**SQL Questions**

1. **Basic Select Query:**
   * **Question:** Retrieve all details of customers who have placed orders.
   * **Task:** Write a SQL query to join the customers and orders tables and select relevant customer details.

**Query:**

**SELECT c.customer\_id, c.first\_name, c.last\_name, c.email**

**FROM customers c**

**JOIN orders o ON c.customer\_id = o.customer\_id;**

* **SELECT** clause: Specifies the columns you want to retrieve from the joined tables. In this case, we're selecting customer\_id, first\_name, last\_name, and email from the customers table.
* **FROM** clause: Specifies the tables involved in the join. We're joining the customers and orders tables.
* **JOIN** clause: Specifies the type of join and the condition for joining the tables. An INNER JOIN is used here, which returns only the rows that have matching values in both tables. The condition c.customer\_id = o.customer\_id ensures that only customers who have placed orders are included in the result.

This query will return a list of customers who have at least one order in the orders table, along with their first name, last name, and email address.

1. **Aggregate Functions:**
   * **Question:** Calculate the total sales for each store.
   * **Task:** Use aggregate functions like SUM and GROUP BY to calculate total sales from the orders and order\_items tables.

Query:

SELECT s.store\_id, SUM(oi.quantity \* oi.list\_price) AS total\_sales

FROM stores s

JOIN orders o ON s.store\_id = o.store\_id

JOIN order\_items oi ON o.order\_id = oi.order\_id

GROUP BY s.store\_id;

Explanation:

1. **SELECT s.store\_id, SUM(oi.quantity \* oi.list\_price) AS total\_sales:**
   * Selects the store\_id from the stores table and calculates the total sales for each store.
   * The SUM(oi.quantity \* oi.list\_price) expression calculates the total sales by multiplying the quantity of each item ordered by its list price and summing the results for all items in each store.
   * The calculated total sales are assigned the alias total\_sales.
2. **FROM stores s:**
   * Specifies that the query will be performed on the stores table, using the alias s for the table name.
3. **JOIN orders o ON s.store\_id = o.store\_id:**
   * Joins the stores table with the orders table based on the store\_id column. This means that only orders with a matching store\_id in the stores table will be included in the calculation.
4. **JOIN order\_items oi ON o.order\_id = oi.order\_id:**
   * Joins the orders table with the order\_items table based on the order\_id column. This means that only order items with a matching order\_id in the orders table will be included in the calculation.
5. **GROUP BY s.store\_id:**
   * Groups the results by the store\_id column. This means that the total sales will be calculated separately for each unique store.

**Output:** The query will return a result set with two columns:

* **store\_id:** The unique identifier for each store.
* **total\_sales:** The total sales for that store, calculated by summing the product quantity and list price for all items sold in that store.

1. **Subqueries:**
   1. **Question:** Find the products that have never been ordered.
   2. **Task:** Write a subquery to identify products in the products table that do not appear in the order\_items table.

Query:

SELECT p.product\_id, p.product\_name

FROM products p

WHERE p.product\_id NOT IN (

SELECT oi.product\_id

FROM order\_items oi);

Explanation:

* The subquery retrieves a list of all product IDs that have been ordered.
* The NOT IN operator checks if the product\_id of each product in the products table is present in the list of ordered product IDs.
* If a product\_id is not found in the list, it means that the product has never been ordered.

**Output:** The query will return a result set with two columns:

* **product\_id:** The unique identifier for the product.
* **product\_name:** The name of the product.

1. **Joins:**
   1. **Question:** Retrieve the names and email addresses of staff along with the names of their managers.
   2. **Task:** Write a SQL query to join the staffs table with itself to get staff and their manager information.

Query:

SELECT s1.staff\_id, s1.first\_name, s1.last\_name, s1.email, s2.first\_name AS manager\_first\_name, s2.last\_name AS manager\_last\_name

FROM staffs s1

LEFT JOIN staffs s2 ON s1.manager\_id = s2.staff\_id;

Explanation:

1. **SELECT s1.staff\_id, s1.first\_name, s1.last\_name, s1.email, s2.first\_name AS manager\_first\_name, s2.last\_name AS manager\_last\_name:**
   * Select the staff\_id, first\_name, last\_name, and email of the staff from the first instance of the staffs table (aliased as s1).
   * Also selects the first\_name and last\_name of the manager from the second instance of the staffs table (aliased as s2). The AS keyword is used to give the columns aliases for clarity.
2. **FROM staffs s1:**
   * Specifies that the query will be performed on the staffs table, using the alias s1 for the table name.
3. **LEFT JOIN staffs s2 ON s1.manager\_id = s2.staff\_id:**
   * Performs a left join between the staffs table with itself. This ensures that all staff members are included in the result, even if they don't have a manager assigned.
   * The join condition s1.manager\_id = s2.staff\_id matches the manager\_id of one staff member with the staff\_id of another, effectively connecting staff to their managers.

