1. Write a Query to Find Duplicate Rows in a Table

Answer:
To find duplicate rows in a table, group by the columns that define a duplicate and use the `HAVING` clause to filter groups with more than one occurrence.

Example:
Suppose you have a table called `employees` with columns `first_name`, `last_name`, and `email`. To find duplicates based on `first_name` and `last_name`:

```
SELECT
    first_name,
    last_name,
    COUNT(*) AS duplicate_count
FROM
    employees
GROUP BY
    first_name,
    last_name
HAVING
    COUNT(*) > 1;
```

Explanation:

- **GROUP BY** groups rows with the same `first_name` and `last_name`.
- **COUNT(*)** counts the number of occurrences for each group.
- **HAVING COUNT(*) > 1** filters only those groups that have duplicates.

> **Tip:** Adjust the columns in the `GROUP BY` clause to match the definition of a duplicate in your specific table.

2. Explain the Difference Between INNER JOIN and OUTER JOIN with Examples

Answer:
INNER JOIN returns only the rows that have matching values in both tables.
OUTER JOIN returns all rows from one or both tables, filling in NULLs where there is no match.

Example:
Suppose you have two tables: `employees` and `departments`.

- `employees(employee_id, name, department_id)`
- `departments(department_id, department_name)`

INNER JOIN Example: Returns only employees who belong to a department.

```
SELECT
    e.name,
    d.department_name
FROM
    employees e
INNER JOIN
    departments d ON e.department_id = d.department_id;
```

OUTER JOIN Example (LEFT OUTER JOIN): Returns all employees, including those who do not belong to any department.

```
SELECT
    e.name,
    d.department_name
FROM
    employees e
LEFT OUTER JOIN
    departments d ON e.department_id = d.department_id;
```

Explanation:

- **INNER JOIN** includes only rows with matching `department_id` in both tables.
- **LEFT OUTER JOIN** includes all rows from `employees`, and fills `department_name` with NULL if there is no matching department.
- You can also use **RIGHT OUTER JOIN** or **FULL OUTER JOIN** to include all rows from the right table or both tables, respectively.

> **Tip:** Use `INNER JOIN` when you need only matching records, and `OUTER JOIN` when you want to include unmatched rows as well.

---

3. Write a Query to Fetch the Second-Highest Salary from an Employee Table

Answer:
To get the second-highest salary, you can use the `ORDER BY` and `LIMIT` clauses, or use a subquery to exclude the highest salary.

Example:
Suppose you have a table called `employees` with a column `salary`.

Using LIMIT/OFFSET (works in MySQL, PostgreSQL):

```
SELECT
    DISTINCT salary
FROM
    employees
```

```sql
ORDER BY
    salary DESC
LIMIT 1 OFFSET 1;
```

Using Subquery (works in most SQL dialects):

```sql
SELECT
    MAX(salary) AS second_highest_salary
FROM
    employees
WHERE
    salary &lt; (SELECT MAX(salary) FROM employees);
```

Explanation:

- The first query orders salaries in descending order, skips the highest, and fetches the next one.
- The second query finds the maximum salary that is less than the overall maximum, effectively giving the second-highest salary.
- **DISTINCT** ensures duplicate salaries are not counted multiple times.

> **Tip:** If there are multiple employees with the same second-highest salary, both queries will return that value. Adjust the query if you need all employees with the second-highest salary.

---

4. How Do You Use GROUP BY and HAVING Together? Provide an Example.

Answer:
The `GROUP BY` clause groups rows that have the same values in specified columns into summary rows. The `HAVING` clause is used to filter groups based on a condition, typically involving aggregate functions.

Example:
Suppose you have a table called `orders` with columns `customer_id` and `order_amount`. To find customers who have placed more than 2 orders:

```sql
SELECT
    customer_id,
    COUNT(*) AS total_orders
FROM
    orders
GROUP BY
    customer_id
HAVING
    COUNT(*) > 2;
```

Explanation:

- **GROUP BY** groups the rows by `customer_id`.
- **COUNT(*)** counts the number of orders for each customer.
- **HAVING COUNT(*) > 2** filters the groups to include only those customers with more than 2 orders.

> **Tip:** Use `HAVING` to filter groups after aggregation, while `WHERE` filters rows before grouping.

---

5. Write a Query to Find Employees Earning More Than Their Managers

Answer:
To find employees who earn more than their managers, you typically need a table where each employee has a `manager_id` referencing another employee's `employee_id`. You can use a self-join to compare each employee's salary with their manager's salary.

