



Dairy Product Company - Haleeb

21.04.2023

Nikita Rajesh Verma - 2021546(CSAI)

Ishwar Babu - 2021532(CSAI)

SCOPE OF PROJECT

The scope of this project is to develop a comprehensive system that can manage the various functionalities and technical requirements of the dairy products company. The system will be responsible for managing inventory, distributor chain, supplier chain, sales, financial management, data analytics and accounts, and more. Additionally, the system will need to be scalable, secure, and available 24/7, with a user-friendly interface. The goal is to improve the efficiency and effectiveness of the company's operations and to support data-driven decision-making and performance measurement.

The objectives of this project are:

1. To improve the efficiency and effectiveness of the company's operations by carrying out processes easily and quickly which is not possible manually.
2. To support data-driven decision-making and performance measurement by providing real-time data and analytics.
3. To enhance the customer experience by providing a web portal/application that allows for easy monitoring and controlling of all functionalities and makes adding, updating, and deleting records easier.
4. Customers can log in, browse the products, place an order, and check the delivery status.
5. Employees can log in, and check sales, inventory, finances, distributor and supplier info.

Technical Requirements -


For the front-end , we plan on using the following:

1. MySQL
2. Python

We need to ensure payment gateway integration, product inventory management, shopping cart and checkout functionality, order tracking and management, delivery management, and raw materials supply to facilitate the smooth running of the factory and website.

Functional Requirements -

For Customer -

- 
1. Creating an account and logging in - verify username and password to ensure the security and protection of data and its access.
 2. Add, delete, and clear products to/from the cart
 3. Placing an order - Will place the order and create a bill for the customer. The distributors will be alerted.
 4. Checking the delivery status
 5. Browsing products available
 6. Payment Management - Have the option to pay using multiple payment modes

For Employee -

1. Creating an account and logging in (Both for customers and Employees) - Will verify username and password to ensure the security and protection of data and its access.
2. Checking the sales, inventory, finances, distributor and supplier info., etc.
3. Add, delete, and edit information about the product.

For Admin -

1. Logging in to existing account - will verify username and password to ensure the security and protection of data and its access.
2. Can check the sales, inventory, finances, distributor and supplier info., along with the customer, supplier, and distributor information.
3. Add, and delete products in the database.
4. Add, and delete suppliers in the database.
5. Add, and delete distributors in the database.

