E-COMMERCE DATABASE MANAGEMENT SYSTEM

NAME :- SANYUKTA NANDKUMAR GADGE

TASK -3 :- SQL for Data Analysis

1. INTRODUCTION : An E-commerce Database Management System (DBMS) is a structured collection of data that stores and manages all information related to an online store. It handles users, products, orders, payments, and categories, enabling efficient and secure transactions between buyers and sellers.

2. QUERY PLANNING

   NUMBER OF OBJECTIVES:

   1. SELECT, WHERE, ORDER BY, GROUP BY
   2. Get all orders with user names(Inner Join)
   3. List all users, along with any orders they have placed(Left Join)
   4. List all orders, and show user details if available(Right Join)
   5. Get users who have placed at least one order
   6. Get products with a price higher than the average price
   7. Get names of users who bought a specific product (e.g., product_id = 1)
   8. Total revenue (SUM of all payments)
   9. Average order amount
   10. Total quantity of items sold (from order_items)

11. Average price of products

12. Total revenue per user (using GROUP BY)

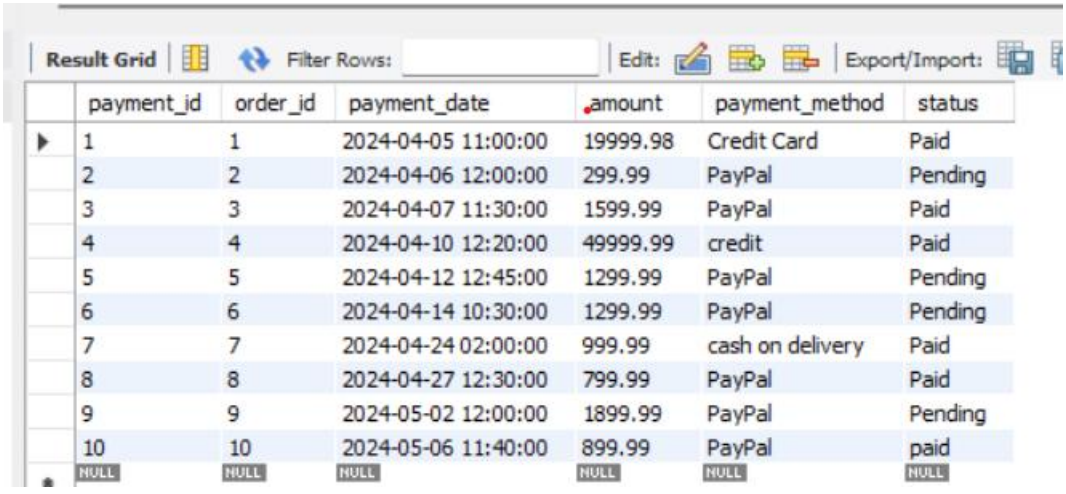
13. Create views for analysis
    1. user_order_summary:
       Summarizes each user's total number of orders and total amount spent
    2. product_sales_summary:
       Summarizes how much of each product has been sold and total revenue.
    3. category_sales_summary:
    4. Shows total revenue and products sold per category.

14. Optimize queries with indexes

3. QUERY CONSTRUCTION

Query 1: SELECT, WHERE, ORDER BY, GROUP BY

1: select * from payments;

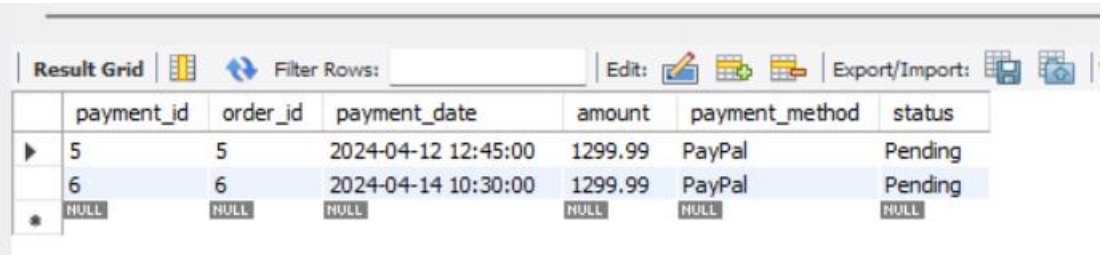


Result Grid | Filter Rows: | Edit: | Export/Import:

payment_id	order_id	payment_date	amount	payment_method	status
1	1	2024-04-05 11:00:00	19999.98	Credit Card	Paid
2	2	2024-04-06 12:00:00	299.99	PayPal	Pending
3	3	2024-04-07 11:30:00	1599.99	PayPal	Paid
4	4	2024-04-10 12:20:00	49999.99	credit	Paid
5	5	2024-04-12 12:45:00	1299.99	PayPal	Pending
6	6	2024-04-14 10:30:00	1299.99	PayPal	Pending
7	7	2024-04-24 02:00:00	999.99	cash on delivery	Paid
8	8	2024-04-27 12:30:00	799.99	PayPal	Paid
9	9	2024-05-02 12:00:00	1899.99	PayPal	Pending
10	10	2024-05-06 11:40:00	899.99	PayPal	paid
NULL	NULL	NULL	NULL	NULL	NULL

2: FIND AMOUNT FROM PAYMENT TABLE WHERE AMOUNT IS 1299.99

SELECT * FROM PAYMENTS WHERE AMOUNT = 1299.99;



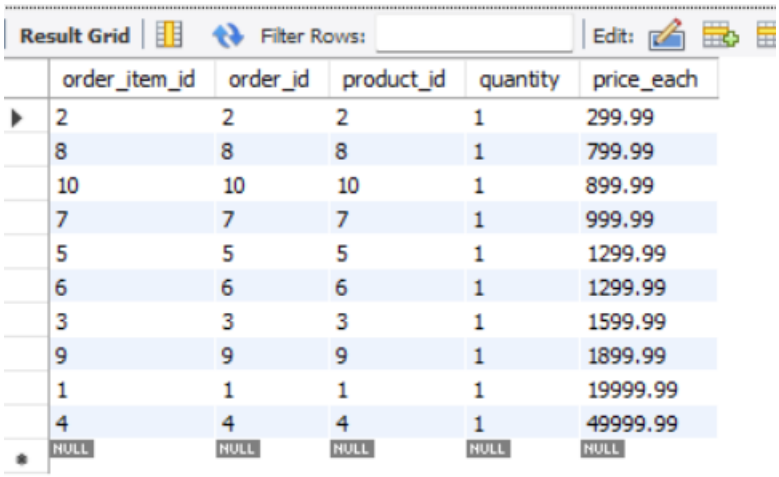
Result Grid | Filter Rows: | Edit: | Export/Import:

payment_id	order_id	payment_date	amount	payment_method	status
5	5	2024-04-12 12:45:00	1299.99	PayPal	Pending
6	6	2024-04-14 10:30:00	1299.99	PayPal	Pending
NULL	NULL	NULL	NULL	NULL	NULL

3: ORDER BY ASCENDING

SELECT * FROM ORDER_ITEMS

ORDER BY PRICE_EACH ASC;



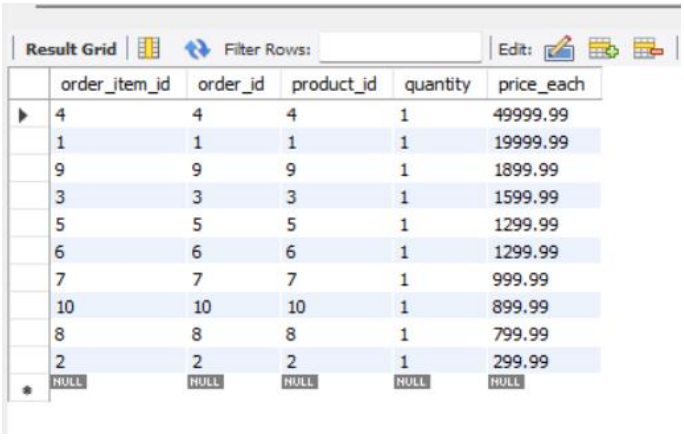
Result Grid | Filter Rows: | Edit:

order_item_id	order_id	product_id	quantity	price_each
2	2	2	1	299.99
8	8	8	1	799.99
10	10	10	1	899.99
7	7	7	1	999.99
5	5	5	1	1299.99
6	6	6	1	1299.99
3	3	3	1	1599.99
9	9	9	1	1899.99
1	1	1	1	19999.99
4	4	4	1	49999.99
NULL	NULL	NULL	NULL	NULL