Example:
Suppose you have an `employees` table with columns `employee_id`, `name`, `salary`, and `manager_id`.

```
SELECT
    e.name AS employee_name,
    e.salary AS employee_salary,
    m.name AS manager_name,
    m.salary AS manager_salary
FROM
    employees e
JOIN
    employees m ON e.manager_id = m.employee_id
WHERE
    e.salary > m.salary;
```

Explanation:

- The table is joined to itself: `e` represents employees, `m` represents their managers.
- The `WHERE` clause filters for employees whose salary is greater than their manager's salary.

> **Tip:** Make sure `manager_id` is not NULL to avoid comparing employees without managers.

---

6. What is a Window Function in SQL? Provide Examples of ROW_NUMBER and RANK.

Answer:
A **window function** performs a calculation across a set of table rows that are somehow related to the current row. Unlike aggregate functions, window functions do not collapse rows; they return a value for each row in the result set. Common window functions include `ROW_NUMBER`, `RANK`, `DENSE_RANK`, `SUM`, and `AVG` used with the `OVER` clause.

Example:
Suppose you have an `employees` table with columns `employee_id`, `name`, and `salary`.

ROW_NUMBER Example: Assigns a unique sequential number to each row within a partition, ordered by salary descending.

```
SELECT
    employee_id,
    name,
    salary,
    ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_num
FROM
    employees;
```

RANK Example: Assigns a rank to each row within the result set, with gaps for ties.

```
SELECT
    employee_id,
    name,
    salary,
    RANK() OVER (ORDER BY salary DESC) AS salary_rank
FROM
    employees;
```

Explanation:

- **ROW_NUMBER()** gives each row a unique number based on the specified order.
- **RANK()** assigns the same rank to rows with equal values, but leaves gaps in the ranking sequence for ties.
- The `OVER (ORDER BY salary DESC)` clause defines the window for the function, ordering employees by salary from highest to lowest.

> **Tip:** Use window functions when you need to perform calculations across rows related to the current row, such as ranking, running totals, or moving averages.

---

7. Write a Query to Fetch the Top 3 Performing Products Based on Sales

Answer:
To find the top 3 performing products based on sales, you can aggregate the sales data by product, order the results by total sales in descending order, and then limit the output to the top 3 products.

Example:
Suppose you have a table called `sales` with columns `product_id` and `sale_amount`, and a `products` table with `product_id` and `product_name`.

```
SELECT
    p.product_name,
```

```sql
    SUM(s.sale_amount) AS total_sales
FROM
    sales s
JOIN
    products p ON s.product_id = p.product_id
GROUP BY
    p.product_name
ORDER BY
    total_sales DESC
LIMIT 3;
```

Explanation:

- **JOIN** combines the `sales` and `products` tables to get product names.
- **SUM(s.sale_amount)** calculates the total sales for each product.
- **GROUP BY** groups the results by product name.
- **ORDER BY total_sales DESC** sorts products from highest to lowest sales.
- **LIMIT 3** returns only the top 3 products.

> **Tip:** Adjust the `LIMIT` value to fetch a different number of top-performing products as needed.

---

8. Explain the Difference Between UNION and UNION ALL

Answer:
`UNION` and `UNION ALL` are used to combine the results of two or more `SELECT` queries. The key difference is that `UNION` removes duplicate rows from the result set, while `UNION ALL` includes all rows, even duplicates.

Example:
Suppose you have two tables, `customers_2023` and `customers_2024`, both with a `customer_id` column.

Using UNION: Removes duplicates.

```sql
SELECT customer_id FROM customers_2023
UNION
SELECT customer_id FROM customers_2024;
```

Using UNION ALL: Includes duplicates.

```sql
SELECT customer_id FROM customers_2023
UNION ALL
SELECT customer_id FROM customers_2024;
```

Comparison Table:

Feature	UNION	UNION ALL

Duplicates	Removes duplicates	Keeps all rows, including duplicates
Performance	Slower (due to duplicate removal)	Faster (no duplicate check)
Use Case	When you want unique results	When you want all results, including duplicates

Explanation:

- **UNION** combines result sets and removes any duplicate rows.
- **UNION ALL** combines result sets and includes all rows, even if they are duplicates.
- Both require the same number of columns and compatible data types in each `SELECT` statement.

> **Tip:** Use `UNION ALL` for better performance when you are sure there are no duplicates or you want to keep them.

---

9. How Do You Use a CASE Statement in SQL? Provide an Example.

Answer:
The `CASE` statement in SQL allows you to perform conditional logic within your queries. It works like an IF-THEN-ELSE statement, letting you return different values based on specified conditions.