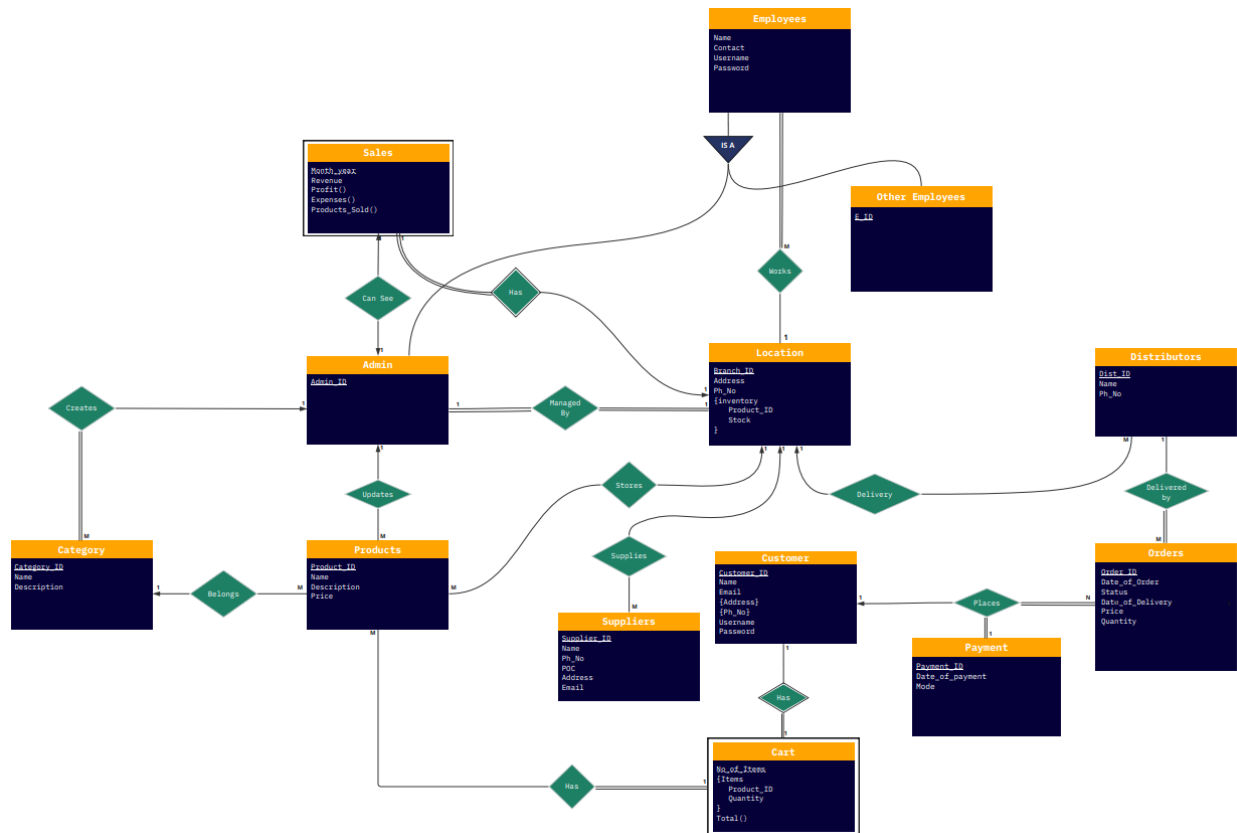
For Distributor -

1. Checking the sales, inventory, finances, distributor and supplier info., etc.
2. Receiving and accepting delivery requests
3. Deliver the product/s to the customer location
4. Update delivery status

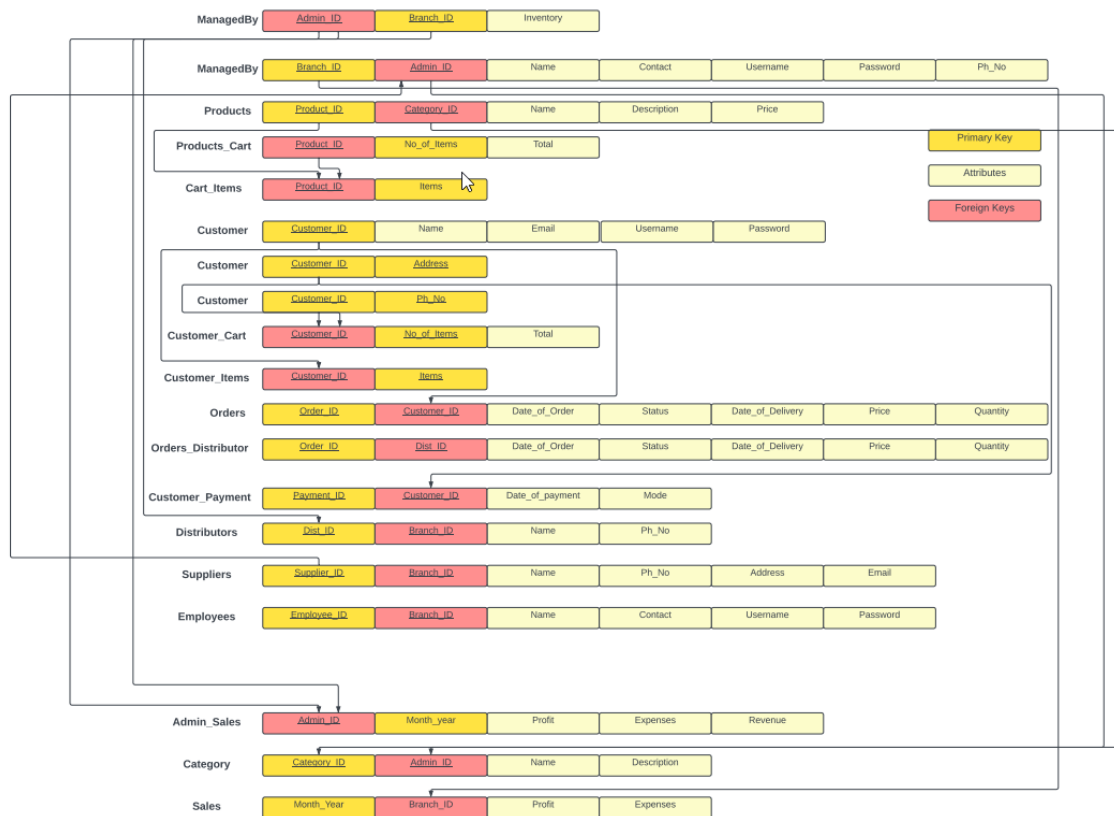
For Supplier -

1. Delivering the raw materials such as skimmed and whole milk powder, whey fats and whey powder concentrates etc., to the company on a daily basis.
2. The ability to schedule and track deliveries of raw materials to companies.
3. To ensure that raw materials meet quality standards and comply with regulations.