4: ORDER BY DESCENDING

SELECT * FROM ORDER_ITEMS

ORDER BY PRICE_EACH desc;



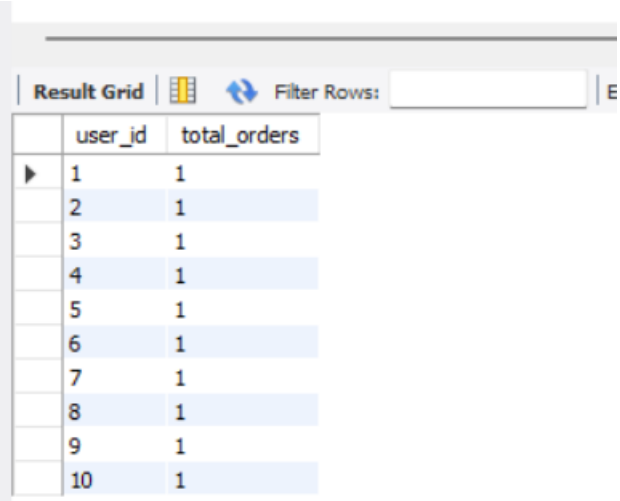
Result Grid | Filter Rows: | Edit:

order_item_id	order_id	product_id	quantity	price_each
4	4	4	1	49999.99
1	1	1	1	19999.99
9	9	9	1	1899.99
3	3	3	1	1599.99
5	5	5	1	1299.99
6	6	6	1	1299.99
7	7	7	1	999.99
10	10	10	1	899.99
8	8	8	1	799.99
2	2	2	1	299.99
NULL	NULL	NULL	NULL	NULL

5: GROUP BY

Total Sales per Product

```
SELECT user_id, COUNT(order_id) AS total_orders
FROM orders
GROUP BY user_id;
```



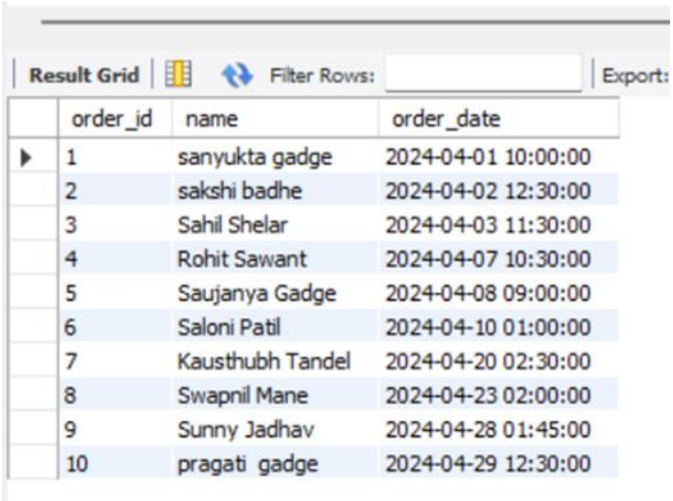
Result Grid | Filter Rows:

user_id	total_orders
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1

Query 2: Get all orders with user names(Inner Join)

```
SELECT orders.order_id, users.name, orders.order_date
FROM orders
INNER JOIN users ON orders.user_id = users.user_id;
```

Explaination: This query fetches the order ID, user name, and order date by joining the orders and users tables. It uses an INNER JOIN, so it only shows orders that have a matching user in the users table.



