DBMS Project Report

PES University

Database Management Systems

UE18CS252

Submitted By

SRN : PES1201801573	Name : Ashish Ramayee Asokan	Evaluation date and time:

Project Title: E-Commerce Transation Processing Database System

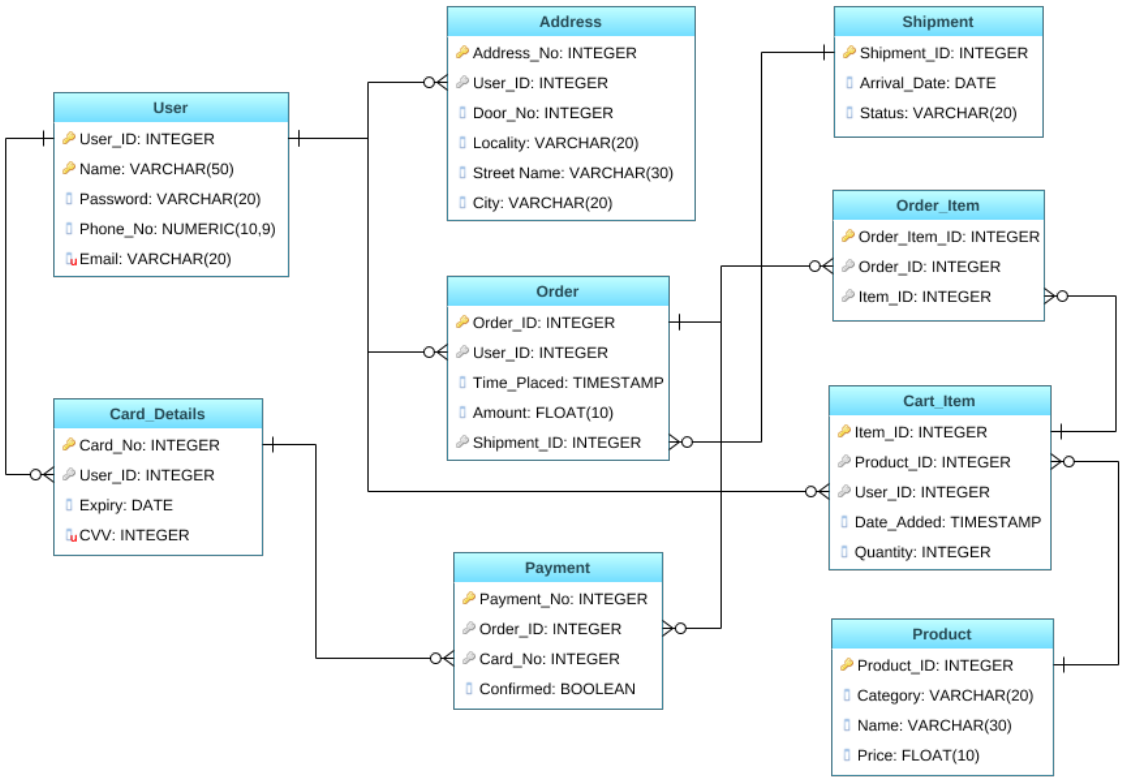
Evaluation Parameter	*Marks Allocated*
Functional Dependencies	2
Identifying keys based on Fds	2
Normalization & Testing for Lossless Join Property	0
DDL : Table Creating with all the constraints	0
Triggers	2
SQL Queries	2
Viva / Modifications (Unit 3 / 4 concepts)	2 + 2

Problem statement:

An E-commerce transaction database has to store information about its registered users (identified by the ***User_ID, Name, Password, Phone no, Email*** and ***address*** as the attributes), available ***products*** (identified by the ***name***, ***category*** and ***price***), cart item (having ***ID*** and ***quantity*** as the attributes), order (having ***ID***, ***total amount*** to be payed and ***payment details*** as attributes) credit card details and shipment (having ***status*** and its ***ID*** as attributes).

Each user ***can have only one*** credit card and can place many orders one after the other. A shipment contains the delivery ***for many orders and a particular order can be split and delivered by multiple*** shipments (as it occurs in real-life situations as well). Each order has a number of order items listed. A particular order item has to be an item from a cart that can belong to only a single user, however not all cart items have to be a part of the final order. Once the order has been placed and the shipment has been confirmed, the ***cart details of the user must be erased*** for a fresh cart to be available. Design an ER diagram and a relational schema for the above situation following all the constraints listed.