ER Diagram -



Relational Schema -



Database Schema -

```
CREATE SCHEMA `haleeb` ;
CREATE TABLE `admin` (
  `admin_id` int NOT NULL,
  `name` varchar(100) NOT NULL,
  `ph_no` varchar(10) NOT NULL,
  `username` varchar(100) NOT NULL,
  `password` varchar(100) NOT NULL,
  PRIMARY KEY (`admin_id`),
  UNIQUE KEY `contact_UNIQUE` (`ph_no`),
```

```
UNIQUE KEY `username_UNIQUE` (`username`)  
)
```

```
CREATE TABLE `cart` (  
  `cust_id` int NOT NULL,  
  `no_of_items` int NOT NULL,  
  PRIMARY KEY (`cust_id`),  
  CONSTRAINT `custome_id` FOREIGN KEY (`cust_id`) REFERENCES `customer`  
  (`customer_id`),  
  CONSTRAINT `cart_chk_1` CHECK ((`no_of_items` >= 0))  
)
```

```
CREATE TABLE `cart_items` (  
  `cust_id` int NOT NULL,  
  `product_id` int NOT NULL,  
  `quantity` int NOT NULL,  
  KEY `custom_id_idx` (`cust_id`),  
  KEY `pro_id_idx` (`product_id`),  
  CONSTRAINT `custom_id` FOREIGN KEY (`cust_id`) REFERENCES `customer`  
  (`customer_id`),  
  CONSTRAINT `pro_id` FOREIGN KEY (`product_id`) REFERENCES `products` (`product_id`)  
)
```

```
CREATE TABLE `category` (  
  `category_id` int NOT NULL AUTO_INCREMENT,  
  `name` varchar(1000) NOT NULL,  
  `description` varchar(2000) NOT NULL,  
  PRIMARY KEY (`category_id`)  
)
```

```
CREATE TABLE `customer` (  
  `customer_id` int NOT NULL AUTO_INCREMENT,  
  `name` varchar(100) NOT NULL,  
  `email` varchar(200) NOT NULL,  
  `username` varchar(100) NOT NULL,  
  `password` varchar(100) NOT NULL,  
  PRIMARY KEY (`customer_id`),  
  UNIQUE KEY `email_UNIQUE` (`email`),  
  UNIQUE KEY `username_UNIQUE` (`username`)  
)
```

```
CREATE TABLE `distributor` (  
  `distributor_id` int NOT NULL,  
  `name` varchar(100) NOT NULL,  
  `ph_no` varchar(10) NOT NULL,  
  `branch_no` int NOT NULL,  
  PRIMARY KEY (`distributor_id`),  
  UNIQUE KEY `ph_no_UNIQUE` (`ph_no`),  
  UNIQUE KEY `branch_no_UNIQUE` (`branch_no`),  
  CONSTRAINT `b_no` FOREIGN KEY (`branch_no`) REFERENCES `location` (`branch_id`)  
)
```

```
CREATE TABLE `location` (  
  `branch_id` int NOT NULL,  
  `address` varchar(100) NOT NULL,  
  `ph_no` varchar(10) NOT NULL,  
  PRIMARY KEY (`branch_id`),  
  UNIQUE KEY `admin_id_UNIQUE` (`branch_id`),
```

```
UNIQUE KEY `b_id_UNIQUE` (`address`),  
UNIQUE KEY `ph_no_UNIQUE` (`ph_no`)  
)
```

```
CREATE TABLE `location_inventory` (  
  `loc_id` int NOT NULL AUTO_INCREMENT,  
  `product_id` int NOT NULL,  
  `stock` int NOT NULL,  
  KEY `loc_id_idx` (`loc_id`),  
  KEY `product_id_idx` (`product_id`),  
  CONSTRAINT `loc_id` FOREIGN KEY (`loc_id`) REFERENCES `location` (`branch_id`),  
  CONSTRAINT `product_id` FOREIGN KEY (`product_id`) REFERENCES `products`  
  (`product_id`)  
)
```

```
CREATE TABLE `orders` (  
  `order_id` int NOT NULL AUTO_INCREMENT,  
  `date_of_order` date NOT NULL,  
  `status` varchar(1000) NOT NULL,  
  `branch_id` int NOT NULL,  
  `payment_id` int NOT NULL,  
  `cust_id` int NOT NULL,  
  PRIMARY KEY (`order_id`),  
  KEY `c_id_idx` (`cust_id`),  
  KEY `b_id_idx` (`branch_id`),  
  KEY `p_id_idx` (`payment_id`),  
  CONSTRAINT `b_id` FOREIGN KEY (`branch_id`) REFERENCES `location` (`branch_id`),  
  CONSTRAINT `c_id` FOREIGN KEY (`cust_id`) REFERENCES `customer` (`customer_id`),  
  CONSTRAINT `p_id` FOREIGN KEY (`payment_id`) REFERENCES `payment` (`payment_id`)  
)
```



```
CREATE TABLE `payment` (  
  `payment_id` int NOT NULL AUTO_INCREMENT,  
  `mode` varchar(1000) NOT NULL,  
  `amount` int NOT NULL,  
  `cust_id` int NOT NULL,  
  PRIMARY KEY (`payment_id`),  
  KEY `c_id_c_idx` (`cust_id`),  
  CONSTRAINT `c_id_c` FOREIGN KEY (`cust_id`) REFERENCES `customer` (`customer_id`)  
)
```

```
CREATE TABLE `products` (  
  `product_id` int NOT NULL AUTO_INCREMENT,  
  `category_no` int NOT NULL,  
  `name` varchar(1000) NOT NULL,  
  `description` varchar(2000) NOT NULL,  
  `price` int NOT NULL,  
  PRIMARY KEY (`product_id`),  
  KEY `cat_id_idx` (`category_no`),  
  CONSTRAINT `cat_id` FOREIGN KEY (`category_no`) REFERENCES `category`  
  (`category_id`)  
)
```

```
CREATE TABLE `supplier` (  
  `supplier_id` int NOT NULL,  
  `name` varchar(100) NOT NULL,  
  `ph_no` varchar(10) NOT NULL,  
  `address` varchar(100) DEFAULT NULL,
```

```

`email` varchar(100) DEFAULT NULL,
`b_no` int NOT NULL,
PRIMARY KEY (`supplier_id`),
UNIQUE KEY `ph_no_UNIQUE` (`ph_no`),
UNIQUE KEY `address_UNIQUE` (`address`),
KEY `branch_id_idx` (`b_no`),
CONSTRAINT `branch_id` FOREIGN KEY (`b_no`) REFERENCES `location` (`branch_id`)
)

