ABSTRACT

A departmental source is an essential community service center. It acts as a source of fulfilling human wants and needs of members of a geographical community. They act as a middleman between the wholesale suppliers of essential goods and the end consumer – for example, households.

This project aims to create an efficient management system for owners and employees of a departmental store, allowing them to carry out various tasks such as refer to the suppliers, existing customers, inventory details, loyalty program depending on the time’s need. It stores all the relevant information in an easy to view manner.

INTRODUCTION

This project report outlines the development of a departmental store management system that utilizes database design and user interface (UI) creation. The system aims to streamline the store's inventory management, sales tracking, and customer relations management.

The database design includes tables for product information, employee data, and sales transactions, among others. The UI was designed with a user-friendly interface, providing quick access to relevant information, and allowing for efficient data entry and retrieval. Overall, this project demonstrates how the integration of database design and UI creation can enhance the management of a departmental store.

PROBLEM STATEMENT

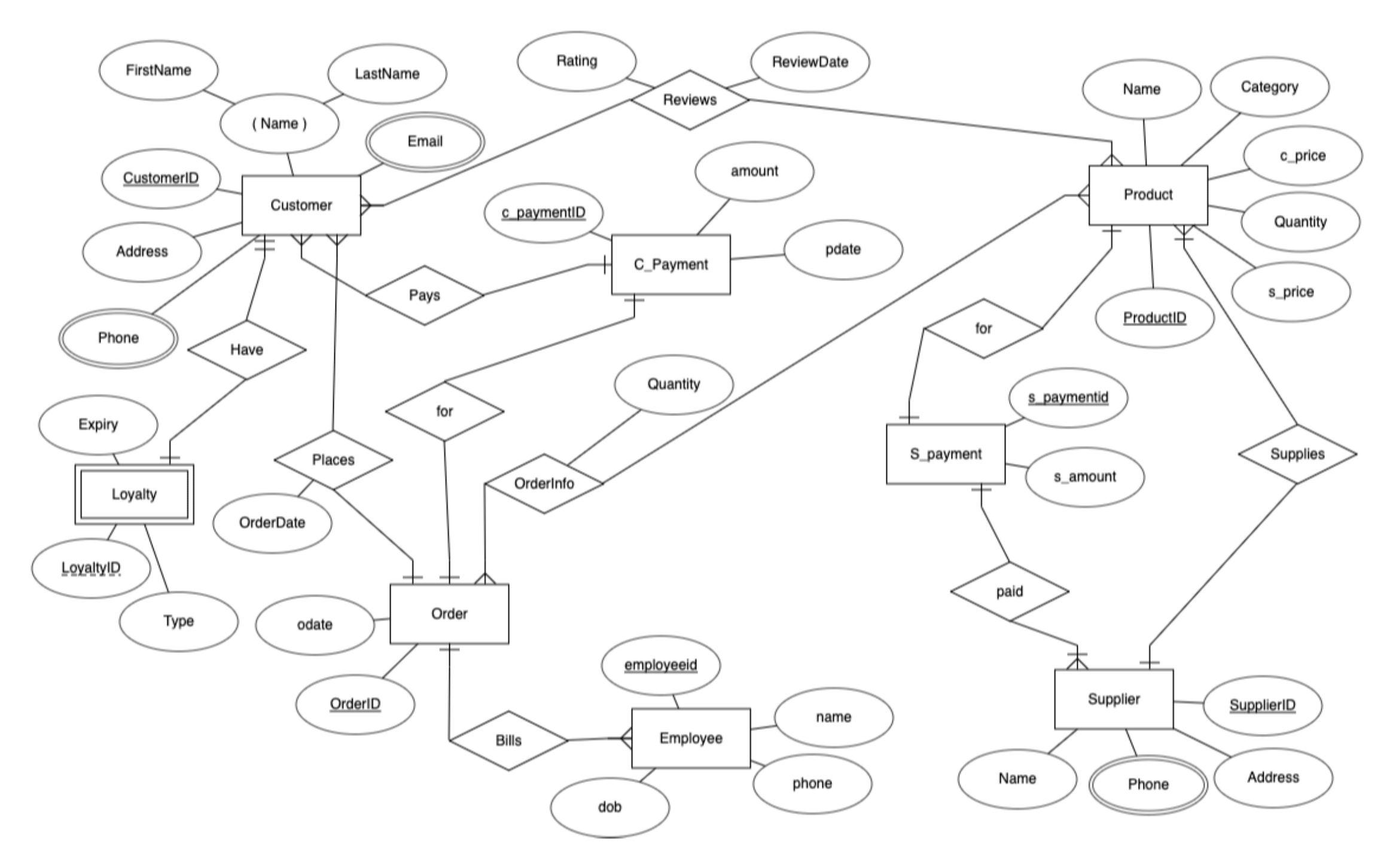
Departmental stores are facing various challenges in managing their inventory, tracking sales, and maintaining customer relations. Many stores rely on manual systems that are time-consuming and error-prone. This can result in poor customer service, stockouts, and loss of sales.

