📘 SQL Mini Project Report

Project Title:
Retail Analytics Using MySQL

Prepared by:
Sonu Raghuwanshi

Date:
20-Aug-2025

Technology Stack:
MySQL 8.x+

Dataset Size:
6 relational tables (~100–400 sample rows each)

- Customers
- Orders
- Order_Items
- Products
- Payments
- Product_Reviews

✨ A collection of 40+ SQL queries demonstrating filtering, aggregations, joins, subqueries, window functions, and set operations, with real-world business use cases.

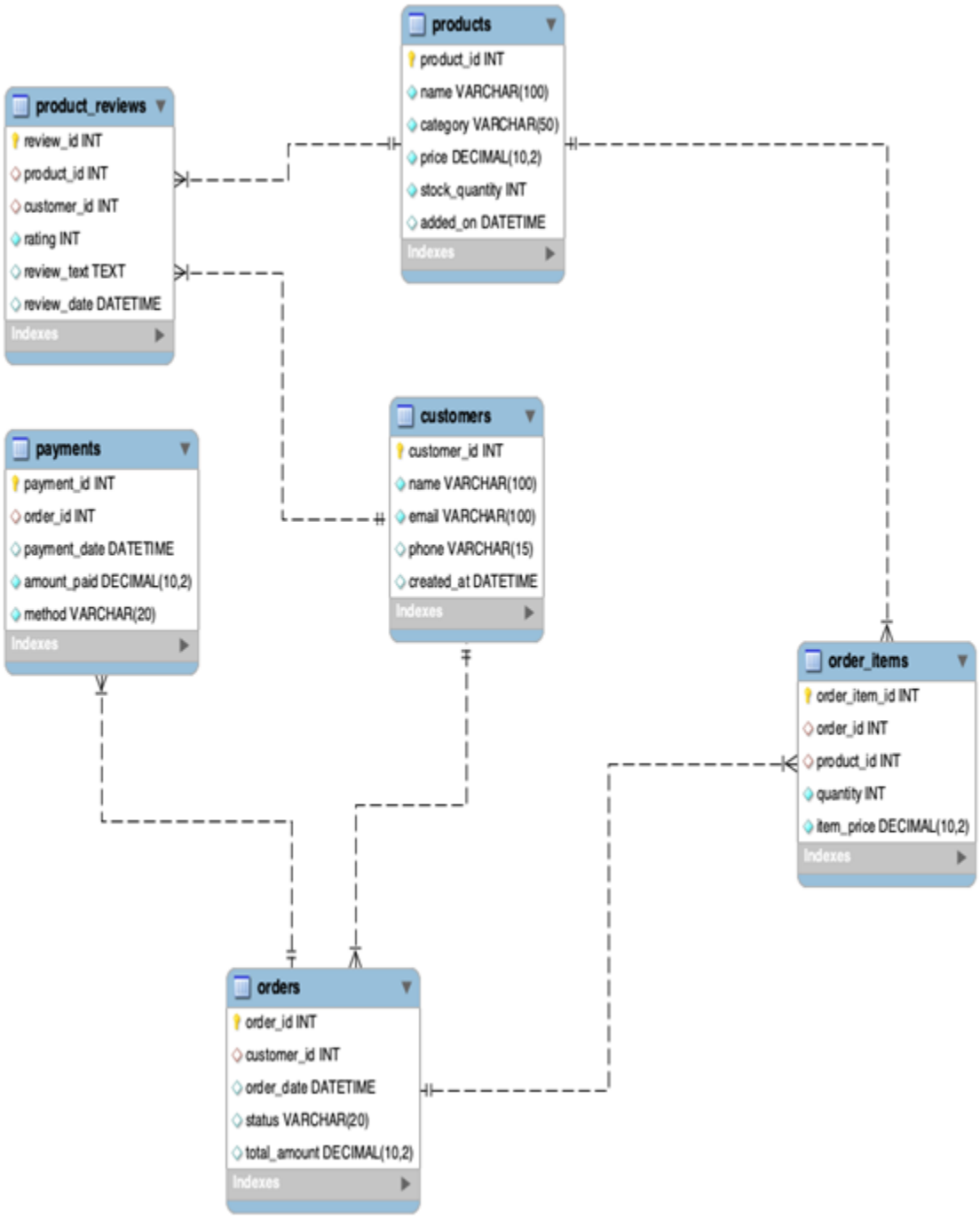
Abstract

This mini-project demonstrates the use of SQL for analyzing a retail database consisting of six interrelated tables: **Customers, Orders, Order_Items, Products, Payments, and Product_Reviews**. The queries are structured in levels of increasing complexity, covering basics, filtering, aggregations, joins, subqueries, window functions, and set operations.

