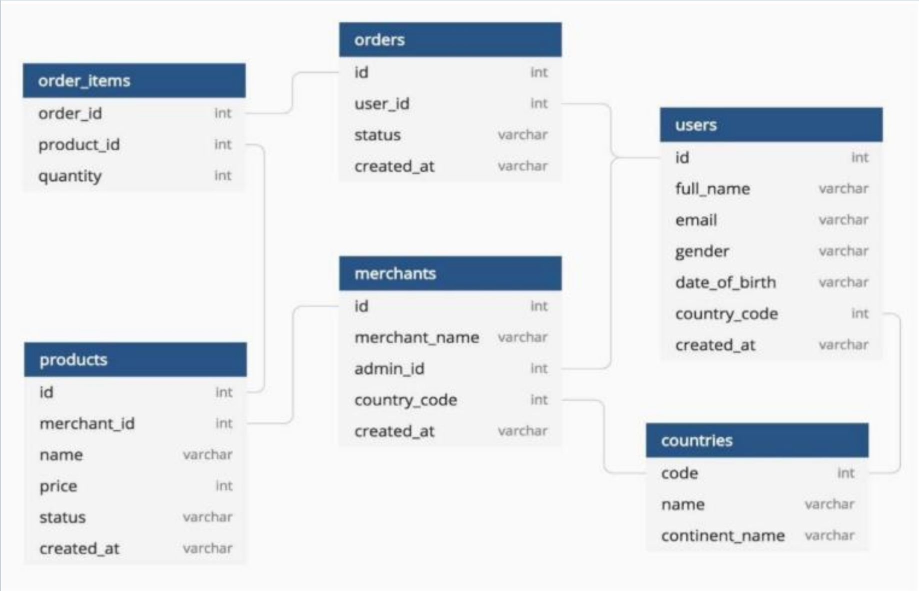
SQL Project"

ISHRAT AMIN
DATA & BUSINESS ANALYTICS
MAY 30, 2025

Database Schema



orders	
id	int
user_id	int
status	varchar
created_at	varchar

order_items	
order_id	int
product_id	int
quantity	int

users	
id	int
full_name	varchar
email	varchar
gender	varchar
date_of_birth	varchar
country_code	int
created_at	varchar

merchants	
id	int
merchant_name	varchar
admin_id	int
country_code	int
created_at	varchar

products	
id	int
merchant_id	int
name	varchar
price	int
status	varchar
created_at	varchar

countries	
code	int
name	varchar
continent_name	varchar

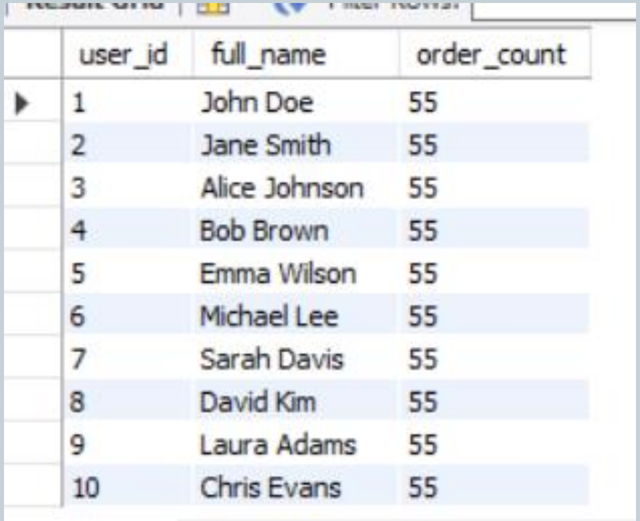
This schema defines six interconnected tables forming the eCommerceDB:

- Users: Customer data with country codes.
- Orders: Linked to users, capturing transactions.
- Order_Items: Lists products per order.
- Products: Connected to merchants.
- Merchants: Linked to users and countries.
- Countries: Holds valid country codes.

Task- 3 : Performing Queries
Query 1: Display the Total Number of Orders Made by Each User

SQL Query Code:

```
SELECT u.user_id, u.full_name,
COUNT(o.order_id) AS
order_count
FROM users u
LEFT JOIN orders o ON u.user_id
= o.user_id
GROUP BY u.user_id,
u.full_name;
```



user_id	full_name	order_count
1	John Doe	55
2	Jane Smith	55
3	Alice Johnson	55
4	Bob Brown	55
5	Emma Wilson	55
6	Michael Lee	55
7	Sarah Davis	55
8	David Kim	55
9	Laura Adams	55
10	Chris Evans	55

The query output shows 10 users, each with their user_id, full_name, and an order_count of 55. This indicates that every user listed (from John Doe to Chris Evans) has placed exactly 55 orders, as counted from the orders table. The LEFT JOIN in the query ensured all users from the users table are included, and the identical order_count suggests either uniform order activity across all users or a potential issue in the data or query logic, as it's unusual for every user to have the exact same number of orders.

Query 2: Display the Names of Products that Have Been Ordered at Least Once

SQL Query Code:

```
SELECT DISTINCT
p.product_name
FROM products p
JOIN order_items oi ON
p.product_id = oi.product_id;
```

product_name	
Laptop	Hat
Smartphone	Scarf
Headphones	Dress
Tablet	Belt
Smartwatch	Sunglasses
Charger	Shirt
Mouse	Drone
Keyboard	Camera
Monitor	Tripod
Speaker	Lens
T-Shirt	Battery Pack
Jeans	VR Headset
Jacket	Game Controller
Sneakers	Smart Light
Earphones	Router

The query outputs a list of unique product names from the products table that appear in at least one order in the order_items table. The INNER JOIN ensures only products that have been ordered are included, and DISTINCT removes any duplicate product names from the result.

Query 2: Display the Names of Products that Have Been Ordered at Least Once

SQL Query Code:

```
SELECT DISTINCT
p.product_name
FROM products p
JOIN order_items oi ON
p.product_id = oi.product_id;
```

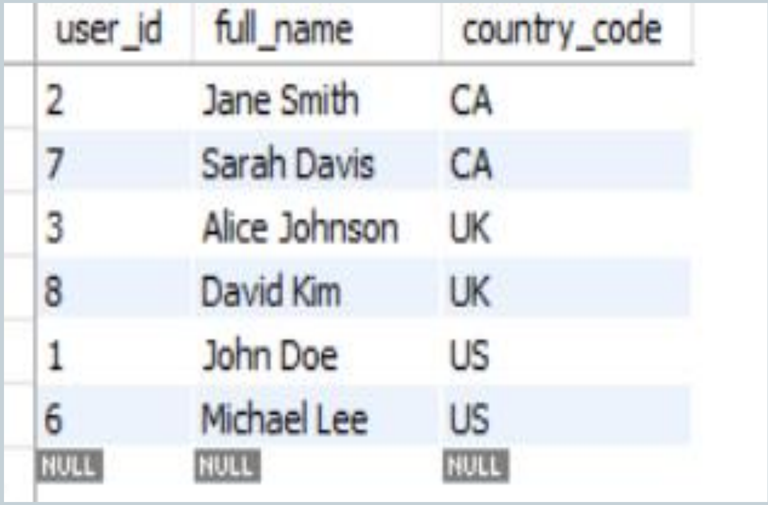
product_name	
Laptop	Hat
Smartphone	Scarf
Headphones	Dress
Tablet	Belt
Smartwatch	Sunglasses
Charger	Shirt
Mouse	Drone
Keyboard	Camera
Monitor	Tripod
Speaker	Lens
T-Shirt	Battery Pack
Jeans	VR Headset
Jacket	Game Controller
Sneakers	Smart Light
Earphones	Router

The query outputs a list of unique product names from the products table that appear in at least one order in the order_items table. The INNER JOIN ensures only products that have been ordered are included, and DISTINCT removes any duplicate product names from the result.

Query 3: :Retrieve the Details of All Users Who Are from the Same Country As the Merchant

SQL Query Code:

```
SELECT u.user_id, u.full_name,
u.country_code
FROM users u
WHERE u.country_code IN
(SELECT country_code FROM
merchants);
```



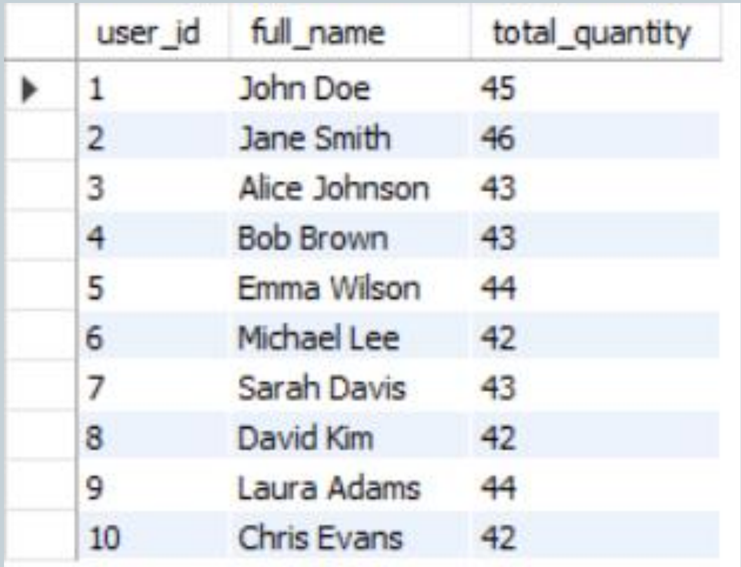
user_id	full_name	country_code
2	Jane Smith	CA
7	Sarah Davis	CA
3	Alice Johnson	UK
8	David Kim	UK
1	John Doe	US
6	Michael Lee	US
NULL	NULL	NULL

The output shows a table with user_id, full_name, and country_code for users from the users table whose country_code matches those in the merchants table. It includes users from CA (Canada) with IDs 2 and 7 (Jane Smith and Sarah Davis), UK (United Kingdom) with IDs 3 and 8 (Alice Johnson and David Kim), and US (United States) with IDs 1 and 6 (John Doe and Michael Lee). The last row with NULL for full_name and country_code indicates a user (ID 10) with missing data, likely included due to the query's structure but not fully matching merchant country criteria.