To address these issues, there is a need for a digital management system that can automate store processes, provide real-time inventory updates, and track sales data. This project aims to create a departmental store management system that uses database design and UI creation to improve the efficiency and effectiveness of store operations, resulting in improved customer satisfaction, increased sales, and reduced costs.

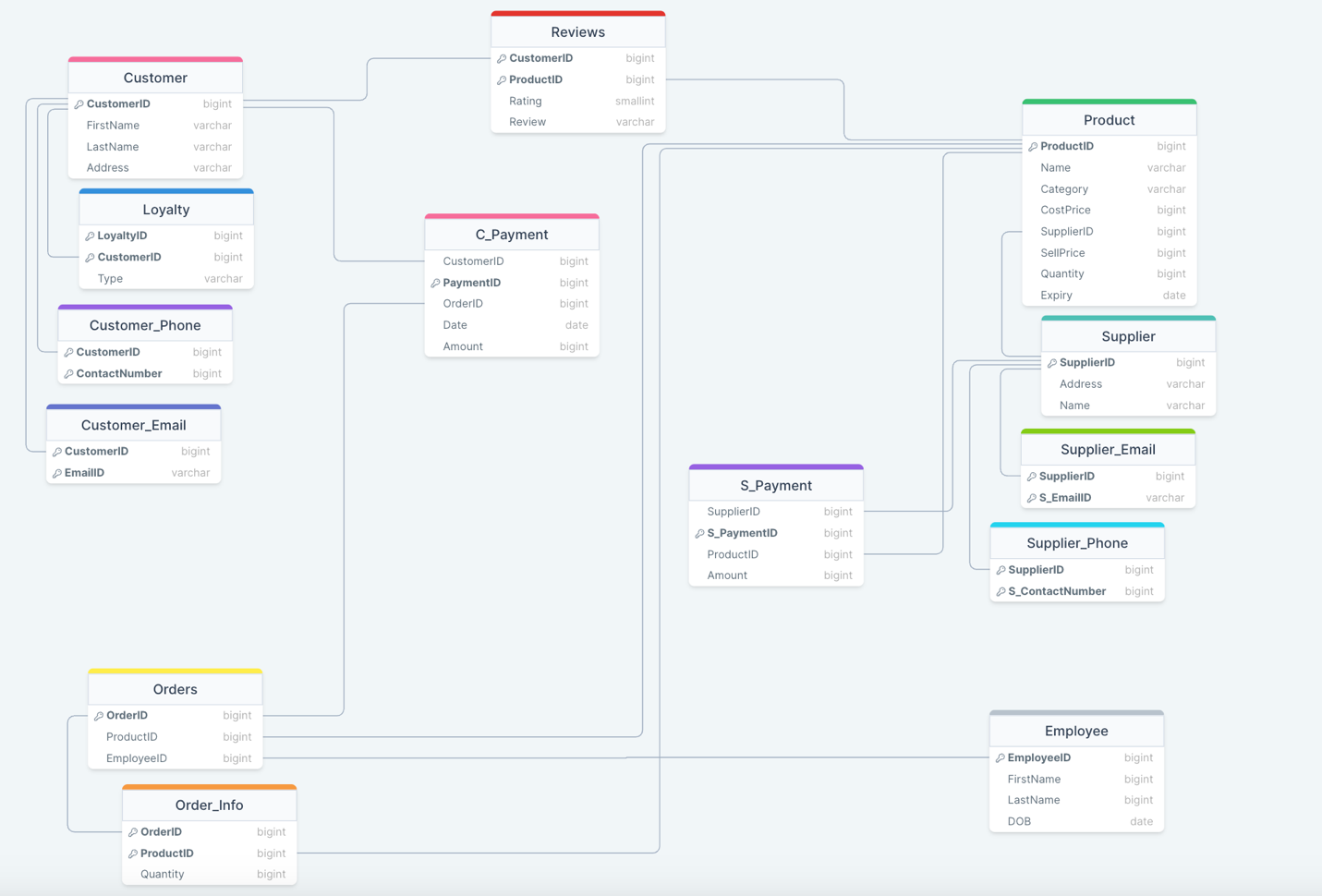
OBJECTIVES

1. Develop a departmental store management system that automates inventory management, sales tracking, and customer relations management processes.
2. Implement a user-friendly UI that allows store employees to easily access and input data into the system.
3. Create a database design that accurately stores and organizes product information, employee data, and sales transactions, among others.
4. Provide real-time updates on inventory levels and sales data to improve decision-making and restocking processes.
5. Improve overall store efficiency and profitability by reducing errors and streamlining operations through the implementation of the digital management system.

DATABASE DESIGN

ER DIAGRAM

SCHEMA DIAGRAM



REDUCTION

Customer(**CustomerID**, FirstName, LastName, Address)

Loyalty(**LoyaltyID, CustomerID,** Type)

Customer\_Phone(**CustomerID, ContactNumber**)

Customer\_Email(**CustomerID, EmailID**)

Orders(**OrderID**, ProductID, EmployeeID)

Order\_Info(**OrderID, ProductID,** Quantity)

Reviews(**CustomerID**, **ProductID**, Rating, Review)

C\_Payment(CustomerID, **PaymentID**, OrderID, Date, Amount)

S\_Payment(SupplierID, S\_**PaymentID**, ProductID, Amount)

Product(**ProductID**, Name, Category, CostPrice, SupplierID, SellPrice, Quantity, Expiry)

Supplier(**SupplierID**, Address, Name)

Supplier\_Phone(**SupplierID, S\_ContactNumber**)

Supplier\_Email(**SupplierID, S\_EmailID**)

Employee(**EmployeeID**, FirstName, LastName, DOB)

NORMALIZATION

The functional dependencies are as follows:

* In the Customer table, CustomerID → FirstName, LastName, Address.
* In the Loyalty table, LoyaltyID → CustomerID, and CustomerID → LoyaltyID, Type (since each customer can have at most one loyalty record).
* In the Customer\_Phone table, (CustomerID, ContactNumber) → none (since a customer can have multiple phone numbers).