Database Schema: (show all the tables and the constraints)



Functional Dependencies: (List based on your application constraints) (The Functional Dependencies in bold are the final dependencies for the relation**)**

➔ ***USER:***

User_ID ⟶ {Username, Password, Email, Phone No}

Email ⟶ {Username, Password, User_ID, Phone No}

{User_ID, Email} ⟶ {Username, Password, Phone No}

➔ ***CARD DETAILS:***

Card_No ⟶ {Expiry, CVV}

Card_No ⟶ User_ID

Card_No ⟶ {User_ID, Expiry, CVV}

➔ ***ADDRESS:***

Address_ID ⟶ User_ID

Address_ID ⟶ {Door_No, Street name, locality, city}

Address_ID ⟶ {User_ID, Door_No, Street name, locality, city}

➔ ***ORDER:***

Order_ID ⟶ {User_ID, Time Placed, Amount, Shipment_ID}

➔ ***SHIPMENT_DETAILS:***

Shipment_ID ⟶ {Arrival_Date, Status}

➔ ***CART_ITEM:***

Item_ID ⟶ {User_ID, Date_Added, Product_ID}

{Item_ID, Product_ID, User_ID} ⟶ Quantity

Item_ID ⟶ {User_ID, Date_Added, Product_ID, Quantity}

➔ ***PRODUCT:***

Product_ID ⟶ {Name, Price}

Product_ID ⟶ {Name, Price, Category}

➔ ***PAYMENT:***

Payment_No ⟶ {Order_ID, Confirmed}

Payment_No ⟶ {Order_ID, Confirmed, Card_No}

➔ ***ORDER_ITEM:***

Order_Item_No ⟶ {Order_ID, Item_ID}

Candidate keys: (Justify how did you get these as keys)

The candidate keys for a given relation can be derived from the functional dependencies for that relation. The attribute closure for all the attributes in the relation should be derived using the functional dependencies. The minimal set of attribute closure that functionally determines all the other attributes in the relation contains the candidate keys of the relation.

Example 1: Considering the relation Cart item

$\{User\_ID\}^+ \longrightarrow \{User\_ID, Quantity\}$

$\{Product\_ID\}^+ \longrightarrow \{Product\_ID, Quantity\}$

$\{Date\_Added\}^+ \longrightarrow \{Date\_Added\}$

$\{Quantity\}^+ \longrightarrow \{Quantity\}$

$\boldsymbol{\{Item\_ID\}^+ \longrightarrow \{Item\_ID, User\_ID, Quantity, Product\_ID\}}$

From the above example, the attribute closure of the Item_ID attribute has all the attributes of the relation, which makes it the only candidate key and thus the primary key. Similarly, the candidate keys for all the other relations of the database can be derived.

Example 2: Considering the relation User

$\{User\_ID\}^+ \longrightarrow \{User\_ID, Email, Phone No, Username, Password\}$

$\{Email\}^+ \longrightarrow \{User\_ID, Email, Phone No, Username, Password\}$

$\{Username\}^+ \longrightarrow \{Username\}$

$\{Password\}^+ \longrightarrow \{Password\}$

$\{Phone No\}^+ \longrightarrow \{Phone No\}$

In the User relation, both User_ID and Email are candidate keys since both are unique and are independent. The attribute closure for both Email and User_ID are equal.

Normalization and testing for lossless join property:

The relational database schema has been obtained from the ER diagram by following the ER to schema mapping rules. So hypothetically the relational database schema obtained should be in the normalised form. The normal forms are discussed below:

1st Normal Form:

According to the 1st Normal Form, the domain of each attribute of a relation should have atomic values, i.e, the attributes cannot have a set of values, tuples or multiple values. In all the relations of the database schema, the domain of all the attributes has atomic, indivisible values according to the constraints.

2nd Normal Form:

For a relation to satisfy the 2nd Normal form, every non-prime attribute of the relation should be fully functionally dependent on the prime attribute. In all the functional dependencies expressed above are either directly or transitively dependent on the prime attribute of each relation.