Each query addresses a **real-world business scenario**, such as identifying top customers, analyzing sales trends, calculating revenue, or reviewing product performance. For every solution, both the SQL query and the corresponding output screenshot are included.

The project highlights the power of **MySQL 8.x** in solving analytical problems and provides a foundation for practical database querying, reporting, and decision-making in a retail environment.

ER Diagram:


products
product_id INT
name VARCHAR(100)
category VARCHAR(50)
price DECIMAL(10,2)
stock_quantity INT
added_on DATETIME
Indexes
product_reviews
review_id INT
product_id INT
customer_id INT
rating INT
review_text TEXT
review_date DATETIME
Indexes
customers
customer_id INT
name VARCHAR(100)
email VARCHAR(100)
phone VARCHAR(15)
created_at DATETIME
Indexes
payments
payment_id INT
order_id INT
payment_date DATETIME
amount_paid DECIMAL(10,2)
method VARCHAR(20)
Indexes
order_items
order_item_id INT
order_id INT
product_id INT
quantity INT
item_price DECIMAL(10,2)
Indexes
orders
order_id INT
customer_id INT
order_date DATETIME
status VARCHAR(20)
total_amount DECIMAL(10,2)
Indexes

QUESTIONS

Level 1: Basics

-- 1. Retrieve customer names and emails for email marketing
-- This helps the marketing team extract basic customer contact details for campaigns.

Query:
```
SELECT name, email FROM customers;
```

Output:

Result Grid | Filter Rows:

name	email
Thomas Owens	user1@example.com
Charles Grant	user2@example.com
Kaitlin Richards	user3@example.com
Christina Williams	user4@example.com
David Allen	user5@example.com
Mark Duke	user6@example.com

Figure 1: Output for Query 1

-- 2. View complete product catalog with all available details
-- The product manager may want to review all product listings in one go.

Query:
```
SELECT * FROM products;
```

Output:

product_id	name	category	price	stock_quantity	added_on
1	Plant No	Home	639.43	152	2024-01-30 06:30:53
2	Population Social	Clothing	4813.68	84	2025-05-30 10:02:50
3	Available Answer	Electronics	2529.51	101	2025-04-13 01:11:46
4	Any Question	Clothing	4759.28	179	2025-06-03 13:34:03
5	Natural Network	Toys	4722.66	75	2023-11-06 00:47:37
6	If Whatever	Electronics	177.40	64	2024-12-19 10:37:14
7	Response Indeed	Clothing	4897.36	36	2025-03-29 02:43:08
8	Every Amount	Home	4173.60	156	2025-04-30 03:11:10
9	Common Study	Toys	985.19	171	2023-07-20 13:06:42
10	Development System	Electronics	4801.78	153	2025-03-12 08:22:57
11	Build Her	Books	1852.64	150	2024-09-08 01:09:15
12	Action Ask	Electronics	4017.01	19	2025-02-14 03:38:06
13	Full West	Books	2112.33	172	2023-09-15 03:13:38
14	Listen Development	Home	296.17	132	2024-10-12 11:49:26
15	Place Low	Electronics	723.97	46	2023-07-05 14:36:07
16	Special Fact	Toys	1094.18	15	2025-03-17 10:42:40
17	Everything Plant	Books	2496.68	120	2023-10-08 20:11:55
18	Some Them	Toys	3673.86	110	2024-07-25 00:33:53
19	Build High	Clothing	4707.14	47	2023-09-01 02:50:01

Figure 2: Output for Query 2

-- 3. List all unique product categories
-- Useful for analyzing the range of departments or for creating filters on the website.

Query:
```
SELECT DISTINCT(category) FROM products;
```
Output:

category
Home
Clothing
Electronics
Toys
Books

Figure 3: Output for Query 3

```
-- 4. Show all products priced above ₹1,000
-- This helps identify high-value items for premium promotions or pricing
strategy reviews.
```

Query:
```
SELECT name, price FROM products
    WHERE price > 1000
    ORDER BY price;
```

Output:

name	price
Answer But	1016.68
Special Fact	1094.18
Despite Win	1340.34
Least Green	1398.26
Between Up	1429.45
Build Her	1852.64
Series Page	2070.37
Full West	2112.33
Life Series	2208.99
Factor Where	2295.14
East Foot	2414.93
Everything Pl...	2496.68
Available An...	2529.51
Future Sort	3043.67
Rest Something	3077.82
Assume Serve	3437.81
Movement F...	3502.62
South Free	3543.58
Suffer We	3657.43

Figure 4: Output for Query 4

```
-- 5. Display products within a mid-range price bracket (₹2,000 to ₹5,000)
-- A merchandising team might need this to create a mid-tier pricing
campaign.
```

