
DELETE FROM product_feedback WHERE
OLD.customer_id=customer_id;
END$$
DELIMITER ;
```

2. **<u>If a cart associated with an order does not exist, then this order can not be placed:</u>**

```
CREATE TRIGGER t1
BEFORE INSERT ON _order
FOR EACH ROW
BEGIN
DECLARE cart_count INT;
SELECT COUNT(*) INTO cart_count FROM Cart WHERE cart_id =
NEW.cart_id;
IF cart_count = 0 THEN
 SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Cannot create
order without a corresponding cart record.';
 END IF;
END;
```

3. **<u>If a category is deleted, all products that belong to that category will also be deleted:</u>**

```
DELIMITER $$ CREATE
TRIGGER delete_product_from_category BEFORE DELETE ON category
FOR EACH ROW
BEGIN
DELETE from product
WHERE OLD.Category_ID=product.Category_ID; END$$
DELIMITER ;
```

TRANSACTIONS:

A. CONFLICTING TRANSACTIONS :

1. **READ-WRITE CONFLICT:**

   Let's say you have a table called products that stores information about the products you sell and another table called cart_items that stores information about the items in customers' shopping carts. To create a read-write conflict, you could have one transaction that reads the price of a product from the products table and then calculates the total price of the items in a customer's cart. At the same time, you could have another transaction that updates the price of the same product in the products table. This would cause the first transaction to have an incorrect total price calculation since it read an outdated price.

TIME	STEP	TRANSACTION	SQL STATEMENT	OPERATIONS
T1	1	TRANSACTION 1	BEGIN;	BEGIN
T1	2	TRANSACTION 1	SELECT price FROM products WHERE product_id = “405”	READ(products: 405)
T2	1	TRANSACTION 2	BEGIN;	BEGIN
T1	3	TRANSACTION 1	UPDATE products SET price = 50	WRITE(products: 405)

			WHERE product_id = ‘405';	
T1	4	TRANSACTION 1	SELECT SUM(quantity*price) FROM Cart_items	READ(Cart_items : 405), READ(products : 405)
T2	3	TRANSACTION 2	COMMIT;	COMMIT;
T1	5	TRANSACTION 1	COMMIT;	WRITE(Cart_items : 405) WRITE(Products : 405) COMMIT

T1
BEGIN TRANSACTION;

```
SELECT price
FROM products
WHERE product_id = 'P01';
```

Calculate the total price of cart items

```
UPDATE cart
SET total_price = total_price + (SELECT price FROM
products WHERE product_id = 'P01')
WHERE cart_id = 'C01';
COMMIT;
```

T2
BEGIN TRANSACTION;

```
UPDATE products
SET price = 200
WHERE product_id = 'P01';
COMMIT;
```

T1: Reads the price of a product from the products table and calculates the total price of items in a cart.
T2: Updation of the price of the SAME product in the products table

T1(R P01) ---> T2(W P01)

Convert the conflict to non - conflicting transaction:
To avoid read-write conflicts, we need to ensure that the transactions are serialized or ordered in a way that prevents conflicting actions from occurring simultaneously.

One way to achieve this is by using serializable isolation level and ensuring that all transactions are scheduled in a way that maintains serializability. This can be done by using a two-phase **locking** protocol, where each transaction must acquire all necessary locks before executing any write operation, and then release all locks once the transaction is complete.

To apply this approach to the scenario you described, we can modify the second transaction that updates the price of the product in the products table to acquire **an exclusive lock on the row for that product** before updating the price. This ensures that **no other transaction can read or write to that row until the lock is released**, preventing any read-write conflicts.

Then, we can modify the first transaction that calculates the total price of the items in the customer's cart to acquire shared locks on all rows in the cart_items table that correspond to the products in the cart. This ensures that no other transaction can modify the cart_items table until the locks are released, preventing any write-read conflicts.

By using this approach, both transactions can execute in a serialized manner, ensuring that no conflicting actions occur simultaneously and preventing any read-write conflicts.

Step wise:
1. Since Transaction 1 already acquired shared locks on the rows in the Cart_Items table, Transaction 2 must wait until Transaction 1 releases the locks before it can acquire the exclusive lock on Product B.
2. Transaction 1 calculates the total price of the items in the customer's cart using the current prices in the Products table.
3. Transaction 1 releases the shared locks on the rows in the Cart_Items table.
4. Transaction 2 acquires the exclusive lock on the row for Product B and updates the price.
5. Transaction 2 releases the exclusive lock on the row for Product B.
6. Both transactions are committed.
7. Using this approach ensures that Transaction 1 completes before Transaction 2 updates the price of Product B, preventing any read-write conflicts.

To avoid deadlocks:
Alternatively, we can also use optimistic concurrency control by allowing both transactions to proceed with their operations without locking any resources, but checking for conflicts before committing the changes. If a conflict is detected, one of the transactions will be rolled back, and the other can proceed with its changes. However, this approach may result in more rollbacks and retries, which can affect performance.

2. WRITE-READ CONFLICT:

Here, an administrator is updating the contact details of a delivery agent while the delivery agent is already allotted to a customer. In this case, the Customer will have access to old contact details, causing miscommunication between the delivery partner and the customer.

SQL STATEMENT:

- **Transaction 1 (T1):**

Updation of Delivery Agent's Contact Details (phone number) by Admin