* In the Customer\_Email table, (CustomerID, EmailID) → none (since a customer can have multiple email addresses).
* In the Orders table, OrderID → ProductID, EmployeeID, and ProductID, EmployeeID → OrderID (since an order can have multiple products and employees, but each product/employee can only be associated with one order).
* In the Order\_Info table, (OrderID, ProductID) → Quantity (since the quantity of a product in an order depends on both the order and the product).
* In the Reviews table, (CustomerID, ProductID) → Rating, Review (since each customer can submit at most one review for a given product).
* In the C\_Payment table, PaymentID → CustomerID, OrderID, Date, Amount, and (CustomerID, OrderID) → PaymentID (since each payment is uniquely identified by a payment ID, and a customer can make multiple payments for the same order).
* In the S\_Payment table, S\_PaymentID → SupplierID, ProductID, Amount, and (SupplierID, ProductID) → S\_PaymentID (since each payment is uniquely identified by a payment ID, and a supplier can receive multiple payments for the same product).
* In the Product table, ProductID → Name, Category, CostPrice, SupplierID, SellPrice, Quantity, Expiry, and SupplierID → Address, Name (since a product is uniquely identified by its ID, and a supplier can supply multiple products).
* In the Supplier table, SupplierID → Address, Name.
* In the Supplier\_Phone table, (SupplierID, S\_ContactNumber) → none (since a supplier can have multiple phone numbers).
* In the Supplier\_Email table, (SupplierID, S\_EmailID) → none (since a supplier can have multiple email addresses).
* In the Employee table, EmployeeID → FirstName, LastName, DOB.

1NF

The given schema is already in 1NF because it satisfies the criteria of atomicity, i.e., each column in a table contains only a single value.

2NF

To convert the given schema to 2NF, we need to ensure that all non-key attributes in each table depend only on the primary key of that table.

We can start by creating a new table for Order\_Info, which has a composite primary key of OrderID and ProductID, and contains only the Quantity attribute. This separates out the non-key attribute Quantity from the Orders table and eliminates the partial dependency.

