SQL SCENARIO BASED CODING Q&A

BY - SHUBHAM WADEKAR

You need to retrieve the names and salaries of employees from the employees table, but only for those working in the Finance department.

Question:
How would you use the SELECT statement with a WHERE clause to retrieve specific data based on a condition?

The SELECT statement is used to query specific columns from a table and retrieve data based on certain conditions. The WHERE clause filters the rows returned by the query, ensuring only records meeting the specified criteria are included in the result set. In this case, we want only the employees working in the Finance department.

Table Structure:

```sql
-- Sample employees table structure.
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    department VARCHAR(50),
    salary DECIMAL(10, 2)
);
```

```sql
-- Query to fetch employees in the Finance department.
SELECT first_name, last_name, salary
FROM employees
WHERE department = 'Finance';
```

This query will return rows where the department column contains the value 'Finance'. If the table contains these records:

first_name	last_name	salary
Alice	Smith	60000
Bob	Johnson	55000

The result set will display only employees belonging to Finance.

You need to add a new employee named John Doe to the employees table with a salary of 50,000 and a department of HR.

Question:
How can you use the INSERT statement to add a new record to the employees table?

The INSERT statement allows you to add a new record to a table by specifying the column names and corresponding values. You must ensure that the values match the data types and constraints (e.g., NOT NULL) defined in the table.

Table Structure:

```sql
-- Creating the employees table.
CREATE TABLE employees (
    employee_id INT PRIMARY KEY AUTO_INCREMENT,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    department VARCHAR(50) NOT NULL,
    salary DECIMAL(10, 2)
);
```

```sql
-- Inserting a new employee.
INSERT INTO employees (first_name, last_name, department, salary)
VALUES ('John', 'Doe', 'HR', 50000);
```

This query inserts John Doe into the HR department with a salary of 50,000. After insertion, the table might look like this:

employee_id	first_name	last_name	department	salary
1	John	Doe	HR	50000

You need to increase the salary of all employees in the IT department by 10%.

Question:
How would you use the UPDATE statement to modify existing records?

The UPDATE statement modifies existing rows in a table. You use the SET clause to specify the column(s) to be updated and the WHERE clause to target specific rows. In this case, we increase the salary of all employees in the IT department by 10%.

Table Structure:

```sql
-- Sample employees table.
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    department VARCHAR(50),
    salary DECIMAL(10, 2)
);
```

```sql
-- Increasing IT department salaries by 10%.
UPDATE employees
SET salary = salary * 1.10
WHERE department = 'IT';
```

If the original table contains:

employee_id	first_name	department	salary
2	Alice	IT	50000
3	Bob	IT	60000

After running the query, the salaries will become:

employee_id	first_name	department	salary
2	Alice	IT	55000
3	Bob	IT	66000

The HR department has been closed, and all employees in HR must be removed from the database.

Question:
How can you use the DELETE statement to remove specific records?

The DELETE statement removes rows from a table. In this case, you must use the WHERE clause to delete only employees belonging to the HR department. Without the WHERE clause, all rows would be deleted.

```sql
-- Deleting all employees from the HR department.
DELETE FROM employees
WHERE department = 'HR';
```

If the original table contains:

employee_id	first_name	department	salary
2	Alice	IT	55000
3	Bob	HR	66000

After running the query, the table will look like:

employee_id	first_name	department	salary
2	Alice	IT	55000

You need to create a table that stores product prices, including whole numbers and decimal values.

Question:
Which data type should you use to store product prices in a table?

The DECIMAL data type is the most appropriate for storing prices since it provides high precision and control over decimal places. Using DECIMAL avoids rounding issues that might arise with floating-point data types like FLOAT.

```sql
-- Creating a table to store product information and prices.
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100),
    price DECIMAL(10, 2) -- Up to 10 digits, 2 after the decimal point.
);
```

This structure ensures that prices like 123.45 or 99999.99 are stored accurately without rounding errors.

You need to filter employees with salaries between 30,000 and 60,000.

Question:
How can you use comparison operators to filter records based on a salary range?

Comparison operators such as >= and <= can filter rows based on specific ranges. The AND operator combines these conditions, ensuring both are satisfied.

For Example:

```
-- Retrieving employees with salaries between 30,000 and 60,000.
SELECT first_name, last_name, salary
FROM employees
WHERE salary >= 30000 AND salary <= 60000;
```

This query returns employees whose salaries fall within the specified range.

You want to retrieve employees whose salaries are not 40,000.

Question:
How can you use the != operator to exclude certain values in SQL?

The != operator filters out rows where the column matches a specific value. This scenario requires excluding employees with a salary of exactly 40,000.

For Example:

```sql
-- Excluding employees with a salary of 40,000.
SELECT first_name, last_name, salary
FROM employees
WHERE salary != 40000;
```

This query returns all employees except those with a salary of 40,000.

You need to find employees whose names contain the letter 'a'.

Question:
How can you use the LIKE operator to search for patterns in SQL?

The LIKE operator allows searching for patterns in text. In this scenario, % acts as a wildcard that matches any sequence of characters.

For Example:

```sql
-- Retrieving employees whose first names contain 'a'.
SELECT first_name, last_name
FROM employees
WHERE first_name LIKE '%a%';
```

This query returns names like Alice or Aaron that contain the letter 'a'.

You want to check if there are any employees in the Sales department.

Question:
How can you use the EXISTS operator to verify if a department has employees?

The EXISTS operator checks if a subquery returns any rows. If it does, the result is TRUE.

For Example:

```sql
-- Checking if there are employees in Sales.
SELECT EXISTS (
    SELECT 1 FROM employees WHERE department = 'Sales'
);
```

If employees exist in the Sales department, the result will be 1 (true).

You want to create a table that stores whether employees are active or inactive.

Question:
Which data type should you use to store true/false values in SQL?

The BOOLEAN data type stores TRUE or FALSE values, ideal for representing binary states.

For Example:

```sql
-- Creating a table to store employee status.
CREATE TABLE employee_status (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    is_active BOOLEAN -- TRUE if active, FALSE if inactive.
);
```

This structure ensures each employee's status is stored accurately.

You need to retrieve the top 5 highest-paid employees from the employees table.

Question:
How would you use the ORDER BY and LIMIT clauses to get the top records?

The ORDER BY clause sorts the data, and the LIMIT clause restricts the number of rows returned. To retrieve the top 5 highest-paid employees, you need to sort the salary in descending order and use LIMIT 5.

For Example:

```sql
-- Retrieving the top 5 highest-paid employees.
SELECT first_name, last_name, salary
FROM employees
ORDER BY salary DESC
LIMIT 5;
```

This query will display the top 5 employees by salary.

You want to calculate the total salary for each department.

Question:
How can you use the GROUP BY clause to aggregate data?

The GROUP BY clause groups rows with the same values, allowing aggregate functions like SUM() to be applied to each group. In this scenario, we group employees by department and calculate the total salary for each department.

