LESSON 5

SUBQUERIES IN SELECT CLAUSE

Ihor Liutak

Content

1. Introduction to Subqueries
2. Why Use Subqueries?
3. Types of Subqueries
4. Best Practices and Common Pitfalls

Short description

Introduce the concept of subqueries and explain their use in sql. Each how to use subqueries in the select clause to fetch additional data or perform calculations.

Kurzbeschreibung

Stellen Sie das Konzept von Unterabfragen vor und erklären Sie deren Verwendung in SQL. Erfahren Sie, wie Sie Unterabfragen in der Select-Klausel verwenden, um zusätzliche Daten abzurufen oder Berechnungen durchzuführen.

Introduction to Subqueries

Definition:

A subquery is a query nested inside another query

Primary Purpose:

Dynamically compute values

Fetch data that depends on results from another query

Why Use Subqueries

Advantages:

Simplify complex queries by breaking them into smaller parts

Make queries more readable, in certain scenarios, than extensive JOINs

Typical Scenarios:

Filtering data based on aggregated results
(e.g., salaries above the average salary)

Calculating values for display without writing multiple queries

Types of Subqueries

Scalar Subqueries: Return a single value (e.g., average salary)

```
SELECT employee_id,
       (SELECT AVG(salary) FROM employees) AS average_salary
FROM employees;
```

Multi-Row Subqueries:

(often used with IN or EXISTS - though those are typically in

WHERE clauses; still good to contrast)

Non-Correlated: Can run independently of the main query.

Correlated: Depend on the main query's row-by-row context

Subqueries in the SELECT Clause

Syntax:

```
SELECT column1,
       (SELECT some_aggregate FROM table2 WHERE condition) AS alias
FROM table1;
```

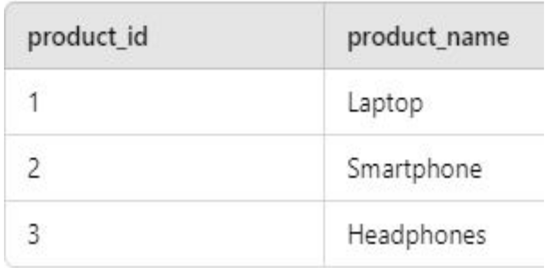
Example:

```
    SELECT p.product_name,
       (SELECT SUM(s.quantity * s.price)
        FROM sales s
        WHERE s.product_id = p.product_id) AS total_revenue
FROM products p;
```

Table Structures

Table **products**:

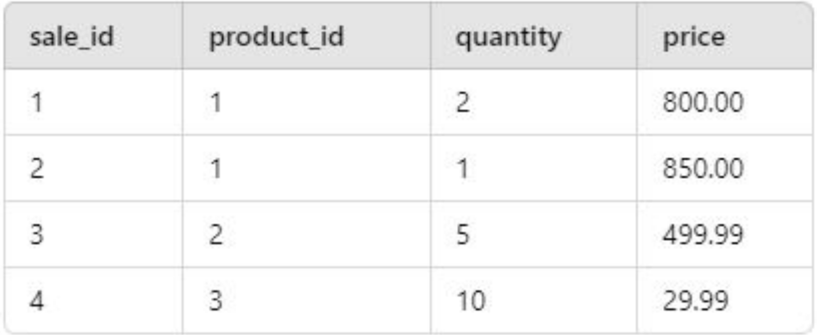
```
CREATE TABLE products (
    product_id INT AUTO_INCREMENT PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL
);
```