Query:
```
SELECT name, price AS mid_range FROM products
    WHERE price BETWEEN 2000 AND 5000
    ORDER BY mid_range;
```

Output:

name	mid_range
Series Page	2070.37
Full West	2112.33
Life Series	2208.99
Factor Where	2295.14
East Foot	2414.93
Everything Plant	2496.68
Available Answer	2529.51
Future Sort	3043.67
Rest Something	3077.82
Assume Serve	3437.81
Movement Future	3502.62
South Free	3543.58
Suffer We	3657.43
Some Them	3673.86
Age Treatment	3728.83
Bed Including	3845.31
Weight Enter	3875.18
Action Ask	4017.01
Data Add	4063.38

Figure 5: Output for Query 5

```
-- 6. Fetch data for specific customer IDs (e.g., from loyalty program list)
-- This is used when customer IDs are pre-selected from another system.
```

Query:
```
SELECT * FROM customers
    WHERE customer_id IN (1,3,5,7,8,12,15);
```

Output:

customer_id	name	email	phone	created_at
1	Thomas Owens	user1@example.com	142-479-1945	2024-10-14 16:01:12
3	Kaitlin Richards	user3@example.com	2073473421	2024-06-23 09:55:22
5	David Allen	user5@example.com	(751)456-8289x1	2023-10-29 02:43:00
7	Briana Wright	user7@example.com	223-833-9635	2023-06-25 00:35:43
8	John Bryan	user8@example.com	045.568.0798x27	2025-02-15 17:57:04
12	Mary Knight	user12@example.com	361-636-3802	2023-08-16 20:05:50
15	Austin Flores	user15@example.com	329.901.1576x66	2024-06-13 09:03:42
NULL	NULL	NULL	NULL	NULL

Figure 6: Output for Query 6

```
-- 7. Identify customers whose names start with the letter 'A'
-- Used for alphabetical segmentation in outreach or app display.
```

Query:
```
SELECT name FROM customers
    WHERE name LIKE 'A%';
```
Output:

name
Austin Flores
Amy Landry
Amanda Bright
Adrienne Green

Figure 7: Output for Query 7

```
-- 8. List electronics products priced under ₹3,000
-- Used by merchandising or frontend teams to showcase budget electronics.
```

Query:
```
SELECT name, category, price FROM products
    WHERE category = 'Electronics' AND price < 3000;
```
Output:

name	category	price
Available Answer	Electronics	2529.51
If Whatever	Electronics	177.40
Place Low	Electronics	723.97
Series Page	Electronics	2070.37
Despite Win	Electronics	1340.34
Actually Term	Electronics	396.11
Southern Thing	Electronics	512.46

Figure 8: Output for Query 8

```
-- 9. Display product names and prices in descending order of price
-- This helps teams easily view and compare top-priced items.
```

Query:
```
SELECT name, price FROM products
    ORDER BY price DESC;
```
Output:

name	price
Response Indeed	4897.36
Population Social	4813.68
Development System	4801.78
Any Question	4759.28
Fire Often	4734.89
Natural Network	4722.66
Build High	4707.14
Serious Recognize	4523.10
Study Total	4413.68
Real Source	4398.66
Involve Officer	4244.09
Group But	4180.23
Every Amount	4173.60
Move Small	4168.08
Drop Remember	4160.45
Data Add	4063.38
Action Ask	4017.01
Weight Enter	3875.18
Bed Including	3845.31

Figure 9: Output for Query 9

```
-- 10. Display product names and prices, sorted by price and then by name
-- The merchandising or catalog team may want to list products from most
expensive to cheapest. If multiple products have the same price, they should
be sorted alphabetically for clarity on storefronts or printed catalogs.
```

Query:
```
SELECT name, price FROM products
    ORDER BY price DESC, name ASC;
```

Output:

name	price
Response Indeed	4897.36
Population Social	4813.68
Development System	4801.78
Any Question	4759.28
Fire Often	4734.89
Natural Network	4722.66
Build High	4707.14
Serious Recognize	4523.10
Study Total	4413.68
Real Source	4398.66
Involve Officer	4244.09
Group But	4180.23
Every Amount	4173.60
Move Small	4168.08
Drop Remember	4160.45
Data Add	4063.38
Action Ask	4017.01
Weight Enter	3875.18
Bed Including	3845.31

Figure 10: Output for Query 10

==========
==========

QUESTIONS

Level 2: Filtering and Formatting

-- 1. Retrieve orders where customer information is missing (possibly due to data migration or deletion)
-- Used to identify orphaned orders or test data where customer_id is not linked.

