# Retail Inventory Database (CS204 Assignment 3)

**--Part 1 #1 Problem:**

--Write an SQL statement that calculates total sales

--by product category using an aggregate function.

-- Logic:

--join orders products and categories to tie each other to a product

--and its respective category and then sum order count per category. The aggregate function is SUM();

```
SELECT
   c.category_name,
   SUM(o.order_amount) AS total_sales
FROM Orders o
        INNER JOIN Products p ON o.product_id = p.product_id
        INNER JOIN Categories c ON p.category_id = c.category_id
GROUP BY c.category_name  --we do group by category name so that the sum
is shown per category
ORDER BY total_sales DESC; -- sort from ascending to descending
```

--Result p1#1:

| | category_name | total_sales |
|---|---|---|
| 1 | Shoes | 90 |
| 2 | Shirts | 60 |
| 3 | Accessories | 50 |
| 4 | Pants | 45 |

**--Part 1 #2**

--Write an SQL statement that formats product names

--and supplier contact information using string functions.

--Logic:

-- we join  products to suppliers by supplier id with UPPER for product names,

-- CONCAT to build the labels and LOWER for email,

-- which results in one row per product with a description and a supplier contact

```
SELECT
   CONCAT(UPPER(p.product_name), ' - supplied by ', s.supplier_name) AS
product_and_supplier,
   CONCAT('Phone: ', s.phone, ' | Email: ', LOWER(s.email)) AS
supplier_contact
FROM Products p
        INNER JOIN Suppliers s ON p.supplier_id = s.supplier_id;
```

-- Result p1#2:

| | product_and_supplier | supplier_contact |
|---|---|---|
| 1 | COTTON T-SHIRT - supplied by Metro Wear | Phone: +19295551201 \| Email: metro@shop.com |
| 2 | BLUE JEANS - supplied by City Apparel | Phone: +19295551202 \| Email: city@shop.com |
| 3 | RUNNING SHOES - supplied by Prime Clothing | Phone: +19295551204 \| Email: prime@shop.com |
| 4 | LEATHER BELT - supplied by Basics Co | Phone: +19295551205 \| Email: basics@shop.com |

## --Part 1 #3

--Write an SQL statement that lists orders by purchase month

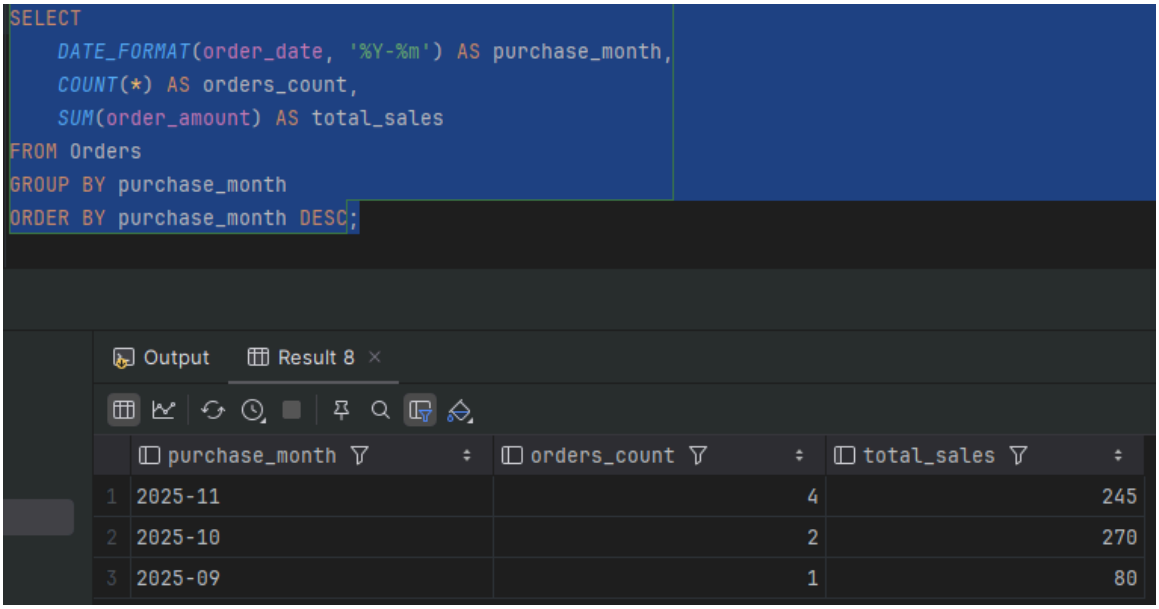
--in descending order using date/time functions.

