

Hi! Are u familair with the classic educational customer products orders order details SQL educatioal entry level dataset?



Yes, I am familiar with the classic educational "classicmodels" database. It's a widely used sample database for learning and practicing SQL. This database represents the business operations of a fictional retailer of scale model cars. It includes several interconnected tables that are essential for understanding fundamental database concepts.

Key Tables in the Classicmodels Database

- **Customers:** This table stores information about the customers, such as their names, contact information, and credit limits.
- **Products:** Here, you'll find details about the products sold, including the product name, product line, description, and buy price.
- **ProductLines:** This table categorizes the products into different lines, like "Classic Cars" or "Motorcycles."
- **Orders:** This contains information about customer orders, including the order date, required date, and status.
- **OrderDetails:** This table links the `Orders` and `Products` tables and specifies the quantity and price for each product within an order.
- **Employees:** This table holds information about the company's employees, including their roles and reporting structure.
- **Offices:** This contains data about the company's sales offices.
- **Payments:** This table records the payments made by customers.

Common Educational Uses

The "classicmodels" database is a popular choice for teaching and learning SQL because its structure allows for a wide range of queries, from simple single-table lookups to complex multi-table joins. It's an excellent tool for practicing concepts such

as:

- **Basic** `SELECT` **statements**
- **Filtering data with** `WHERE` **clauses**
- **Sorting results with** `ORDER BY`
- **Joining multiple tables (** `INNER JOIN` **,** `LEFT JOIN` **,** **etc.)**
- **Aggregate functions (** `COUNT` **,** `SUM` **,** `AVG` **,** **etc.)**
- **Grouping data with** `GROUP BY`
- **Subqueries**

Its clear and intuitive schema makes it an ideal starting point for anyone new to SQL and relational databases.

```
SELECT order_details.order_id, SUM(products.price)
FROM order_details
LEFT JOIN products ON order_details.product_id =
```



The correct implementation is the **fifth query**:

SQL



```
SELECT order_details.order_id,
sum(products.price*order_details.quantity) as total_price
FROM order_details
LEFT JOIN products
ON order_details.product_id = products.product_id
group by order_id
having sum(products.price*order_details.quantity) > 400;
```

This query is correct because it properly calculates the total price for each order by

order_details.quantity	sums	products.price * order_d
		GROUP BY order_id
VING		HA

What the Other Queries Are Lacking

Here's a breakdown of what each of the other queries does and what it's missing.

Query 1

SQL



```
SELECT order_details.order_id, SUM(products.price)
FROM order_details
LEFT JOIN products ON order_details.product_id = products.product_id
GROUP BY order_id
HAVING SUM(products.price) > 400.00;
```

- **Lacking:** It fails to multiply the product's price by the **quantity** ordered. This query incorrectly sums the unit prices of the products in an order, rather than the actual total cost.

Query 2

SQL



```
SELECT order_details.product_id, SUM(products.price)
FROM order_details
LEFT JOIN products ON order_details.product_id = products.product_id
GROUP BY order_details.product_id;
```

- ```
GROUP BY order_details.product_id
```

  
condition ( 

```
HAVING ... > 400
```

 ).

## Query 3

SQL



```
SELECT order_details.order_id, SUM(products.price)
FROM order_details
LEFT JOIN products ON order_details.product_id = products.product_id
GROUP BY order_id;
```

- **Lacking:** This is similar to the first query but is missing the final filtering step. It

```
HAVING
```

## Query 4

SQL



```
SELECT order_details.order_id,
products.price*order_details.quantity as total_price
FROM order_details
```

```
LEFT JOIN products
ON order_details.product_id = products.product_id;
```

- **Lacking:** This query correctly calculates the total price for each **individual line item** but fails to **aggregate** ( `SUM` and `GROUP BY` ) them to get the total for the entire order. It also doesn't filter the results.

```
SELECT customers.customer_name, products.price
FROM order_details
LEFT JOIN products ON order_details.product_id =
```



No, that query **does not** correctly return how much each customer spent on each order.

The query you provided will return a long list showing the customer's name next to the unit price of every single item they have ever purchased, across all their orders.

Here is a breakdown of what's wrong with the original query and the correct way to write it.

### What's Wrong with Your Query

1. **No Calculation for Quantity:** It only selects `products.price` , but it ignores the `order_details.quantity` . The total for a line item is `price * quantity` .
2. **No Aggregation (SUM):** It doesn't sum up the totals of all the items within a single order to get the order's grand total.
3. **No Grouping (GROUP BY):** Without `GROUP BY` , you get one row for every product in an order instead of one row for the entire order.

4. 

```
order_i
d
```

### The Correct Implementation

To get the total amount spent by each customer on each of their orders, you need to calculate the cost of each line item, and then group by both the customer and the specific order.

Here is the correct query:



```
SELECT
 customers.customer_name,
 orders.order_id,
 SUM(products.price * order_details.quantity) AS total_order_price
FROM customers
JOIN orders
 ON customers.customer_id = orders.customer_id
JOIN order_details
 ON orders.order_id = order_details.order_id
JOIN products
 ON order_details.product_id = products.product_id
GROUP BY
 customers.customer_name,
 orders.order_id
ORDER BY
 customers.customer_name,
 total_order_price DESC;
```

### Why This Correct Query Works

- `SUM(products.price * order_details.quantity)` : This correctly calculates the total cost for each line item ( `price * quantity` ) and then adds them all up ( `SUM` ) for each group.
- `GROUP BY customers.customer_name, orders.order_id` : This is the crucial step. It tells the database to perform the `SUM()` calculation for each unique combination of a customer and their order ID, effectively creating a total for each individual order.
- `ORDER BY` : This is an optional but helpful clause that organizes the results, making them easier to read.

#### Continue with Google AI Pro

You've reached your limit on 2.5 Pro until Jun 19, 4:09 a.m.. Try Google AI Pro for higher limits.