Query:
```
SELECT * FROM Orders
    WHERE customer_id IS NULL;
```
Output:

order_id	customer_id	order_date	status	total_amount
NULL	NULL	NULL	NULL	NULL

Figure 1: Output for Query 1

-- 2. Display customer names and emails using column aliases for frontend readability
-- Useful for feeding into frontend displays or report headings that require user-friendly labels.

Query:
```
SELECT name AS customer_name, email FROM customers;
```
Output:

customer_name	email
Thomas Owens	user1@example.com
Charles Grant	user2@example.com
Kaitlin Richards	user3@example.com
Christina Williams	user4@example.com
David Allen	user5@example.com
Mark Duke	user6@example.com
Briana Wright	user7@example.com
John Bryan	user8@example.com
Jason Thompson	user9@example.com
Shawn Hill	user10@example.com
Walter Jenkins	user11@example.com
Mary Knight	user12@example.com

Figure 2: Output for Query 2

-- 3. Calculate total value per item ordered by multiplying quantity and item price
-- This can help generate per-line item bill details or invoice breakdowns.

Query:
```
SELECT order_item_id, quantity, item_price, (quantity*item_price) AS
total_value
FROM order_items;
```
Output:

order_item_id	quantity	item_price	total_value
1	2	4707.14	9414.28
2	3	177.40	532.20
3	3	985.19	2955.57
4	1	2208.99	2208.99
5	2	4734.89	9469.78
6	1	4707.14	4707.14
7	2	4897.36	9794.72
8	3	4897.36	14692.08
9	1	1429.45	1429.45
10	1	723.97	723.97
11	3	4734.89	14204.67
12	1	3043.67	3043.67
13	3	4801.78	14405.34
14	3	4722.66	14167.98
15	2	2070.37	4140.74
16	3	4897.36	14692.08
17	1	396.11	396.11
18	3	3043.67	9131.01

Figure 3: Output for Query 3

```
-- 4. Combine customer name and phone number in a single column
-- Used to show brief customer summaries or contact lists.
```

Query:
```
SELECT CONCAT(name, '-' , phone) AS customer_detail
FROM customers;
```

Output:

customer_detail
Thomas Owens-142-479-1945
Charles Grant-9153947511
Kaitlin Richards-2073473421
Christina Williams-586-605-5061x06
David Allen-(751)456-8289x1
Mark Duke-(144)957-2811
Briana Wright-223-833-9635
John Bryan-045.568.0798x27
Jason Thompson-1862659420
Shawn Hill-(268)113-3152x7
Walter Jenkins-536-329-0817x71
Mary Knight-361-636-3802
Leslie Wilson-+1-256-261-1984
Deborah Arias-811-821-2144x97

Figure 4: Output for Query 4

```
-- 5. Extract only the date part from order timestamps for date-wise
reporting
```

```
-- Helps group or filter orders by date without considering time.
```

Query:
```
SELECT order_id, DATE(order_date) AS date
FROM orders;
```
Output:

order_id	date
1	2025-03-02
2	2024-10-09
3	2025-05-08
4	2024-09-19
5	2025-04-08
6	2024-10-25
7	2024-07-29
8	2024-07-30
9	2025-06-10
10	2025-02-16

Figure 5: Output for Query 5

```
-- 6. List products that do not have any stock left
-- This helps the inventory team identify out-of-stock items.
```

Query:
```
SELECT product_id, name, stock_quantity FROM products
WHERE stock_quantity = 0;
```
Output:

product_id	name	stock_quantity
NULL	NULL	NULL

Figure 6: Output for Query 6

==========
==========

QUESTIONS
Level 3: Aggregations

```
-- 1. Count the total number of orders placed
-- Used by business managers to track order volume over time.
```

Query:
```
SELECT COUNT(order_id) FROM orders;
```
Output:

COUNT(order_id)
400

Figure 1: Output for Query 1

```
-- 2. Calculate the total revenue collected from all orders
-- This gives the overall sales value.
```

Query:
```
SELECT SUM(total_amount) AS total_revenue FROM orders;
```
Output:

total_revenue
6960973.66

Figure 2: Output for Query 2

```
-- 3. Calculate the average order value
-- Used for understanding customer spending patterns.
```

Query:
```
SELECT AVG(total_amount) AS avg_order_value
FROM orders;
```
Output:

avg_order_value
17402.434150

Figure 3: Output for Query 3

```
-- 4. Count the number of customers who have placed at least one order
-- This identifies active customers.
```

Query:
```
SELECT COUNT(DISTINCT customer_id) AS active_customers FROM orders;
```
Output:

active_customers
30

Figure 4: Output for Query 4

```
-- 5. Find the number of orders placed by each customer
-- Helpful for identifying top or repeat customers.
```