Example:
Suppose you have an `employees` table with a `salary` column, and you want to categorize employees as 'High', 'Medium', or 'Low' earners based on their salary.

```sql
SELECT
    name,
    salary,
    CASE
        WHEN salary >= 100000 THEN 'High'
        WHEN salary >= 50000 THEN 'Medium'
        ELSE 'Low'
    END AS salary_category
FROM
    employees;
```

Explanation:

- **CASE** checks each condition in order and returns the corresponding value for the first true condition.
- If none of the conditions are met, the **ELSE** value is returned.
- The result is a new column (`salary_category`) that classifies each employee based on their salary.

> **Tip:** Use `CASE` for conditional transformations, custom groupings, or to replace IF/ELSE logic in your SQL queries.

---

10. Write a Query to Calculate the Cumulative Sum of Sales

10. Write a query to calculate the cumulative sum of sales.

Answer:
To calculate the cumulative sum (running total) of sales, you can use the `SUM()` window function with the `OVER` clause, ordering by the relevant column (such as date or transaction ID).

Example:
Suppose you have a table called `sales` with columns `sale_date` and `sale_amount`.

```
SELECT
    sale_date,
    sale_amount,
    SUM(sale_amount) OVER (ORDER BY sale_date) AS cumulative_sales
FROM
    sales;
```

Explanation:

- **SUM(sale_amount) OVER (ORDER BY sale_date)** calculates the running total of `sale_amount` up to the current row, ordered by `sale_date`.
- This provides a cumulative sum for each row in the result set.
- You can partition the results (e.g., by customer or product) using `PARTITION BY` if needed.

> **Tip:** Cumulative sums are useful for tracking running totals, trends over time, or progress toward goals.

---

11. What is a CTE (Common Table Expression), and How Is It Used?

Answer:
A **Common Table Expression (CTE)** is a temporary result set defined within the execution scope of a single SQL statement. CTEs make complex queries easier to read and maintain by allowing you to break them into logical building blocks. They are defined using the `WITH` keyword and can be referenced like a table or view within the main query.

Example:
Suppose you want to find employees who earn more than the average salary. You can use a CTE to calculate the average salary first, then reference it in your main query.

```
WITH avg_salary_cte AS (
    SELECT AVG(salary) AS avg_salary
    FROM employees
)
SELECT
    name,
    salary
FROM
```

```
    employees,
    avg_salary_cte
WHERE
    employees.salary > avg_salary_cte.avg_salary;
```

Explanation:

- The `WITH` clause defines a CTE named `avg_salary_cte` that calculates the average salary.
- The main query selects employees whose salary is greater than the average, referencing the CTE as if it were a table.
- CTEs can simplify queries, especially when you need to reuse a subquery or perform recursive operations.

> **Tip:** Use CTEs to improve query readability and maintainability, especially for complex or multi-step data transformations.

12. Write a Query to Identify Customers Who Have Made Transactions Above $5,000 Multiple Times

Answer:
To find customers who have made transactions greater than $5,000 more than once, filter the transactions above $5,000 and group by `customer_id`, then use the `HAVING` clause to select those with a count greater than 1.

Example:
Suppose you have a table called `transactions` with columns `customer_id` and `transaction_amount`.

```
SELECT
    customer_id,
    COUNT(*) AS high_value_transactions
FROM
    transactions
WHERE
    transaction_amount > 5000
GROUP BY
    customer_id
HAVING
    COUNT(*) > 1;
```

Explanation:

- **WHERE transaction_amount > 5000** filters for transactions above $5,000.
- **GROUP BY customer_id** groups the results by customer.
- **COUNT(*)** counts the number of high-value transactions per customer.
- **HAVING COUNT(*) > 1** selects only those customers who have made such transactions more than once.

> **Tip:** Adjust the threshold or grouping as needed to match your business requirements.

13. Explain the Difference Between DELETE and TRUNCATE Commands

Answer:
Both `DELETE` and `TRUNCATE` are used to remove data from a table, but they differ in how they operate and their effects on the table and its data.

Comparison Table:

Feature	DELETE	TRUNCATE
Removes Rows	Removes specified rows (can use WHERE clause)	Removes all rows (cannot use WHERE clause)
Transaction Logging	Row-by-row logging (slower for large tables)	Minimal logging (faster for large tables)
Can Be Rolled Back	Yes, if used within a transaction	Yes, in most databases if used within a transaction
Resets Identity Column	No	Yes
Triggers	Activates DELETE triggers	Does not activate DELETE triggers

Explanation:

- **DELETE** is used when you need to remove specific rows and can be filtered using a `WHERE` clause.
- **TRUNCATE** quickly removes all rows from a table and resets identity columns, but cannot be used to delete specific rows.
- Use `DELETE` for selective removal and `TRUNCATE` for fast, complete data removal.

