

# Coding challenge

## SQL

**Name:** Rohan Vinayak Chaudhari

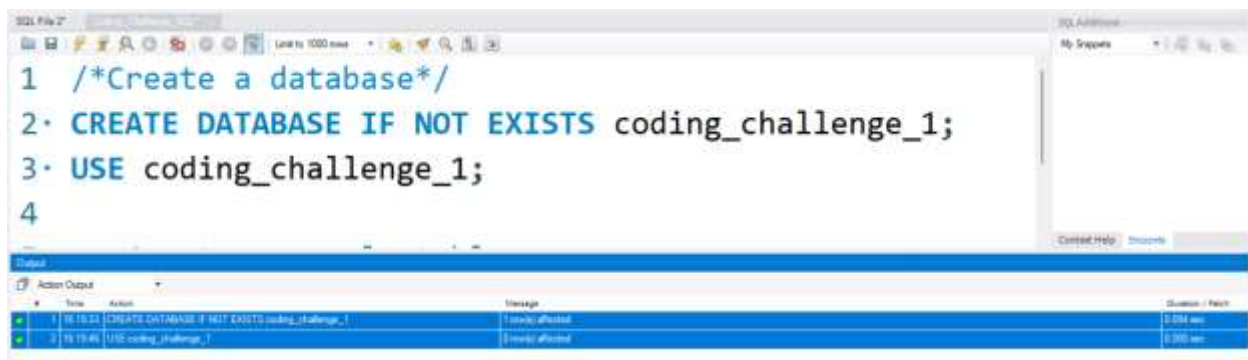
**Batch:** Data Engineering 1

**Question1:** Execute OVER and PARTITION BY Clause in SQL Queries , creating subtotals &Total Aggregations using SQL Queries.

1)Create database & using it:

Creates a database named coding\_challenge\_1.

Switches to using the newly created database.

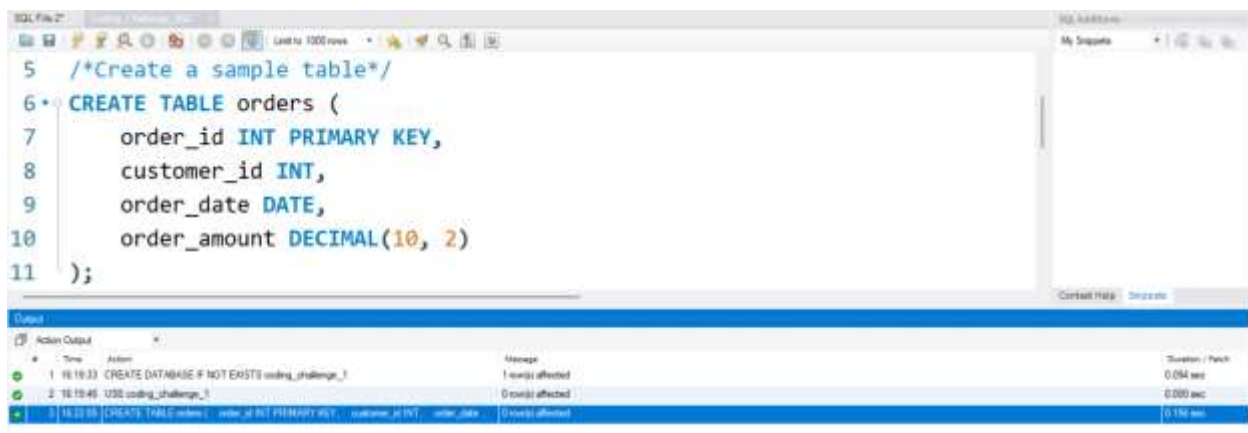


```
1 /*Create a database*/
2 CREATE DATABASE IF NOT EXISTS coding_challenge_1;
3 USE coding_challenge_1;
4
```

Time	Action	Message	Duration / Patch
16:16:33	CREATE DATABASE IF NOT EXISTS coding_challenge_1	1 row(s) affected	0.094 sec
16:16:46	USE coding_challenge_1	0 row(s) affected	0.000 sec

2)Creating table Order:

Creates a table named orders with columns order\_id, customer\_id, order\_date, and order\_amount.

