****

**Day 3: Advanced Joins, Nested Queries, and Advanced Subqueries**

**Objective**

Dive deeper into SQL joins, nested queries, and advanced subqueries to enhance your data retrieval and manipulation skills. Understand how to perform complex joins, utilize nested queries to extract data from multiple tables, and leverage advanced subqueries for advanced filtering and data transformation.

**Part 1: Advanced Joins**

In this part, we will explore advanced join techniques to enhance your data retrieval capabilities. We'll dive into self joins and cross joins. Self joins are useful when comparing records within the same table. Cross joins create a Cartesian product of two tables. By understanding these advanced join types, you'll have more flexibility in combining data from multiple tables and solving complex data retrieval problems.

**Self Join**

A self join is a technique used to join a table with itself. It is useful when you need to compare records within the same table.

SELECT

t1.column,

t2.column

FROM table

t1

JOIN table t2

ON t1.key = t2.key;

**Cross Join**

A cross join, also known as a Cartesian join, combines each row from the first table with every row from the second table. It results in a Cartesian product of both tables.

SELECT \*

FROM

table1

CROSS JOIN

table2;

**Part 2: Nested Queries**

A nested select statement, also known as a subquery, is a query nested within another query. It can be used in the SELECT, FROM, WHERE, or HAVING clause to retrieve data based on specific conditions.

SELECT

column1

FROM

table1

WHERE

column2 IN (SELECT column3 FROM table2);

**Part 3: Advanced Subqueries**

In this part, we will uncover the potential of advanced subqueries. We'll explore correlated subqueries, which enable dynamic filtering and calculations by referencing columns from the outer query. Scalar subqueries allow us to retrieve a single value as an expression within the SELECT clause or use it within a WHERE clause. We'll also dive into derived tables, which are subqueries used as virtual tables within the main query, allowing us to perform complex calculations or filtering before joining or further processing the data. Understanding advanced subqueries will enhance your ability to perform sophisticated data analysis and manipulation tasks in SQL.

**Correlated Subquery**

A correlated subquery is a subquery that refers to a column from the outer query. The subquery is evaluated for each row of the outer query, allowing for more dynamic and context-dependent filtering or calculations.

SELECT

column1

FROM

table1 t1

WHERE column2 = (SELECT MAX(column2) FROM table2 WHERE t1.column3 = table2.column3);

**Scalar Subquery**

A scalar subquery is a subquery that returns a single value and can be used as an expression in the SELECT clause or within a WHERE clause.

SELECT

column1,

(SELECT MAX(column2) FROM table2) AS max\_value

FROM

table1;

**Derived Table**

A derived table, also known as an inline view, is a subquery that is used as a virtual table within the main query. It allows complex calculations or filtering to be performed before joining or further processing the data.

SELECT

dt.column1

FROM

(SELECT column1, column2 FROM table1) AS dt

JOIN

table2 ON dt.column2 = table2.column2;

**Case Study: Sales Data Analysis for an E-commerce Platform**

In this case study, we will analyze sales data for an e-commerce platform. The dataset consists of three tables: **orders**, **customers**, and **products**. Our objective is to gain insights into customer purchasing behavior, product performance, and sales trends. We will utilize advanced joins, nested queries, and advanced subqueries to combine data from multiple tables, extract specific information, and perform complex filtering and calculations. By applying these techniques, we will analyze sales performance by customer and product, identify top-selling products within categories, and determine average order value. This case study will provide hands-on experience in analyzing sales data using advanced SQL techniques.

**Questions**

1. Retrieve the customer names and their corresponding orders.
2. Find the total quantity and revenue generated from each product category.
3. Retrieve the top-selling products in each category.
4. Retrieve the average order value for each customer.
5. Retrieve the customers who have made more than the average order quantity.

\*\*Schema (MySQL v8.0)\*\*

CREATE TABLE customers (

customer\_id INT PRIMARY KEY,

customer\_name VARCHAR(50),

email VARCHAR(100),

location VARCHAR(100)

);

CREATE TABLE products (

product\_id INT PRIMARY KEY,

product\_name VARCHAR(100),

category VARCHAR(50),

price DECIMAL(10, 2)

);

CREATE TABLE orders (

order\_id INT PRIMARY KEY,

customer\_id INT,

product\_id INT,

quantity INT,

order\_date DATE,

FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id),

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

);

INSERT INTO customers (customer\_id, customer\_name, email, location)

VALUES

(1, 'John Doe', 'johndoe@example.com', 'New York'),

(2, 'Jane Smith', 'janesmith@example.com', 'Los Angeles'),

(3, 'Mike Johnson', 'mikejohnson@example.com', 'Chicago'),

(4, 'Emily Brown', 'emilybrown@example.com', 'Houston'),

(5, 'David Wilson', 'davidwilson@example.com', 'Miami');