-- Logic : DATE_FORMAT is used to format a full date to year/month and then form there

-- we count how many orders per month and then sum their amounts ordered from latest month to oldest

```sql
SELECT
   DATE_FORMAT(order_date, '%Y-%m') AS purchase_month,
   COUNT(*) AS orders_count,
   SUM(order_amount) AS total_sales
FROM Orders
GROUP BY purchase_month
ORDER BY purchase_month DESC;
```

-- Result p1#3:



**--Part 1 #4**

--Write an SQL query to calculate the

--20% discounted price for the most expensive product

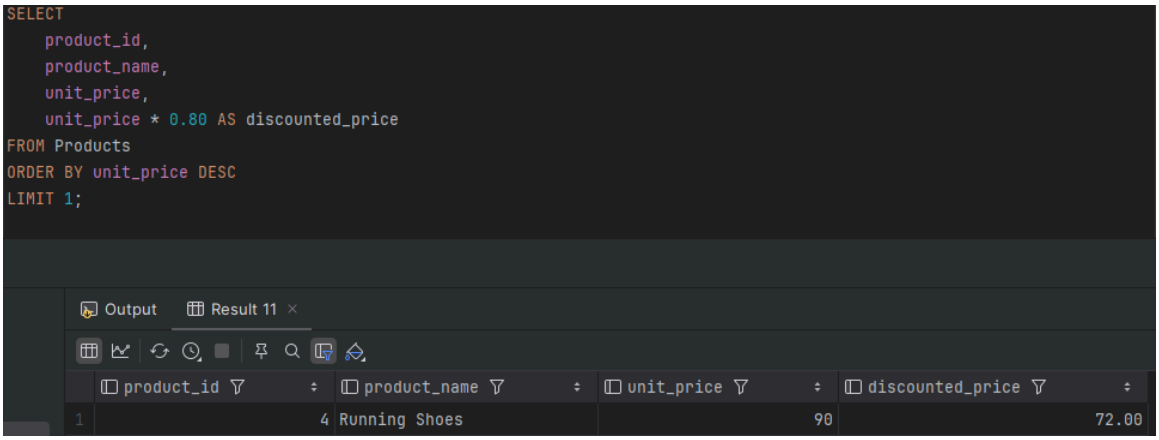
--Logic: We sort by unitprice from Highest to Lowest and the we only show the first result

-- via LIMIT 1 and retrieve the highest product and then we apply a multiplier to the price

-- to add a 20% discount

```sql
SELECT
    product_id,
    product_name,
    unit_price,
    unit_price * 0.80 AS discounted_price
FROM Products
ORDER BY unit_price DESC
LIMIT 1;
```

-- Result p1#4:



**--Part 2 #1**

--Create a view that returns the top 5 best-selling products.

-- Logic : we created a view with a LEFT JOIN of orders so that we can capture all products even the ones with no sales

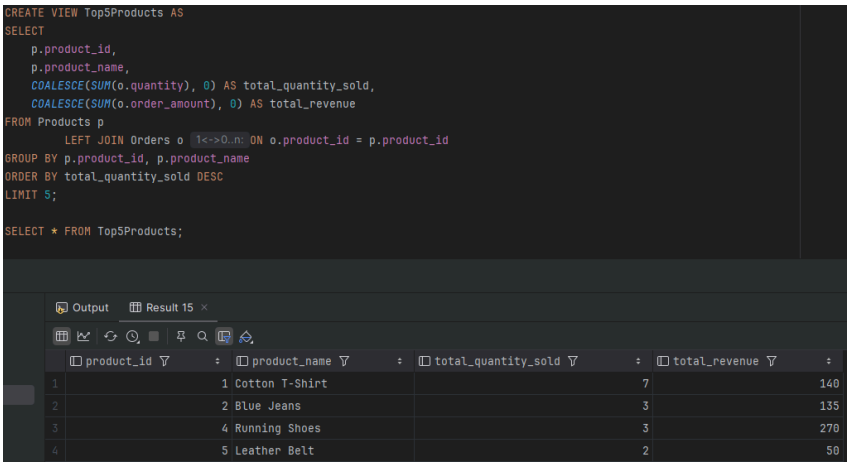
-- we leverage SUM() for the quantity and order_amount per product to record total quantity sold & total_revenue