Query:
```
SELECT customer_id, COUNT(order_id) AS number_of_orders
        FROM orders
        GROUP BY customer_id
        ORDER BY number_of_orders DESC;
```
Output:

customer_id	number_of_orders
29	20
2	17
3	17
17	17
20	17
28	17
6	16
16	16
30	16
14	15
15	15
19	15
22	15
24	15

Figure 5: Output for Query 5

```
-- 6. Find total sales amount made by each customer
```

Query:
```
SELECT customer_id, SUM(total_amount) AS total_sales
FROM orders
GROUP BY customer_id
ORDER BY total_sales DESC;
```

Output:

customer_id	total_sales
29	435408.89
24	379286.82
28	362584.19
14	360324.18
25	335100.94
2	284420.07
30	281841.38
16	278469.73
5	262504.19
17	262189.71
15	260499.91
3	253783.31
10	252722.75
22	237879.99

Figure 6: Output for Query 6

```
-- 7. List the number of products sold per category
-- This helps category managers assess performance by department.
```

Query:
```
SELECT p.category, SUM(o.quantity) AS total_products_sold FROM products
p
JOIN order_items o ON p.product_id = o.product_id
GROUP BY p.category
ORDER BY total_products_sold DESC;
```

Output:

category	total_products_sold
Electronics	687
Clothing	559
Home	443
Toys	405
Books	350

Figure 7: Output for Query 7

```
-- 8. Find the average item price per category
-- Useful to compare pricing across departments.
```

Query:
```
SELECT category, AVG(price) AS avg_price FROM products
      GROUP BY category
      ORDER BY avg_price DESC;
```

Output:

category	avg_price
Clothing	3434.581538
Books	3167.084286
Electronics	2653.388571
Toys	2516.526250
Home	2146.367500

Figure 8: Output for Query 8

```
-- 9. Show number of orders placed per day
-- Used to track daily business activity and demand trends.
```

Query:
```
SELECT DATE(order_date) AS order_day, COUNT(order_id) AS total_orders
FROM orders
      GROUP BY order_day
      ORDER BY total_orders DESC;
```

Output:

order_day	total_orders
2024-12-07	5
2025-06-06	5
2025-01-15	4
2025-05-05	4
2024-11-10	4
2025-02-07	4
2024-09-23	4
2025-03-02	3
2024-09-19	3
2025-04-08	3
2024-10-25	3
2025-03-11	3
2024-09-09	3
2024-08-24	3

Figure 9: Output for Query 9

```
-- OR
```

Query:
```
SELECT
    DAYNAME(order_date) AS weekday,
    COUNT(order_id) AS total_orders
        FROM
    orders
        GROUP BY weekday
        ORDER BY total_orders DESC;
```

Output:

weekday	total_orders
Friday	65
Wednesday	63
Monday	61
Tuesday	59
Thursday	51
Saturday	51
Sunday	50

Figure 9: Output for Query 9

```
-- 10. List total payments received per payment method
-- Helps the finance team understand preferred transaction modes.
```

Query:
```
SELECT method, SUM(amount_paid) AS total_amount FROM payments
GROUP BY method
ORDER BY total_amount DESC;
```

Output:

method	total_amount
Debit Card	1930577.88
Credit Card	1754603.10
Net Banking	1658383.90
UPI	1617408.78

Figure 10: Output for Query 10

==========

QUESTIONS

Level 4: Multi-Table Queries (JOINS)

```
-- 1. Retrieve order details along with the customer name (INNER JOIN)
-- Used for displaying which customer placed each order.
```

Query:
```
SELECT c.customer_id, c.name, o.order_id, date(o.order_date),
o.total_amount FROM
customers c
JOIN orders o ON c.customer_id = o.customer_id;
```
Output:

customer_id	name	order_id	date(o.order_date)	total_amount
1	Thomas Owens	14	2024-09-24	15803.34
1	Thomas Owens	17	2024-08-19	11173.77
1	Thomas Owens	61	2024-12-26	13053.00
1	Thomas Owens	76	2025-01-14	23506.81
1	Thomas Owens	92	2024-09-26	834.75
1	Thomas Owens	109	2025-03-17	12190.14
1	Thomas Owens	127	2025-06-10	20160.64
1	Thomas Owens	135	2025-03-11	8891.79
1	Thomas Owens	144	2024-09-19	12093.54
1	Thomas Owens	221	2024-09-25	12437.61
1	Thomas Owens	278	2025-02-01	32015.16
1	Thomas Owens	379	2025-03-22	21586.89
2	Charles Grant	56	2024-06-23	6828.61
2	Charles Grant	169	2025-02-11	9426.30
2	Charles Grant	209	2024-10-16	7690.62
2	Charles Grant	223	2024-11-25	41020.75
2	Charles Grant	230	2024-07-25	22655.32
2	Charles Grant	232	2025-05-02	9583.84
2	Charles Grant	248	2025-06-06	9829.50