INSERT INTO products (product\_id, product\_name, category, price)

VALUES

(1, 'iPhone 12', 'Electronics', 999.99),

(2, 'Samsung Galaxy S21', 'Electronics', 899.99),

(3, 'Nike Air Max', 'Fashion', 129.99),

(4, 'Sony PlayStation 5', 'Gaming', 499.99),

(5, 'MacBook Pro', 'Electronics', 1499.99);

INSERT INTO orders (order\_id, customer\_id, product\_id, quantity, order\_date)

VALUES

(1, 1, 1, 2, '2023-01-01'),

(2, 2, 3, 1, '2023-01-02'),

(3, 3, 2, 3, '2023-01-03'),

(4, 4, 4, 1, '2023-01-04'),

(5, 5, 1, 1, '2023-01-05'),

(6, 1, 3, 2, '2023-01-06'),

(7, 2, 2, 1, '2023-01-07'),

(8, 3, 5, 1, '2023-01-08'),

(9, 4, 4, 2, '2023-01-09'),

(10, 5, 1, 3, '2023-01-10');

---

\*\*Query #1\*\*

SELECT c.customer\_name,p.product\_name, o.quantity

FROM products p

JOIN orders o

ON p.product\_id = o.product\_id

JOIN customers c

ON c.customer\_id = o.customer\_id;

| customer\_name | product\_name | quantity |

| ------------- | ------------------ | -------- |

| John Doe | iPhone 12 | 2 |

| Jane Smith | Nike Air Max | 1 |

| Mike Johnson | Samsung Galaxy S21 | 3 |

| Emily Brown | Sony PlayStation 5 | 1 |

| David Wilson | iPhone 12 | 1 |

| John Doe | Nike Air Max | 2 |

| Jane Smith | Samsung Galaxy S21 | 1 |

| Mike Johnson | MacBook Pro | 1 |

| Emily Brown | Sony PlayStation 5 | 2 |

| David Wilson | iPhone 12 | 3 |

---

\*\*Query #2\*\*

SELECT p.category,

SUM(o.product\_id) AS total\_quantity,

SUM(price\*quantity) AS total\_revenue

FROM products p

JOIN orders o

ON p.product\_id = o.product\_id

GROUP BY p.category;

| category | total\_quantity | total\_revenue |

| ----------- | -------------- | ------------- |

| Electronics | 12 | 11099.89 |

| Fashion | 6 | 389.97 |

| Gaming | 8 | 1499.97 |

---

\*\*Query #3\*\*

WITH ProductTotalQuantity AS (

SELECT p.category, p.product\_name, SUM(o.quantity) AS total\_quantity

FROM products p

JOIN orders o ON p.product\_id = o.product\_id

GROUP BY p.category, p.product\_name

)

SELECT category, product\_name, total\_quantity

FROM ProductTotalQuantity

WHERE (category, total\_quantity) IN (

SELECT category, MAX(total\_quantity)

FROM ProductTotalQuantity

GROUP BY category

);

| category | product\_name | total\_quantity |

| ----------- | ------------------ | -------------- |

| Electronics | iPhone 12 | 6 |

| Fashion | Nike Air Max | 3 |

| Gaming | Sony PlayStation 5 | 3 |

---

\*\*Query #4\*\*

SELECT c.customer\_id,

c.customer\_name,

ROUND(avg(price\*quantity),2) as average\_order\_value

FROM products p

JOIN orders o

ON p.product\_id = o.product\_id

JOIN customers c

ON c.customer\_id = o.customer\_id

GROUP BY c.customer\_id, c.customer\_name

ORDER BY average\_order\_value DESC;

| customer\_id | customer\_name | average\_order\_value |

| ----------- | ------------- | ------------------- |

| 3 | Mike Johnson | 2099.98 |

| 5 | David Wilson | 1999.98 |

| 1 | John Doe | 1129.98 |

| 4 | Emily Brown | 749.99 |

| 2 | Jane Smith | 514.99 |

---

\*\*Query #5\*\*

SELECT c.customer\_id,

c.customer\_name

FROM customers c

JOIN

(SELECT customer\_id, AVG(quantity) AS avg\_quantity

FROM orders

GROUP BY customer\_id) as avg\_orders

ON c.customer\_id = avg\_orders.customer\_id

JOIN orders o ON c.customer\_id = o.customer\_id

WHERE o.quantity > avg\_quantity;

| customer\_id | customer\_name |

| ----------- | ------------- |

| 3 | Mike Johnson |

| 4 | Emily Brown |

| 5 | David Wilson |

---

[View on DB Fiddle](https://www.db-fiddle.com/f/xtvT8co9LVD2K41UcpYcaE/9)