**Output:** The query will return a result set with the following columns:

* **staff\_id:** The unique identifier for the staff member.
* **first\_name:** The first name of the staff member.
* **last\_name:** The last name of the staff member.
* **email:** The email address of the staff member.
* **manager\_first\_name:** The first name of the staff member's manager, or NULL if the staff member has no manager assigned.
* **manager\_last\_name:** The last name of the staff member's manager, or NULL if the staff member has no manager assigned.

This query provides a valuable tool for understanding the hierarchy and relationships within a company's staff structure.

1. Question: Rank stores based on their total sales.

Task: Use window functions like ROW\_NUMBER() or RANK() to rank stores based on total sales.

Query:

SELECT s.store\_id, SUM(oi.quantity \* oi.list\_price) AS total\_sales,

RANK() OVER (ORDER BY SUM(oi.quantity \* oi.list\_price) DESC) AS sales\_rank

FROM stores s

JOIN orders o ON s.store\_id = o.store\_id

JOIN order\_items oi ON o.order\_id = oi.order\_id

GROUP BY s.store\_id;

Explanation:

* Calculate Total Sales: The inner query calculates the total sales for each store using the same logic as the previous response:

Joins the stores, orders, and order\_items tables.

Calculates the total sales for each store by multiplying quantity and list price of items.

Groups the results by store\_id.

* Assign Ranks:

The outer query applies the RANK() window function to the calculated total sales:

RANK() OVER (ORDER BY SUM(oi.quantity \* oi.list\_price) DESC) assigns a rank to each store based on its total sales, with the highest-selling store having a rank of 1.

* The DESC keyword ensures that the ranking is in descending order, so the store with the highest total sales is ranked first.

Output:

The query will return a result set with three columns:

store\_id: The unique identifier for the store.

total\_sales: The total sales for that store.

sales\_rank: The rank of the store based on its total sales.

Categorizing Orders Based on Status Using CASE Statements

Question: Categorize orders based on their status.

Task: Use CASE statements to categorize orders in the orders table into different status groups.

Query:

SELECT order\_id, order\_status, CASE

WHEN order\_status = 'pending' THEN 'Pending Order'

WHEN order\_status = 'processing' THEN 'Processing Order'

WHEN order\_status = 'shipped' THEN 'Shipped Order'

ELSE 'Unknown Status'

END AS status\_category

FROM orders;

Explanation

SELECT order\_id, order\_status, CASE ... END AS status\_category:

* Selects the order\_id and order\_status columns from the orders table.

The CASE expression is used to conditionally categorize orders based on their order\_status.

WHEN order\_status = 'pending' THEN 'Pending Order':

* If the order\_status is 'pending', the status\_category is set to 'Pending Order'.

WHEN order\_status = 'processing' THEN 'Processing Order':

* If the order\_status is 'processing', the status\_category is set to 'Processing Order'.

WHEN order\_status = 'shipped' THEN 'Shipped Order':

* If the order\_status is 'shipped', the status\_category is set to 'Shipped Order'.

ELSE 'Unknown Status':

* If the order\_status doesn't match any of the previous conditions, the status\_category is set to 'Unknown Status'.

Output:

The query will return a result set with three columns:

order\_id: The unique identifier for the order.

order\_status: The original order status.

status\_category: The categorized order status.

Complex Joins:

Question: Retrieve all orders along with the product names and the store names.

Task: Write a SQL query to join the orders, order\_items, products, and stores tables.

Query:

SELECT o.order\_id, o.order\_date, p.product\_name, s.store\_name

FROM orders o

JOIN order\_items oi ON o.order\_id = oi.order\_id

JOIN products p ON oi.product\_id = p.product\_id

JOIN stores s ON o.store\_id = s.store\_id;

Explanation:

* The query starts with the orders table, using the alias o.
* It then joins the order\_items table using the order\_id column, ensuring that only the items associated with each order are included.
* Next, the products table is joined using the product\_id column, linking each order item with its corresponding product.
* Finally, the stores table is joined using the store\_id column, connecting each order with the store where it was placed.

Output:

The query will return a result set with the following columns:

* order\_id: The unique identifier for the order.
* order\_date: The date the order was placed.
* product\_name: The name of the product ordered.
* store\_name: The name of the store where the order was placed.
* This query provides a comprehensive view of all orders, including their associated products and store locations.