For Example:

```
-- Calculating total salary for each department.
SELECT department, SUM(salary) AS total_salary
FROM employees
GROUP BY department;
```

This query returns the total salary for each department.

You need to display departments with a total salary greater than 100,000.

Question:
How can you filter grouped data using the HAVING clause?

The HAVING clause filters grouped data after aggregation. In this scenario, it ensures only departments with a total salary greater than 100,000 are displayed.

For Example:

```
-- Filtering departments with a total salary > 100,000.
SELECT department, SUM(salary) AS total_salary
FROM employees
GROUP BY department
HAVING SUM(salary) > 100000;
```

This query displays departments whose combined salaries exceed 100,000.

You need to combine employees from two tables: employees_2023 and employees_2024.

Question:
How would you use the UNION operator to combine the data?

The UNION operator combines the result sets of two SELECT queries and removes duplicates. Both queries must return the same number of columns with compatible data types.

For Example:

```sql
-- Combining employees from two years.
SELECT first_name, last_name FROM employees_2023
UNION
SELECT first_name, last_name FROM employees_2024;
```

This query returns a combined list of employees from both tables, without duplicates.

You want to keep duplicates while combining two datasets.

Question:
How can you use UNION ALL to retain all rows, including duplicates?

The UNION ALL operator combines result sets and keeps all rows, including duplicates. It is more efficient since it doesn't check for duplicates.

For Example:

```sql
-- Using UNION ALL to combine employees with duplicates.
SELECT first_name, last_name FROM employees_2023
UNION ALL
SELECT first_name, last_name FROM employees_2024;
```

This query retains all duplicate rows from both datasets.

You need to create a list of employees who belong to either the Finance or IT department.

Question:
How can you use the IN operator to filter multiple values?

The IN operator allows filtering a column against multiple values. In this scenario, it ensures only employees from the Finance or IT departments are retrieved.

For Example:

```sql
-- Retrieving employees from Finance or IT.
SELECT first_name, department
FROM employees
WHERE department IN ('Finance', 'IT');
```

This query returns employees who belong to either department.

You need to calculate the average salary across all employees.

Question:
How can you use the AVG() function to find the average value?

The AVG() function calculates the average value of a numeric column. In this scenario, it finds the average salary across all employees.

For Example:

```sql
-- Calculating the average salary.
SELECT AVG(salary) AS average_salary
FROM employees;
```

This query returns the average salary of all employees.

You need to find employees who don’t belong to the HR department.

Question:
How can you use the NOT IN operator to exclude certain values?

The NOT IN operator excludes rows where the column value matches any value in a specified list. In this case, it filters out employees from the HR department.

For Example:

```sql
-- Retrieving employees not in HR.
SELECT first_name, department
FROM employees
WHERE department NOT IN ('HR');
```

This query returns employees who belong to departments other than HR.

You need to retrieve only the first three employees in alphabetical order by their first names.

Question:
How can you use ORDER BY and LIMIT together?

The ORDER BY clause sorts the results, and LIMIT restricts the number of rows returned. This scenario retrieves the first three employees alphabetically by their first name.

For Example:

```sql
-- Retrieving the first three employees in alphabetical order.
SELECT first_name, last_name
FROM employees
ORDER BY first_name ASC
LIMIT 3;
```

This query returns the first three employees sorted by their first names.

You want to join the employees table with the departments table to display employee names along with their department names.

Question:
How can you use the JOIN clause to combine data from multiple tables?

The JOIN clause combines rows from two or more tables based on a related column. In this scenario, the department_id column links the employees and departments tables.

Table Structures::

```sql
-- Creating employees table.
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    department_id INT
);

-- Creating departments table.
CREATE TABLE departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(50)
);
```

For Example:

```sql
-- Joining employees with departments.
SELECT e.first_name, d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id;
```

This query returns employee names along with their corresponding department names. If the tables contain:

employees Table:

employee_id	first_name	department_id
1	Alice	1
2	Bob	2

departments Table:

department_id	department_name
1	IT
2	Finance

The result will be:

first_name	department_name
Alice	IT
Bob	Finance

You need to retrieve employees who joined within the last six months.

Question:
How can you use date functions and the WHERE clause to filter records by date range?

In SQL, date filtering is achieved using functions like CURRENT_DATE or NOW(). You can subtract six months using the INTERVAL keyword to find employees who joined recently.

Table Structure:

```sql
CREATE TABLE employees (

    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    join_date DATE
);
```

For Example:

```sql
-- Retrieving employees who joined within the last six months.
SELECT first_name, join_date
FROM employees
WHERE join_date >= CURRENT_DATE - INTERVAL '6 months';
```

You want to generate a list of employees with their total annual bonuses. Bonuses are 10% of their salary.

Question:

How can you use arithmetic operators to calculate derived values?

The * operator allows you to calculate the annual bonus as 10% of the employee's salary.

Table Structure:

```sql
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    salary DECIMAL(10, 2)
);
```

This query returns the average salary of all employees.

For Example:

```sql
-- Calculating the annual bonus for each employee.
SELECT first_name, salary, salary * 0.10 AS bonus
FROM employees;
```

Sample Data and Result:

first_name	salary	bonus
Alice	80000	8000
Bob	60000	6000

This query calculates each employee’s bonus by multiplying the salary by 0.10.

You need to assign unique IDs to each result row, grouped by department and ordered by salary.

Question:
How can you use ROW_NUMBER() with PARTITION BY?

ROW_NUMBER() assigns a unique number to each row, and PARTITION BY ensures the numbering resets for each department.

Table Structure:

```sql
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    department VARCHAR(50),
    salary DECIMAL(10, 2)
);
```

For Example:

```sql
-- Assigning row numbers to employees within each department.
SELECT department, first_name,
       ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS
row_number
FROM employees;
```

Sample Data and Result:

department	first_name	row_number
IT	Alice	1
IT	Bob	2
HR	Charlie	1

Each department has its own row numbers based on salary order.

You need to delete duplicate rows from a table, keeping only one instance of each duplicate.

Question:
How can you identify and remove duplicate rows using ROW_NUMBER()?

Using ROW_NUMBER() with PARTITION BY, you can mark duplicate rows and delete them by keeping only the first occurrence.

Table Structure:

```sql
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    department VARCHAR(50)
);
```

This query returns the first three employees sorted by their first names.

For Example:

```sql
-- Identifying and deleting duplicate rows.
WITH CTE AS (
    SELECT *,
           ROW_NUMBER() OVER (PARTITION BY first_name, department ORDER BY
employee_id) AS row_num
    FROM employees
)
DELETE FROM employees
WHERE employee_id IN (
    SELECT employee_id FROM CTE WHERE row_num > 1
);
```

This query ensures only one instance of each duplicate is kept.

You want to list employees with their managers. Managers are also stored in the employees table.

Question:
How can you use SELF JOIN to connect employees with their managers?

A SELF JOIN allows you to join a table with itself. Here, each employee has a manager_id that references another employee's employee_id.