Task- 4: Aggregate Functions
Query 1: Total Quantity of Products Ordered by Each User

SQL Query Code:

```
SELECT u.user_id, u.full_name,
SUM(oi.quantity) AS total_quantity
FROM users u
JOIN orders o ON u.user_id =
o.user_id
JOIN order_items oi ON o.order_id =
oi.order_id
GROUP BY u.user_id, u.full_name;
```



user_id	full_name	total_quantity
1	John Doe	45
2	Jane Smith	46
3	Alice Johnson	43
4	Bob Brown	43
5	Emma Wilson	44
6	Michael Lee	42
7	Sarah Davis	43
8	David Kim	42
9	Laura Adams	44
10	Chris Evans	42

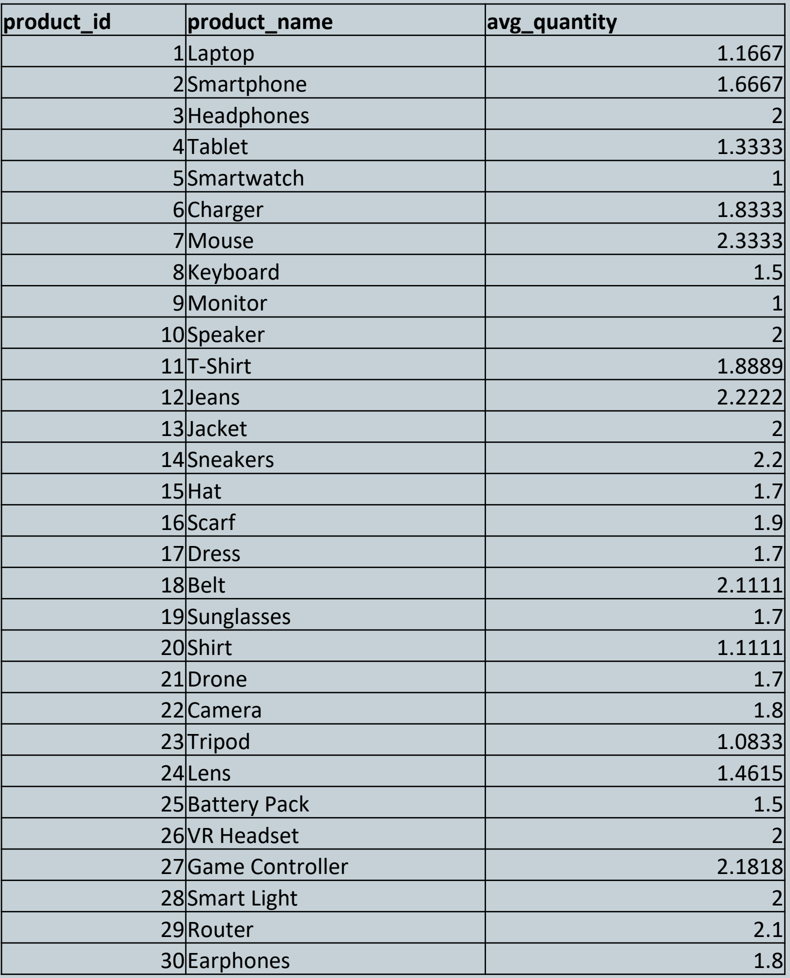
The output lists each user (with their user_id and full_name) and the total quantity of items they've ordered across all their orders. Only users who have placed at least one order are included due to the INNER JOINs. For example, if Jane Smith ordered 5 items in one order and 3 in another, her total_quantity would be 8.

Query 2: Average Order Quantity for Each Product

SQL Query Code:

```
SELECT p.product_id,
p.product_name, AVG(oi.quantity)
AS avg_quantity
FROM products p
JOIN order_items oi ON
p.product_id = oi.product_id
GROUP BY p.product_id,
p.product_name;
```

The output lists each product (by product_name) and the average quantity ordered per order for that product. For example, if "Laptop" was ordered 3 times with quantities 2, 4, and 6, the avg_quantity for "Laptop" would be (2 + 4 + 6) / 3 = 4. Only products that have been ordered at least once are included due to the INNER JOIN.

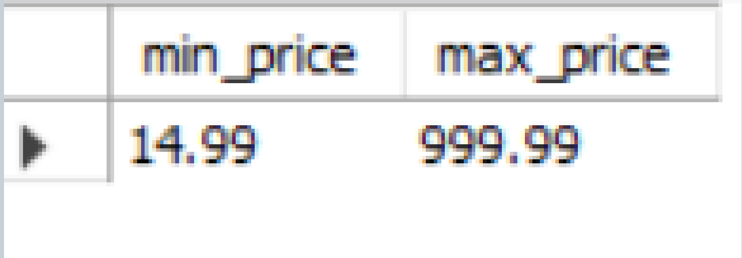


product_id	product_name	avg_quantity
1	Laptop	1.1667
2	Smartphone	1.6667
3	Headphones	2
4	Tablet	1.3333
5	Smartwatch	1
6	Charger	1.8333
7	Mouse	2.3333
8	Keyboard	1.5
9	Monitor	1
10	Speaker	2
11	T-Shirt	1.8889
12	Jeans	2.2222
13	Jacket	2
14	Sneakers	2.2
15	Hat	1.7
16	Scarf	1.9
17	Dress	1.7
18	Belt	2.1111
19	Sunglasses	1.7
20	Shirt	1.1111
21	Drone	1.7
22	Camera	1.8
23	Tripod	1.0833
24	Lens	1.4615
25	Battery Pack	1.5
26	VR Headset	2
27	Game Controller	2.1818
28	Smart Light	2
29	Router	2.1
30	Earphones	1.8

Query 3: Min and Max Prices of Products

SQL Query Code:

```
SELECT MIN(price) AS
min_price, MAX(price) AS
max_price
FROM products;
```



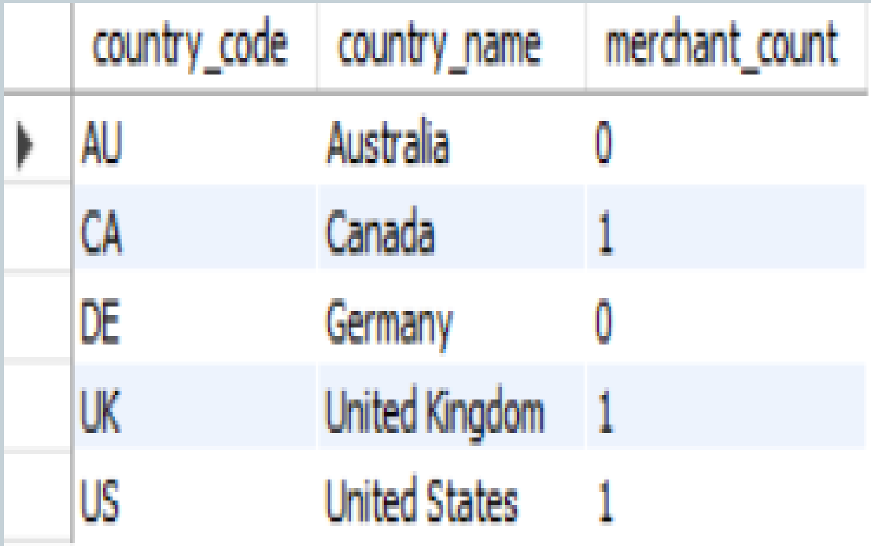
```
  min_price   max_price
▶ 14.99       999.99
```

The output is a single row with two columns: min_price (the lowest price of any product; 14.99) and max_price (the highest price of any product; 999.99).

Query 4: Count Number of Merchants Per Country

SQL Query Code:

```
SELECT c.country_code,
c.country_name,
COUNT(m.merchant_id) AS
merchant_count
FROM countries c
LEFT JOIN merchants m ON
c.country_code = m.country_code
GROUP BY c.country_code,
c.country_name;
```



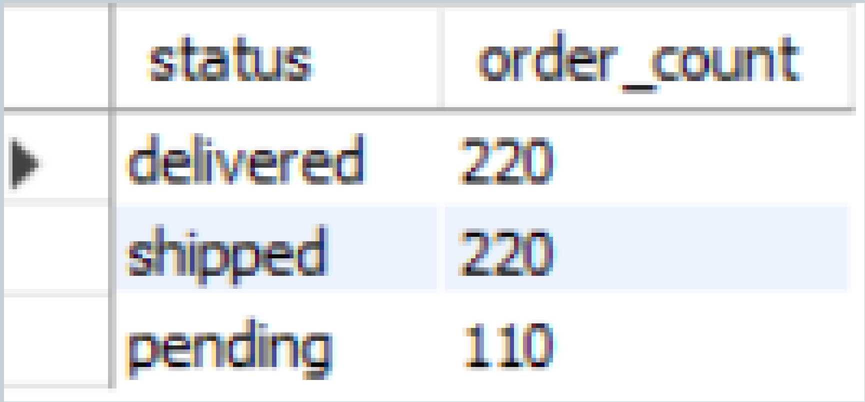
country_code	country_name	merchant_count
AU	Australia	0
CA	Canada	1
DE	Germany	0
UK	United Kingdom	1
US	United States	1

The output lists each country (with its country_code and country_name) and the number of merchants in that country (merchant_count). Countries with no merchants will show a merchant_count of 0 due to the LEFT JOIN. Here, the CA,UK and US has 1 merchant respectively.

Task 5: GROUP BY & HAVING Clauses
Query 1: Group Orders by Status and Count Them

SQL Query Code:

```
SELECT status,
COUNT(order_id) AS
order_count FROM orders
GROUP BY status;
```



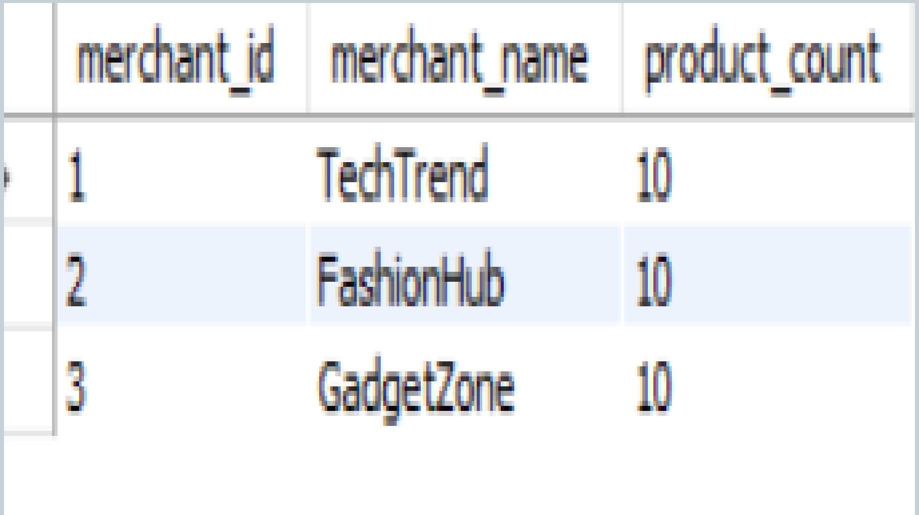
status	order_count
delivered	220
shipped	220
pending	110

The output lists each unique order status (e.g., "Pending", "Shipped", "Delivered") and the total number of orders with that status (order_count).

Query 2: Group Products by Merchant and Count Them

SQL Query Code:

```
SELECT m.merchant_id,
m.merchant_name,
COUNT(p.product_id) AS
product_count
FROM merchants m
LEFT JOIN products p ON
m.merchant_id = p.merchant_id
GROUP BY m.merchant_id,
m.merchant_name;
```



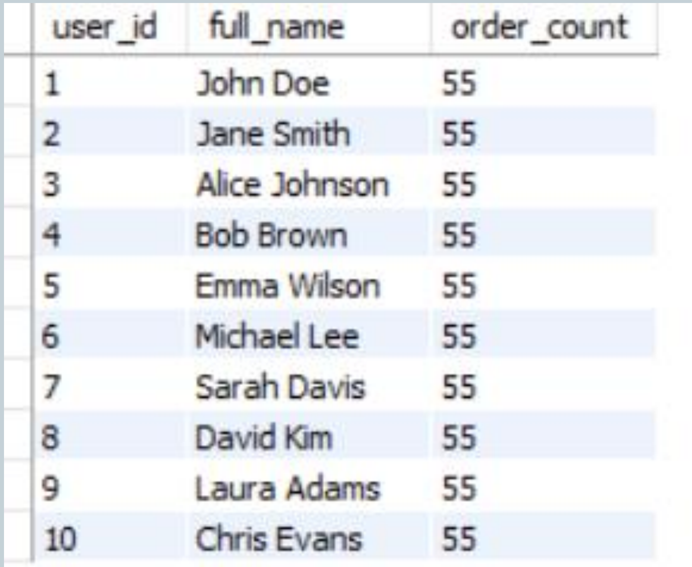
merchant_id	merchant_name	product_count
1	TechTrend	10
2	FashionHub	10
3	GadgetZone	10

The output lists each merchant (with their merchant_id and merchant_name) and the number of products they offer (product_count). Merchants with no products will show a product_count of 0 due to the LEFT JOIN.