```

```


CREATE TABLE `restock_request` (
  `id` int NOT NULL AUTO_INCREMENT,
  `product_id` int NOT NULL,
  `supplier_id` int NOT NULL,
  `quantity` int NOT NULL,
  `b_no` int NOT NULL,
  PRIMARY KEY (`id`),
  KEY `restock_product_id` (`product_id`),
  KEY `restock_supplier_id` (`supplier_id`),
  KEY `branch_id_no_idx` (`b_no`),
  CONSTRAINT `restock_product_id` FOREIGN KEY (`product_id`) REFERENCES `products`
(`product_id`),
  CONSTRAINT `restock_supplier_id` FOREIGN KEY (`supplier_id`) REFERENCES `supplier`
(`supplier_id`)
)

```

Integrity Constraints:

Table -

Location -



primary key = branch_id

Customer -

Primary key = customer_id

Suppliers -

primary key = supplier_id

Delivery -

primary key = delivery_id

adminprimary key = admin_id

categoryprimary key = category_id

Products -

primary key = product_id

Sales -

Partial key = month_year

Foreign key = branch_id

Every sales entity will be uniquely identified by the month of the year and the branch it belongs to. Hence, branch_id becomes the foreign key.

Cart -

Foreign key = c_id(customer_id)

Payment -

Primary key = payment_id

Orders -

Primary key = order_id

Employees

Primary key = id

Foreign key = e_id

Foreign key = admin_id

Integrity Constraints:

INSERT INTO `admin` VALUES

(1,'Melisandra Fleetwood','3252262931','mfleetwood0','sWMUciw'),

(2,'Mariana Hambly','9633045402','mhambly1','MSdzZqTZb'),

(3,'Em Itskovitz','5472941684','eitskovitz2','sv1Pwr6Z')

INSERT INTO `cart` VALUES

(1,0),

(2,13),

(3,0)

INSERT INTO `cart_items` VALUES

(2,34,9),

(2,54,4)



```
INSERT INTO `category` VALUES
```

```
(1,'Cheese',"Dairy products made from milk curd"),
(2,'Milk',"A white liquid produced by the mammary glands of mammals"),
(3,'Yogurt',"A creamy, tangy dairy product made from milk")
```

```
INSERT INTO `customer` VALUES
```

```
(1,"John Doe","john Doe@example.com","john Doe","password1"),
(2,"Jane Doe","jane Doe@example.com","jane Doe","password2"),
(3,"Jim Smith","jim Smith@example.com","jim Smith","password3")
```

```
INSERT INTO `distributor` VALUES
```

```
(1,'Heloise Shaxby','2605059055',1),
(2,'Madalena De La Haye','9384116161',2),
(3,'Hilly Gorey','2099514621',3)
```

```
INSERT INTO `location` VALUES
```

```
(1,"Address 1","1234567890"),
(2,"Address 2","1234567891"),
(3,"Address 3","1234567892")
```

```
INSERT INTO `location_inventory` VALUES
```

```
(1,1,50),
(1,2,50),
(1,3,50)
```

```
INSERT INTO `orders` VALUES
(2,'2022-07-06','Delivered',14,2),
(3,'2022-03-29','Delivered',72,3),
(4,'2022-04-14','Delivered',77,4)
```

```
INSERT INTO `payment` VALUES
(1,'Credit Card',199),
(2,'Cash',299),
(3,'Cash',499)
```

```
INSERT INTO `products` VALUES
(1,2,\"Whole Milk\", \"A creamy and rich whole milk perfect for cooking and drinking.\",299),
(2,2,\"Low Fat Milk\", \"A healthier alternative to whole milk with reduced fat content.\",329),
(3,3,\"Greek Yogurt\", \"A thick and creamy yogurt with a tangy flavor and high protein content.\",199)
```

SQL Queries:

/Query 1(Delete)/

```
DELETE FROM orders
WHERE date_of_order >= '2022-01-01' AND date_of_order <= '2022-02-01';
```

/Query 2(ALTER)/

```
ALTER TABLE customer ADD address varchar(80);
```

The SQL query ALTER TABLE customer ADD address varchar(80); adds a new column called address to the customer table with a data type of VARCHAR(80).