Table Structures::

```sql
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    manager_id INT,
    FOREIGN KEY (manager_id) REFERENCES employees(employee_id)
);
```

For Example:

```sql
-- Listing employees with their managers.
SELECT e.first_name AS employee_name, m.first_name AS manager_name
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.employee_id;
```

Sample Data and Result:

employee_name	manager_name
Alice	Bob
Charlie	Bob
Bob	NULL

This query shows each employee along with their manager's name.

You need to merge new employee records with existing ones. If an employee exists, update the salary; otherwise, insert the new employee.

Question:
How can you use the MERGE statement to perform an upsert operation?

The MERGE statement allows you to insert or update records based on whether the employee already exists.

Table Structure:

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    salary DECIMAL(10, 2)
);

CREATE TABLE new_employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    salary DECIMAL(10, 2)
);
```

For Example:

```
-- Merging new employee data.
MERGE INTO employees e
USING new_employees n ON e.employee_id = n.employee_id
WHEN MATCHED THEN
    UPDATE SET e.salary = n.salary
WHEN NOT MATCHED THEN
    INSERT (employee_id, first_name, salary)
    VALUES (n.employee_id, n.first_name, n.salary);
```

This query synchronizes both employee tables.

You need to find all employees who have not submitted their projects.

Question:
How can you use LEFT JOIN and IS NULL to find unmatched rows?

A LEFT JOIN helps find rows in one table that have no matching rows in another.

Table Structure:

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50)
);

CREATE TABLE project_submissions (
    submission_id INT PRIMARY KEY,
    employee_id INT,
    FOREIGN KEY (employee_id) REFERENCES employees(employee_id)
);
```

This query returns the average salary of all employees.

For Example:

```sql
-- Finding employees without project submissions.
SELECT e.first_name
FROM employees e
LEFT JOIN project_submissions p ON e.employee_id = p.employee_id
WHERE p.employee_id IS NULL;
```

This query retrieves employees who haven't submitted projects.

You want to display the total number of employees in each department, including departments with no employees.

Question:
How can you use LEFT OUTER JOIN to retrieve all departments?

Using a LEFT OUTER JOIN, all departments are retrieved, even if no employees exist.

Table Structure:

```sql
CREATE TABLE departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(50)
);

CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES departments(department_id)
);
```

For Example:

```sql
-- Counting employees by department.
SELECT d.department_name, COUNT(e.employee_id) AS employee_count
FROM departments d
LEFT JOIN employees e ON d.department_id = e.department_id
GROUP BY d.department_name;
```

This query ensures all departments are listed, with zero counts if they have no employees.

You need to create a table that records the current timestamp when a row is inserted.

Question:
How can you use DEFAULT with the TIMESTAMP column?

Using DEFAULT CURRENT_TIMESTAMP, SQL automatically records the insertion time.

Table Structure:

```
CREATE TABLE logs (
    log_id INT PRIMARY KEY,
    description VARCHAR(255),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

You want to ensure that each employee email is unique.

Question:
How can you use the UNIQUE constraint?

The UNIQUE constraint ensures that no two employees have the same email.

Table Structures::

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    email VARCHAR(100) UNIQUE
);
```

This ensures every employee has a unique email.

You need to find the second highest salary from the employees table.

Question:
How can you retrieve the second highest salary using SQL?

You can use ORDER BY and LIMIT with OFFSET to retrieve the second highest salary.ORDER BY sorts the rows by salary in descending order. OFFSET 1 skips the first row (the highest salary), and LIMIT 1 ensures only the second highest salary is returned.

Sample Data:

employee_id	first_name	salary
1	Alice	100000
2	Bob	95000
3	Charlie	90000

For Example:

```sql
-- Retrieving the second highest salary.
SELECT salary
FROM employees
ORDER BY salary DESC
LIMIT 1 OFFSET 1;
```

Result:

salary
95000

This query skips the first row (highest salary) and returns the second highest salary.

You need to compare two columns in the same table to find discrepancies.

Question:
How can you compare two columns within the same table using SQL?

SQL allows you to compare two columns using the WHERE clause. This is helpful for checking if values in two columns differ, such as salary and expected_salary.

Sample Data:

employee_id	first_name	salary	expected_salary
1	Alice	80000	85000
2	Bob	95000	95000

For Example:

```sql
-- Finding discrepancies between salary and expected salary.
SELECT employee_id, first_name, salary, expected_salary
FROM employees
WHERE salary != expected_salary;
```

Result:

employee_id	first_name	salary	expected_salary
1	Alice	80000	85000

This query shows employees where the actual salary doesn't match the expected salary.

You need to identify duplicate employee names in the employees table.

Question:
How can you use GROUP BY and HAVING to find duplicate entries?

The GROUP BY clause groups rows with identical values, and the HAVING clause filters groups with more than one row.

Sample Data:

employee_id	first_name	department
1	Alice	IT
2	Alice	HR
3	Bob	Finance

For Example:

```sql
-- Finding duplicate employee names.
SELECT first_name, COUNT(*) AS count
FROM employees
GROUP BY first_name
HAVING COUNT(*) > 1;
```

Result:

first_name	count
Alice	2

This query identifies employee names that appear more than once.

You want to update multiple columns for an employee in one query.

Question:
How can you use the UPDATE statement to modify multiple columns?

The UPDATE statement can modify multiple columns by listing them in the SET clause.

Sample Data Before Update:

employee_id	first_name	salary	department
1	Alice	70000	Sales

For Example:

```sql
-- Updating salary and department for an employee.
UPDATE employees
SET salary = 75000, department = 'IT'
WHERE employee_id = 1;
```

Data After Update:

employee_id	first_name	salary	department
1	Alice	75000	IT

This query updates Alice’s salary and department.

You need to insert data into one table by selecting it from another.

Question:
How can you use INSERT INTO ... SELECT to copy data between tables?

The INSERT INTO ... SELECT statement inserts data from one table into another. Both tables must have matching column types.

For Example:

```sql
-- Copying data from employees_2023 to employees.
INSERT INTO employees (employee_id, first_name, salary)
SELECT employee_id, first_name, salary
FROM employees_2023;
```

This query copies data from the employees_2023 table to the employees table.

You want to calculate the cumulative salary for employees ordered by department.

Question:
How can you use SUM() with OVER() to compute cumulative totals?

The SUM() function with OVER() computes a running total for each row within a partition.

Sample Data:

employee_id	first_name	department	salary
1	Alice	IT	60000
2	Bob	IT	70000
3	Charlie	HR	50000

For Example:

```sql
-- Calculating cumulative salary within each department.
SELECT department, first_name,
       SUM(salary) OVER (PARTITION BY department ORDER BY salary) AS
cumulative_salary
FROM employees;
```

Result:

department	first_name	salary
IT	Alice	60000
IT	Bob	130000
HR	Charlie	170000

This query shows the cumulative salary within each department.