Query 3: Show Users Who Have Placed More Than 3 Orders

SQL Query Code:

```
SELECT u.user_id, u.full_name,
COUNT(o.order_id) AS order_count
FROM users u
JOIN orders o ON u.user_id =
o.user_id
GROUP BY u.user_id, u.full_name
HAVING COUNT(o.order_id) > 3;
```



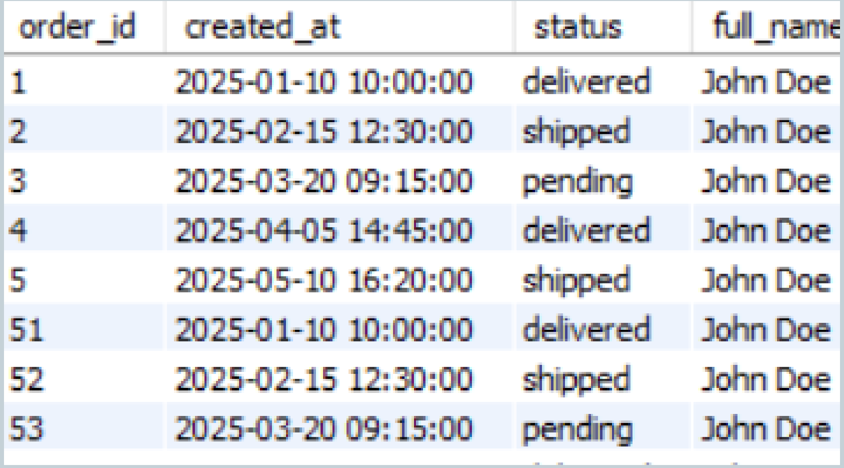
user_id	full_name	order_count
1	John Doe	55
2	Jane Smith	55
3	Alice Johnson	55
4	Bob Brown	55
5	Emma Wilson	55
6	Michael Lee	55
7	Sarah Davis	55
8	David Kim	55
9	Laura Adams	55
10	Chris Evans	55

The output lists users (with their user_id, full_name, and order_count) who have placed more than 3 orders.

Task-6: SQL Joins
Query 1: Inner Join Between Orders and Users

SQL Query Code:

```
SELECT o.order_id,
o.created_at, o.status,
u.full_nameFROM orders
oINNER JOIN users u ON
o.user_id = u.user_id;
```



order_id	created_at	status	full_name
1	2025-01-10 10:00:00	delivered	John Doe
2	2025-02-15 12:30:00	shipped	John Doe
3	2025-03-20 09:15:00	pending	John Doe
4	2025-04-05 14:45:00	delivered	John Doe
5	2025-05-10 16:20:00	shipped	John Doe
51	2025-01-10 10:00:00	delivered	John Doe
52	2025-02-15 12:30:00	shipped	John Doe
53	2025-03-20 09:15:00	pending	John Doe

The output lists each order with its order_id, created_at (e.g., a timestamp like "2025-05-23 11:20 PM +06"), status (e.g., "Pending"), and the full_name of the user who placed it. Only orders linked to existing users are included, as the INNER JOIN excludes unmatched rows.

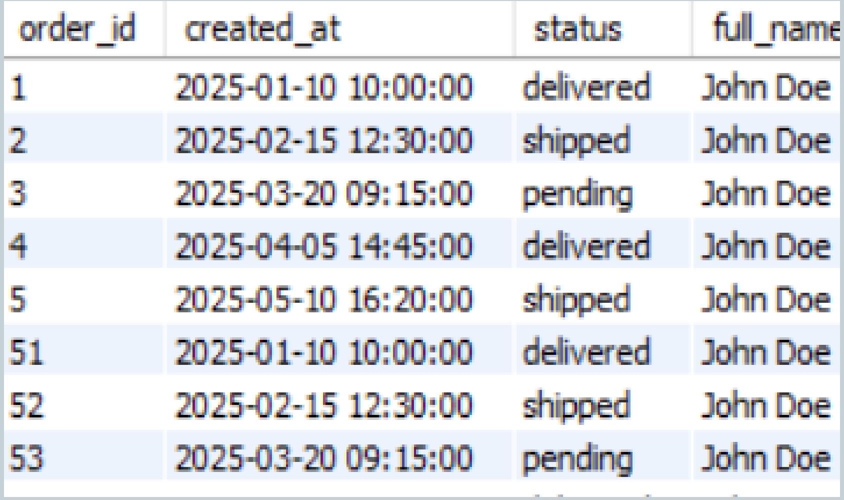
Task-6: SQL Joins
Query 1: Inner Join Between Orders and Users

SQL Query Code:

SELECT o.order_id, o.created_at, o.status, u.full_name

FROM orders o

INNER JOIN users u ON o.user_id = u.user_id;



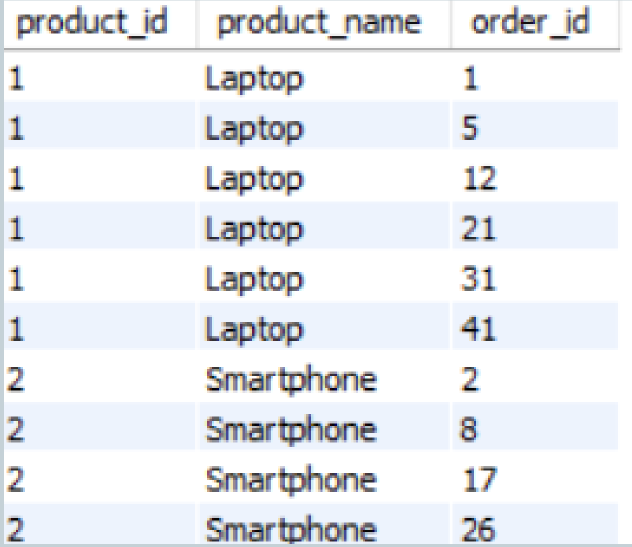
order_id	created_at	status	full_name
1	2025-01-10 10:00:00	delivered	John Doe
2	2025-02-15 12:30:00	shipped	John Doe
3	2025-03-20 09:15:00	pending	John Doe
4	2025-04-05 14:45:00	delivered	John Doe
5	2025-05-10 16:20:00	shipped	John Doe
51	2025-01-10 10:00:00	delivered	John Doe
52	2025-02-15 12:30:00	shipped	John Doe
53	2025-03-20 09:15:00	pending	John Doe

The output lists each order with its order_id, created_at (e.g., a timestamp like "2025-05-23 11:20 PM +06"), status (e.g., "Pending"), and the full_name of the user who placed it. Only orders linked to existing users are included, as the INNER JOIN excludes unmatched rows.

Query 2: Left Join to Get All Products (Including Unordered Ones)

SQL Query Code:

```
SELECT p.product_id,
p.product_name, o.order_id
FROM products p
LEFT JOIN order_items oi ON
p.product_id = oi.product_id
LEFT JOIN orders o ON
oi.order_id = o.order_id;
```



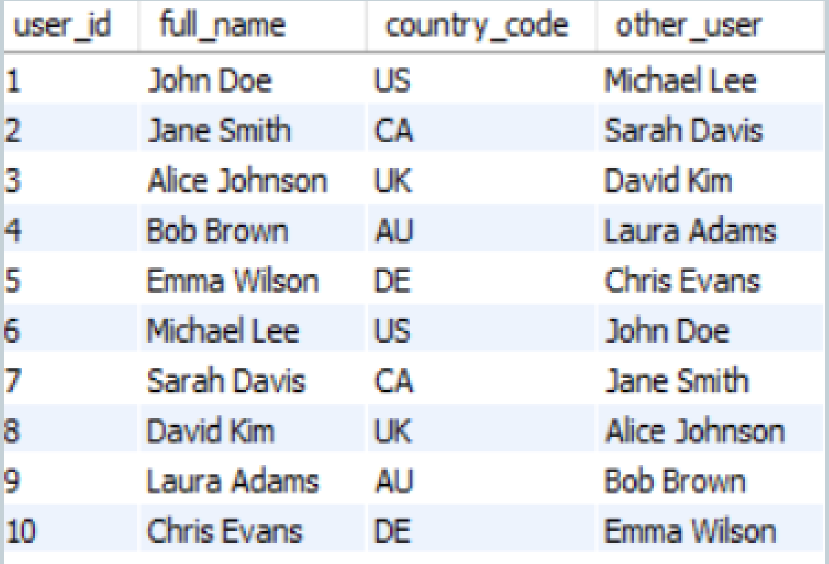
product_id	product_name	order_id
1	Laptop	1
1	Laptop	5
1	Laptop	12
1	Laptop	21
1	Laptop	31
1	Laptop	41
2	Smartphone	2
2	Smartphone	8
2	Smartphone	17
2	Smartphone	26

The output lists every product_id and product_name from the products table, with an order_id if the product has been ordered, or NULL if it hasn’t. For example, a product like "Laptop" with an order might show order_id 1, while an unordered product like "Tablet" would show NULL, ensuring all products are included regardless of order status.