-- we limit the view to 5 to show the top 5 products, then we use the SELECT * FROM Top5Products to see the results of the view

```sql
CREATE VIEW Top5Products AS
SELECT
    p.product_id,
    p.product_name,
    COALESCE(SUM(o.quantity), 0) AS total_quantity_sold,
    COALESCE(SUM(o.order_amount), 0) AS total_revenue
FROM Products p
        LEFT JOIN Orders o ON o.product_id = p.product_id
GROUP BY p.product_id, p.product_name
ORDER BY total_quantity_sold DESC
LIMIT 5;

SELECT * FROM Top5Products;
```

-- Result p2#1:



**--Part 2 #2**

--Create a stored procedure that accepts a product ID and returns total quantity sold and revenue.

--Then, write a command to execute the stored procedure.

-- Logic : we initialize our procedure with CREATE PROCEDURE and give it a name as GetProductSalesSummary

-- with a required variable int containing the in_product_id , we then use coalesce to safeguard our check for a product with no orders

-- as that scenario would return null, so we return 0 instead. , then besides that logic we just

-- SUM() product order quantity and their respective order_amounts to prepare total_quantity_sold and total_revenue

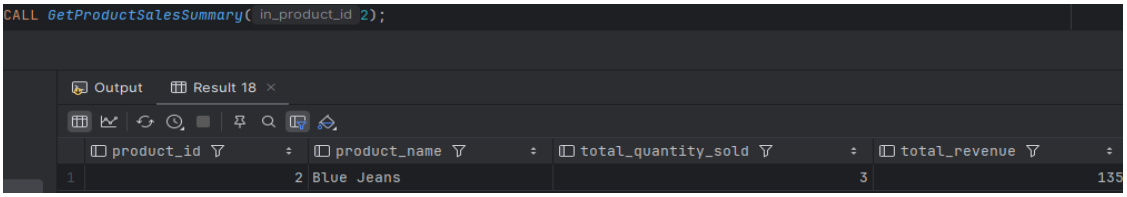
-- we use delimiter to clearly show the start and end of our statement. and finally we call our procedure with CALL GetProductSalesSummary(1);

```
DELIMITER $$

CREATE PROCEDURE GetProductSalesSummary(IN in_product_id INT)
BEGIN
   SELECT
       p.product_id,
       p.product_name,
       COALESCE(SUM(o.quantity), 0) AS total_quantity_sold,
       COALESCE(SUM(o.order_amount), 0) AS total_revenue
   FROM Products p
            LEFT JOIN Orders o ON o.product_id = p.product_id
   WHERE p.product_id = in_product_id
   GROUP BY p.product_id, p.product_name;
END$$
DELIMITER ;

CALL GetProductSalesSummary(1);
```

-- Result p2#2:



## --Part 2 #3

--Part 2 #3 : Create a trigger that logs an entry into an Inventory Audit table every time a product's QOH is updated. Then, add a rule to prevent negative values in the QOH field.

-- Logic: we first create an Inventory_Audit table to log every QOH as that did not exist we then capture the previous QOH and the updated QOH and add a timestamp when the trigger occurs we have a check for if QOH ever becomes negative as that would be an edge case we want covered and our happy path is if the quantity on hand is positive we resume as BAU. results were tested by updating the QOH for a product and inspecting the inventory audit table

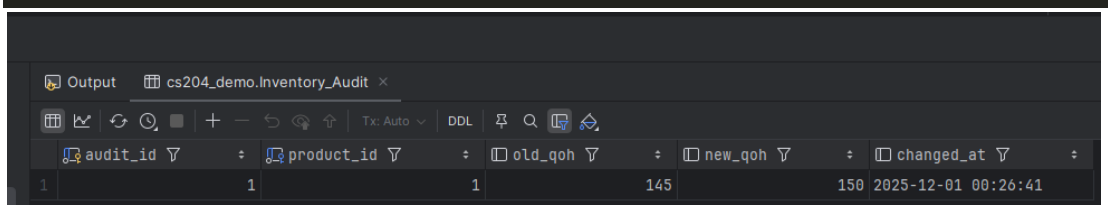
```
CREATE TABLE IF NOT EXISTS Inventory_Audit (
   audit_id INT AUTO_INCREMENT PRIMARY KEY,
   product_id INT NOT NULL,
   old_qoh INT,
   new_qoh INT,
   changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

DROP TRIGGER IF EXISTS trg_products_qoh_audit;
DELIMITER $$

CREATE TRIGGER trg_products_qoh_audit
   BEFORE UPDATE ON Products
   FOR EACH ROW
BEGIN
   IF NEW.qoh < 0 THEN
       SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'QOH cannot be
negative';
   END IF;

   INSERT INTO Inventory_Audit (product_id, old_qoh, new_qoh)
   VALUES (OLD.product_id, OLD.qoh, NEW.qoh);
END$$
DELIMITER ;

ALTER TABLE Products
   ADD CONSTRAINT chk_products_qoh_nonnegative CHECK (qoh >= 0);
```