Orders(OrderID, EmployeeID)

Order\_Product(OrderID, ProductID)

Product(ProductID, Name, Category, CostPrice, SupplierID, SellPrice, Quantity, Expiry)

C\_Payment(PaymentID, CustomerID, OrderID, Date, Amount)

S\_Payment(S\_PaymentID, SupplierID, ProductID, Amount)

Next, we can create a new table for Loyalty, which has a composite primary key of LoyaltyID and CustomerID, and contains only the Type attribute. This separates out the non-key attribute Type from the Customer table and eliminates the partial dependency.

Customer(CustomerID, FirstName, LastName, Address)

Loyalty(LoyaltyID, CustomerID, Type)

Customer\_Phone(CustomerID, ContactNumber)

Customer\_Email(CustomerID, EmailID)

Reviews(CustomerID, ProductID, Rating, Review)

Supplier(SupplierID, Address, Name)

Supplier\_Phone(SupplierID, S\_ContactNumber)

Supplier\_Email(SupplierID, S\_EmailID)

Employee(EmployeeID, FirstName, LastName, DOB)

3NF

To convert the schema to 3NF, we need to eliminate transitive dependencies. A transitive dependency occurs when a non-key attribute depends on another non-key attribute rather than the primary key.

We can start by creating a new table for the Supplier\_Products, which has a composite primary key of SupplierID and ProductID, and contains only the SupplierCost attribute. This separates out the non-key attribute SupplierCost from the Product table and eliminates the transitive dependency between the SupplierID and the CostPrice.

Customer(CustomerID, FirstName, LastName, Address)

Loyalty(LoyaltyID, CustomerID, Type)

Customer\_Phone(CustomerID, ContactNumber)

Customer\_Email(CustomerID, EmailID)

Orders(OrderID, EmployeeID)

Order\_Product(OrderID, ProductID)

Order\_Info(OrderID, ProductID, Quantity)

Reviews(CustomerID, ProductID, Rating, Review)

C\_Payment(PaymentID, CustomerID, OrderID, Date, Amount)

S\_Payment(S\_PaymentID, SupplierID, ProductID, Amount)

Product(ProductID, Name, Category, SellPrice, Quantity, Expiry)

Supplier(SupplierID, Address, Name)

Supplier\_Products(SupplierID, ProductID, SupplierCost)

Supplier\_Phone(SupplierID, S\_ContactNumber)

Supplier\_Email(SupplierID, S\_EmailID)

Employee(EmployeeID, FirstName, LastName, DOB)

BCNF

To convert the schema to BCNF, we need to ensure that all functional dependencies are dependencies on the primary key, and there are no non-trivial functional dependencies between non-key attributes.

There are no non-trivial functional dependencies between non-key attributes in the current schema, so it is already in BCNF.

METHODOLOGY

To develop the departmental store management system, a comprehensive methodology was followed, encompassing the stages of data gathering, schema design, and user interface (UI) creation. This section provides an overview of the step-by-step process undertaken to achieve the desired outcomes.

Based on the gathered data, a detailed database schema was created to represent the various entities and their relationships. The team analyzed the requirements and identified the key entities, including products, employees, customers, suppliers, orders, and payments. These entities were organized into separate tables, ensuring the appropriate attributes and relationships were defined.

The schema consisted of primary keys, foreign keys, and the necessary constraints to maintain data integrity. For example, the Customer table contained fields such as CustomerID, FirstName, LastName, and Address, while the Orders table included attributes like OrderID, ProductID, and EmployeeID. The schema design phase aimed to capture all the essential information needed for effective store management.

Simultaneously with the schema design, we focused on developing the user interface (UI) for the management system. The UI was created using a combination of HTML, CSS, JavaScript, and Bootstrap framework. The goal was to design an interface that would facilitate easy navigation and efficient data entry.

The UI design incorporated a visually appealing layout with well-organized sections for different functionalities. The team worked on creating elements to facilitate data entry, retrieval, and manipulation. CSS styling was applied to enhance the visual presentation, while JavaScript and Bootstrap were utilized to add functionality and responsiveness to the UI.

In order to ensure data integrity, eliminate redundancy, and improve database performance, the tables in the departmental store management system were normalized. Normalization is organizing data into efficient structures by minimizing data duplication and establishing relationships between entities. The normalization process involved analysing the data requirements and applying the principles of normalization, specifically up to the third normal form (3NF). This involved breaking down tables into smaller, more focused entities and arranging them in a way that reduced redundancy.