```
BEGIN TRANSACTION;
UPDATE Delivery SET phone_num = '9195551234'
WHERE agent_id = '19024';
 COMMIT;
```

- **Transaction 2 (T2):**

```
BEGIN TRANSACTION;
SELECT * FROM Delivery WHERE agent_id = '19024';
COMMIT;
```

Serialization Table :

TIME	TRANSACTION	SQL STATEMENT	OPERATIONS
T1	BEGIN		

T1	UPDATE	BEGIN TRANSACTION; UPDATE Delivery SET phone_num = '9195551234' WHERE agent_id = '19024';COMMIT;	WRITE(Agent_id = 19024)
T2	BEGIN		
T2	SELECT	BEGIN TRANSACTION; SELECT * FROM Delivery WHERE agent_id = '19024'; COMMIT;	READ(Agent_id = 19024)
T1	COMMIT		
T2	COMMIT		

As we can see, T1 has a WRITE operation, whereas T2 has a READ operation, As T2 is reading the non-updated values while T1 is updating them, This has caused a WRITE-READ Conflict.

Convert the conflict to non - conflicting transaction:

1. Transaction 1 (T1) begins by acquiring an exclusive lock on the rows it is going to update, which prevents any other transactions from accessing or modifying the same rows until the lock is released.
2. Transaction 2 (T2) begins by acquiring a shared lock on the rows it is going to read, which allows other transactions to read the same rows but prevents them from updating them until the lock is released.

3. Transaction 2 (T2) reads the data without any conflicts because it has acquired a shared lock on the rows being read.
4. Transaction 1 (T1) updates the data and commits the transaction, releasing the exclusive lock it had acquired on the rows being updated.
5. Transaction 2 (T2) commits the transaction, releasing the shared lock it had acquired on the rows being read.

RESOLVING THE CONFLICT :

In order to resolve this conflict, we can either :

1. ABORT one transaction.
2. DELAY a transaction until the other one is complete.

Conflicts are usually resolved via the following methods :
1. Locking: Prevents multiple transactions from accessing the same data at the same time.
2. Timestamping: Ordering of transactions, the earlier time stamped transaction is prioritized.
3. Conflict Detection and Resolution: This can be practiced using specialized algorithms.
4. Optimistic Concurrency Control: Assumes conflicts are rare and checks if other users have made changes while one user has; if changes are found, the user is notified that changes were not saved and has to do the process again.

B. NON-CONFLICTING TRANSACTIONS :

1. **Insert a new delivery agent :**

```
BEGIN TRANSACTION T1;
INSERT INTO Agents (agent_id, first_name, last_name, password, phone_number, email, average_rating)
VALUES ('D001', 'Jane', 'Smith', 'password', '123-456-7890', 'janesmith@email.com', 4.5);

COMMIT TRANSACTION T1;
```

2. **Insert a new offer**

```
BEGIN TRANSACTION T2;
```

```
INSERT INTO Offers (offer_id, discount_percentage, promo_code, min_orderValue,
max_discountValue)
VALUES ('O001', 10, 'OFF10', 5000, 1000);
COMMIT TRANSACTION T2;
```

3. **Apply a discount to the price of the cart and update the order price**

```
START TRANSACTION;
UPDATE _order
JOIN cart ON _order.cart_id = cart.cart_id
JOIN offer ON cart.offer_id = offer.offer_id
SET _order.amount = cart.total_price * (1 - offer.percentagediscount/100)
WHERE _order.order_id = 15091;

COMMIT;
```

4. **Updating the prices of all products in a category**

```
BEGIN;
UPDATE product
SET price = price * 1.1
WHERE product.category_id = (
SELECT category.category_id
FROM category
WHERE category.category_name = 'Dairy'
);
COMMIT;
```