Figure 1: Output for Query 1

```
-- 2. Get list of products that have been sold (INNER JOIN with order_items)
-- Used to find which products were actually included in orders.
```

Query:
```
SELECT DISTINCT(p.product_id), p.name FROM order_items o
        JOIN products p ON p.product_id = o.product_id;
```

Output:

product_id	name
1	Plant No
2	Population Social
3	Available Answer
4	Any Question
5	Natural Network
6	If Whatever
7	Response Indeed
8	Every Amount
9	Common Study
10	Development System
11	Build Her
12	Action Ask
13	Full West
14	Listen Development
15	Place Low
16	Special Fact
17	Everything Plant
18	Some Them
19	Build High

Figure 2: Output for Query 2

```
-- 3. List all orders with their payment method (INNER JOIN)
-- Used by finance or audit teams to see how each order was paid for.
```

Query:
```
SELECT o.order_id, date(o.order_date) AS order_date, o.total_amount,
p.method FROM orders o
JOIN payments p ON o.order_id = p.order_id;
```

Output:

order_id	order_date	total_amount	method
1	2025-03-02	9414.28	Credit Card
2	2024-10-09	532.20	Net Banking
3	2025-05-08	5164.56	Credit Card
4	2024-09-19	9469.78	UPI
5	2025-04-08	14501.86	UPI
6	2024-10-25	31050.17	UPI
7	2024-07-29	3043.67	UPI
8	2024-07-30	32714.06	Net Banking
9	2025-06-10	24219.20	Net Banking
10	2025-02-16	24342.52	Debit Card
11	2025-03-11	16196.13	Credit Card
12	2025-01-15	9890.72	Credit Card
13	2025-05-12	9627.36	UPI
14	2024-09-24	15803.34	UPI
15	2024-07-18	12421.88	UPI
16	2024-09-09	22856.73	UPI

Figure 3: Output for Query 3

```
-- 4. Get list of customers and their orders (LEFT JOIN)
-- Used to find all customers and see who has or hasn't placed orders.
```

Query:
```
SELECT c.customer_id, c.name, o.order_id, date(o.order_date) AS
order_date FROM customers c
LEFT JOIN orders o
ON c.customer_id = o.customer_id;
```

Output:

customer_id	name	order_id	order_date
1	Thomas Owens	14	2024-09-24
1	Thomas Owens	17	2024-08-19
1	Thomas Owens	61	2024-12-26
1	Thomas Owens	76	2025-01-14
1	Thomas Owens	92	2024-09-26
1	Thomas Owens	109	2025-03-17
1	Thomas Owens	127	2025-06-10
1	Thomas Owens	135	2025-03-11
1	Thomas Owens	144	2024-09-19
1	Thomas Owens	221	2024-09-25
1	Thomas Owens	278	2025-02-01
1	Thomas Owens	379	2025-03-22
2	Charles Grant	56	2024-06-23
2	Charles Grant	169	2025-02-11
2	Charles Grant	209	2024-10-16
2	Charles Grant	223	2024-11-25
2	Charles Grant	230	2024-07-25

Figure 4: Output for Query 4

```
-- 5. List all products along with order item quantity (LEFT JOIN)
-- Useful for inventory teams to track what sold and what hasn't.
```

Query:
```
SELECT p.product_id, p.name, p.category, SUM(oi.quantity) AS
total_quantity FROM products p
LEFT JOIN order_items oi ON p.product_id = oi.product_id
GROUP BY p.product_id,  p.name, p.category;
```

Output:

product_id	name	category	total_quantity
1	Plant No	Home	43
2	Population Social	Clothing	33
3	Available Answer	Electronics	47
4	Any Question	Clothing	30
5	Natural Network	Toys	44
6	If Whatever	Electronics	54
7	Response Indeed	Clothing	39
8	Every Amount	Home	61
9	Common Study	Toys	57
10	Development System	Electronics	53
11	Build Her	Books	50
12	Action Ask	Electronics	47
13	Full West	Books	46
14	Listen Development	Home	43

Figure 5: Output for Query 5

```
-- 6. List all payments including those with no matching orders (RIGHT JOIN)
-- Rare but used when ensuring all payments are mapped correctly.
```

Query:
```
SELECT o.order_id, date(o.order_date), p.payment_id,
date(p.payment_date) AS payment_date FROM orders o
RIGHT JOIN payments p ON p.order_id = o.order_id;
```