Overall, the methodology encompassed data gathering, schema design, and UI creation, all aimed at developing a robust departmental store management system. By combining the insights gathered from stakeholders, a well-structured database schema was created, enabling efficient data management. Simultaneously, the UI design process focused on delivering a user-friendly interface that met the needs of the store staff. The seamless integration of database design and UI creation laid the foundation for an effective system that streamlines inventory management, sales tracking, and customer relations in the departmental store.

CODE FOR FUNCTIONAL DESIGN

* DDL

create table customer(

customerid int,

firstname varchar(20),

lastname varchar(20),

address varchar(40),

primary key(customerid)

);

create table customer\_phone(

customerid int,

phone varchar(30),

primary key(phone,customerid),

foreign key(customerid)references customer(customerid) on delete cascade

);

create table customer\_email(

customerid int,

email varchar(30),

primary key(customerid,email),

foreign key(customerid)references customer(customerid) on delete cascade

);

create table loyalty(

customerid int,

loyaltyid int,

type varchar(20),

expiry date,

primary key(customerid,loyaltyid),

foreign key(customerid)references customer(customerid) on delete cascade

);

create table supplier(

supplierid int,

address varchar(30),

name varchar(20),

primary key(supplierid)

);

create table supplier\_phone(

supplierid int,

phone varchar(30),

primary key(supplierid,phone),

foreign key(supplierid)references supplier(supplierid) on delete cascade);

create table product(

productid int,

name varchar(20),

category varchar(30),

c\_price int,

supplierid int,

s\_price int,

quantity int,

expiry date,

primary key(productid,supplierid),

foreign key(supplierid)references supplier(supplierid) on delete cascade

);

create table employee(

employeeid int,

name varchar(20),

dob date,

primary key(employeeid)

);

create table orders(

orderid int,

customerid int,

orderdate date,

employeeid int,

primary key(orderid,customerid,employeeid),

foreign key(customerid)references customer(customerid) on delete cascade,

foreign key(employeeid)references employee(employeeid) on delete cascade

);

create table Order\_Info(

orderid INT,

productid INT,

quantity INT,

PRIMARY KEY(orderid, productid),

FOREIGN KEY(orderid) references orders(orderid) on delete cascade,

FOREIGN KEY(productid) references product(productid) on delete cascade

);

create table reviews(

customerid INT,

productid INT,

rating INT,

PRIMARY KEY (customerid, productid),

FOREIGN KEY(customerid) references customer(customerid) on delete cascade,

FOREIGN KEY(productid) references product(productid) on delete cascade

);

create table c\_payment(

customerid INT,

c\_paymentid INT,

orderid INT,

pdate date,

amount INT,

PRIMARY KEY (c\_paymentid),

FOREIGN KEY(customerid) references customer(customerid) on delete cascade,

FOREIGN KEY(orderid) references orders(orderid) on delete cascade

);

create table s\_payment(

supplierid INT,

s\_paymentid INT,

productid INT,

amount INT,

PRIMARY KEY (s\_paymentid),

FOREIGN KEY(supplierid) references supplier(supplierid) on delete cascade,

FOREIGN KEY(productid) references product(productid) on delete cascade

);

* DML

select \* from customer natural join customer\_phone natural join customer\_email;

select \* from product

insert into customer values(${customerId},'${customerFName}','${customerLName}','${customerAddress}')

insert into customer\_phone values(${customerId},${custPhone})

insert into customer\_email values(${customerId},'${custEmail}')

insert into product(productid,name,category,c\_price,supplierid,s\_price,quantity,expiry) SELECT ${productId},'${productName}','${productCategory}',${productCPrice},${productSuppId},${productSuppPrice},${productQty},'${productExpiryDate}' FROM dual

WHERE (SELECT COUNT(\*) FROM supplier where supplierid='${productSuppId}') > 0;

delete from product where productid=${product\_id}

delete from customer where customerid=${customer\_id}

insert into supplier values(${supplierid},'${supplieraddress}','${suppliername}')

insert into supplier\_phone values(${supplierid},'${supplierphone}');

select \* from supplier natural join supplier\_phone;

insert into employee values(${employeeid},'${employeename}','${employeedob}')