However, the 2nd Normal Form can be violated if any unnecessary attributes are added to the normalised relation. For example, the relation ***User*** had the prime attributes ***{User_ID, Email}***. If an attribute Name is added to the User relation, an additional functional dependency would be ***User_ID → Name***. But the prime attribute combination is ***{Username, Email}***. In such a situation, the 2nd Normal Form can be violated.

3rd Normal Form:

A given relation is in the 3rd Normal Form if it is already in the 2nd Normal Form and if there are no transitive dependencies between the attributes of the relation. Since all the relations in the schema are already in the 1st and 2nd Normal Forms, the first condition is satisfied. From the functional dependencies, it can be inferred that none of the non-prime attributes have transitive dependencies on other non-prime attributes.

Testing for Lossless Join Property

A decomposition D of a given relation R is said to possess the lossless join property with a given set of functional dependencies F if the natural join of the instances in the decomposition D gives back the original relation without the addition of spurious tuples. Given below is an example considering the *User* relation.

Functional Dependency: (Taking only the 1st one for ease of explanation)
User_ID ⟶ {Username, Password, Email, Phone No}

Relations after Decomposition:
R1 : {User_ID, Username, Password}
R2 : {User_ID, Email, Phone No}

Initial Table before applying the algorithm

User_ID	Username	Password	Email	Phone No
a_1	a_2	a_3	b_{14}	b_{15}
a_1	b_{22}	b_{23}	a_4	a_5

Table after applying the algorithm:

User_ID	Username	Password	Email	Phone No
a_1	a_2	a_3	b_{14}	b_{15}
$\boldsymbol{a_1}$	$\boldsymbol{a_2}$	$\boldsymbol{a_3}$	$\boldsymbol{a_4}$	$\boldsymbol{a_5}$

A given relation satisfies the lossless join decomposition if, after applying all the steps, atleast one row in the table consists only of ‘a’ symbols. In the above table, the last row consists only of ‘a’s which indicates that the decomposition into R1 and R2 listed above satisfies the lossless join property.

DDL:

Table User_Details:

```
CREATE TABLE User_Details(
    User_ID INT NOT NULL,
    Username VARCHAR(40),
    Pass VARCHAR(30),
    Email VARCHAR(40) NOT NULL UNIQUE,
    Phone_No CHAR(10),

    PRIMARY KEY (User_ID),
    CONSTRAINT check_Phone_No CHECK (Phone_No > 99999999)
);
```

Table Address:

```
CREATE TABLE Address(
    Address_ID INT NOT NULL UNIQUE,
    User_ID INT NOT NULL,
    Door_No INT NOT NULL,
    Locality VARCHAR(50),
    Street_Name VARCHAR(50),
    City VARCHAR(30),

    PRIMARY KEY (Address_ID),
    FOREIGN KEY (User_ID) REFERENCES User_Details(User_ID) ON DELETE
CASCADE ON UPDATE CASCADE
);
```

Table Card_Details:

```
CREATE TABLE Card_Details(
    Card_No BIGINT  NOT NULL UNIQUE,
    User_ID INT NOT NULL,
    Expiry DATE,
    CVV INT,

    CONSTRAINT check_Card_No CHECK (Card_No > 999999999999999 and
Card_No < 10000000000000000),
    CONSTRAINT check_CVV CHECK (CVV > 99 and CVV < 1000),
    PRIMARY KEY (Card_No),
    FOREIGN KEY (User_ID) REFERENCES User_Details (User_ID) ON
DELETE CASCADE ON UPDATE CASCADE
);
```

Table Shipment:

```
CREATE TABLE Shipment(
    Shipment_ID INT NOT NULL UNIQUE,
    Shipment_Status VARCHAR(30) NOT NULL,
    Arrival_Date DATE,
    PRIMARY KEY (Shipment_ID)
);
```

Table Order_Details:

```
CREATE TABLE Order_Details(
    Order_ID INT NOT NULL UNIQUE,
    User_ID INT NOT NULL,
    Amount FLOAT,
    Shipment_ID INT,

    PRIMARY KEY (Order_ID),
    FOREIGN KEY (User_ID) REFERENCES User_Details(User_ID) ON DELETE
CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (Shipment_ID) REFERENCES Shipment(Shipment_ID) ON
DELETE CASCADE ON UPDATE CASCADE
);
```