Query 3: Self Join – Find Users Who Share the Same Country

SQL Query Code:

```
SELECT u1.user_id, u1.full_name, u1.country_code, u2.full_name AS other_user
FROM users u1
JOIN users u2 ON u1.country_code = u2.country_code AND u1.user_id != u2.user_id;
```



user_id	full_name	country_code	other_user
1	John Doe	US	Michael Lee
2	Jane Smith	CA	Sarah Davis
3	Alice Johnson	UK	David Kim
4	Bob Brown	AU	Laura Adams
5	Emma Wilson	DE	Chris Evans
6	Michael Lee	US	John Doe
7	Sarah Davis	CA	Jane Smith
8	David Kim	UK	Alice Johnson
9	Laura Adams	AU	Bob Brown
10	Chris Evans	DE	Emma Wilson

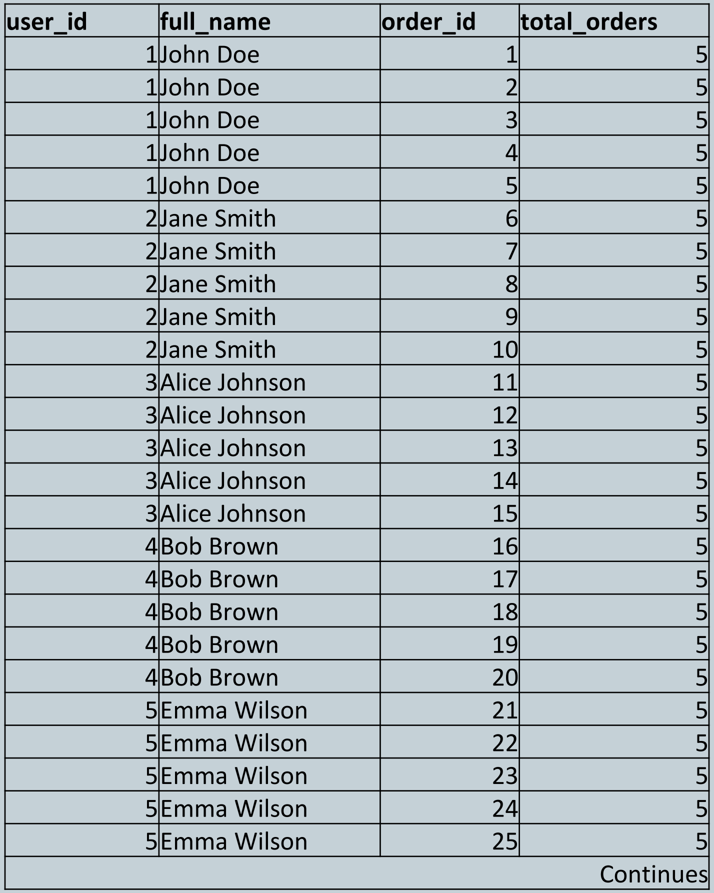
The output lists each user (with user_id, full_name, and country_code) alongside the full_name of another user from the same country (as other_user).

Task-7: Window Functions
Query 1: Total Number of Orders Per User Using a Window Function

SQL Query Code:

```
SELECT u.user_id, u.full_name,
o.order_id,
COUNT(o.order_id) OVER
(PARTITION BY u.user_id) AS
total_orders
FROM users u
LEFT JOIN orders o ON u.user_id
= o.user_id;
```

The output lists each user (with user_id and full_name) and their associated order_id (or NULL if no orders), with the total_orders column showing the total number of orders for that user. For example, if John Doe has orders 101 and 102, both rows will show total_orders as 2, while a user with no orders will show total_orders as 0.



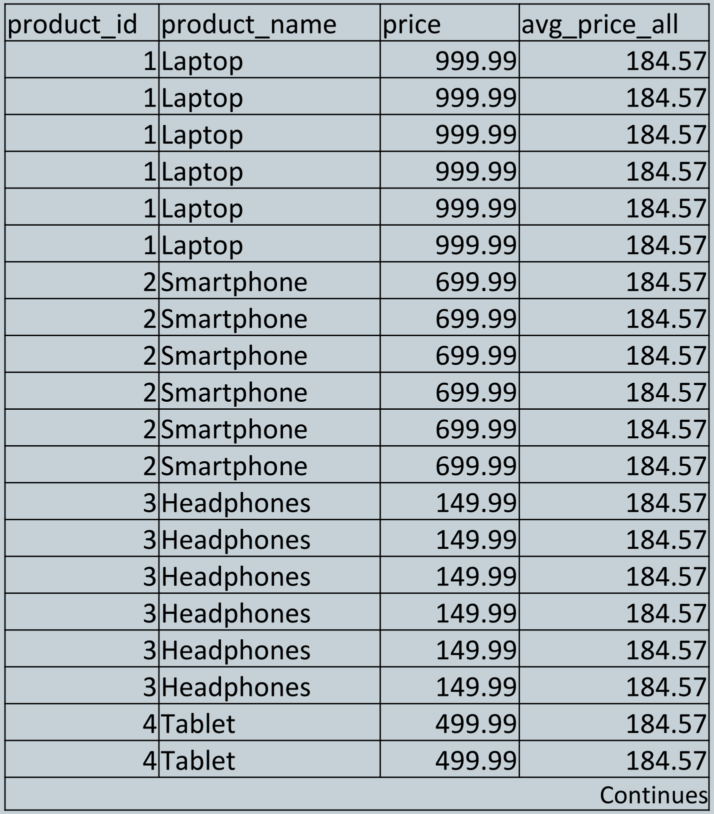
user_id	full_name	order_id	total_orders
1	John Doe	1	5
1	John Doe	2	5
1	John Doe	3	5
1	John Doe	4	5
1	John Doe	5	5
2	Jane Smith	6	5
2	Jane Smith	7	5
2	Jane Smith	8	5
2	Jane Smith	9	5
2	Jane Smith	10	5
3	Alice Johnson	11	5
3	Alice Johnson	12	5
3	Alice Johnson	13	5
3	Alice Johnson	14	5
3	Alice Johnson	15	5
4	Bob Brown	16	5
4	Bob Brown	17	5
4	Bob Brown	18	5
4	Bob Brown	19	5
4	Bob Brown	20	5
5	Emma Wilson	21	5
5	Emma Wilson	22	5
5	Emma Wilson	23	5
5	Emma Wilson	24	5
5	Emma Wilson	25	5
			Continues

Query 2: Average Price of Products Across All Orders Using a Window Function

SQL Query Code:

```
SELECT p.product_id,
p.product_name, p.price,
AVG(p.price) OVER () AS
avg_price_all
FROM products p
JOIN order_items oi ON
p.product_id = oi.product_id;
```

The output lists each ordered product (with product_id, product_name, and price) and the avg_price_all, which is the same average price of all products across all rows. For example, if products have prices 10, 20, and 30, each row will show avg_price_all as (10 + 20 + 30) / 3 = 20, alongside the individual product details. Only products that were ordered are included due to the INNER JOIN.



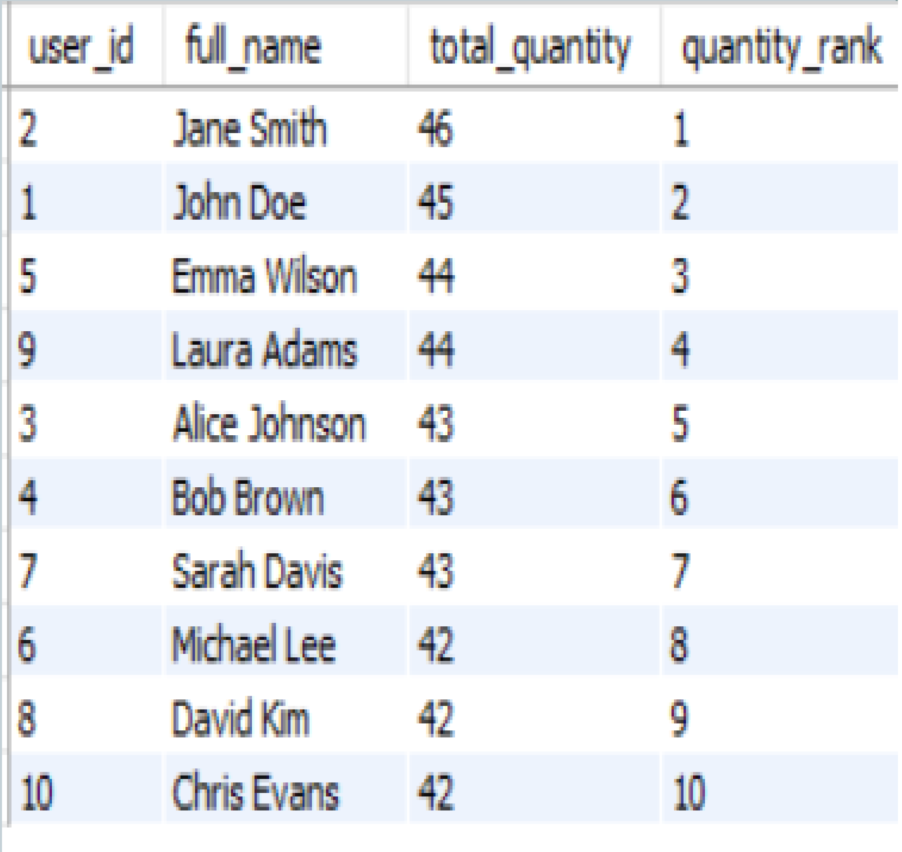
product_id	product_name	price	avg_price_all
1	Laptop	999.99	184.57
1	Laptop	999.99	184.57
1	Laptop	999.99	184.57
1	Laptop	999.99	184.57
1	Laptop	999.99	184.57
1	Laptop	999.99	184.57
2	Smartphone	699.99	184.57
2	Smartphone	699.99	184.57
2	Smartphone	699.99	184.57
2	Smartphone	699.99	184.57
2	Smartphone	699.99	184.57
2	Smartphone	699.99	184.57
3	Headphones	149.99	184.57
3	Headphones	149.99	184.57
3	Headphones	149.99	184.57
3	Headphones	149.99	184.57
3	Headphones	149.99	184.57
3	Headphones	149.99	184.57
4	Tablet	499.99	184.57
4	Tablet	499.99	184.57
			Continues

Query 3: Rank Users by Quantity of Products Ordered Using a Window Function

SQL Query Code:

```
SELECT u.user_id, u.full_name,
SUM(oi.quantity) AS total_quantity,
ROW_NUMBER() OVER (ORDER BY
SUM(oi.quantity) DESC) AS
quantity_rank
FROM users u
JOIN orders o ON u.user_id = o.user_id
JOIN order_items oi ON o.order_id =
oi.order_id
GROUP BY u.user_id, u.full_name;
```

The output lists each user (with user_id and full_name), their total_quantity of ordered products, and a quantity_rank where 1 is assigned to the user with the highest total quantity, 2 to the next, and so on. Only users with at least one order are included due to the INNER JOINs.