You want to delete employees who don’t belong to a specified list of departments.

Question:
How can you use the NOT IN operator to filter and delete specific rows?

The NOT IN operator excludes rows matching any value in the specified list.

Sample Data:

```sql
-- Deleting employees not in Finance or IT departments.
DELETE FROM employees
WHERE department NOT IN ('Finance', 'IT');
```

This query removes employees not in Finance or IT.

You need to count employees in each department but only display departments with more than five employees.

Question:
How can you use GROUP BY with HAVING to filter grouped data?

The GROUP BY clause groups rows by department, and the HAVING clause filters those with more than five employees.

For Example:

```
-- Counting employees in departments with more than five employees.
SELECT department, COUNT(employee_id) AS employee_count
FROM employees
GROUP BY department
HAVING COUNT(employee_id) > 5;
```

This query returns departments with more than five employees.

You want to check if there are employees earning more than 100,000.

Question:
How can you use the EXISTS clause to verify the presence of specific records?

The EXISTS clause checks if a subquery returns any rows.

For Example:

```sql
-- Checking if any employees earn more than 100,000.
SELECT EXISTS (
    SELECT 1
    FROM employees
    WHERE salary > 100000
);
```

If employees earning more than 100,000 exist, the result is TRUE (1).

You want to rank employees within each department based on their salary.

Question:
How can you use the RANK() function with PARTITION BY?

The RANK() function assigns ranks to rows within each department, resetting for each partition.

Sample Data:

employee_id	first_name	department	salary
1	Alice	IT	60000
2	Bob	IT	70000
3	Charlie	HR	50000

For Example:

```sql
-- Ranking employees by salary within each department.
SELECT department, first_name,
       RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS
salary_rank
FROM employees;
```

Result:

department	first_name	salary_rank
IT	Alice	1
IT	Bob	2
HR	Charlie	1

This query ranks employees within each department by their salary. If two employees have the same salary, they get the same rank, and the next rank is skipped.

Inserting new employee data into a table

Question:
How would you insert a new employee record into the employees table with details such as id, name, department_id, and salary?

The INSERT statement adds new data to a table.

Table Structure:

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    department_id INT,
     salary DECIMAL(10, 2)
);
```

For Example:

```
INSERT INTO employees (id, name, department_id, salary)
VALUES (1, 'Alice', 101, 50000);
```

Resulting Table:

id	name	department_id	salary
1	Alice	101	50000

Updating employee salaries based on performance

Question:
How can you increase the salary of employees in the employees table by 10% for those in department 101?

The UPDATE statement modifies rows with a condition.

For Example:

```
UPDATE employees
SET salary = salary * 1.10
WHERE department_id = 101;
```

Resulting Table (After Update):

id	name	department_id	salary
1	Alice	101	55000

Deleting records of employees who left the company

Question:
How would you delete all employees with IDs between 10 and 20?

For Example:

```
DELETE FROM employees
WHERE id BETWEEN 10 AND 20;
```

Resulting Table (After Delete):

id	name	department_id	salary
1	Alice	101	55000
9	Bob	109	60000
22	Ben	122	65000

Inserting multiple rows at once

Question:
How would you add multiple employees to the employees table?

For Example:

```
INSERT INTO employees (id, name, department_id, salary)
VALUES
(2, 'Bob', 102, 55000),
(3, 'Charlie', 101, 48000),
(4, 'Diana', 103, 60000);
```

Resulting Table:

id	name	department_id	salary
1	Alice	101	55000
2	Bob	102	55000
3	Charlie	101	48000
4	Diana	103	60000

Ensuring all employees have unique email addresses

Table Structure:

```sql
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE,
    department_id INT
);
```

For Example:

```sql
INSERT INTO employees (id, name, email, department_id)
VALUES (5, 'Eve', 'eve@example.com', 102);
```

Resulting Table:

id	name	email	department_id
5	Eve	eve@example.com	102

Scenario: Managing failed transactions with ROLLBACK

For Example:

```sql
BEGIN TRANSACTION;
UPDATE employees SET salary = 65000 WHERE id = 1;

-- Error: Cannot set salary to NULL
UPDATE employees SET salary = NULL WHERE id = 2;

ROLLBACK;
```

Resulting Table (After ROLLBACK):

id	name	department_id	salary
1	Alice	101	50000
2	Bob	102	55000

Scenario: Grouping employee records by department

For Example:

```
SELECT department_id, AVG(salary) AS avg_salary
FROM employees
GROUP BY department_id;
```

Resulting Aggregated Output:

department_id	salary
101	51500
102	55000
103	60000

Scenario: Automatically updating a log table when employees are added

Log Table Structure:

```sql
CREATE TABLE employee_logs (
    employee_id INT,
    action VARCHAR(50)
);
```

For Example:

```sql
INSERT INTO employees (id, name, department_id, salary)
VALUES (6, 'Frank', 104, 70000);
```

Resulting Logs Table:

employee_id	action
6	Inserted

Scenario: Ensuring data consistency with a foreign key constraint

Table Structure:

```sql
CREATE TABLE departments (
    id INT PRIMARY KEY,
    name VARCHAR(50)
);

CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES departments(id)
);
```

For Example:

```sql
INSERT INTO departments VALUES (104, 'Finance');
INSERT INTO employees (id, name, department_id) VALUES (7, 'George', 104);
```

Resulting Table:

Departments:

id	name
104	Finance

Employees:

id	name	department_id
7	George	104

Scenario: Creating a view to show high-earning employees

Table Structure:

```sql
CREATE VIEW high_earning_employees AS
SELECT name, salary
FROM employees
WHERE salary > 50000;

SELECT * FROM high_earning_employees;
```

Resulting View:

name	salary
Alice	55000
Diana	60000
Frank	70000

Scenario: Adding a new column to an existing table

Question: How would you add a new column, email, to the employees table?

To store additional information, you can use the ALTER TABLE statement to add a new column. This ensures the table structure can evolve with changing data requirements. Newly added columns will have NULL values for existing records unless a DEFAULT value is specified.

Table Structure Before:

id	name	department_id	salary
1	Alice	101	55000

For Example:

```
ALTER TABLE employees
ADD email VARCHAR(100);
```

Table Structure Before:

id	name	department_id	salary	email
1	Alice	101	55000	NULL

Scenario: Removing a column from a table

Question: How can you remove the email column from the employees table?

Use the ALTER TABLE statement with DROP COLUMN to remove an unnecessary column from the table. Be careful, as this action is irreversible and will delete all data in that column.

Table Structure Before:

id	name	department_id	salary	email
1	Alice	101	55000	NULL

For Example:

```
ALTER TABLE employees
DROP COLUMN email;
```

This query will remove the coulmn email from the table.

Scenario: Renaming a column in a table

Question: How would you rename the name column to employee_name in the employees table?

You can use ALTER TABLE with RENAME COLUMN to modify the name of a column for better clarity or consistency.

Table Structure Before:

id	name	department_id	salary
1	Alice	101	55000

For Example:

```
ALTER TABLE employees
RENAME COLUMN name TO employee_name;
```

This query will rename the coulmn name to employee_name to the table.

Scenario: Changing the data type of a column

Question: How can you change the data type of the salary column from DECIMAL to INTEGER?

If you need to change a column's data type, use ALTER TABLE with MODIFY or ALTER COLUMN. Be cautious as incompatible changes may require data adjustments.

Table Structure Before:

id	name	department_id	salary
1	Alice	101	55000.00

For Example:

```
ALTER TABLE employees
MODIFY salary INT;
```

This query will change the salary data type from decimal to integer.

Scenario: Setting a default value for a column

Question: How would you set a default value of 0 for the salary column?

Setting a default value ensures that if no value is provided during an insert, the column will automatically be assigned the default.

For Example:

```
ALTER TABLE employees
ALTER COLUMN salary SET DEFAULT 0;
```

Inserting a New Record Without Salary:

```
INSERT INTO employees (id, employee_name, department_id)
VALUES (2, 'Bob', 102);
```

Resulting Table:

id	name	department_id	salary
1	Alice	101	55000.00
2	Bob	102	0

Scenario: Dropping a default value from a column

Question: How can you remove the default value from the salary column?

Use ALTER TABLE to drop a default value. After this, if no value is provided, the column will store NULL.

Table Structure Before:

id	name	department_id	salary
1	Alice	101	55000

For Example:

```
ALTER TABLE employees
ALTER COLUMN salary DROP DEFAULT;
```

Inserting Without a Salary Now:

```
INSERT INTO employees (id, employee_name, department_id)
VALUES (3, 'Charlie', 103);
```

Table Structure Before:

id	name	department_id	salary
1	Alice	101	55000
2	Bob	102	0
3	Charlie	103	NULL

Scenario: Adding a primary key to a table

Question: How would you add a primary key to the employees table using the id column?

A primary key uniquely identifies each row and ensures no duplicates.

For Example:

```
ALTER TABLE employees
ADD CONSTRAINT pk_employee_id PRIMARY KEY (id);
```

Resulting Table:

id	name	department_id	salary
1	Alice	101	55000
2	Bob	102	0

Scenario: Adding a foreign key constraint to a table

Question: How would you add a foreign key on department_id referencing the departments table?

Foreign keys ensure referential integrity between tables.

Departments Table:

id	name
101	HR
102	Finance

For Example:

```
ALTER TABLE employees
ADD CONSTRAINT fk_department
FOREIGN KEY (department_id) REFERENCES departments(id);
```

Scenario: Creating an index on the salary column

Question: How would you create an index on the salary column to improve query performance?

An index speeds up searches on a specific column.

For Example:

```
CREATE INDEX idx_salary ON employees(salary);
```

Query Using the Index:

```
CREATE INDEX idx_salary ON employees(salary);
```

Resulting Table:

id	name	department_id	salary
1	Alice	101	55000

Scenario: Dropping an index from a table

Question: How would you drop the idx_salary index?

Use the DROP INDEX statement to remove an index.

For Example:

```
DROP INDEX idx_salary;
```

After dropping the index, queries on the salary column may become slower.

Scenario: Handling duplicate rows in a table

Question: How can you remove duplicate rows from a table, keeping only the first occurrence?

Duplicate rows can lead to incorrect results in reports or analytics. To remove them, we use ROW_NUMBER() with a PARTITION BY clause to assign a unique number to each duplicate row. We keep the first occurrence and delete the others.

Table Before Deletion:

id	name	department_id	salary
1	Alice	101	55000
2	Bob	102	60000
3	Alice	101	55000

For Example:

```
WITH CTE AS (
    SELECT id, employee_name, department_id, salary,
           ROW_NUMBER() OVER (PARTITION BY employee_name, department_id
ORDER BY id) AS row_num
    FROM employees
)
DELETE FROM employees
WHERE id IN (SELECT id FROM CTE WHERE row_num > 1);
```

Resulting Table:

id	name	department_id	salary
1	Alice	101	55000
2	Bob	102	60000

Scenario: Combining multiple result sets with UNION

Question: How would you combine employees and contractors into a single result set without duplicates?

The UNION operator combines rows from multiple queries and removes duplicates. It ensures unique entries in the final result set.

For Example:

```
SELECT name, salary FROM employees
UNION
SELECT name, salary FROM contractors;
```

Employees Table:

name	salary
Alice	55000
Bob	60000

Contractors Table:

name	salary
David	70000
Bob	60000

Resulting Table:

name	salary
Alice	55000
Bob	60000
David	70000

Scenario: Fetching the top N salaries

Question: How would you retrieve the top 2 highest salaries from the employees table?

Using ORDER BY with LIMIT retrieves the top N rows sorted by a specific column.

For Example:

```
SELECT employee_name, salary
FROM employees
ORDER BY salary DESC
LIMIT 2;
```

Resulting Table:

employee_name	salary
Bob	60000
Alice	55000

Scenario: Using a self-join to find employees reporting to the same manager

Question: How would you find pairs of employees reporting to the same manager?

A self-join joins a table with itself, enabling comparisons within the same table.

Employees Table:

id	name	manager_id
1	Alice	NULL
2	Bob	1
3	Charlie	1

For Example:

```sql
SELECT e1.employee_name AS employee1, e2.employee_name AS employee2
FROM employees e1
JOIN employees e2 ON e1.manager_id = e2.manager_id
WHERE e1.id < e2.id;
```

Resulting Table:

employee1	employee2
Bob	Charlie

Scenario: Recursive query to find the hierarchy of employees

Question: How can you retrieve the hierarchy of employees in an organization?

A recursive CTE helps you query hierarchical data, such as employees and their managers.

For Example:

```
WITH RECURSIVE EmployeeHierarchy AS (
    SELECT id, employee_name, manager_id
    FROM employees
    WHERE manager_id IS NULL
    UNION ALL
    SELECT e.id, e.employee_name, e.manager_id
    FROM employees e
    INNER JOIN EmployeeHierarchy eh ON e.manager_id = eh.id
)
SELECT * FROM EmployeeHierarchy;
```

After dropping the index, queries on the salary column may become slower.

Resulting Table:

id	name	manager_id
1	Alice	NULL
2	Bob	1
3	Charlie	1

Scenario: Calculating running totals using a window function

Question: How would you calculate a running total of salaries?

The SUM() window function with OVER() calculates cumulative sums for each row in sequence.

For Example:

```
SELECT employee_name, salary,
       SUM(salary) OVER (ORDER BY salary) AS running_total
FROM employees;
```

Resulting Table:

name	salary	running_total
Alice	55000	55000
Bob	60000	115000

Scenario: Identifying employees with the same salary

Question: How would you find employees earning the same salary?

Use GROUP BY with HAVING to filter groups with more than one employee sharing the same salary.

For Example:

```sql
SELECT salary, COUNT(*) AS employee_count
FROM employees
GROUP BY salary
HAVING COUNT(*) > 1;
```

Resulting Table:

salary	employee_count
55000	2

Scenario: Handling NULL values in SQL

Question: How can you replace NULL salaries with a default value of 0 when retrieving data?

The COALESCE() function returns the first non-null value from a list, making it useful for replacing NULL values.

For Example:

```
SELECT employee_name, COALESCE(salary, 0) AS salary
FROM employees;
```

Resulting Table:

employee_name	salary
Bob	60000
Charlie	0

Scenario: Creating a materialized view for performance optimization

Question: How would you create a materialized view to store high-salary employees?

A materialized view stores query results physically, improving performance for repeated queries.

For Example:

```
CREATE MATERIALIZED VIEW HighSalaryEmployees AS
SELECT employee_name, salary
FROM employees
WHERE salary > 50000;
```

Resulting Table:

employee_name	salary
Alice	55000
Bob	60000

Scenario: Refreshing a materialized view

Question: How do you manually refresh a materialized view to reflect the latest data?

A materialized view must be refreshed to reflect changes in the underlying data.

For Example:

```
REFRESH MATERIALIZED VIEW HighSalaryEmployees;
```

Result: This command updates the view with the latest data from the employees table.

Scenario: Locking a table to prevent other transactions from accessing it

Question: How would you lock a table to ensure that only your transaction can make changes to it?

When performing sensitive operations, locking a table ensures that other transactions cannot modify or read from it until your operation is complete. SQL uses LOCK TABLE to prevent conflicts between concurrent transactions.

Employees Table (Before Lock):

id	name	department_id	salary
1	Alice	101	55000
2	Bob	102	60000

For Example:

```
LOCK TABLE employees IN EXCLUSIVE MODE;

-- Now perform updates
UPDATE employees SET salary = salary * 1.10;
```

Resulting Table (After Update):

id	name	department_id	salary
1	Alice	101	60500
2	Bob	102	60000

Scenario: Using isolation levels to manage concurrent transactions

Question: How do isolation levels control data consistency in concurrent transactions?

Isolation levels control how transactions interact with each other. SQL provides the
following levels:

- READ UNCOMMITTED: Allows dirty reads.
- READ COMMITTED: Only committed changes are visible.
- REPEATABLE READ: Ensures the same data is read within a transaction.
- SERIALIZABLE: Prevents concurrent access, ensuring complete isolation.

Employees Table:

id	name	salary
1	Alice	60500
2	Bob	60000

For Example:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

BEGIN TRANSACTION;
UPDATE employees SET salary = 70000 WHERE id = 2;
COMMIT;
```

Resulting Table (After Commit):

id	name	salary
1	Alice	60500
2	Bob	70000

Scenario: Enforcing a CHECK constraint

Question: How can you ensure that employee salaries cannot be less than 30,000?

A CHECK constraint ensures that values in a column meet a specific condition.

For Example:

```
ALTER TABLE employees
ADD CONSTRAINT check_salary
CHECK (salary >= 30000);
```

Attempt to Insert Invalid Data: INSERT

```
INSERT INTO employees (id, employee_name, salary)
VALUES (3, 'Charlie', 25000); -- This will fail due to the CHECK constraint
```

Resulting Table (No Change):

id	name	salary
1	Alice	60500
2	Bob	70000

Scenario: Implementing soft deletes using a status column

Question: How would you implement a soft delete for employees?

Instead of physically deleting rows, soft deletes mark records as inactive using a status column.

For Example:

```
ALTER TABLE employees
ADD is_active BOOLEAN DEFAULT TRUE;

-- Mark an employee as inactive (soft delete)
UPDATE employees SET is_active = FALSE WHERE id = 2;
```

Resulting Table (After Soft Delete):

id	name	salary	is_active
1	Alice	60500	TRUE
2	Bob	70000	FALSE

Scenario: Automatically updating timestamps using a trigger

Question: How can you automatically update a timestamp when an employee's details change?

Use a trigger to set the last_updated column whenever an employee's record is modified.

For Example:

```
ALTER TABLE employees
ADD last_updated TIMESTAMP;

CREATE TRIGGER update_timestamp
BEFORE UPDATE ON employees
FOR EACH ROW
BEGIN
    SET NEW.last_updated = CURRENT_TIMESTAMP;
END;
```

Updating Employee Record:

```
UPDATE employees SET salary = 71000 WHERE id = 2;
```

Resulting Table (After Update):

id	name	salary	last_updated
1	Alice	60500	NULL
2	Bob	71000	2024-10-21 12:30:00

Scenario: Using a partitioned table for performance optimization

Question: How would you partition a table by department?

Partitioning divides large tables into smaller, manageable pieces based on a column, improving query performance.

For Example:

```
CREATE TABLE employees_partitioned (
    id INT,
    employee_name VARCHAR(100),
    salary DECIMAL(10, 2),
    department_id INT
) PARTITION BY RANGE (department_id) (
    PARTITION p1 VALUES LESS THAN (102),
    PARTITION p2 VALUES LESS THAN (200)
);
```

Inserting Data:

```
INSERT INTO employees_partitioned VALUES (1, 'Alice', 55000, 101);
INSERT INTO employees_partitioned VALUES (2, 'Bob', 71000, 102);
```

Resulting Partitioned Table:

id	name	salary	department_id
1	Alice	55000	101
2	Bob	71000	102

Scenario: Using JSON data in SQL

Question: How would you store and query JSON data in a SQL table?

SQL databases support JSON columns to store semi-structured data.

For Example:

```
CREATE TABLE employee_data (
    id INT PRIMARY KEY,
    details JSON
);

-- Insert JSON data
INSERT INTO employee_data (id, details)
VALUES (1, '{"name": "Alice", "salary": 55000}');
```

Querying JSON Data:

```
SELECT details->>'$.name' AS employee_name FROM employee_data;
```

Resulting Table:

employee_name
Alice

Scenario: Creating an updatable view

Question: How can you create a view that allows updates to the underlying table?

An updatable view allows changes to flow through the view into the base table.

For Example:

```
CREATE VIEW active_employees AS
SELECT id, employee_name, salary
FROM employees
WHERE is_active = TRUE
WITH CHECK OPTION;
```

Updating Through the View:

```
UPDATE active_employees SET salary = 75000 WHERE id = 1;
```

Resulting Table:

id	name	salary	is_active
1	Alice	75000	TRUE

Scenario: Implementing row-level security

Question: How can you restrict access to employees based on their department?

Row-level security (RLS) allows filtering rows based on user permissions.

For Example:

```
ALTER TABLE employees ENABLE ROW LEVEL SECURITY;

CREATE POLICY department_policy
ON employees
USING (department_id = 101);
```

Query (Executed by User with Access to Department 101):

```
SELECT * FROM employees;
```

Resulting Table:

id	employee_name	salary
1	Alice	75000

Scenario: Using temporary tables for intermediate calculations

Question: How would you use a temporary table to store intermediate results?