Output:

order_id	date(o.order_date)	payment_id	payment_date
1	2025-03-02	1	2025-03-02
2	2024-10-09	2	2024-10-09
3	2025-05-08	3	2025-05-08
4	2024-09-19	4	2024-09-19
5	2025-04-08	5	2025-04-08
6	2024-10-25	6	2024-10-25
7	2024-07-29	7	2024-07-29
8	2024-07-30	8	2024-07-30
9	2025-06-10	9	2025-06-10
10	2025-02-16	10	2025-02-16
11	2025-03-11	11	2025-03-11
12	2025-01-15	12	2025-01-15
13	2025-05-12	13	2025-05-12
14	2024-09-24	14	2024-09-24

Figure 6: Output for Query 6

```
-- 7. Combine data from three tables: customer, order, and payment
-- Used for detailed transaction reports.
```

Query:
```
SELECT c.customer_id, c.name, o.order_id, date(o.order_date) AS
order_date, p.payment_id, p.amount_paid FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
LEFT JOIN payments p ON o.order_id = p.order_id;
```

Output:

customer_id	name	order_id	order_date	payment_id	amount_paid
1	Thomas Owens	14	2024-09-24	14	15803.34
1	Thomas Owens	17	2024-08-19	17	11173.77
1	Thomas Owens	61	2024-12-26	61	13053.00
1	Thomas Owens	76	2025-01-14	76	23506.81
1	Thomas Owens	92	2024-09-26	92	834.75
1	Thomas Owens	109	2025-03-17	109	12190.14
1	Thomas Owens	127	2025-06-10	127	20160.64
1	Thomas Owens	135	2025-03-11	135	8891.79
1	Thomas Owens	144	2024-09-19	144	12093.54
1	Thomas Owens	221	2024-09-25	221	12437.61
1	Thomas Owens	278	2025-02-01	278	32015.16
1	Thomas Owens	379	2025-03-22	379	21586.89
2	Charles Grant	56	2024-06-23	56	6828.61
2	Charles Grant	169	2025-02-11	169	9426.30
2	Charles Grant	209	2024-10-16	209	7690.62
2	Charles Grant	223	2024-11-25	223	41020.75

Figure 7: Output for Query 7

==========
==========

QUESTIONS

Level 5: Subqueries (Inner Queries)

-- 1. List all products priced above the average product price
-- Used by pricing analysts to identify premium-priced products.

Query:
```
SELECT product_id, name, category, price
FROM products
WHERE price > (SELECT AVG(price) FROM products)
ORDER BY price;
```
Output:

product_id	name	category	price
30	Future Sort	Home	3043.67
49	Rest Something	Books	3077.82
24	Assume Serve	Home	3437.81
26	Movement Future	Electronics	3502.62
25	South Free	Clothing	3543.58
45	Suffer We	Electronics	3657.43
18	Some Them	Toys	3673.86
46	Age Treatment	Home	3728.83
21	Bed Including	Clothing	3845.31
42	Weight Enter	Clothing	3875.18
12	Action Ask	Electronics	4017.01
39	Data Add	Books	4063.38
29	Drop Remember	Electronics	4160.45
22	Move Small	Books	4168.08
8	Every Amount	Home	4173.60

Figure 1: Output for Query 1

-- 2. Find customers who have placed at least one order
-- Used to identify active customers for loyalty campaigns.

Query:
```
SELECT customer_id, name FROM customers
WHERE customer_id IN (SELECT customer_id FROM orders);
```
Output:

customer_id	name
1	Thomas Owens
2	Charles Grant
3	Kaitlin Richards
4	Christina Williams
5	David Allen
6	Mark Duke
7	Briana Wright
8	John Bryan
9	Jason Thompson
10	Shawn Hill
11	Walter Jenkins
12	Mary Knight
13	Leslie Wilson

Figure 2: Output for Query 2

```
-- 3. Show orders whose total amount is above the average for that customer
-- Used to detect unusually high purchases per customer.
```

Query:
```
SELECT *
FROM (
    SELECT
        o.order_id,
        o.customer_id,
        SUM(oi.quantity * oi.item_price) AS order_total,
        AVG(SUM(oi.quantity * oi.item_price)) OVER (PARTITION BY
o.customer_id) AS avg_order_total
    FROM orders o
    JOIN order_items oi ON o.order_id = oi.order_id
    GROUP BY o.order_id, o.customer_id
) t
WHERE t.order_total > t.avg_order_total;
```

Output:

order_id	customer_id	order_total	avg_order_total
14	1	15803.34	15312.286667
76	1	23506.81	15312.286667
127	1	20160.64	15312.286667
278	1	32015.16	15312.286667
379	1	21586.89	15312.286667
223	2	41020.75	16730.592353
230	2	22655.32	16730.592353
315	2	42056.04	16730.592353
329	2	31387.01	16730.592353
334	2	27857.86	16730.592353
394	2	18154.30	16730.592353
42	3	21622.22	14928.430000
63	3	36236.03	14928.430000
134	3	15483.53	14928.430000
234	3	18473.60	14928.430000
281	3	41679.11	14928.430000
369	3	33503.41	14928.430000