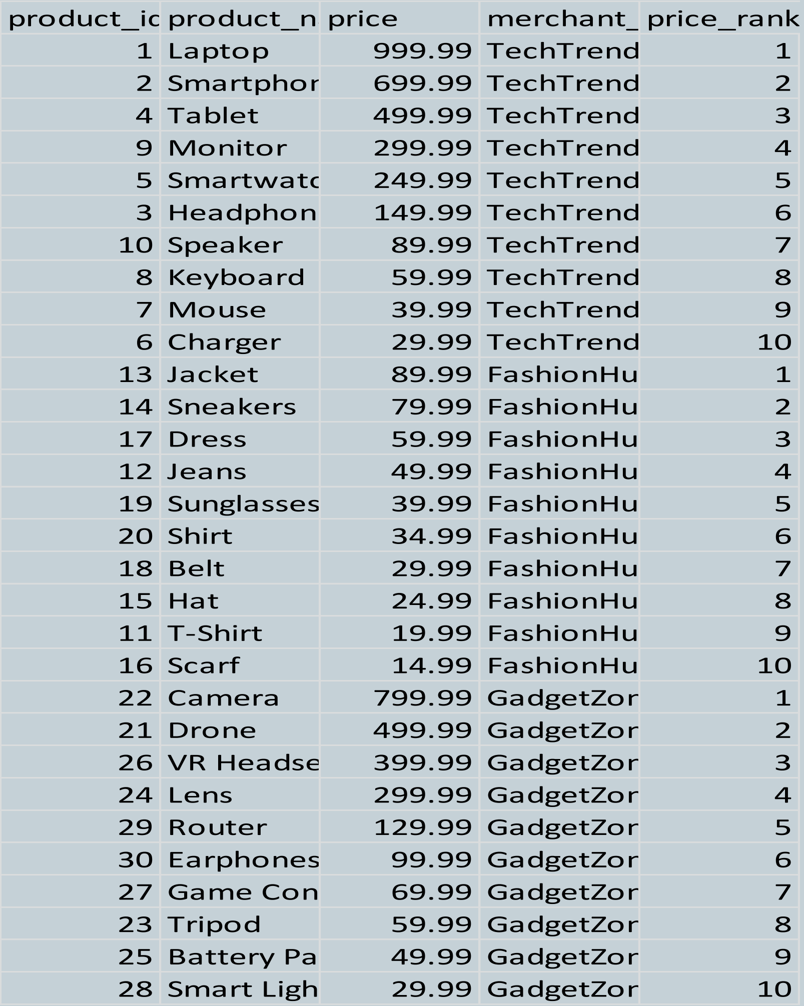
user_id	full_name	total_quantity	quantity_rank
2	Jane Smith	46	1
1	John Doe	45	2
5	Emma Wilson	44	3
9	Laura Adams	44	4
3	Alice Johnson	43	5
4	Bob Brown	43	6
7	Sarah Davis	43	7
6	Michael Lee	42	8
8	David Kim	42	9
10	Chris Evans	42	10

Query 4: Rank Products by Price Within Each Merchant Using RANK()

SQL Query Code:

```
SELECT p.product_id,
p.product_name, p.price,
m.merchant_name,
RANK() OVER (PARTITION BY
p.merchant_id ORDER BY p.price
DESC) AS price_rank
FROM products p
JOIN merchants m ON p.merchant_id
= m.merchant_id;
```

The output lists each product (with product_id, product_name, price, and the merchant_name) along with a price_rank indicating its price ranking within its merchant (e.g., 1 for the highest-priced product, 2 for the next, etc.). Only products linked to merchants are included due to the INNER JOIN.



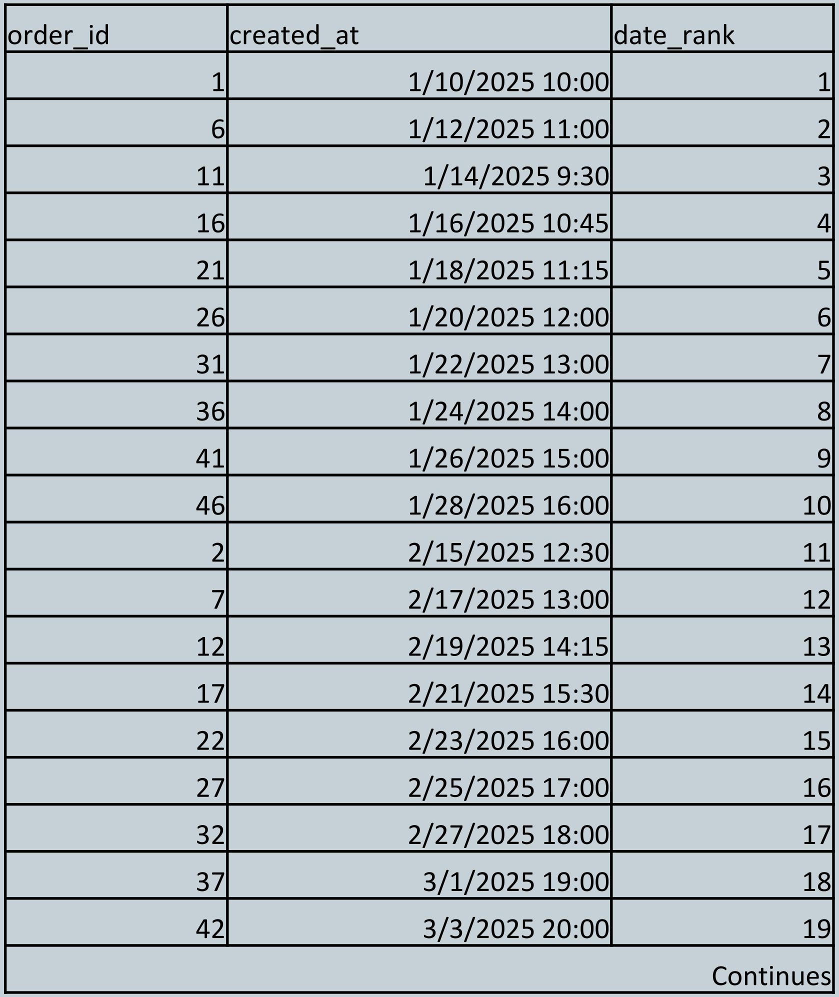
product_ic	product_n	price	merchant_	price_rank
1	Laptop	999.99	TechTrend	1
2	Smartphor	699.99	TechTrend	2
4	Tablet	499.99	TechTrend	3
9	Monitor	299.99	TechTrend	4
5	Smartwatc	249.99	TechTrend	5
3	Headphon	149.99	TechTrend	6
10	Speaker	89.99	TechTrend	7
8	Keyboard	59.99	TechTrend	8
7	Mouse	39.99	TechTrend	9
6	Charger	29.99	TechTrend	10
13	Jacket	89.99	FashionHu	1
14	Sneakers	79.99	FashionHu	2
17	Dress	59.99	FashionHu	3
12	Jeans	49.99	FashionHu	4
19	Sunglasses	39.99	FashionHu	5
20	Shirt	34.99	FashionHu	6
18	Belt	29.99	FashionHu	7
15	Hat	24.99	FashionHu	8
11	T-Shirt	19.99	FashionHu	9
16	Scarf	14.99	FashionHu	10
22	Camera	799.99	GadgetZor	1
21	Drone	499.99	GadgetZor	2
26	VR Headse	399.99	GadgetZor	3
24	Lens	299.99	GadgetZor	4
29	Router	129.99	GadgetZor	5
30	Earphones	99.99	GadgetZor	6
27	Game Con	69.99	GadgetZor	7
23	Tripod	59.99	GadgetZor	8
25	Battery Pa	49.99	GadgetZor	9
28	Smart Ligh	29.99	GadgetZor	10

Query 5: Use DENSE_RANK() to Rank Orders by Created Date

SQL Query Code:

```
SELECT order_id, created_at,
DENSE_RANK() OVER (ORDER
BY created_at) AS date_rank
FROM orders;
```

The output lists each order with its order_id, created_at (e.g., "2025-05-23 10:00 AM +06"), and a date_rank where 1 is the earliest date, 2 the next, and so on. Unlike RANK(), DENSE_RANK() ensures consecutive ranks for ties (e.g., if two orders share the same created_at, both get rank 1, and the next order gets 2), avoiding gaps. All orders in the table are included.



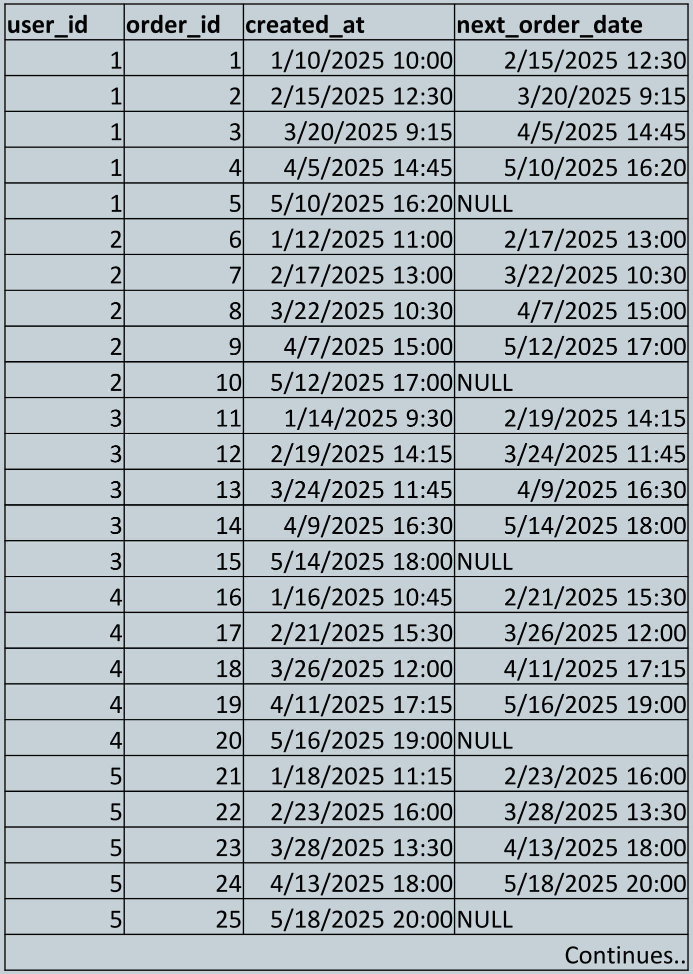
order_id	created_at	date_rank
1	1/10/2025 10:00	1
6	1/12/2025 11:00	2
11	1/14/2025 9:30	3
16	1/16/2025 10:45	4
21	1/18/2025 11:15	5
26	1/20/2025 12:00	6
31	1/22/2025 13:00	7
36	1/24/2025 14:00	8
41	1/26/2025 15:00	9
46	1/28/2025 16:00	10
2	2/15/2025 12:30	11
7	2/17/2025 13:00	12
12	2/19/2025 14:15	13
17	2/21/2025 15:30	14
22	2/23/2025 16:00	15
27	2/25/2025 17:00	16
32	2/27/2025 18:00	17
37	3/1/2025 19:00	18
42	3/3/2025 20:00	19
		Continues

Task 7 (Value/Analytic): Advanced Window Functions Query 1: Use LEAD() to Find the Next Order Date for Each User

SQL Query Code:

```
SELECT user_id, order_id, created_at,
LEAD(created_at) OVER (PARTITION BY user_id ORDER BY created_at) AS next_order_date
FROM orders;
```

The output lists each order with its user_id, order_id, created_at (e.g., "2025-05-23 10:00 AM +06"), and the next_order_date for that user, which is the created_at of their subsequent order. If it's the user's last order, next_order_date will be NULL.