Table Payment:

```
CREATE TABLE Payment(
    Payment_No INT NOT NULL UNIQUE,
    Order_ID INT NOT NULL,
    Card_No BIGINT,
    Confirmed BOOLEAN,

    PRIMARY KEY (Payment_No),
    FOREIGN KEY (Order_ID) REFERENCES Order_Details(Order_ID) ON
DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (Card_No) REFERENCES Card_Details(Card_No) ON DELETE
CASCADE ON UPDATE CASCADE
);
```

Table Product:

```
CREATE TABLE Product(
    Product_ID INT NOT NULL UNIQUE,
    Category VARCHAR(30),
    Product_name VARCHAR(30),
    Price FLOAT,
```

```
    PRIMARY KEY (Product_ID)
);
```

Table Cart_Item:

```
CREATE TABLE Cart_Item(
    Item_ID INT NOT NULL UNIQUE,
    Product_ID INT NOT NULL,
    User_ID INT NOT NULL,
    Time_Added TIMESTAMP,
    Quantity INT DEFAULT 0,

    PRIMARY KEY (Item_ID),
    FOREIGN KEY (User_ID) REFERENCES User_Details(User_ID) ON DELETE
CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (Product_ID) REFERENCES Product(Product_ID) ON
DELETE CASCADE ON UPDATE CASCADE
);
```

Table Order_Item:

```
CREATE TABLE Order_Item(
    Order_Item_ID INT NOT NULL UNIQUE,
    Order_ID INT NOT NULL,
    Item_ID INT NOT NULL,

    PRIMARY KEY (Order_Item_ID),
    FOREIGN KEY (Order_ID) REFERENCES Order_Details(Order_ID) ON
DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (Item_ID) REFERENCES Cart_Item(Item_ID) ON DELETE
CASCADE ON UPDATE CASCADE
);
```

Sample Insert Statements:

```
insert into Order_Item values (214, 42, 23);
```

The above insert statement will demonstrate the working of the trigger that has been defined in the next section. When the above statement is executed, the price of cart item 23 will be added to the order labeled 41.

Triggers:

A trigger is used to perform an automated update before or after the addition of a record in a particular table. In this project, a trigger has been used to update the Amount attribute in the Order_Details table whenever an item is added to the specific order from the cart. When a

record is inserted into the Order_Item table, the corresponding cart Item_ID is used to retrieve the price and the quantity of the item added from the cart to the order.

To obtain the price and the quantity, the Item_ID foreign key of the added entry is used to refer to the Quantity attribute in the Cart_Item table. The price of the product is retrieved using a natural join between the Cart_Item and Product tables. Below is the code used to create the above mentioned trigger.

```
delimiter //

CREATE TRIGGER Increase_Price
AFTER INSERT ON
Order_Item FOR EACH ROW

BEGIN

    -- Declaring the Variables to be used
    DECLARE price_new FLOAT;
    DECLARE quan INT;

    --  Getting Price and Quantity of the item added
    SET @price_new = (SELECT Price FROM Cart_Item NATURAL JOIN Product
WHERE Item_ID = new.Item_ID);
    SET @quan = (SELECT Quantity FROM Cart_Item WHERE Item_ID =
new.Item_ID);

    -- Updating order amount
    UPDATE Order_Details
    SET Order_Details.Amount = Order_Details.Amount + (@price_new * @quan)
    WHERE Order_Details.Order_ID = new.Order_ID;

END //
```

SQL Queries:

1. Display the Username, Phone No and Address of the min amount order in each shipment

```
SELECT User_ID, Phone_No, Door_No FROM User_Details NATURAL JOIN
Address WHERE User_ID in (SELECT User_ID FROM Order_Details GROUP BY
Shipment_ID ORDER BY Amount);
```

2. Retrieve all the user details whose orders have been dispatched

```
SELECT * FROM User_Details WHERE User_ID IN (SELECT User_ID FROM
Order_Details NATURAL JOIN Shipment WHERE Shipment_Status LIKE
"Dispatched");
```

3. For each user display the most expensive item in his/her cart and its quantity

```
select User_ID, Username, max(Price) from User_Details natural join
Cart_Item natural join Product group by User_ID;
```

4. Display all the usernames who have paid for their orders

```
SELECT * FROM User_Details WHERE User_ID IN (SELECT User_ID FROM
Order_Details WHERE Order_ID IN (SELECT Order_ID FROM Payment WHERE
Confirmed=True));
```