Figure 3: Output for Query 3

```
-- 4. Display customers who haven't placed any orders
-- Used for re-engagement campaigns targeting inactive users.
```

Query:
```
SELECT c.customer_id, c.name
        FROM customers c
        LEFT JOIN orders o ON c.customer_id = o.customer_id
        WHERE o.order_id IS NULL;
```

Output:

customer_id	name

Figure 4: Output for Query 4

```
-- 5. Show products that were never ordered
-- Helps with inventory clearance decisions or product deactivation.
```

Query:
```
SELECT product_id, name
        FROM products
        WHERE product_id NOT IN (
        SELECT DISTINCT product_id
        FROM order_items);
```

Output:

product_id	name
NULL	NULL

Figure 5: Output for Query 5

```
-- 6. Show highest value order per customer
-- Used to identify the largest transaction made by each customer.
```

Query:
```
SELECT DISTINCT(o.customer_id),
        MAX(oi.quantity * oi.item_price) OVER (PARTITION BY o.customer_id) AS
max_order_value
    FROM orders o
    JOIN order_items oi ON o.order_id = oi.order_id;
```

Output:

customer_id	max_order_value
1	14167.98
2	14692.08
3	14441.04
4	14692.08
5	12540.69
6	14692.08
7	10507.86
8	13195.98
9	14167.98
10	13569.30
11	12732.27
12	14167.98
13	14441.04
14	14441.04
15	14405.34

Figure 6: Output for Query 6

```
-- 7. Highest Order Per Customer (Including Names)
-- Used to identify the largest transaction made by each customer. Outputs
name as well.
```

Query:

```
SELECT
    ot.customer_id,
    ot.name,
    MAX(ot.order_total) AS max_order_value
FROM (
    SELECT
        o.customer_id,
        c.name,
        o.order_id,
        SUM(oi.quantity * oi.item_price) AS order_total
    FROM orders o
    JOIN order_items oi ON o.order_id = oi.order_id
    JOIN customers c ON c.customer_id = o.customer_id
    GROUP BY o.customer_id, o.order_id, c.name
) AS ot
GROUP BY ot.customer_id, ot.name;
```

Output:

customer_id	name	max_order_value
1	Thomas Owens	32015.16
2	Charles Grant	42056.04
3	Kaitlin Richards	41679.11
4	Christina Williams	25747.34
5	David Allen	39921.78
6	Mark Duke	39003.19
7	Briana Wright	28589.04
8	John Bryan	36147.09
9	Jason Thompson	21414.44
10	Shawn Hill	35723.17
11	Walter Jenkins	37129.82
12	Mary Knight	38656.26
13	Leslie Wilson	32001.24
14	Deborah Arias	48042.06
15	Austin Flores	40510.54
16	Amy Landry	37415.67
17	Randy Mooney	40944.10

Figure7: Output for Query 7

==========
==========

QUESTIONS

Level 6: SET Operations

-- Q1. List all unique product categories and customer names in a single column.

Query:
```
SELECT category AS info FROM products
    UNION
    SELECT name AS info FROM customers;
```

Output:

info
Home
Clothing
Electronics
Toys
Books
Thomas Owens
Charles Grant
Kaitlin Richards
Christina Williams
David Allen
Mark Duke
Briana Wright
John Bryan
Jason Thompson
Shawn Hill

Figure 1: Output for Query 1

```
-- Q2. List all order IDs and payment IDs together, keeping duplicates if any exist.
```

Query:
```
SELECT order_id AS info FROM orders
        UNION ALL
        SELECT payment_id AS info FROM payments;
```

Output:

info
14
17
61
76
92
109
127
135
144
221
278
379

Figure 2: Output for Query 2

Assumptions & Limitations

Assumptions

1. The dataset represents a **simplified retail scenario** with six core tables: customers, products, orders, order_items, payments, and product_reviews.
2. All dates are stored and interpreted in the **server's local timezone**.
3. Order amounts are assumed to be **final (no partial returns or discounts)** unless explicitly mentioned.
4. Payments are recorded only for completed or confirmed orders.
5. Reviews are assumed to be genuine and linked to valid customers and products.

Limitations

1. The dataset is **sample-sized (~100–200 rows per table)** and may not fully represent real-world complexities.
2. Advanced business cases like **refunds, cancellations, and loyalty programs** are not included.
3. Currency is assumed to be uniform (single currency system).
4. Some queries (especially those using **window functions**) require **MySQL 8.x or above**.

5. The focus is on **query correctness and learning SQL concepts**, not on database optimization or indexing.