Result Grid | Filter Rows: | Export:

order_id	name	order_date
1	sanyukta gadge	2024-04-01 10:00:00
2	sakshi badhe	2024-04-02 12:30:00
3	Sahil Shelar	2024-04-03 11:30:00
4	Rohit Sawant	2024-04-07 10:30:00
5	Saujanya Gadge	2024-04-08 09:00:00
6	Saloni Patil	2024-04-10 01:00:00
7	Kausthubh Tandel	2024-04-20 02:30:00
8	Swapnil Mane	2024-04-23 02:00:00
9	Sunny Jadhav	2024-04-28 01:45:00
10	pragati gadge	2024-04-29 12:30:00

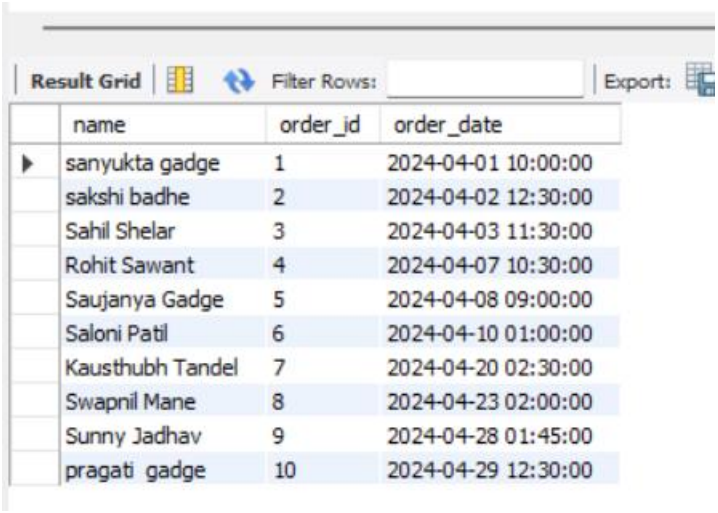
Query 3: List all users, along with any orders they have placed(Left Join)

```
SELECT users.name, orders.order_id, orders.order_date

FROM users

LEFT JOIN orders ON users.user_id = orders.user_id;
```

Explanation: This query shows all users and their orders, if any.
It uses a LEFT JOIN, so users without orders will still appear, but with NULL for order_id and order_date.



Result Grid | Filter Rows: | Export:

name	order_id	order_date
sanyukta gadge	1	2024-04-01 10:00:00
sakshi badhe	2	2024-04-02 12:30:00
Sahil Shelar	3	2024-04-03 11:30:00
Rohit Sawant	4	2024-04-07 10:30:00
Saujanya Gadge	5	2024-04-08 09:00:00
Saloni Patil	6	2024-04-10 01:00:00
Kausthubh Tandel	7	2024-04-20 02:30:00
Swapnil Mane	8	2024-04-23 02:00:00
Sunny Jadhav	9	2024-04-28 01:45:00
pragati gadge	10	2024-04-29 12:30:00

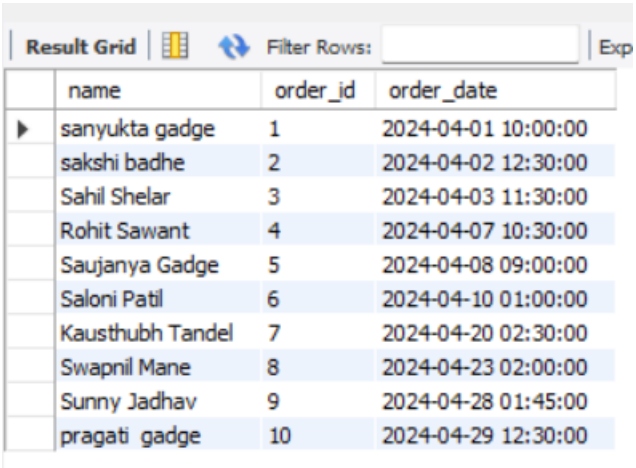
Query 4 :List all orders, and show user details if available(Right Join)

```
SELECT users.name, orders.order_id, orders.order_date

FROM users

RIGHT JOIN orders ON users.user_id = orders.user_id;
```

Explaination : This query lists all orders, along with the user's name if available.
It uses a RIGHT JOIN, so even if a user is missing, the order still shows up with NULL for the user name.



