1. Create a Sales table using sale\_id, product\_id, quantity\_sold , sale\_date , total\_price with FOREIGN KEY (product\_id) REFERENCES Products(product\_id), and insert 10 data.

CREATE TABLE sales (

sale\_id BIGINT AUTO\_INCREMENT PRIMARY KEY,

product\_id BIGINT NOT NULL,

FOREIGN KEY (product\_id) REFERENCES products(product\_id),

sale\_date DATE NOT NULL,

quantity\_sold INT NOT NULL,

total\_price DECIMAL (20,2) NOT NULL

);

INSERT INTO sales (product\_id, sale\_date, quantity\_sold, total\_price) VALUES

(1, '2025-01-01', 10, 900.00),

(2, '2025-01-02', 5, 600.00),

(3, '2025-01-03', 2, 3000.00),

(4, '2025-05-04', 8, 640.00),

(5, '2025-05-05', 4, 600.00),

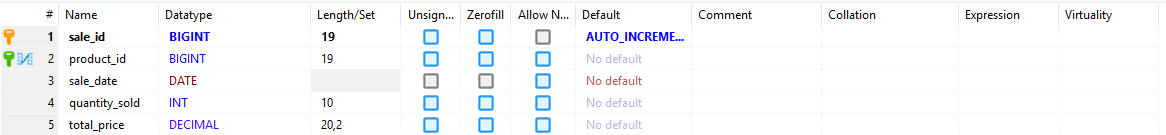
(6, '2025-05-06', 20, 500.00),

(7, '2025-05-07', 15, 900.00),

(8, '2025-05-08', 1, 25000.00),

(9, '2025-05-09', 3, 10500.00),

(10, '2025-05-10', 2, 4000.00);



1. Create a Product table using product\_id, product\_name , category, unit\_price and insert 10 data.

CREATE TABLE products (

product\_id BIGINT AUTO\_INCREMENT PRIMARY KEY,

product\_name VARCHAR(255) NOT NULL,

category VARCHAR(255) NOT NULL,

unit\_price DECIMAL (20,2) NOT NULL

);

INSERT INTO products (product\_name, category, unit\_price) VALUES

('Milk', 'Grocery', 90.00),

('Sugar', 'Grocery', 120.00),

('Rice (Milled)', 'Grocery', 1500.00),

('Toothpaste', 'Daily Use', 80.00),

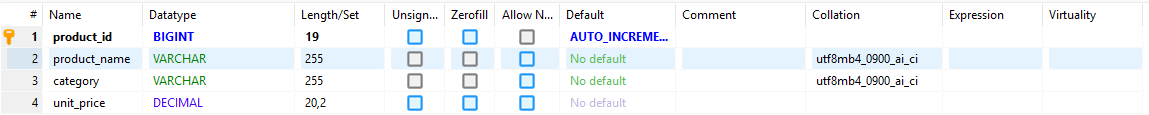
('Shampoo', 'Daily Use', 150.00),

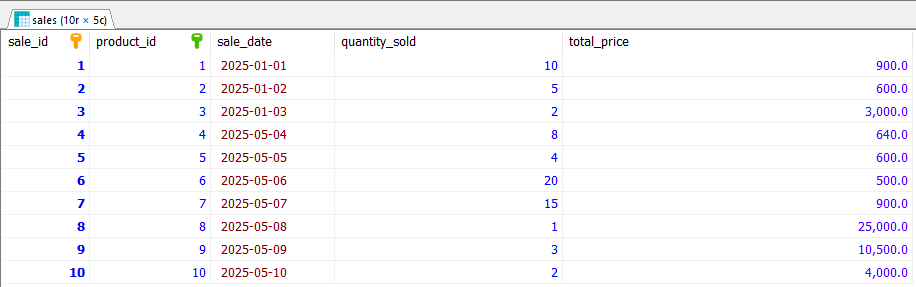
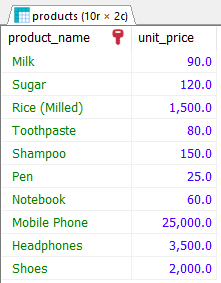
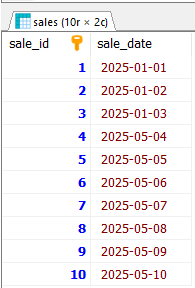
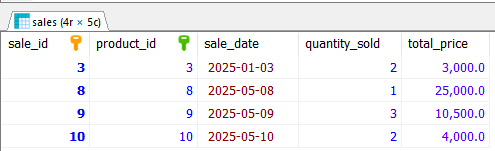
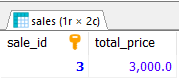
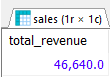
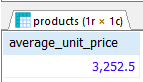
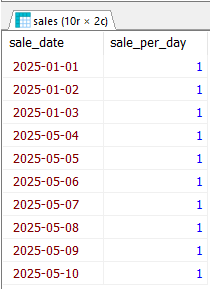
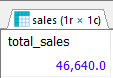
('Pen', 'Stationery', 25.00),