**--Part 2 #4**

--Create a transaction block that updates inventory and inserts a sales record, rolling back if any part fails.

Logic : we create a function called ProcessSale, in the event of an error we initiate a roll back, we create a new unique order every time ProcessSale is called, we can improve this method by accepting variables so that we do not hardcode it to the same quantity, price, and product each time but this should solve the problem at hand. if there is no error we commit the transaction and do not perform a rollback

```
DROP PROCEDURE IF EXISTS ProcessSale;

DELIMITER $$

CREATE PROCEDURE ProcessSale()
BEGIN

   DECLARE new_order_id INT;

   DECLARE EXIT HANDLER FOR SQLEXCEPTION
       BEGIN
           ROLLBACK;
       END;

   SELECT COALESCE(MAX(order_id), 0) + 1
   INTO new_order_id
   FROM Orders;

   START TRANSACTION;

   UPDATE Products
   SET qoh = qoh - 3
   WHERE product_id = 1;

   INSERT INTO Orders (
       order_id,
       customer_id,
       product_id,
       quantity,
       order_amount,
       order_date,
       order_time
   )
   VALUES (
              new_order_id,
```

```
            10,
            1,
            3,
            3 * 19.99,
            CURRENT_DATE,
            CURRENT_TIME
        );

    COMMIT;
END$$

DELIMITER ;


SELECT QOH FROM Products WHERE product_id=1;

CALL ProcessSale();
```

-- Result p2#4: showing results for before and after process sale:

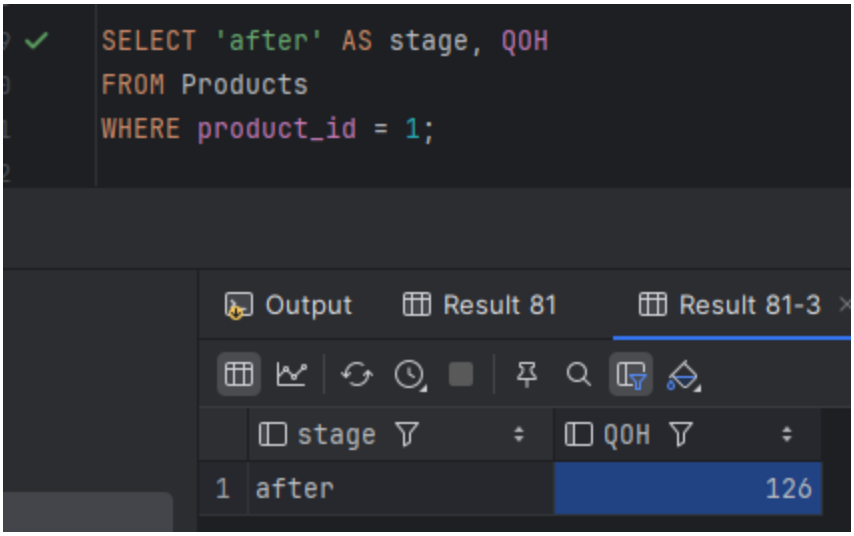
```
SELECT 'before' AS stage, QOH
FROM Products
WHERE product_id = 1;

CALL ProcessSale();

SELECT 'after' AS stage, QOH
FROM Products
WHERE product_id = 1;
```

Output | Result 81 | Result 81-3

| | stage | QOH |
|---|---|---|
| 1 | before | 129 |


SELECT 'after' AS stage, QOH
FROM Products
WHERE product_id = 1;
Output
Result 81
Result 81-3
stage
QOH
1
after
126

**--Part 2 #5**

--Create an index on a frequently queried column and explain why you chose it.

-- Logic : product_id is used in joins and lookups between Orders and Products,so indexing it should help optimize those queries.

```
CREATE INDEX index_orders_product_id ON Orders(product_id);
```