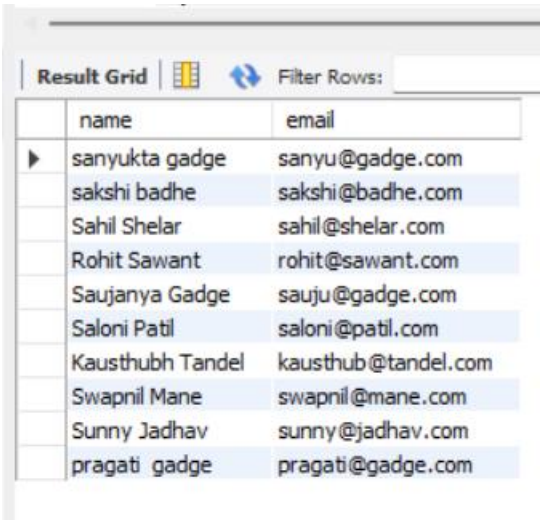
Result Grid | Filter Rows: | Exp

name	order_id	order_date
sanyukta gadge	1	2024-04-01 10:00:00
sakshi badhe	2	2024-04-02 12:30:00
Sahil Shelar	3	2024-04-03 11:30:00
Rohit Sawant	4	2024-04-07 10:30:00
Saujanya Gadge	5	2024-04-08 09:00:00
Saloni Patil	6	2024-04-10 01:00:00
Kausthubh Tandel	7	2024-04-20 02:30:00
Swapnil Mane	8	2024-04-23 02:00:00
Sunny Jadhav	9	2024-04-28 01:45:00
pragati gadge	10	2024-04-29 12:30:00

Query 5: Get users who have placed at least one order (sub query)

```
SELECT name, email
FROM users
WHERE user_id IN (
    SELECT DISTINCT user_id
    FROM orders);
```

Explanation : This query lists the name and email of users who have placed at least one order.It uses a subquery to find user_ids from the orders table.



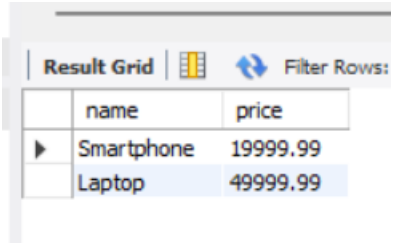
Result Grid | Filter Rows:

name	email
sanyukta gadge	sanyu@gadge.com
sakshi badhe	sakshi@badhe.com
Sahil Shelar	sahil@shelar.com
Rohit Sawant	rohit@sawant.com
Saujanya Gadge	sauju@gadge.com
Saloni Patil	saloni@patil.com
Kausthubh Tandel	kausthub@tandel.com
Swapnil Mane	swapnil@mane.com
Sunny Jadhav	sunny@jadhav.com
pragati gadge	pragati@gadge.com

Query 6: Get products with a price higher than the average price(sub query)

```
SELECT name, price
FROM products
WHERE price > (
    SELECT AVG(price)
    FROM products);
```

Explanation: This query shows products that have a price higher than the average product price.It uses a subquery to calculate the average price first.



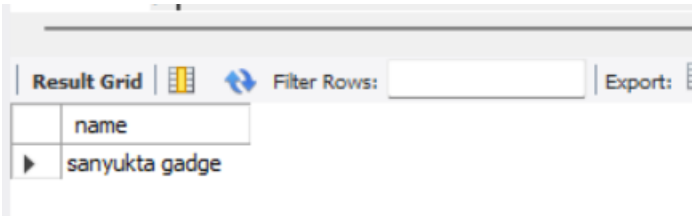
Result Grid | Filter Rows:

name	price
Smartphone	19999.99
Laptop	49999.99

Query 6: Get names of users who bought a specific product (e.g., product_id = 1)(sub query)

```
SELECT name
FROM users
WHERE user_id IN (
    SELECT o.user_id
    FROM orders o
    JOIN order_items oi ON o.order_id = oi.order_id
    WHERE oi.product_id = 1);
```

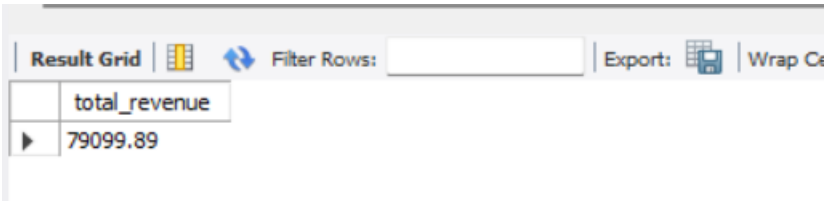
Explanation: This query shows the names of users who bought product with ID 1.
It uses a subquery with JOIN to link orders and order items.