product_id	product_name
1	Laptop
2	Smartphone
3	Headphones

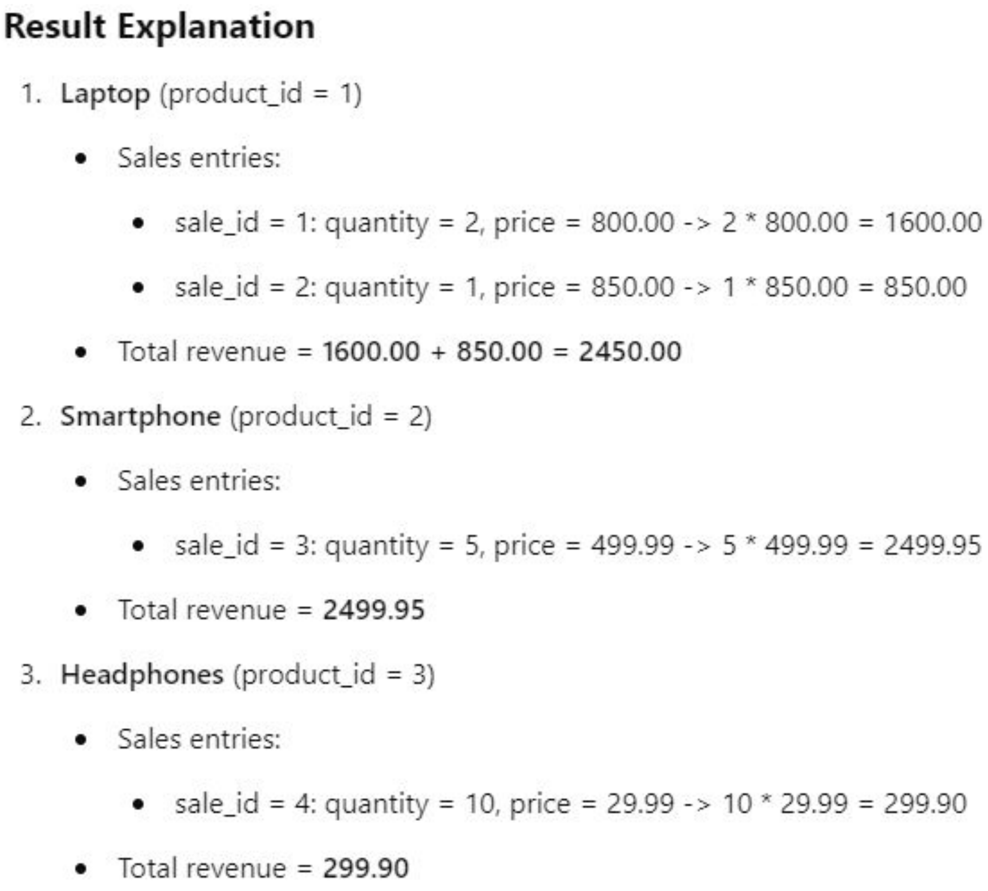
Table **sales**:

```
CREATE TABLE sales (
    sale_id INT AUTO_INCREMENT PRIMARY KEY,
    product_id INT NOT NULL,
    quantity INT NOT NULL,
    price DECIMAL(10,2) NOT NULL,
    FOREIGN KEY (product_id)
REFERENCES products (product_id)
);
```



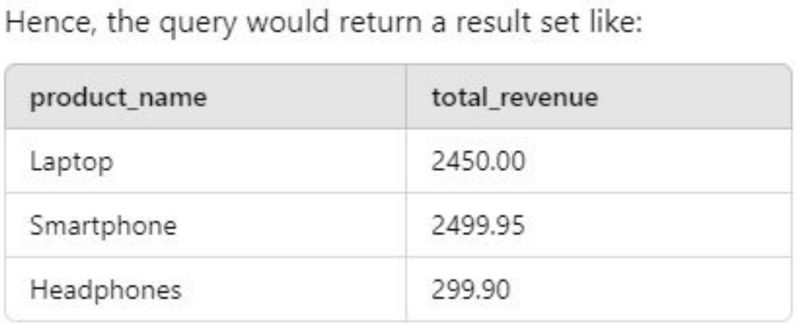
sale_id	product_id	quantity	price
1	1	2	800.00
2	1	1	850.00
3	2	5	499.99
4	3	10	29.99

Result Explanation



Result Explanation

1. **Laptop** (product_id = 1)
   - Sales entries:
     - sale_id = 1: quantity = 2, price = 800.00 -> 2 * 800.00 = 1600.00
     - sale_id = 2: quantity = 1, price = 850.00 -> 1 * 850.00 = 850.00
   - Total revenue = **1600.00 + 850.00 = 2450.00**
2. **Smartphone** (product_id = 2)
   - Sales entries:
     - sale_id = 3: quantity = 5, price = 499.99 -> 5 * 499.99 = 2499.95
   - Total revenue = **2499.95**
3. **Headphones** (product_id = 3)
   - Sales entries:
     - sale_id = 4: quantity = 10, price = 29.99 -> 10 * 29.99 = 299.90
   - Total revenue = **299.90**



Hence, the query would return a result set like:

product_name	total_revenue
Laptop	2450.00
Smartphone	2499.95
Headphones	299.90

Best Practices and Common Pitfalls

Best Practices:

Use descriptive aliases for subqueries

Keep subqueries simple and ensure they return only one value if placed in the SELECT list as a scalar subquery

Common Pitfalls:

Returning multiple rows in a scalar subquery, causing errors (ERROR 1242 (21000): Subquery returns more than 1 row).

Performance impact when subqueries are used in place of more efficient JOINs

Unclear logic if nesting becomes too deep