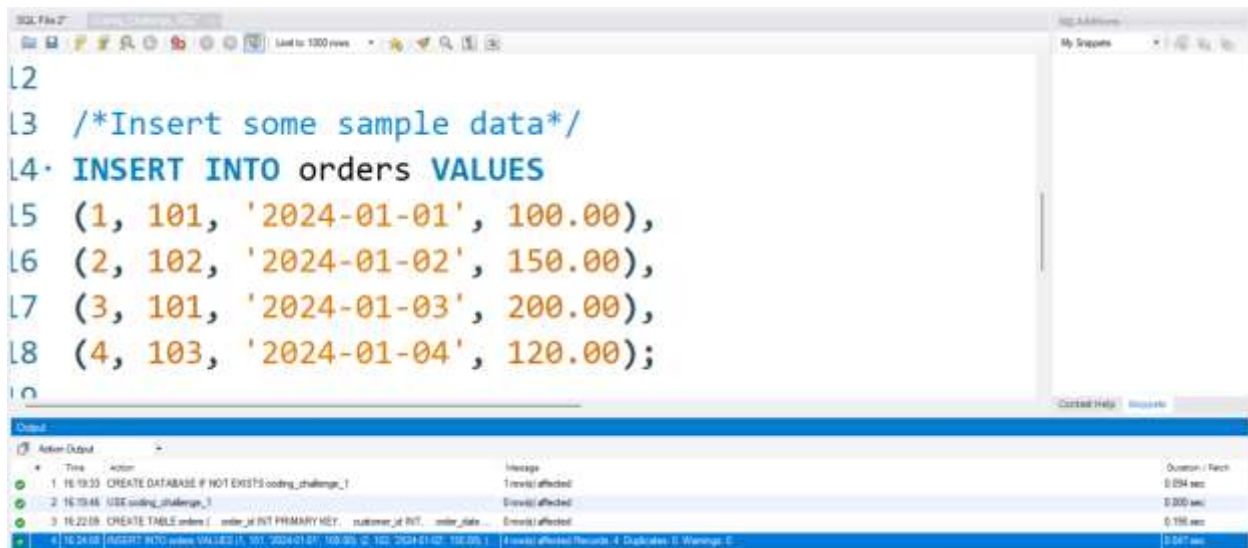


```
5 /*Create a sample table*/
6 CREATE TABLE orders (
7     order_id INT PRIMARY KEY,
8     customer_id INT,
9     order_date DATE,
10    order_amount DECIMAL(10, 2)
11 );
```

Time	Action	Message	Duration / Patch
16:16:33	CREATE DATABASE IF NOT EXISTS coding_challenge_1	1 row(s) affected	0.094 sec
16:16:46	USE coding_challenge_1	0 row(s) affected	0.000 sec
16:22:55	CREATE TABLE orders ( order_id INT PRIMARY KEY, customer_id INT, order_date DATE, order_amount DECIMAL(10, 2) )	0 row(s) affected	0.150 sec

### 3) Inserting data into order table

Inserts sample data into the orders table.



```
12
13 /*Insert some sample data*/
14 INSERT INTO orders VALUES
15 (1, 101, '2024-01-01', 100.00),
16 (2, 102, '2024-01-02', 150.00),
17 (3, 101, '2024-01-03', 200.00),
18 (4, 103, '2024-01-04', 120.00);
19
```