/Query 3(GROUP BY)/

```
SELECT name, AVG(price) as average_price, COUNT(*) as num_products
```

```
FROM products
```

```
WHERE price > 150
```

```
GROUP BY name
```

```
HAVING COUNT(*) >= 2 AND AVG(price) < 500;
```

This query retrieves the name of all products that have a price greater than 150, groups the results by name, and then applies two conditions in the HAVING clause to filter the results.

/Query 4(SUM + AVG + COUNT + GROUP BY + ORDER BY)/

```
SELECT category_no, SUM(price), avg(price), count(category_no) FROM products GROUP BY  
category_no order by category_no;
```

This query gets the sum of price and average price of products from each category to estimate which category has the most expensive items

/Query 5(WILDCARD)/

```
SELECT * From supplier WHERE name LIKE '%O%';
```

This query selects all columns from the suppliers table where the name column contains the letter O anywhere in the name.

/Query 6(AVG)/

```
SELECT AVG(price)FROM products where product_id >= "50" AND product_id <= "60";
```

This query calculates the average value of the price column from the products table, for the products with a product_id between 50 and 60 (inclusive).

/Query 7(ORDER BY)/

```
SELECT distributor_id AS distributor_id, name
```

```
FROM distributor
```

```
ORDER BY name ASC;
```

This query selects the "dist_id" and "name" columns from the "distributor" table, and aliases "dist_id" to "distributor_id" to match the original column name in the first query. The results are ordered by the "name" column in ascending order.

/Query 8(COUNT)/

```
SELECT COUNT(products_id) as num_products, category_no, price
```

```
FROM products2
```

```
GROUP BY category_no, price;
```

This query counts the number of products that have the same price, groups the results by both the category_no and price columns, and returns the number of products in each category at each price point.

/Query 9(DISTINCT)/

```
SELECT DISTINCT mode FROM payment;
```

This query selects unique/data without duplicate values from the table payment.

/QUERY 10(JOIN) /


```
SELECT products.name, category.name, price
FROM products
JOIN category ON products.category_no = category.category_id;
```

This table will serve as a brochure for customer so they can see the products category wise along with the product price

/Query 11(INNER JOIN + CONDITIONAL SUBQUERY)/

```
SELECT orders.order_id, search.distributor_id, search.name FROM orders, (
SELECT distributor.distributor_id, distributor.name, location.branch_id FROM distributor
INNER JOIN location ON distributor.branch_no = location.branch_id)search WHERE
orders.branch_id=search.branch_id ;
```

Shows the distributor corresponding to each branch which can be used to find the distributor of a particular order by comparing the branch_id in order and in the table created above.

/Query 12('OR')/

```
SELECT * FROM orders WHERE status = 'Cancelled' OR status ='Returned'
```

The "or" operator is used to retrieve rows that meet at least one of the two conditions, while the "nor" operator is used implicitly in the second condition to exclude rows where the status is "delivered" and the date of delivery is not null.

/Query 13(RANK + PARTITION + OVER)/

```
SELECT product_id,category_no, name,price,RANK() OVER (PARTITION BY category_no
ORDER BY price desc) "Rank" FROM products;
```

Orders the products by price.This query uses the RANK function to calculate the rank of each product's price within its category.

Embedded Queries:

/Embedded Query 1/

```
SELECT orders.order_id, search.distributor_id, search.name FROM orders, (  
SELECT distributor.distributor_id, distributor.name, location.branch_id FROM distributor  
INNER JOIN location ON distributor.branch_no = location.branch_id)search WHERE  
orders.branch_id=search.branch_id ;
```

This SQL query joins two tables, orders and a subquery aliased as search, to retrieve order details and distributor information based on branch location. Here's a breakdown of the code:


- FROM orders, (
SELECT distributor.distributor_id, distributor.name, location.branch_id
FROM distributor
INNER JOIN location ON distributor.branch_no = location.branch_id) search:

This line specifies the source tables for the query. The first table is orders, and the second table is a subquery aliased as search. The subquery joins the distributor and location tables using an inner join on the branch_no and branch_id columns, respectively, and selects the distributor_id, name, and branch_id columns from the distributor and location tables.

- WHERE orders.branch_id = search.branch_id:

This line specifies the join condition for the query, which links the orders and search tables using the branch_id column. The result set includes rows where the branch_id column in the orders table matches the branch_id column in the search subquery.