Query 7: Total revenue (SUM of all payments)

```
SELECT SUM(amount) AS total_revenue
FROM payments;
```

Explanation:This query calculates the total revenue by adding up all payment amounts using the SUM() function.

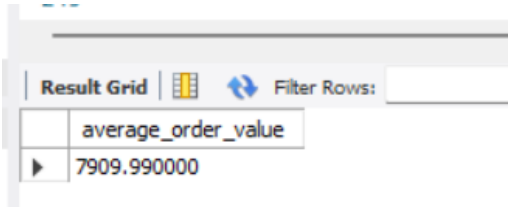


Query 8: Average order amount

SELECT AVG(total_amount) AS average_order_value

FROM orders;

Explanation: This query calculates the average value of all orders using the AVG() function on total_amount.

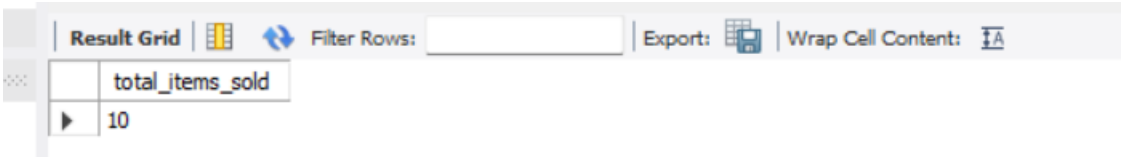


Query 9: Total quantity of items sold (from order_items)

SELECT SUM(quantity) AS total_items_sold

FROM order_items;

Explanation :This query gives the **total number of items sold** by summing up the quantity from all order items.

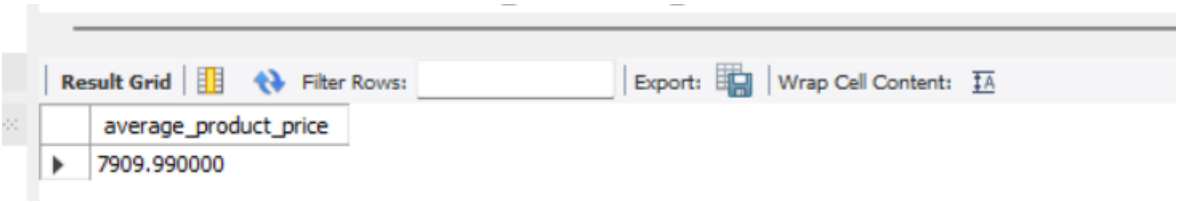


Query 9: Average price of products

SELECT AVG(price) AS average_product_price

FROM products;

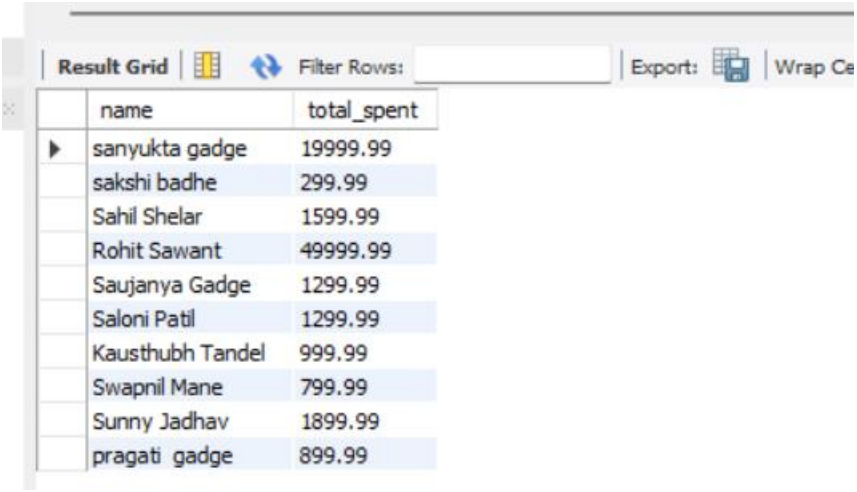
Ecplanation: This version omits the alias (AS average_product_price) and will return the average price directly.