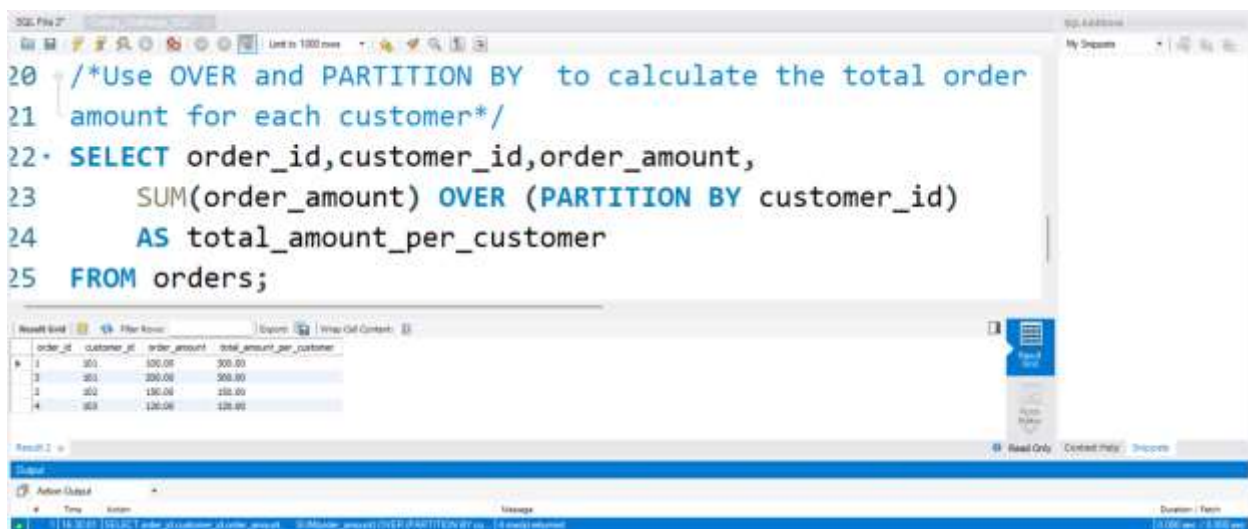
Output

Time	Action	Message	Duration / Path
1 16:19:33	CREATE DATABASE IF NOT EXISTS coding_challenge_1	1 row(s) affected	0.034 sec
2 16:19:46	USE coding_challenge_1	0 row(s) affected	0.000 sec
3 16:22:08	CREATE TABLE orders ( order_id INT PRIMARY KEY, customer_id INT, order_date DATE, order_amount DECIMAL(10,2))	0 row(s) affected	0.198 sec
4 16:24:08	INSERT INTO orders VALUES (1, 101, '2024-01-01', 100.00), (2, 102, '2024-01-02', 150.00), (3, 101, '2024-01-03', 200.00), (4, 103, '2024-01-04', 120.00);	4 row(s) affected Rows: 4 Duplicates: 0 Warnings: 0	0.547 sec

### 4) Execute OVER and PARTITION BY Clause in SQL Queries

Use of the OVER and PARTITION BY clauses with the SUM aggregation(window)) function to calculate the total order amount for each customer.

The query calculates the total order amount for each customer using the SUM window function with the OVER and PARTITION BY clauses.



```
20 /*Use OVER and PARTITION BY to calculate the total order
21 amount for each customer*/
22 SELECT order_id, customer_id, order_amount,
23        SUM(order_amount) OVER (PARTITION BY customer_id)
24        AS total_amount_per_customer
25 FROM orders;
```

Result Set

order_id	customer_id	order_amount	total_amount_per_customer
1	101	100.00	300.00
2	102	150.00	300.00
3	101	200.00	300.00
4	103	120.00	120.00

Output

Time	Action	Message	Duration / Path
1 16:30:01	SELECT order_id, customer_id, order_amount, SUM(order_amount) OVER (PARTITION BY customer_id) AS total_amount_per_customer FROM orders;	4 row(s) returned	0.000 sec / 0.000 sec

### 5) creating subtotals & Total Aggregations using SQL Queries.

This query is useful for obtaining a summary of total order amounts, total orders, and average order amounts, min, max amount for each customer, along with subtotals and a grand total.

**GROUP BY:** A clause used in SQL to arrange identical data into groups. In this query, it groups rows based on the `customer_id` column.

**SUM():** An aggregate function that adds up the values in a specified column. Here, it calculates the total order amount for each customer.

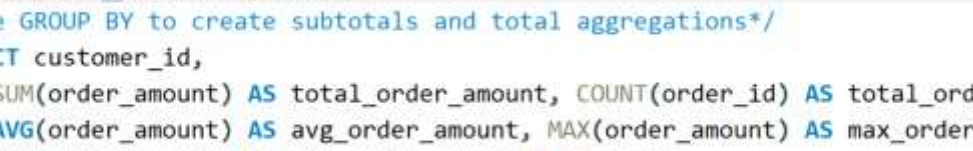
**COUNT():** An aggregate function that counts the number of rows in a result set. In this query, it counts the total number of orders for each customer.

**AVG():** An aggregate function that calculates the average of values in a specified column. Here, it computes the average order amount for each customer.

**MAX():** An aggregate function that calculates the MAX of order value for a particular customer id. Here, it computes the average order amount for each customer.

**MIN():** An aggregate function that calculates the MIN of order value for a particular customer id. Here, it computes the average order amount for each customer.

**WITH ROLLUP:** An extension to the GROUP BY clause that includes extra rows that represent subtotals and a grand total. The subtotals are generated for each level of grouping specified in the GROUP BY clause.



The screenshot shows a SQL IDE with a query editor and a results pane. The query in the editor is:

```

28 /*Use GROUP BY to create subtotals and total aggregations*/
29 SELECT customer_id,
30        SUM(order_amount) AS total_order_amount, COUNT(order_id) AS total_orders,
31        AVG(order_amount) AS avg_order_amount, MAX(order_amount) AS max_order_amount,
32        MIN(order_amount) AS min_order_amount FROM orders
33 GROUP BY customer_id WITH ROLLUP ORDER BY min_order_amount desc;

```

The results pane displays the following data:

customer_id	total_order_amount	total_orders	avg_order_amount	max_order_amount	min_order_amount
101	100.00	1	100.000000	100.00	100.00
102	100.00	1	100.000000	100.00	100.00
103	300.00	3	100.000000	300.00	100.00
1000	100.00	4	25.000000	100.00	100.00