/Embedded Query 2/



```
SELECT product_id,category_no, name,price,RANK() OVER (PARTITION BY category_no  
ORDER BY price desc) "Rank" FROM products;
```

This query selects data from a table named products. It returns the product_id, category_no, name, price, and a computed value that represents the rank of each product within its category based on its price. Here is the breakdown of the code.

- product_id, category_no, name, price: These are the columns we want to select from the products table.

- RANK() OVER (PARTITION BY category_no ORDER BY price DESC) AS "Rank":

This is a window function that calculates the rank of each product based on its price within its category. The RANK() function assigns a rank to each row based on the ordering specified by the ORDER BY clause. The PARTITION BY clause divides the rows into partitions based on the values in the category_no column. The AS keyword assigns the name "Rank" to the column that contains the computed ranks.

OLAP Queries:

/OLAP Query 1/

```
SELECT b_no, p_no, SUM(quantity) AS total_quantity  
FROM inventory  
GROUP BY b_no, p_no WITH ROLLUP;
```

The given SQL statement is used to retrieve the total quantity of dairy products and the grand total.

Explanation of the SQL statement:

- SELECT: specifies the columns to be returned in the result set.
- b_no, p_no, SUM(quantity) AS total_quantity: specifies the columns to be grouped by and the aggregate function to calculate the total quantity of products.

- FROM: specifies the table from which the data will be retrieved.
- inventory: the name of the table.
- GROUP BY: groups the rows based on the specified columns.
- b_no, p_no: the columns used for grouping.
- WITH ROLLUP: adds a row at the end of the result set that shows the grand total for each book.

/OLAP Query 2/


```
SELECT category_no, SUM(price) as total_price  
FROM products  
GROUP BY category_no WITH ROLLUP  
ORDER BY category_no;
```

The given SQL statement is used to retrieve the total price of products in each category and the grand total of each category, sorted by category number and product name.

- SELECT: specifies the columns to be returned in the result set.
- category_no, name, SUM(price) as total_price: specifies the columns to be grouped by and the aggregate function to calculate the total price of products.
- GROUP BY: groups the rows based on the specified columns.
- WITH ROLLUP: adds a row at the end of the result set that shows the grand total for each category.
- ORDER BY: sorts the result set in ascending order based on the specified columns.
- category_no, name: the columns used for sorting.

/OLAP Query 3/:

```
SELECT p.category_no,  
MONTH(o.date_of_order) AS month,
```



```
YEAR(o.date_of_order) AS year,  
SUM(i.stock) AS total_inventory  
FROM products p  
JOIN orders o ON p.product_id = o.order_id  
JOIN location_inventory i ON p.product_id = i.product_id  
GROUP BY p.category_no, month, year;
```

This query retrieves the total sales amount and inventory quantity of products by category and month, but it has a small typo which I'll correct.

- `SELECT p.category_no, MONTH(o.date_of_order) AS month, SUM(o.total)`
AS total_sales, SUM(i.quantity) AS total_inventory: This line specifies the columns to be returned in the result set. It selects the category_no column from the products2 table, the month from the date_of_order column in the orders table, the total sales amount from the total column in the orders table, and the total inventory quantity from the quantity column in the inventory table.
- `JOIN orders o ON p.products_id = o.products_id`: This line joins the orders table aliased as o to the main table products2 aliased as p, using the products_id column in both tables.
- `JOIN inventory i ON p.products_id = i.p_no`: This line joins the inventory table aliased as i to the main table products2 aliased as p, using the products_id column in p and the p_no column in i.
- `GROUP BY p.category_no, month`: This line groups the results by category_no and month, so that the sales and inventory data is aggregated for each unique combination of category_no and month.

/OLAP Query 4/

```
SELECT ioc_id, product_id, SUM(stock) AS total_stock
```



```
FROM inventory
```

```
GROUP BY ioc_id, product_id
```

This query aggregates the stock column by ioc_id and product_id and calculates the total stock for each combination of IOCID and product. The result set will include three columns: ioc_id, product_id, and total_stock.

SELECT ioc_id, product_id, SUM(stock) AS total_stock: This line selects the ioc_id and product_id columns from the inventory table, and calculates the sum of the stock column for each combination of ioc_id and product_id. The AS total_stock clause renames the calculated sum as total_stock.

GROUP BY ioc_id, product_id: This line groups the results by ioc_id and product_id, so that the sum of the stock column is calculated for each unique combination of ioc_id and product_id.