SELECT

employeeid,

name,

dob,

TIMESTAMPDIFF(YEAR, dob, CURDATE()) AS age

FROM

employee;

insert into loyalty (customerid,loyaltyid,type,expiry) SELECT ${custId},${loyaltyId},'${loyaltyCategory}','${loyaltyExpiryDate}' FROM dual

WHERE (SELECT COUNT(\*) FROM customer where customerid='${custId}') > 0;

select \* from loyalty natural join customer natural join customer\_phone natural join customer\_email;

SELECT oi.orderid, c.firstname, c.lastname, c.customerid,

SUM(p.c\_price \* oi.quantity) \*

CASE l.type

WHEN 'Silver' THEN 0.95

WHEN 'Gold' THEN 0.9

WHEN 'Platinum' THEN 0.8

ELSE 1

END AS total\_amt,

l.type AS loyalty\_type

FROM orders o

JOIN customer c ON o.customerid = c.customerid

JOIN Order\_Info oi ON o.orderid = oi.orderid

JOIN product p ON oi.productid = p.productid

LEFT JOIN loyalty l ON c.customerid = l.customerid

GROUP BY oi.orderid, c.firstname, c.lastname, c.customerid, l.type;

;

SELECT s.name as Supplier\_Name,s.supplierid as id,p.category, COUNT(\*) AS num\_products\_ordered

FROM supplier s

JOIN product p ON s.supplierid = p.supplierid

JOIN order\_info oi ON p.productid = oi.productid

WHERE p.category = '${productCategoryFilter}'

GROUP BY s.name, p.category,s.supplierid

ORDER BY num\_products\_ordered DESC;

UPDATE product

SET name = '${productName}',

category = '${productCategory}',

c\_price = '${productCPrice}',

s\_price = '${productSPrice}',

quantity = '${productQty}',

expiry = '${productExpiryDate}'

WHERE productid = ${productId};

UPDATE customer c

JOIN customer\_phone cp ON c.customerid = cp.customerid

SET c.firstname = '${customerFName}', c.lastname = '${customerLName}', c.address = '${customerAddress}', cp.phone = ${custPhone}

WHERE c.customerid = ${customerId}`;

SELECT c.customerid, c.firstname, c.lastname, COUNT(o.orderid) as order\_count

FROM customer c

JOIN orders o ON c.customerid = o.customerid

GROUP BY c.customerid, c.firstname, c.lastname

ORDER BY order\_count DESC;

SELECT c.customerid, c.firstname, c.lastname, SUM(cp.amount) as total\_spent

FROM customer c

JOIN c\_payment cp ON c.customerid = cp.customerid

GROUP BY c.customerid, c.firstname, c.lastname

ORDER BY total\_spent DESC

LIMIT 10;

UPDATE supplier

JOIN supplier\_phone ON supplier.supplierid = supplier\_phone.supplierid

SET supplier.name = '${suppliername}',

supplier.address = '${supplieraddress}', supplier\_phone.phone = ${supplierphone}

WHERE supplier.supplierid = ${supplierid}`;

SELECT oi.orderid, c.firstname, c.lastname, c.customerid,o.orderdate,

SUM(p.c\_price \* oi.quantity) \*

CASE l.type

WHEN 'Silver' THEN 0.95

WHEN 'Gold' THEN 0.9

WHEN 'Platinum' THEN 0.8

ELSE 1

END AS total\_amt,

l.type AS loyalty\_type

FROM orders o

JOIN customer c ON o.customerid = c.customerid

JOIN Order\_Info oi ON o.orderid = oi.orderid

JOIN product p ON oi.productid = p.productid

LEFT JOIN loyalty l ON c.customerid = l.customerid

where oi.orderid=${orderId}

GROUP BY oi.orderid, c.firstname, c.lastname, c.customerid, l.type,o.orderdate;

;

INSERT INTO c\_payment (customerid, c\_paymentid, orderid, pdate, amount)

VALUES (${customerid}, ${orderid}, ${orderid}, '${finalDate}',${amount} );

UPDATE employee

SET name = '${employeename}',

dob ='${employeedob}'

WHERE employeeid = ${employeeid};`

UPDATE Order\_Info

SET quantity = ${orderQty}

WHERE orderid =${orderid} AND productid = ${orderProduct};

INSERT INTO reviews (customerid, productid, rating,review)

VALUES (${customerId}, ${productId}, ${rating},'${review}');

SELECT \*

FROM reviews

SELECT COUNT(\*) AS total\_orders

FROM orders;