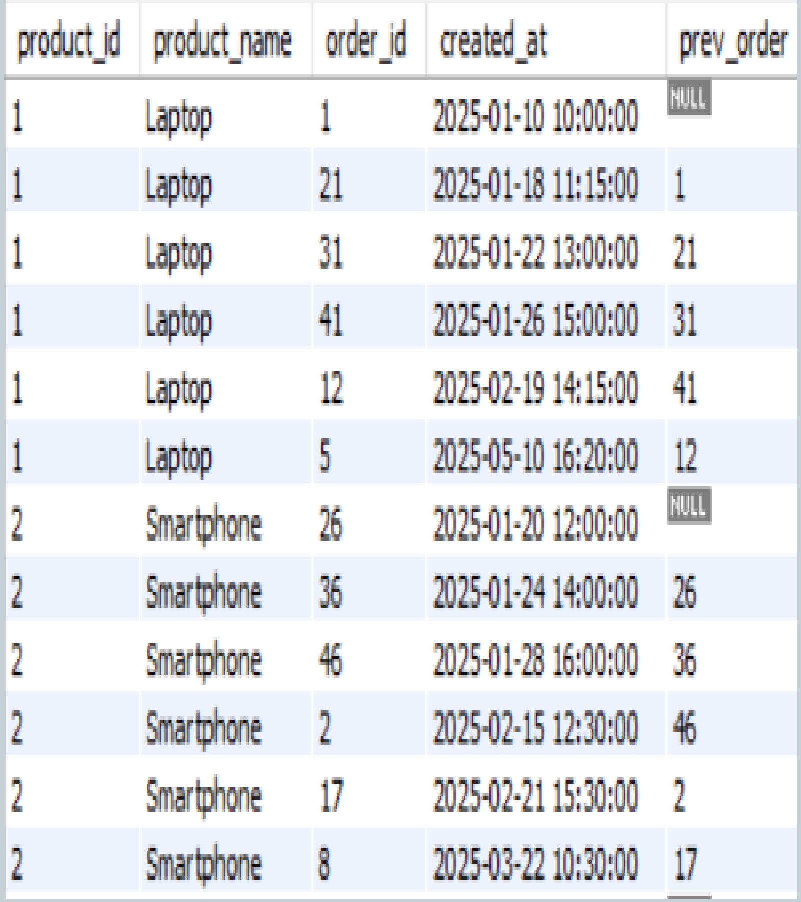
user_id	order_id	created_at	next_order_date
1	1	1/10/2025 10:00	2/15/2025 12:30
1	2	2/15/2025 12:30	3/20/2025 9:15
1	3	3/20/2025 9:15	4/5/2025 14:45
1	4	4/5/2025 14:45	5/10/2025 16:20
1	5	5/10/2025 16:20	NULL
2	6	1/12/2025 11:00	2/17/2025 13:00
2	7	2/17/2025 13:00	3/22/2025 10:30
2	8	3/22/2025 10:30	4/7/2025 15:00
2	9	4/7/2025 15:00	5/12/2025 17:00
2	10	5/12/2025 17:00	NULL
3	11	1/14/2025 9:30	2/19/2025 14:15
3	12	2/19/2025 14:15	3/24/2025 11:45
3	13	3/24/2025 11:45	4/9/2025 16:30
3	14	4/9/2025 16:30	5/14/2025 18:00
3	15	5/14/2025 18:00	NULL
4	16	1/16/2025 10:45	2/21/2025 15:30
4	17	2/21/2025 15:30	3/26/2025 12:00
4	18	3/26/2025 12:00	4/11/2025 17:15
4	19	4/11/2025 17:15	5/16/2025 19:00
4	20	5/16/2025 19:00	NULL
5	21	1/18/2025 11:15	2/23/2025 16:00
5	22	2/23/2025 16:00	3/28/2025 13:30
5	23	3/28/2025 13:30	4/13/2025 18:00
5	24	4/13/2025 18:00	5/18/2025 20:00
5	25	5/18/2025 20:00	NULL
			Continues..

Query 2: Use LAG() to Determine the Previous Order for Each Product

<u>SQL Query Code:</u>

```
SELECT p.product_id, p.product_name,
o.order_id, o.created_at,
LAG(o.order_id) OVER (PARTITION BY
p.product_id ORDER BY o.created_at)
AS prev_order
FROM products p
JOIN order_items oi ON p.product_id =
oi.product_id
JOIN orders o ON oi.order_id =
o.order_id;
```

The output lists each ordered product (with product_id, product_name, order_id, and created_at, e.g., "2025-05-23 10:00 AM +06") and the prev_order (the order_id of the prior order for that product). If it's the product's first order, prev_order is NULL.



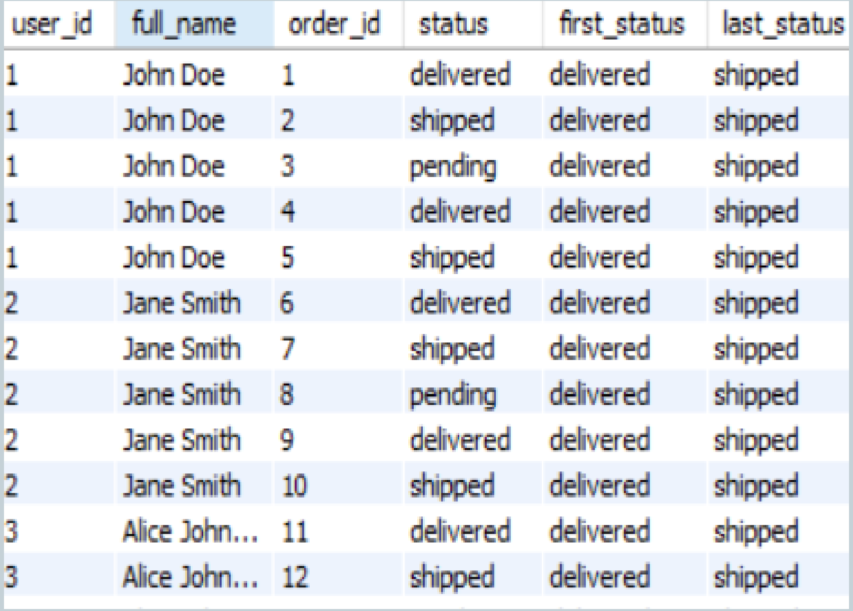
product_id	product_name	order_id	created_at	prev_order
1	Laptop	1	2025-01-10 10:00:00	NULL
1	Laptop	21	2025-01-18 11:15:00	1
1	Laptop	31	2025-01-22 13:00:00	21
1	Laptop	41	2025-01-26 15:00:00	31
1	Laptop	12	2025-02-19 14:15:00	41
1	Laptop	5	2025-05-10 16:20:00	12
2	Smartphone	26	2025-01-20 12:00:00	NULL
2	Smartphone	36	2025-01-24 14:00:00	26
2	Smartphone	46	2025-01-28 16:00:00	36
2	Smartphone	2	2025-02-15 12:30:00	46
2	Smartphone	17	2025-02-21 15:30:00	2
2	Smartphone	8	2025-03-22 10:30:00	17

Query 3: Retrieve FIRST_VALUE() and LAST_VALUE() of Order Statuses for Each User

SQL Query Code:

```
SELECT u.user_id, u.full_name, o.order_id,
o.status,
FIRST_VALUE(o.status) OVER (PARTITION
BY u.user_id ORDER BY o.created_at) AS
first_status,
LAST_VALUE(o.status) OVER (PARTITION
BY u.user_id ORDER BY o.created_at
ROWS BETWEEN UNBOUNDED PRECEDING
AND UNBOUNDED FOLLOWING) AS
last_status
FROM users u
JOIN orders o ON u.user_id = o.user_id;
```

The output lists each user (with user_id, full_name) and their orders (with order_id and status, e.g., "Pending" or "Delivered"), along with first_status (status of their earliest order, e.g., "Pending") and last_status (status of their latest order, e.g., "Delivered") for every row.



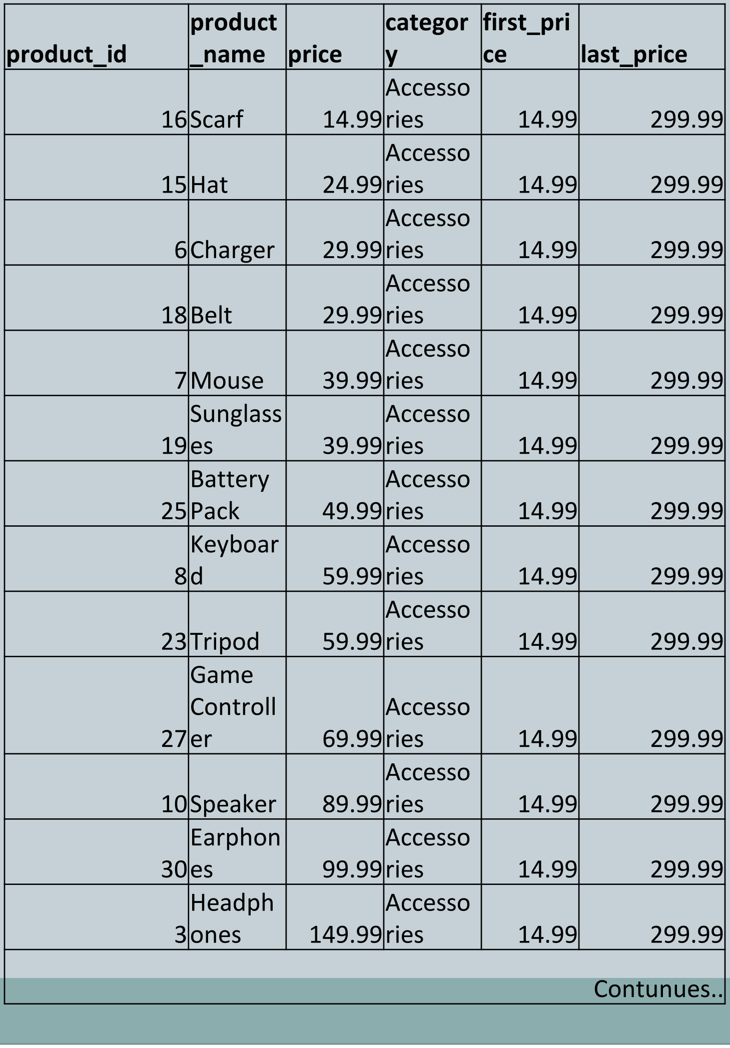
user_id	full_name	order_id	status	first_status	last_status
1	John Doe	1	delivered	delivered	shipped
1	John Doe	2	shipped	delivered	shipped
1	John Doe	3	pending	delivered	shipped
1	John Doe	4	delivered	delivered	shipped
1	John Doe	5	shipped	delivered	shipped
2	Jane Smith	6	delivered	delivered	shipped
2	Jane Smith	7	shipped	delivered	shipped
2	Jane Smith	8	pending	delivered	shipped
2	Jane Smith	9	delivered	delivered	shipped
2	Jane Smith	10	shipped	delivered	shipped
3	Alice John...	11	delivered	delivered	shipped
3	Alice John...	12	shipped	delivered	shipped

Query 4: Find FIRST_VALUE() and LAST_VALUE() of Prices in Each Product Category

SQL Query Code:

```
SELECT p.product_id, p.product_name,
p.price, p.category,      FIRST_VALUE(p.price)
OVER (PARTITION BY p.category ORDER BY
p.price) AS first_price,
LAST_VALUE(p.price) OVER (PARTITION BY
p.category ORDER BY p.price
ROWS BETWEEN UNBOUNDED PRECEDING
AND UNBOUNDED FOLLOWING) AS
last_price
FROM products p;
```

The output lists each product (with product_id, product_name, price, and category) along with first_price (the lowest price in that category, e.g., 10) and last_price (the highest price, e.g., 50), repeated for every product row. For example, in the "Electronics" category, if prices range from 10 to 50, all rows for that category will show first_price as 10 and last_price as 50, including all products regardless of order status.



product_id	product_name	price	category	first_price	last_price
16	Scarf	14.99	Accessories	14.99	299.99
15	Hat	24.99	Accessories	14.99	299.99
6	Charger	29.99	Accessories	14.99	299.99
18	Belt	29.99	Accessories	14.99	299.99
7	Mouse	39.99	Accessories	14.99	299.99
19	Sunglasses	39.99	Accessories	14.99	299.99
25	Battery Pack	49.99	Accessories	14.99	299.99
8	Keyboard	59.99	Accessories	14.99	299.99
23	Tripod	59.99	Accessories	14.99	299.99
27	Game Controller	69.99	Accessories	14.99	299.99
10	Speaker	89.99	Accessories	14.99	299.99
30	Earphones	99.99	Accessories	14.99	299.99
3	Headphones	149.99	Accessories	14.99	299.99

Contunues..