Query 10: Total revenue per user (using GROUP BY)

```
SELECT u.name, SUM(o.total_amount) AS total_spent
FROM users u
JOIN orders o ON u.user_id = o.user_id
GROUP BY u.user_id, u.name;
```

Explanation: The query sums the total amount spent by each user by joining the users and orders tables, then grouping by user_id. It shows the total revenue per user.


Result Grid | Filter Rows: | Export: | Wrap Ce

name	total_spent
sanyukta gadge	19999.99
sakshi badhe	299.99
Sahil Shelar	1599.99
Rohit Sawant	49999.99
Saujanya Gadge	1299.99
Saloni Patil	1299.99
Kausthubh Tandel	999.99
Swapnil Mane	799.99
Sunny Jadhav	1899.99
pragati gadge	899.99

Query 11: user_order_summary

Summarizes each user's total number of orders and total amount spent.

```
CREATE VIEW user_order_summary AS
SELECT
    u.user_id,
    u.name,
    COUNT(o.order_id) AS total_orders,
    SUM(o.total_amount) AS total_spent
FROM users u
LEFT JOIN orders o ON u.user_id = o.user_id
GROUP BY u.user_id, u.name;
```

Explanation

Query 12: product_sales_summary

Summarizes how much of each product has been sold and total revenue.

```
CREATE VIEW product_sales_summary AS
SELECT
    p.product_id,
    p.name,
    SUM(oi.quantity) AS total_quantity_sold,
    SUM(oi.quantity * oi.price_each) AS total_revenue
FROM products p
LEFT JOIN order_items oi ON p.product_id = oi.product_id
GROUP BY p.product_id, p.name;
```

Explanation : This query creates a view summarizing each product's total quantity sold and total revenue, using a LEFT JOIN with order_items and grouping by product_id and name.

Query 13: category_sales_summary

Shows total revenue and products sold per category.

```
CREATE VIEW category_sales_summary AS
SELECT
    c.category_id,
    c.name AS category_name,
    SUM(oi.quantity) AS total_items_sold,
    SUM(oi.quantity * oi.price_each) AS category_revenue
FROM categories c
JOIN products p ON c.category_id = p.category_id
JOIN order_items oi ON p.product_id = oi.product_id
GROUP BY c.category_id, c.name;
```

Explanation: This query creates a view showing total items sold and revenue per category by joining categories, products, and order_items, grouped by category_id and category_name.

Query 14: Optimize queries with indexes

```
CREATE INDEX idx_orders_user_id ON orders(user_id);

CREATE INDEX idx_order_items_order_id ON order_items(order_id);

CREATE INDEX idx_order_items_product_id ON order_items(product_id);

CREATE INDEX idx_products_category_id ON products(category_id);

CREATE INDEX idx_payments_order_id ON payments(order_id);
```

Explanation: These queries create indexes on key columns to speed up queries involving user_id, order_id, product_id, and category_id across relevant tables.

ERROR HANDLING & PERFORMANCE CONSIDERATION:

1. Use TRY...CATCH blocks for handling SQL errors (in supported databases like SQL Server).
2. Implement proper constraints (NOT NULL, CHECK, FOREIGN KEY) to ensure data integrity.
3. Log errors using a separate error_log table or built-in error logging mechanisms.
4. Use indexing on frequently queried columns (e.g., user_id, order_id, product_id).
5. Optimize queries by avoiding SELECT *, using joins efficiently, and applying WHERE filters.
6. Normalize data to avoid redundancy but consider denormalization for read-heavy operations.
7. Regularly update statistics and rebuild indexes for maintaining performance.