SELECT SUM(amount) AS total\_revenue

FROM orders o

JOIN c\_payment p ON o.orderid = p.orderid;

SELECT COUNT(customerid) AS total\_customers

FROM customer;

SELECT COUNT(\*) as total\_reviews FROM reviews;

SELECT AVG(rating) AS average\_rating FROM reviews;

//INDEX PAGE

SELECT SUM(Order\_info.quantity \* c\_price) as revenue

FROM orders

INNER JOIN Order\_Info ON orders.orderid = Order\_Info.orderid

INNER JOIN product ON Order\_Info.productid = product.productid

WHERE orders.orderdate >= DATE\_SUB(NOW(), INTERVAL 30 DAY);

SELECT o.orderid,c.firstname, c.lastname, SUM(p.amount) AS total\_amount, MAX(p.pdate) AS payment\_date

FROM c\_payment AS p

INNER JOIN orders AS o ON p.orderid = o.orderid

INNER JOIN customer AS c ON o.customerid = c.customerid

GROUP BY o.orderid,c.firstname, c.lastname

ORDER BY payment\_date DESC

LIMIT 10;

SELECT COUNT(DISTINCT c.customerid) AS customers\_with\_loyalty,

COUNT(DISTINCT c2.customerid) AS total\_customers

FROM customer c

LEFT JOIN loyalty l ON c.customerid = l.customerid

JOIN customer c2 ON c.customerid = c2.customerid;

SELECT COUNT(\*) as total\_reviews FROM reviews;

SELECT AVG(rating) AS average\_rating FROM reviews;

SELECT COUNT(\*) AS total\_orders

FROM orders;

SELECT SUM(amount) AS total\_revenue

FROM orders o

JOIN c\_payment p ON o.orderid = p.orderid;

SELECT COUNT(customerid) AS total\_customers

FROM customer;

DELIMITER //

CREATE TRIGGER delete\_expired\_loyalty

BEFORE INSERT OR UPDATE ON orders

FOR EACH ROW

BEGIN

DELETE FROM loyalty

WHERE customerid = NEW.customerid AND expiry < CURDATE();

END//

DELIMITER ;

DELIMITER //

CREATE TRIGGER reduce\_product\_quantity AFTER INSERT ON Order\_Info

FOR EACH ROW

BEGIN

UPDATE product

SET quantity = quantity - NEW.quantity

WHERE productid = NEW.productid;

END$$

DELIMITER ;

DELIMITER //

CREATE PROCEDURE add\_order\_and\_info(

IN customer\_id INT,

IN employee\_id INT,

IN order\_id INT,

IN product\_id INT,

IN quantity INT

)

BEGIN

DECLARE available\_quantity INT;

SELECT quantity INTO available\_quantity FROM product WHERE productid = product\_id;

-- Check if the available quantity is enough

IF available\_quantity >= quantity THEN

-- Check if the order exists

IF NOT order\_exists(order\_id) THEN

-- If it doesn't, add a new order for the customer and employee

INSERT INTO orders (orderid, customerid, employeeid, orderdate)

VALUES (order\_id, customer\_id, employee\_id, NOW());

END IF;

INSERT INTO order\_info (orderid, productid, quantity)

VALUES (order\_id, product\_id, quantity);

-- Reduce the product quantity

UPDATE product SET quantity = quantity - quantity WHERE productid = product\_id;

ELSE

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'Insufficient quantity available for product';

END IF;

END //

DELIMITER ;

CREATE PROCEDURE `filter\_products123`(IN min\_price INT, IN max\_price INT)

BEGIN

DECLARE done INT DEFAULT FALSE;

DECLARE product\_id INT;

DECLARE product\_name VARCHAR(255);

DECLARE product\_category VARCHAR(255);

DECLARE product\_c\_price DECIMAL(10, 2);

DECLARE product\_supplier\_id INT;

DECLARE product\_s\_price DECIMAL(10, 2);

DECLARE product\_quantity INT;

DECLARE product\_expiry DATE;

DECLARE cur\_products CURSOR FOR SELECT productid, name, category, c\_price, supplierid, s\_price, quantity, expiry FROM product WHERE c\_price BETWEEN min\_price AND max\_price;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

OPEN cur\_products;

products\_loop: LOOP

FETCH cur\_products INTO product\_id, product\_name, product\_category, product\_c\_price, product\_supplier\_id, product\_s\_price, product\_quantity, product\_expiry;