> **Tip:** Use `TRUNCATE` with caution, as it cannot be used if the table is referenced by a foreign key constraint.

14. How Do You Optimize SQL Queries for Better Performance?

Answer:
Optimizing SQL queries involves improving their efficiency to reduce execution time and resource usage. This can be achieved through indexing, query rewriting, and analyzing execution plans.

Example Strategies:

- **Use Indexes:** Create indexes on columns used in `WHERE`, `JOIN`, and `ORDER BY` clauses to speed up data retrieval.

- **Write Selective Queries:** Retrieve only the columns and rows you need using specific `SELECT` and `WHERE` clauses.
- **Avoid SELECT *:** Specify only required columns to reduce data transfer and processing.
- **Use Joins Efficiently:** Prefer appropriate join types and ensure join columns are indexed.
- **Analyze Execution Plans:** Use tools like `EXPLAIN` to understand how queries are executed and identify bottlenecks.
- **Optimize Subqueries:** Replace correlated subqueries with joins or CTEs when possible.
- **Limit Result Sets:** Use `LIMIT` or `TOP` to restrict the number of rows returned.

Explanation:

- Indexes help the database find data faster, especially for large tables.
- Efficient queries reduce unnecessary data processing and network load.
- Reviewing execution plans helps identify slow operations like full table scans.

> **Tip:** Regularly monitor query performance and update indexes or rewrite queries as your data grows and changes.

---

15. Identify Customers with Consecutive Months of Purchases

Answer:
To find customers who have made purchases in consecutive months, use window functions to compare each purchase month with the previous one for the same customer. If the difference is 1 month, it indicates consecutive purchases.

Example:
Suppose you have a table called `purchases` with columns `customer_id` and `purchase_date`.

```
SELECT
    customer_id,
    purchase_date,
    LAG(purchase_date) OVER (PARTITION BY customer_id ORDER BY purchase_date) AS
prev_purchase_date
FROM
    purchases
```

To identify consecutive months, filter where the difference between `purchase_date` and `prev_purchase_date` is exactly 1 month:

```
SELECT
    customer_id,
    purchase_date,
    prev_purchase_date
FROM (
    SELECT
        customer_id,
        purchase_date,
```

```
        LAG(purchase_date) OVER (PARTITION BY customer_id ORDER BY purchase_date)
AS prev_purchase_date
    FROM
        purchases
) t
WHERE
    prev_purchase_date IS NOT NULL
    AND DATE_PART('month', AGE(purchase_date, prev_purchase_date)) = 1
    AND DATE_PART('year', AGE(purchase_date, prev_purchase_date)) = 0;
```

Explanation:

- **LAG()** gets the previous purchase date for each customer.
- The `WHERE` clause checks if the current and previous purchases are exactly one month apart in the same year.
- This identifies customers with purchases in consecutive months.

> **Tip:** Adjust the date difference logic for your SQL dialect (e.g., use `DATEDIFF` or `TIMESTAMPDIFF` in MySQL).

16. Calculate Average Order Value (AOV) by Month

Answer:
To calculate the Average Order Value (AOV) by month, group your orders by month and divide the total order amount by the number of orders in each month.

Example:
Suppose you have an `orders` table with columns `order_id`, `order_date`, and `order_amount`.

```
SELECT
    DATE_TRUNC('month', order_date) AS order_month,
    AVG(order_amount) AS average_order_value
FROM
    orders
GROUP BY
    DATE_TRUNC('month', order_date)
ORDER BY
    order_month;
```

Explanation:

- **DATE_TRUNC('month', order_date)** extracts the month from each order date (syntax may vary by SQL dialect).
- **AVG(order_amount)** calculates the average order value for each month.
- **GROUP BY** groups the results by month.
- **ORDER BY** sorts the results chronologically.

> **Tip:** Adjust the date truncation function for your SQL dialect (e.g., use `FORMAT(order_date, 'yyyy-MM')` in SQL Server or `TO_CHAR(order_date, 'YYYY-MM')` in Oracle).

17. Rank Sales Representatives by Quarterly Performance

Answer:
To rank sales representatives by their performance each quarter, aggregate sales by representative and quarter, then use a window function like `RANK()` or `ROW_NUMBER()` to assign rankings within each quarter.