Triggers:

1) This trigger helps maintain the cart table's integrity by automatically updating the number of items in a customer's cart whenever a new item is added to the cart_items table. It updates the no_of_items column in the cart table whenever a new row is inserted into the cart_items table by adding the value of the quantity column in the newly inserted row.

```
CREATE DEFINER='root'@'localhost'
```

```
TRIGGER `cart_items_AFTER_INSERT`
```

```
AFTER INSERT ON `cart_items`
```

```
FOR EACH ROW
```

```
BEGIN
```

```
UPDATE cart
```

```
SET no_of_items = no_of_items+NEW.quantity WHERE NEW.cust_id = cart.cust_id;
```


END

2) The purpose of this trigger is to update the stock level in the location_inventory table whenever a customer checks out their cart and removes items from the cart_items table. For each deleted row, the trigger updates the location_inventory table. Specifically, it updates the stock column by subtracting the value of the quantity column in the deleted row. This helps to ensure that the stock levels in the inventory are accurate and up-to-date, which can prevent overselling or stockouts.

```
CREATE DEFINER='root'@'localhost'
TRIGGER `cart_items_BEFORE_DELETE`
BEFORE DELETE ON `cart_items`
FOR EACH ROW
BEGIN
    update location_inventory
    set stock = stock - OLD.quantity where OLD.product_id = location_inventory.product_id
    and location_inventory.loc_id = 1;
END
```

3) This trigger automatically creates an order after a payment is made, with default values for date_of_order, status, and branch_id, and using the payment_id from the newly inserted payment row.

```
CREATE DEFINER='root'@'localhost'
TRIGGER `payment_AFTER_INSERT`
AFTER INSERT ON `payment`
FOR EACH ROW
BEGIN
    insert into orders(date_of_order, status, branch_id, payment_id) values
    (curdate(), "In Transit", 1, NEW.payment_id);
```

END

Transactions

Consider the following two transactions:

Transaction T1:

- a. Read the stock for a product from the location_inventory table.
- b. Decrease the stock by the number of items bought.
- c. Update the location_inventory table with the new stock.

Transaction T2:

- a. Read the stock for the same product from the location_inventory table.
- b. Increase the stock by the number of items added.
- c. Update the location_inventory table with the new stock.

Conflict Serializable Schedule:

Order	T1	T2
1	R(stock)	

2		R(stock)
3		W(stock)
4	W(stock)	

This schedule is conflict serializable because it is equivalent to the serial schedule T1 -> T2, in which T1 executes entirely before T2.

Not Conflict Serializable Schedule:

Order	T1	T2
1	R(Stock)	
2		R(Stock)
3	W(Stock)	
4		W(Stock)

This schedule is not conflict serializable because there is no serial schedule that is equivalent to this one in terms of conflicts. In this schedule, the write of T1 conflicts with the read of T2, and the write of T2 conflicts with the write of T1. Therefore, this schedule does not preserve the same order of conflicting operations as any serial schedule.

Here are the SQL queries for the conflict serializable schedule and the not conflict serializable schedule.

Conflict Serializable Schedule:

Transaction T1:

1. START TRANSACTION;
2. SELECT stock FROM location_inventory WHERE product_id = 1 AND loc_id = 1 FOR UPDATE;
-- Decrease stock (assuming the quantity bought is 5)
3. SET @new_stock = stock - 5;
4. UPDATE location_inventory SET stock = @new_stock WHERE product_id = 1 AND loc_id = 1;
5. COMMIT;

Transaction T2:

1. START TRANSACTION;
2. SELECT stock FROM location_inventory WHERE product_id = 1 AND loc_id = 1 FOR UPDATE;
-- Increase stock (assuming the quantity added is 3)
3. SET @new_stock = stock + 3;
4. UPDATE location_inventory SET stock = @new_stock WHERE product_id = 1 AND loc_id = 1;
5. COMMIT;

Not Conflict Serializable Schedule:**Transaction T1:**

1. START TRANSACTION;
2. SELECT stock FROM location_inventory WHERE product_id = 1 AND loc_id = 1 FOR UPDATE;
-- Decrease stock (assuming the quantity bought is 5)
3. SET @new_stock = stock - 5;
4. COMMIT;

Transaction T2:

1. START TRANSACTION;
2. SELECT stock FROM location_inventory WHERE product_id = 1 AND loc_id = 1 FOR UPDATE;
-- Increase stock (assuming the quantity added is 3)
3. SET @new_stock = stock + 3;
4. COMMIT;

Here are the explanations for the transactions using locking mechanisms:

Conflict Serializable Schedule:

Transaction T1:

START TRANSACTION;

Acquire X-lock on the product row in the location_inventory table.

SELECT stock FROM location_inventory WHERE product_id = 1 AND loc_id = 1 FOR UPDATE;

Decrease stock (assuming the quantity bought is 5).

UPDATE location_inventory SET stock = @new_stock WHERE product_id = 1 AND loc_id = 1;

Release X-lock on the product row.

COMMIT;

Transaction T2:

START TRANSACTION;

Wait for the X-lock acquired by T1 to be released.