('Notebook', 'Stationery', 60.00),

('Mobile Phone', 'Electronics', 25000.00),

('Headphones', 'Electronics', 3500.00),

('Shoes', 'Apparel', 2000.00);  


1. Retrieve all columns from the Sales table.   
     
   SELECT \* from sales;  
   
2. Retrieve the product\_name and unit\_price from the Products table.  
     
   SELECT product\_name, unit\_price FROM products;  
   
3. Retrieve the sale\_id and sale\_date from the Sales table.   
     
   SELECT sale\_id, sale\_date FROM sales;  
   
4. Filter the Sales table to show only sales with a total\_price greater than $1570.   
     
   SELECT \* from sales WHERE total\_price > 1570;  
   
5. Retrieve the sale\_id and total\_price from the Sales table for sales made on January 3, 2025.  
     
   SELECT sale\_id, total\_price FROM sales WHERE sale\_date='2025-01-03';  
   
6. Calculate the total revenue generated from all sales in the Sales table.  
     
   SELECT SUM(total\_price) AS total\_revenue FROM sales;  
   
7. Calculate the average unit\_price of products in the Products table.   
     
   SELECT SUM(unit\_price)/COUNT(\*) AS average\_unit\_price FROM products;  
   
8. Count Sales Per Day from the Sales table.  
     
   SELECT sale\_date, COUNT(\*) AS sale\_per\_day FROM sales GROUP BY sale\_date;  
   
9. Retrieve the total\_price of all sales, rounding the values to two decimal places.  
     
   SELECT ROUND(SUM(total\_price), 2) AS total\_sales FROM sales;  
   
10. Retrieve the sale\_id and sale\_date from the Sales table, formatting the sale\_date as 'YYYY-MM DD'.   
      
    SELECT sale\_id, DATE\_FORMAT(sale\_date, '%Y-%m-%d') AS formatted\_sale\_date FROM sales;  
    

1. Rank products based on total sales revenue.

SELECT

p.product\_id,

p.product\_name,

SUM(s.total\_price) AS total\_revenue,

RANK() OVER (ORDER BY SUM(s.total\_price) DESC) AS revenue\_rank

FROM

products p

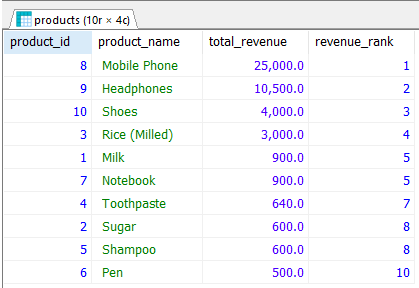
JOIN

sales s ON p.product\_id = s.product\_id

GROUP BY

p.product\_id

ORDER BY

total\_revenue DESC;  


1. Categorize sales as "High", "Medium", or "Low" based on total price (e.g., > $2000 is High, $1100-$1999 is Medium, < $1099 is Low).

SELECT

sale\_id,

sale\_date,

total\_price,

CASE

WHEN total\_price > 2000 THEN 'High'

WHEN total\_price BETWEEN 1100 AND 1999 THEN 'Medium'

ELSE 'Low'

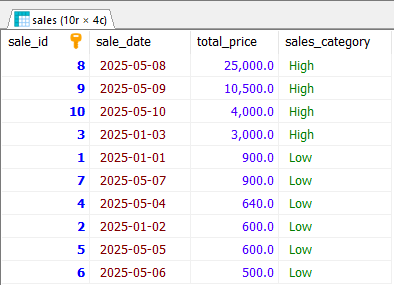
END AS sales\_category

FROM

sales

ORDER BY

total\_price DESC;



1. Retrieve the product details (name, category, unit price) for products that have a quantity sold greater than the average quantity sold across all products.

SELECT

p.product\_name,

p.category,

p.unit\_price

FROM

products p

JOIN

sales s ON p.product\_id = s.product\_id

GROUP BY

p.product\_id

HAVING

SUM(s.quantity\_sold) > (

SELECT AVG(total\_quantity)

FROM (

SELECT SUM(quantity\_sold) AS total\_quantity

FROM sales

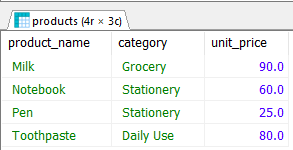
GROUP BY product\_id

) AS avg\_sales

)

ORDER BY

p.product\_name;



1. Add a foreign key constraint to the Sales table that references the product\_id column in the Products table.  
     
   Already done in question 1.
2. Implement a transaction that deducts the quantity sold from the Products table when a sale is made in the Sales table, ensuring that both operations are either committed or rolled back together.