Example:
Suppose you have a `sales` table with columns `rep_id`, `sale_date`, and `sale_amount`, and a `representatives` table with `rep_id` and `rep_name`.

```
SELECT
    r.rep_name,
    DATE_TRUNC('quarter', s.sale_date) AS quarter,
    SUM(s.sale_amount) AS total_sales,
    RANK() OVER (
        PARTITION BY DATE_TRUNC('quarter', s.sale_date)
        ORDER BY SUM(s.sale_amount) DESC
    ) AS sales_rank
FROM
    sales s
JOIN
    representatives r ON s.rep_id = r.rep_id
GROUP BY
    r.rep_name,
    DATE_TRUNC('quarter', s.sale_date)
ORDER BY
    quarter,
    sales_rank;
```

Explanation:

- **DATE_TRUNC('quarter', s.sale_date)** extracts the quarter from each sale date.
- **SUM(s.sale_amount)** calculates total sales per representative per quarter.
- **RANK() OVER (PARTITION BY quarter ORDER BY total_sales DESC)** ranks representatives within each quarter based on their total sales.
- **ORDER BY** ensures results are sorted by quarter and rank.

> **Tip:** Adjust the date truncation function for your SQL dialect (e.g., use `QUARTER(sale_date)` in MySQL or `TO_CHAR(sale_date, 'YYYY-Q')` in Oracle).

18. Find the Month with the Highest Revenue in Each Year

Answer:
To find the month with the highest revenue for each year, aggregate sales by year and month, then use a

window function to rank months within each year by total revenue. Select the top-ranked month for each year.

Example:
Suppose you have a `sales` table with columns `sale_date` and `sale_amount`.

```
SELECT
    year,
    month,
    total_revenue
FROM (
    SELECT
        EXTRACT(YEAR FROM sale_date) AS year,
        EXTRACT(MONTH FROM sale_date) AS month,
        SUM(sale_amount) AS total_revenue,
        RANK() OVER (
            PARTITION BY EXTRACT(YEAR FROM sale_date)
            ORDER BY SUM(sale_amount) DESC
        ) AS revenue_rank
    FROM
        sales
    GROUP BY
        EXTRACT(YEAR FROM sale_date),
        EXTRACT(MONTH FROM sale_date)
) ranked
WHERE
    revenue_rank = 1
ORDER BY
    year;
```

Explanation:

- Aggregate sales by year and month using `SUM(sale_amount)`.
- **RANK()** assigns a rank to each month within a year based on total revenue.
- Filter for `revenue_rank = 1` to get the month(s) with the highest revenue per year.
- Adjust `EXTRACT` or date functions for your SQL dialect as needed.

> **Tip:** If multiple months tie for highest revenue in a year, all will be shown. Use `ROW_NUMBER()` if you want only one month per year.

---

19. Identify Items with Stockouts

Answer:
To identify items that have experienced stockouts (i.e., inventory levels dropped to zero), query the inventory or stock movement table for records where the stock quantity is zero.

Example:
Suppose you have an `inventory` table with columns `item_id`, `item_name`, `stock_date`, and `quantity`.

```
SELECT
    item_id,
    item_name,
    stock_date
FROM
    inventory
WHERE
    quantity = 0;
```

Explanation:

- This query selects all items and dates where the quantity on hand is zero, indicating a stockout event.
- If you want only the most recent stockout per item, use a window function or subquery to get the latest `stock_date` where `quantity = 0`.

> **Tip:** Adjust the table and column names as needed. For ongoing stock status, check the latest inventory record per item.

---

20. Calculate Average Time Between Orders by Customer

Answer:
To calculate the average time between orders for each customer, use window functions to find the difference between consecutive order dates, then average those differences per customer.

Example:
Suppose you have an `orders` table with columns `customer_id` and `order_date`.

```
SELECT
    customer_id,
    AVG(days_between) AS avg_days_between_orders
FROM (
    SELECT
        customer_id,
        order_date,
        LAG(order_date) OVER (PARTITION BY customer_id ORDER BY order_date) AS
prev_order_date,
        EXTRACT(DAY FROM order_date - LAG(order_date) OVER (PARTITION BY
customer_id ORDER BY order_date)) AS days_between
    FROM
        orders
) t
WHERE
    prev_order_date IS NOT NULL
GROUP BY
    customer_id;
```

Explanation:

- **LAG(order_date)** gets the previous order date for each customer.
- **EXTRACT(DAY FROM ...)** calculates the number of days between consecutive orders.
- Rows without a previous order are excluded from the average.
- **AVG(days_between)** gives the average interval between orders for each customer.
- Adjust date difference logic for your SQL dialect (e.g., use `DATEDIFF` in SQL Server/MySQL).

> **Tip:** This metric helps analyze customer purchase frequency and can inform retention strategies.