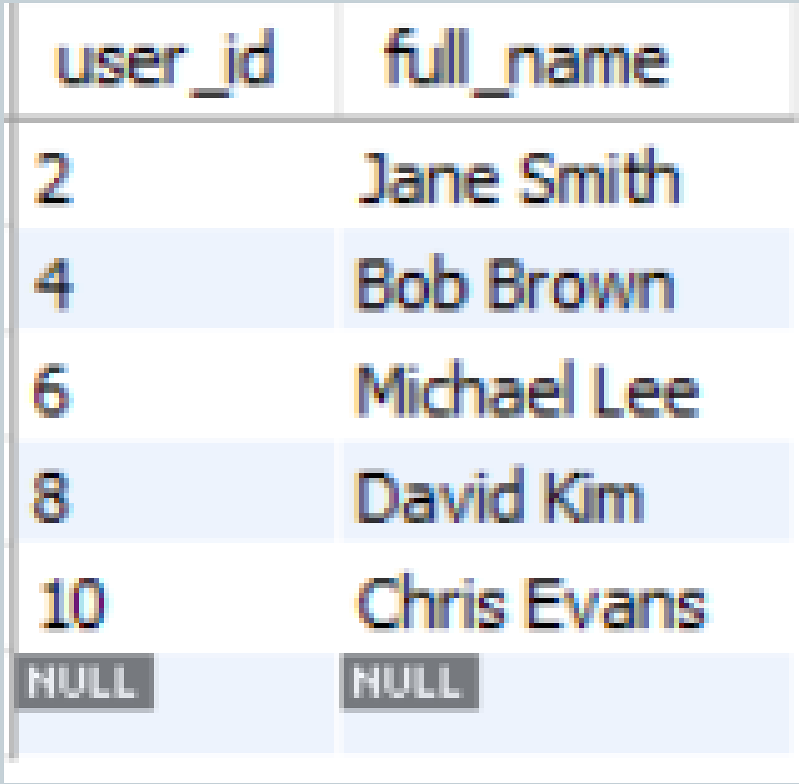
Task 8: Subqueries

Query 1: Find Users Who Have Placed Orders But Have Not Ordered a Particular Product

SQL Query Code:

```
SELECT u.user_id, u.full_name
FROM users u
WHERE u.user_id NOT IN (
SELECT o.user_id
FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
WHERE oi.product_id = 1);
```

The output lists users (with user_id and full_name) who have placed at least one order but have not ordered the product with product_id = 1. For example, if Product 1 is a "Laptop" and John Doe ordered other products but not a Laptop, he would appear in the result. Users who never placed any orders are not explicitly included unless they are filtered in the outer query's context.



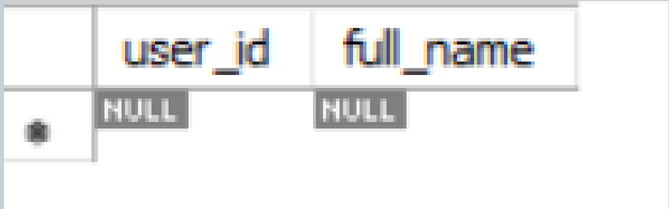
user_id	full_name
2	Jane Smith
4	Bob Brown
6	Michael Lee
8	David Kim
10	Chris Evans
NULL	NULL

Query 2: Find Users Who Have Placed Orders But Have Not Ordered a Particular Product

SQL Query Code:

```
SELECT u.user_id, u.full_name
FROM users u
WHERE (
SELECT COUNT(*)
FROM orders o
WHERE o.user_id = u.user_id
) > (
SELECT AVG(order_count)
FROM (
SELECT COUNT(*) AS order_count
FROM orders
GROUP BY user_id
) sub
);
```

The output lists users (with user_id and full_name) who have placed more orders than the average number of orders per user. For example, if the average order count is 3 and John Doe has 5 orders while Jane Smith has 2, only John Doe would appear. This does not directly address the "not ordered a particular product" condition but rather identifies users with above-average order activity. Users with no orders are excluded due to the COUNT(*) > condition.



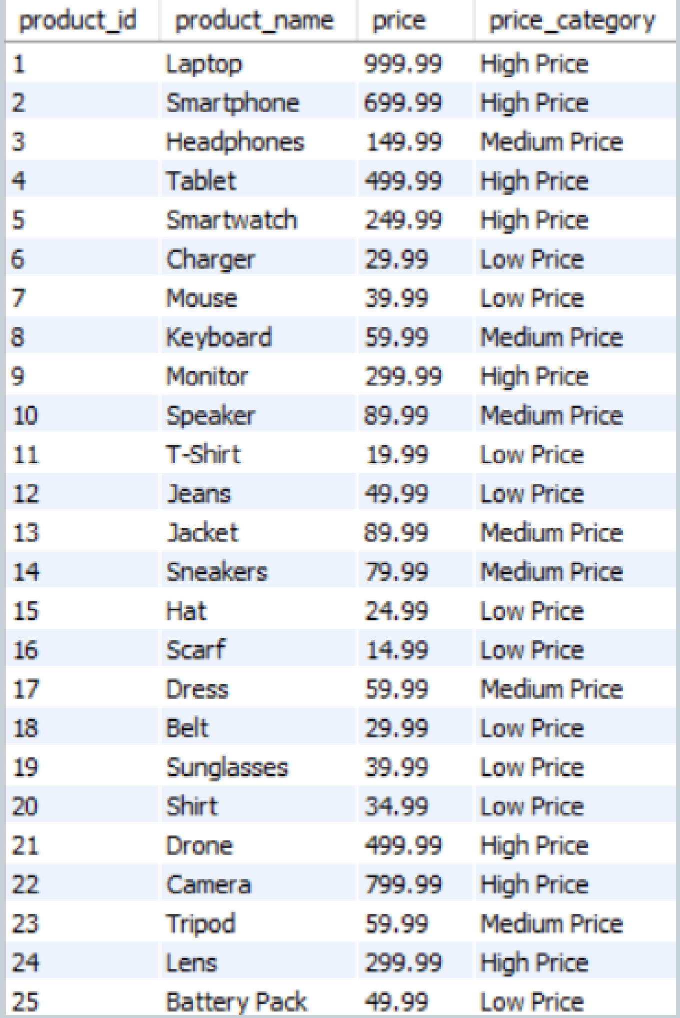
user_id	full_name
NULL	NULL

Task 9: Case Statements
Query 1: Categorize Products by Price

SQL Query Code:

```
SELECT product_id, product_name, price,
CASE
WHEN price < 50 THEN 'Low Price'          WHEN
price BETWEEN 50 AND 200 THEN 'Medium
Price'
ELSE 'High Price'
END AS price_category
FROM products;
```

The output lists each product (with product_id, product_name, and price) along with its price_category. For example, a product priced at 30 would be labeled "Low Price," one at 100 would be "Medium Price," and one at 250 would be "High Price," providing a clear price-based classification for all products in the table.



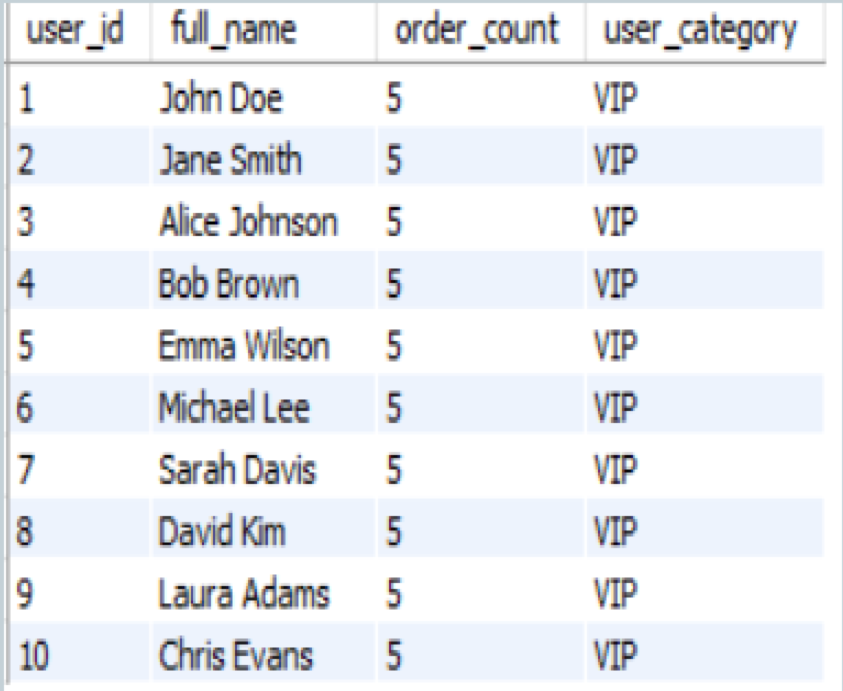
product_id	product_name	price	price_category
1	Laptop	999.99	High Price
2	Smartphone	699.99	High Price
3	Headphones	149.99	Medium Price
4	Tablet	499.99	High Price
5	Smartwatch	249.99	High Price
6	Charger	29.99	Low Price
7	Mouse	39.99	Low Price
8	Keyboard	59.99	Medium Price
9	Monitor	299.99	High Price
10	Speaker	89.99	Medium Price
11	T-Shirt	19.99	Low Price
12	Jeans	49.99	Low Price
13	Jacket	89.99	Medium Price
14	Sneakers	79.99	Medium Price
15	Hat	24.99	Low Price
16	Scarf	14.99	Low Price
17	Dress	59.99	Medium Price
18	Belt	29.99	Low Price
19	Sunglasses	39.99	Low Price
20	Shirt	34.99	Low Price
21	Drone	499.99	High Price
22	Camera	799.99	High Price
23	Tripod	59.99	Medium Price
24	Lens	299.99	High Price
25	Battery Pack	49.99	Low Price

Query 2: Categorize Users Based on Number of Orders

SQL Query Code:

```
SELECT u.user_id, u.full_name,
COUNT(o.order_id) AS order_count,
CASE
WHEN COUNT(o.order_id) = 1 THEN 'New'
WHEN COUNT(o.order_id) BETWEEN 2 AND
4 THEN 'Regular'
ELSE 'VIP'
END AS user_category
FROM users u
LEFT JOIN orders o ON u.user_id = o.user_id
GROUP BY u.user_id, u.full_name;
```

The output lists each user (with user_id, full_name, and order_count) along with their user_category. For example, a user with 1 order is "New," one with 3 orders is "Regular," and one with 5 orders is "VIP." Users with 0 orders (due to the LEFT JOIN) would be categorized as "New" since COUNT(o.order_id) is 0, but the CASE logic starts at 1, so they may need additional handling in practice.