ALTER TABLE products

ADD COLUMN stock\_quantity INT NOT NULL DEFAULT 0;

DROP PROCEDURE IF EXISTS Process\_Sale;

DELIMITER $$

CREATE PROCEDURE Process\_Sale (

IN p\_product\_id BIGINT,

IN p\_sale\_date DATE,

IN p\_quantity\_sold INT

)

BEGIN

DECLARE p\_total\_price DECIMAL(20,2);

DECLARE v\_unit\_price DECIMAL(20,2);

DECLARE v\_stock\_quantity INT;

DECLARE v\_error\_message VARCHAR(255);

START TRANSACTION;

SELECT unit\_price, stock\_quantity INTO v\_unit\_price, v\_stock\_quantity

FROM products

WHERE product\_id = p\_product\_id

FOR UPDATE;

IF v\_stock\_quantity >= p\_quantity\_sold THEN

SET p\_total\_price = v\_unit\_price \* p\_quantity\_sold;

INSERT INTO sales (product\_id, sale\_date, quantity\_sold, total\_price)

VALUES (p\_product\_id, p\_sale\_date, p\_quantity\_sold, p\_total\_price);

UPDATE products

SET stock\_quantity = stock\_quantity - p\_quantity\_sold

WHERE product\_id = p\_product\_id;

COMMIT;

ELSE

SET v\_error\_message = 'Not enough stock available.';

ROLLBACK;

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = v\_error\_message;

END IF;

END$$

DELIMITER ;

CALL Process\_Sale(3, '2025-05-20', 5);



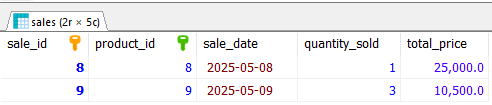
1. Write a query to find all sales where the total price is greater than the average total price of all sales.

SELECT \*

FROM sales

WHERE total\_price > (

SELECT AVG(total\_price)

FROM sales);  


1. Develop a stored procedure named Update\_Unit\_Price that updates the unit price of a product in the Products table based on the provided product\_id.

DROP PROCEDURE IF EXISTS Update\_Unit\_Price;

DELIMITER $$

CREATE PROCEDURE Update\_Unit\_Price (

IN p\_product\_id BIGINT,

IN p\_new\_price DECIMAL(20,2)

)

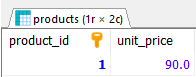
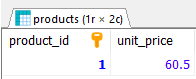
BEGIN

UPDATE products

SET unit\_price = p\_new\_price

WHERE product\_id = p\_product\_id;

END$$

DELIMITER ;  
  
SELECT product\_id,unit\_price FROM products WHERE product\_id=1;  
  
CALL Update\_Unit\_Price(1,60.50);  
SELECT product\_id,unit\_price FROM products WHERE product\_id=1;  


1. Implement a transaction that inserts a new product into the Products table and then adds a corresponding sale record into the Sales table, ensuring that both operations are either fully completed or fully rolled back.

DROP PROCEDURE IF EXISTS Add\_Product\_And\_Sale;

DELIMITER $$

CREATE PROCEDURE Add\_Product\_And\_Sale(

IN p\_product\_name VARCHAR(255),

IN p\_category VARCHAR(255),

IN p\_unit\_price DECIMAL(20,2),

IN p\_stock\_quantity INT,

IN qty\_sold INT

)

BEGIN

DECLARE new\_pid INT;

START TRANSACTION;

INSERT INTO Products (product\_name, category, unit\_price, stock\_quantity)

VALUES (p\_product\_name, p\_category, p\_unit\_price, p\_stock\_quantity);

SET new\_pid = LAST\_INSERT\_ID();

INSERT INTO Sales (product\_id, quantity\_sold, sale\_date, total\_price)

VALUES (new\_pid, qty\_sold, CURDATE(), qty\_sold \* p\_unit\_price);

UPDATE Products

SET stock\_quantity = stock\_quantity - qty\_sold

WHERE product\_id = new\_pid

AND stock\_quantity >= qty\_sold;

IF ROW\_COUNT() = 0 THEN

ROLLBACK;

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Insufficient stock for new product sale';

ELSE

COMMIT;

END IF;

END$$

DELIMITER ;

CALL Add\_Product\_And\_Sale('Potato', 'Grocery','20',10.5,5);  