IF done THEN

LEAVE products\_loop;

END IF;

-- Do something with the fetched values, for example, print them out

SELECT product\_id as id, product\_name as name, product\_category as category, product\_c\_price as c\_price, product\_supplier\_id as supplierid, product\_s\_price as s\_price, product\_quantity as quantity, product\_expiry as expiry;

END LOOP;

CLOSE cur\_products;

END $$

DELIMITER ;

CREATE PROCEDURE `filter\_suppliers\_by\_category`(IN category\_name VARCHAR(30))

BEGIN

SELECT s.name AS supplier\_name, COUNT(p.product\_id) AS num\_of\_products

FROM supplier s

JOIN product p ON s.supplierid = p.supplierid

WHERE p.category = category\_name

GROUP BY s.supplierid

ORDER BY num\_of\_products DESC;

END

* Database Connectivity

var createError = require('http-errors');

var express = require('express');

var path = require('path');

var cookieParser = require('cookie-parser');

var logger = require('morgan');

const bodyParser = require('body-parser');

var indexRouter = require('./routes/index');

var usersRouter = require('./routes/users');

var dataRouter = require('./test');

var app = express();

// view engine setup

app.set('views', path.join(\_\_dirname, 'views'));

app.set('view engine', 'hbs');

app.use(logger('dev'));

app.use(express.json());

app.use(express.urlencoded({ extended: false }));

app.use(cookieParser());

app.use(express.static(path.join(\_\_dirname, 'public')));

app.use(bodyParser.urlencoded({ extended: true }));

console.log("9");

app.use('/index', indexRouter);

app.use('/users', usersRouter);

app.use('/',dataRouter);

// catch 404 and forward to error handler

app.use(function(req, res, next) {

next(createError(404));

});

// error handler

app.use(function(err, req, res, next) {

// set locals, only providing error in development

res.locals.message = err.message;

res.locals.error = req.app.get('env') === 'development' ? err : {};

// render the error page

res.status(err.status || 500);

res.render('error');

});

module.exports = app;

//test.js Connecting To MySql

var express = require("express");

var router = express.Router();

var mysql = require("mysql");

var connection = mysql.createConnection({

host: "localhost",

user: "root",

password: "password",

database: "dbs",

});

const bodyParser = require('body-parser');

router.use(bodyParser.urlencoded({ extended: true }));

connection.connect(function (err) {

if (err) throw err;

console.log("connected!");

});

CONCLUSION

In conclusion, the development of an efficient management system for departmental stores will be highly beneficial for both the store owners and employees. By providing easy access to information related to suppliers, customers, inventory, and loyalty programs, this system can streamline daily operations and increase productivity. It can also lead to better customer satisfaction by ensuring that essential goods are readily available and that customers are rewarded for their loyalty. Therefore, it is recommended that departmental stores consider investing in such a system to improve their services and overall performance.

LIMITATIONS

Despite the potential benefits of an efficient management system for departmental stores, there are also some limitations to consider. One limitation is the initial cost of implementing the system, which may be high and require significant investment. Additionally, the system may require ongoing maintenance and updates, which can add to the overall cost. Another limitation is the need for employees to be trained on how to use the system effectively, which may take time and resources. Furthermore, there may be technological limitations or compatibility issues with existing systems that could cause disruptions or delays during the implementation process. Finally, the system may not be able to account for unforeseen changes in the market or sudden shifts in demand for certain products. Therefore, while an efficient management system can provide many benefits, it is important to carefully consider these limitations before investing in such a system.

FUTURE WORK

Future work for this departmental store management system project involves several potential areas for improvement and expansion. One area for further development could be the incorporation of machine learning algorithms to predict customer demand and optimize inventory levels. Another possibility is to integrate the system with e-commerce platforms to provide an omnichannel shopping experience for customers.

Additionally, the system could be extended to include supply chain management features, allowing for better coordination with suppliers and more efficient order fulfillment. Finally, further improvements to the UI could be made to enhance user experience and improve the system's usability. These potential future developments have the potential to further increase the system's efficiency and effectiveness in managing departmental   
store operations.

REFERENCES

* Stack Overflow - https://stackoverflow.com/
* BootStrap Documentation - https://getbootstrap.com/docs/5.3/getting-started/introduction/
* Geeks For Geeks - https://www.geeksforgeeks.org/plsql-introduction/