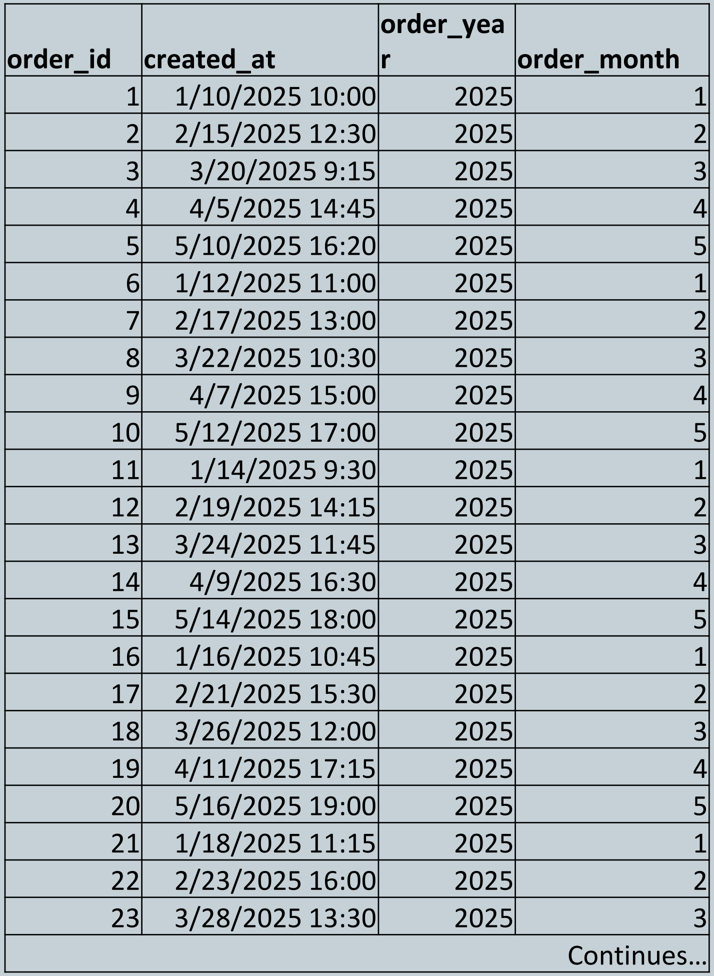
user_id	full_name	order_count	user_category
1	John Doe	5	VIP
2	Jane Smith	5	VIP
3	Alice Johnson	5	VIP
4	Bob Brown	5	VIP
5	Emma Wilson	5	VIP
6	Michael Lee	5	VIP
7	Sarah Davis	5	VIP
8	David Kim	5	VIP
9	Laura Adams	5	VIP
10	Chris Evans	5	VIP

Task 10: Date and Time Functions
Query 1: Extract Year and Month from Created at Column in Orders

SQL Query Code:

```
SELECT order_id, created_at,
EXTRACT(YEAR FROM created_at)
AS order_year,
EXTRACT(MONTH FROM created_at)
AS order_month
FROM orders;
```

The output lists each order with its order_id, created_at (e.g., "2025-05-24 14:33:00 +06"), order_year (e.g., 2025), and order_month (e.g., 5 for May). For example, an order created on "2024-03-15" would show order_year as 2024 and order_month as 3, breaking down the timestamp into year and month components for analysis.

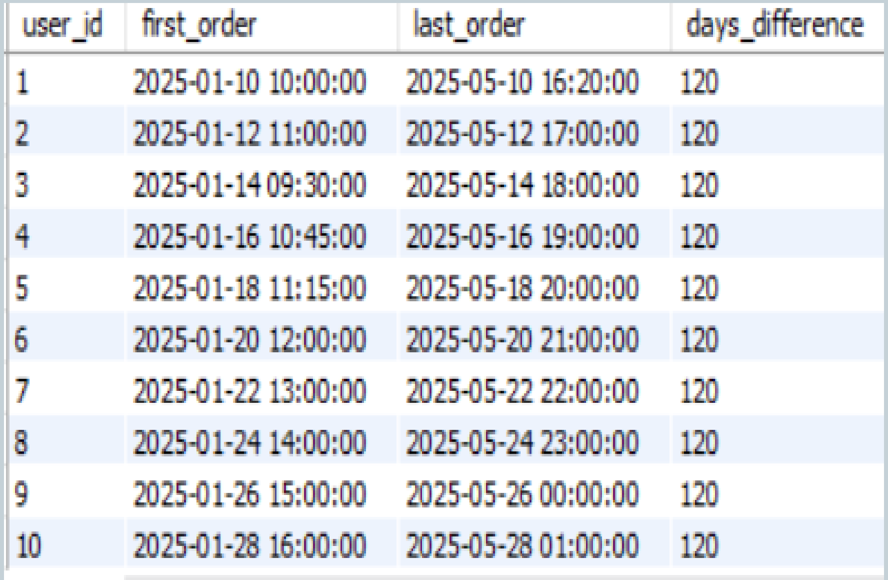


order_id	created_at	order_year	order_month
1	1/10/2025 10:00	2025	1
2	2/15/2025 12:30	2025	2
3	3/20/2025 9:15	2025	3
4	4/5/2025 14:45	2025	4
5	5/10/2025 16:20	2025	5
6	1/12/2025 11:00	2025	1
7	2/17/2025 13:00	2025	2
8	3/22/2025 10:30	2025	3
9	4/7/2025 15:00	2025	4
10	5/12/2025 17:00	2025	5
11	1/14/2025 9:30	2025	1
12	2/19/2025 14:15	2025	2
13	3/24/2025 11:45	2025	3
14	4/9/2025 16:30	2025	4
15	5/14/2025 18:00	2025	5
16	1/16/2025 10:45	2025	1
17	2/21/2025 15:30	2025	2
18	3/26/2025 12:00	2025	3
19	4/11/2025 17:15	2025	4
20	5/16/2025 19:00	2025	5
21	1/18/2025 11:15	2025	1
22	2/23/2025 16:00	2025	2
23	3/28/2025 13:30	2025	3
			Continues...

Query 2: Calculate Day Difference Between First and Last Order Per User

SQL Query Code:

```
SELECT user_id,
MIN(created_at) AS first_order,
MAX(created_at) AS last_order,
DATEDIFF(MAX(created_at),
MIN(created_at)) AS
days_difference
FROM orders
GROUP BY user_id;
```



user_id	first_order	last_order	days_difference
1	2025-01-10 10:00:00	2025-05-10 16:20:00	120
2	2025-01-12 11:00:00	2025-05-12 17:00:00	120
3	2025-01-14 09:30:00	2025-05-14 18:00:00	120
4	2025-01-16 10:45:00	2025-05-16 19:00:00	120
5	2025-01-18 11:15:00	2025-05-18 20:00:00	120
6	2025-01-20 12:00:00	2025-05-20 21:00:00	120
7	2025-01-22 13:00:00	2025-05-22 22:00:00	120
8	2025-01-24 14:00:00	2025-05-24 23:00:00	120
9	2025-01-26 15:00:00	2025-05-26 00:00:00	120
10	2025-01-28 16:00:00	2025-05-28 01:00:00	120

The output lists each user (by user_id) with their first_order (e.g., "2024-01-01 10:00:00 +06"), last_order (e.g., "2025-05-24 14:35:00 +06"), and days_difference (e.g., 509 days). For example, if a user's first order was on "2024-01-01" and last on "2025-05-24," the days_difference would be 509 days, reflecting the time span of their order history. Users with only one order will show 0 days. Only users with orders are included

Task 11: Common Table Expressions (CTEs)
Query 1: CTE to Calculate Cumulative Total of Products Ordered by Each User

SQL Query Code:

```
WITH UserOrderQuantities AS (
SELECT u.user_id, u.full_name, o.order_id, SUM(oi.quantity) AS
order_quantity
FROM users u
JOIN orders o ON u.user_id = o.user_id
JOIN order_items oi ON o.order_id = oi.order_id
GROUP BY u.user_id, u.full_name, o.order_id)
SELECT user_id, full_name, order_id, order_quantity,
SUM(order_quantity) OVER (PARTITION BY user_id ORDER BY
order_id) AS cumulative_quantity
FROM UserOrderQuantities;
```

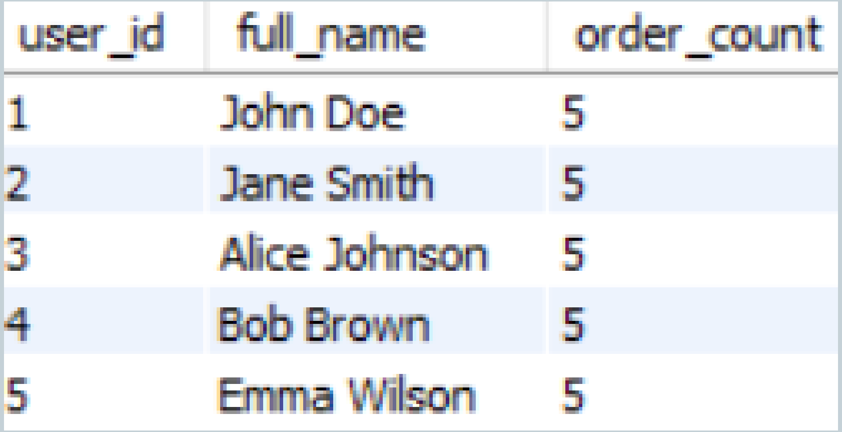
The output lists each order (with user_id, full_name, order_id, and order_quantity, e.g., 5 items) and the cumulative_quantity, which accumulates the order_quantity for that user as orders progress. For example, if User 1 ordered 5 items in Order 101 and 3 in Order 102, the rows would show cumulative_quantity as 5 for Order 101 and 8 for Order 102, reflecting the running total. Only users with orders are included due to the INNER JOINs.

user_id	full_name	order_id	order_quantity	cumulative_quantity
1	John Doe	1	8	8
1	John Doe	2	9	17
1	John Doe	3	10	27
1	John Doe	4	9	36
1	John Doe	5	9	45
2	Jane Smith	6	9	9
2	Jane Smith	7	9	18
2	Jane Smith	8	10	28
2	Jane Smith	9	9	37
2	Jane Smith	10	9	46
3	Alice Johnson	11	9	9
3	Alice Johnson	12	7	16
3	Alice Johnson	13	9	25
3	Alice Johnson	14	9	34
3	Alice Johnson	15	9	43
4	Bob Brown	16	9	9
				Continue..

Query 2: CTE to Find Top 5 Users by Number of Orders

SQL Query Code:

```
WITH UserOrderCounts AS (    SELECT
u.user_id, u.full_name, COUNT(o.order_id) AS
order_count    FROM users u
LEFT JOIN orders o ON u.user_id = o.user_id
GROUP BY u.user_id, u.full_name
)
SELECT user_id, full_name, order_count
FROM UserOrderCounts
ORDER BY order_count DESC
LIMIT 5;
```



user_id	full_name	order_count
1	John Doe	5
2	Jane Smith	5
3	Alice Johnson	5
4	Bob Brown	5
5	Emma Wilson	5

This query lists the top 5 users with the highest number of orders. It counts orders per user using a CTE, sorts them in descending order of order count, and returns the top 5 results with user ID, name, and order count.

Thank You!