Acquire X-lock on the product row in the location_inventory table.

```
SELECT stock FROM location_inventory WHERE product_id = 1 AND loc_id = 1 FOR UPDATE;
```

Increase stock (assuming the quantity added is 3).

```
UPDATE location_inventory SET stock = @new_stock WHERE product_id = 1 AND loc_id = 1;
```

Release X-lock on the product row.

```
COMMIT;
```

In the conflict serializable schedule, the transactions execute one after the other, ensuring data integrity and preventing conflicts.

Not Conflict Serializable Schedule:

Transaction T1:

```
START TRANSACTION;
```

Acquire X-lock on the product row in the location_inventory table.

```
SELECT stock FROM location_inventory WHERE product_id = 1 AND loc_id = 1 FOR UPDATE;
```

Decrease stock (assuming the quantity bought is 5).

Release X-lock on the product row.

```
COMMIT;
```

Transaction T2:

```
START TRANSACTION;
```

Acquire X-lock on the product row in the location_inventory table before T1 has released its lock, leading to a deadlock situation.

```
SELECT stock FROM location_inventory WHERE product_id = 1 AND loc_id = 1 FOR UPDATE;
```

Increase stock (assuming the quantity added is 3).

Release X-lock on the product row.

COMMIT;

In the not conflict serializable schedule, T2 tries to acquire the X-lock on the product row while T1 still holds its X-lock, which leads to a deadlock. In a real-world scenario, a deadlock detection and resolution mechanism would step in to resolve the deadlock by aborting one of the transactions and allowing the other to proceed.

USER GUIDE (Features and workings)

When the code is run, a menu is displayed with login options for our stakeholders i.e Admin, Customer, Distributor and Supplier.

Upon logging in using username and password, several features are available.

For Admin -

- 1) Add Product - insert into products values (%s,%s,%s,%s,%s)
- 2) Add Category - insert into category values (%s,%s,%s)
- 3) View Categories - SELECT * FROM category;
- 4) View Suppliers - SELECT * FROM supplier;
- 5) Add Supplier - INSERT INTO supplier (name, ph_no, address, email, b_no) VALUES (%s, %s, %s, %s, %s)
- 6) Update Supplier - UPDATE supplier SET name=%s, ph_no=%s, address=%s, email=%s, b_no=%s WHERE supplier_id=%s
- 7) Delete Supplier - DELETE FROM supplier WHERE supplier_id=%s
- 8) View Distributors - SELECT * FROM distributor;
- 9) Add Distributor INSERT INTO distributor (name, ph_no, branch_no) VALUES (%s, %s, %s)
- 10) Update Distributor - UPDATE distributor SET name=%s, ph_no=%s, branch_no=%s WHERE distributor_id=%s
- 11) Delete Distributor - DELETE FROM distributor WHERE distributor_id=%s

12) View Branch Stock - SELECT product_id, stock FROM location_inventory where loc_id = '%s'

13) View Sales - select sum(amount) from payment, orders where payment.payment_id = orders.payment_id and orders.branch_id = '%s'

14) View Orders - SELECT * from orders where branch_id = '%s'

For Customer-

1) Browse Products -

- (a) See best selling item - select * from products, (select product_id from location_inventory where loc_id = '%s' order by stock asc limit 3) as table2 where products.product_id = table2.product_id;
- (b) Browse category wise - select * from products where category_no = '%s'
- (c) See catalogue - SELECT products.product_id, products.name, price FROM products JOIN category ON products.category_no = category.category_id order by products.product_id asc

2) Add item to cart - INSERT INTO cart_items(cust_id, product_id, quantity) VALUES (%s, %s, %s)

3) View cart - select products.product_id, products.name, quantity, price*quantity as total_price from cart_items, products where cust_id = '%s' and products.product_id=cart_items.product_id

4) Checkout Cart - insert into payment(mode, amount, cust_id) values(%s, %s, %s)

delete from cart_items where cust_id = '%s'

update cart set no_of_items = 0 where cust_id = '%s'

5) Update quantity/Remove items in cart -

- (a) Decrease quantity - update cart_items set quantity = '%s' where cust_id = '%s' and product_id = '%s';
- (b) Add quantity - update cart_items set quantity = '%s' where cust_id = '%s' and product_id = '%s'
- (c) Remove product - delete from cart_items where cust_id = '%s' and product_id = '%s';

6) Cancel order - delete from orders where order_id = '%s'

7) View previous orders - select order_id, date_of_order, status from orders where cust_id = '%s'

For Distributor-

1) See all orders - select * from orders where branch_id = '%s'

2) See pending orders - select * from orders where branch_id = '%s' and status = "In Transit"

3) Change status of order to delivered - update orders set status = "Delivered" where order_id = '%s'

4) See Orders delivered - select * from orders where branch_id = '%s' and status = "Delivered"

For Supplier -

1) Request Restock - SELECT * from restock_request WHERE supplier_id = %s

2) Accept Restock request - update location_inventory set stock = stock + '%s' where loc_id = '%s' and product_id = %s;