Temporary tables store data temporarily for intermediate processing and are dropped automatically after the session ends.

For Example:

```sql
CREATE TEMPORARY TABLE temp_high_salaries AS
SELECT * FROM employees WHERE salary > 60000;

-- Use the temporary table for further analysis
SELECT COUNT(*) FROM temp_high_salaries;
```

Resulting Temporary Table:

id	employee_name	salary
2	Bob	71000

This table will be removed when the session ends.

Scenario: Creating a new table to store employee data

Question: How would you create a new table named employees with columns for employee ID, first name, last name, and hire date, ensuring that the employee ID is the primary key?

The CREATE TABLE statement allows us to define the structure of a new table. In this case, the employee_id column is set as the primary key, meaning that it must contain unique, non-null values to uniquely identify each row. The NOT NULL constraint ensures that first_name, last_name, and hire_date cannot be left empty.

For Example:

```sql
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    hire_date DATE NOT NULL
);
```

Resulting Table Structure:

employee_id	first_name	last_name	hire_date
INT (PK)	VARCHAR(50)	VARCHAR(50)	DATE

This creates the employees table where every row must have a unique employee_id.

Scenario: Altering an existing table to add a new column

Question: How can you add a new column email to the employees table, ensuring it stores unique values?

The ALTER TABLE statement allows modifications to an existing table. Here, the email column is added with a UNIQUE constraint to prevent duplicate emails. Adding new columns with constraints helps maintain data quality and ensures uniqueness.

For Example:

```
ALTER TABLE employees
ADD email VARCHAR(100) UNIQUE;
```

Updated Table Structure:

employee_id	first_name	last_name	hire_date	email
INT (PK)	VARCHAR(50)	VARCHAR(50)	DATE	VARCHAR(100)

Scenario: Creating a table with a composite primary key

Question: How would you create a student_courses table with student_id and course_id as a composite primary key?

A composite primary key uses multiple columns to uniquely identify a row. This is useful when no single column alone can guarantee uniqueness, but a combination of columns can.

For Example:

```
CREATE TABLE student_courses (
    student_id INT,
    course_id INT,
    PRIMARY KEY (student_id, course_id)
);
```

Resulting Table Structure:

student_id	course_id
INT (PK)	INT (PK)

This ensures that each unique combination of student_id and course_id is allowed only once.

Scenario: Creating a foreign key relationship between two tables

Question: How do you create a foreign key relationship where the department_id column in the employees table references the departments table?

The FOREIGN KEY constraint ensures that data in the employees table's department_id column corresponds to valid entries in the departments table.

For Example:

```sql
CREATE TABLE departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(100)
);

CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES departments(department_id)
);
```

departments Table Structure:

department_id	department_name
INT (PK)	VARCHAR(100)

employees Table Structure:

employee_id	department_name
INT (PK)	VARCHAR(100)

This ensures that department_id in the employees table must match a valid entry in departments.

Scenario: Dropping an unwanted column from a table

Question: How do you remove the email column from the employees table?

The ALTER TABLE statement with DROP COLUMN allows us to remove a column. This is useful when a column is no longer needed or if requirements change.

For Example:

```
ALTER TABLE employees
DROP COLUMN email;
```

Resulting Table Structure:

employee_id	first_name	last_name	hire_date
INT (PK)	VARCHAR(50)	VARCHAR(50)	DATE

After executing this command, the email column is no longer part of the table.

Scenario: Handling deletion of related rows with foreign keys

Question: How can you ensure that when a department is deleted, all employees in that department are also deleted?

The ON DELETE CASCADE clause ensures that child rows (employees) are automatically deleted when the parent row (department) is deleted.

For Example:

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES departments(department_id) ON
DELETE CASCADE
);
```

Now, if a department is removed from the departments table, all employees assigned to that department will also be deleted automatically.

Scenario: Creating a clustered index on a primary key column

Question: How would you create a clustered index on the employee_id column of the employees table?

A clustered index determines the physical storage order of the rows based on the indexed column. Typically, a primary key is a clustered index by default, but it can be explicitly created as well.

For Example:

```
CREATE CLUSTERED INDEX idx_employee_id ON employees (employee_id);
```

Resulting Table:

The rows in the employees table are now stored in the order of employee_id.

Scenario: Adding a non-clustered index to improve query performance

Question: How can you create a non-clustered index on the last_name column to speed up queries?

A non-clustered index improves search performance without altering the physical order of the data. It provides pointers to the actual data location.

For Example:

```
CREATE NONCLUSTERED INDEX idx_last_name ON employees (last_name);
```

Now, searches on last_name will be faster, as the database uses the index for
lookups.

Scenario: Preventing duplicate values in a column

Question: How can you ensure that the email column in the employees table contains unique values only?

The UNIQUE constraint ensures that no duplicate values are entered into the email column.

For Example:

```
ALTER TABLE employees
ADD CONSTRAINT unique_email UNIQUE (email);
```

Resulting Table Structure:

employee_id	first_name	last_name	email
INT (PK)	VARCHAR(50)	VARCHAR(50)	VARCHAR(100)

Each email value must be unique across all rows.

Scenario: Dropping an index from a table

Question: How do you drop an index named idx_last_name from the employees table?

The DROP INDEX statement is used to remove an index when it is no longer needed, freeing up space and reducing maintenance overhead.

For Example:

```
DROP INDEX idx_last_name ON employees;
```

After this command, the index on last_name is removed, potentially slowing down
queries that rely on it.

Scenario: Ensuring a column cannot have NULL values

Question: How would you ensure that the last_name column in the employees table cannot store NULL values?

The NOT NULL constraint ensures that a column must contain a value in every row. If you attempt to insert a row without a value in the last_name column, the database will return an error. This constraint helps maintain data consistency and integrity by ensuring critical fields are always populated.

For Example:

```
ALTER TABLE employees
MODIFY last_name VARCHAR(50) NOT NULL;
```

Resulting Table Structure:

Now, if a department is removed from the departments table, all employees assigned to that department will also be deleted automatically.

Scenario: Renaming a table in the database

Question: How would you rename the employees table to staff?

The ALTER TABLE command with the RENAME TO clause allows us to rename a table while keeping its structure and data intact.

For Example:

```
ALTER TABLE employees RENAME TO staff;
```

Renamed Table Structure:

employee_id	first_name	last_name	hire_date
INT (PK)	VARCHAR(50)	VARCHAR(50)	DATE

After this, the table is referenced as staff instead of employees.

Scenario: Assigning a default value to a column

Question: How would you set the default value of the hire_date column to the current date in the employees table?

The DEFAULT constraint assigns a default value to a column when no value is explicitly provided during insertion. This ensures that even if the hire_date is not specified, the system will use the current date as the default value.

For Example:

```
ALTER TABLE employees
ALTER COLUMN hire_date SET DEFAULT CURRENT_DATE;
```

Resulting Table:

employee_id	first_name	last_name	hire_date(DEFAULT CURRENT_DATE)
INT (PK)	VARCHAR(50)	VARCHAR(50)	DATE

When inserting a new employee without specifying hire_date, the system will automatically assign the current date.

Scenario: Dropping a table from the database

Question: How do you permanently delete the departments table and all its data?

The DROP TABLE command removes the table from the database, including its structure and all the data it contains. This action is irreversible, so use it carefully.

For Example:

```
DROP TABLE departments;
```

After executing this command, the departments table and all its data are permanently deleted. Any references to this table, such as foreign keys, will also need to be handled to avoid integrity issues.

Scenario: Creating a table with a UNIQUE constraint on multiple columns

Question: How would you ensure that the combination of first_name and last_name in the employees table is unique?

A UNIQUE constraint on multiple columns ensures that the combination of values across those columns is unique across all rows.

For Example:

```
ALTER TABLE employees
ADD CONSTRAINT unique_full_name UNIQUE (first_name, last_name);
```

Resulting Table:

employee_id	first_name	last_name(UNIQUE)	hire_date
INT (PK)	VARCHAR(50)	VARCHAR(50)	DATE

This ensures that no two employees can have the same first and last name combination.

Scenario: Creating a bitmap index for performance improvement

Question: How would you create a bitmap index on the department_id column to improve query performance?

A bitmap index is efficient for columns with low cardinality, such as department_id, because it uses bitmaps to store information, improving query speed for filtering.

For Example:

```
CREATE BITMAP INDEX idx_department_id ON employees (department_id);
```

This index improves the performance of queries filtering by department_id.

Scenario: Using a sequence to generate unique employee IDs

Question: How would you use a sequence to automatically generate unique employee_id values in the employees table?

A sequence generates unique numbers, typically used for primary keys. The NEXTVAL function retrieves the next value in the sequence.

For Example:

```
CREATE SEQUENCE employee_seq START WITH 1 INCREMENT BY 1;

INSERT INTO employees (employee_id, first_name, last_name)
VALUES (employee_seq.NEXTVAL, 'John', 'Doe');
```

Resulting Data

employee_id	first_name	last_name	hire_date
1	John	Doe	NULL

Each time you insert a new employee, the employee_id is incremented automatically.

Scenario: Modifying a column's data type

Question: How would you change the salary column’s data type from INT to DECIMAL(10,2)?

The ALTER TABLE command allows changing the data type of an existing column. This is useful when more precision is needed for a column.

For Example:

```
ALTER TABLE employees
ALTER COLUMN salary DECIMAL(10, 2);
```

Resulting Table:

employee_id	first_name	last_name	salary (DECIMAL)
INT (PK)	VARCHAR(50)	VARCHAR(50)	DECIMAL(10,2)

This ensures that salaries can now have two decimal places for precision.

Scenario: Deleting all rows from a table without removing its structure

Question: How can you remove all rows from the employees table but retain the table structure?

The TRUNCATE TABLE statement removes all rows from a table while keeping its structure intact. It is faster than DELETE as it does not log individual row deletions.

For Example:

```
TRUNCATE TABLE employees;
```

Resulting Table:

employee_id	first_name	last_name	salary (DECIMAL)

All data is removed, but the table remains available for new entries.

Scenario: Handling conflicts when inserting rows with unique constraints

Question: How do you handle conflicts when inserting data into a table with a UNIQUE constraint?

The ON CONFLICT clause in PostgreSQL helps resolve conflicts during insertion by specifying what action to take if a conflict occurs.

For Example:

```
INSERT INTO employees (employee_id, first_name, last_name)
VALUES (1, 'John', 'Doe')
ON CONFLICT (employee_id)
DO UPDATE SET first_name = 'Updated John';
```

Resulting Data:

employee_id	first_name	last_name	hire_date
INT (PK)	Updated John	Doe	NULL

If an employee with the same employee_id already exists, the first_name is updated instead of causing an error.

Scenario: Handling conflicts when inserting rows with unique constraints

Question: How do you handle conflicts when inserting data into a table with a UNIQUE constraint?

The ON CONFLICT clause in PostgreSQL helps resolve conflicts during insertion by specifying what action to take if a conflict occurs.

For Example:

```
INSERT INTO employees (employee_id, first_name, last_name)
VALUES (1, 'John', 'Doe')
ON CONFLICT (employee_id)
DO UPDATE SET first_name = 'Updated John';
```

Resulting Data

employee_id	first_name	last_name	hire_date
1	John	Doe	NULL

If an employee with the same employee_id already exists, the first_name is updated instead of causing an error.

Scenario: Enforcing a Foreign Key with ON DELETE SET NULL

Question: How would you modify the employees table to ensure that if a related department is deleted, the department_id in the

The ON DELETE SET NULL clause ensures that when a row from the parent table (departments) is deleted, any related foreign key in the child table (employees) is set to NULL. This maintains the referential integrity of the database by avoiding orphaned records.

For Example:

```sql
CREATE TABLE departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(100)
);

CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    department_id INT,
    FOREIGN KEY (department_id)
    REFERENCES departments(department_id) ON DELETE SET NULL
);
```

Data Before Deletion:

employee_id	first_name	department_id
1	John	101
2	Jane	102

Resulting Data After Deleting Department 101:

employee_id	first_name	department_id
1	John	NULL
2	Jane	102

Scenario: Preventing Circular References with Foreign Keys

Question: How can you create tables with foreign keys without causing circular reference issues?

Circular references occur when two tables reference each other through foreign keys, which can lead to complex insertion order issues. To avoid this, you can carefully structure relationships, use deferred constraints, or avoid bi-directional foreign keys when possible.

For Example:

```sql
CREATE TABLE managers (
    manager_id INT PRIMARY KEY,
    manager_name VARCHAR(50)
);

CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    manager_id INT,
    FOREIGN KEY (manager_id) REFERENCES managers(manager_id)
);
```

Managers Table:

manager_id	manager_name
1	Alice

Employees Table:

employee_id	manager_id
101	1

This avoids circular references by making only employees reference managers.

Scenario: Using a Trigger to Automatically Update a Timestamp

Question: How can you create a trigger to update the last_modified column whenever a row in the employees table is updated?

A trigger automatically updates a column when a certain event occurs, such as an UPDATE. This ensures that last_modified reflects the latest change.

For Example:

```
ALTER TABLE employees ADD COLUMN last_modified TIMESTAMP;

CREATE OR REPLACE FUNCTION update_timestamp()
RETURNS TRIGGER AS $$
BEGIN
    NEW.last_modified := CURRENT_TIMESTAMP;
    RETURN NEW;
END;
$$ LANGUAGE plpg;

CREATE TRIGGER update_last_modified
BEFORE UPDATE ON employees
FOR EACH ROW EXECUTE FUNCTION update_timestamp();
```

Resulting Table:

employee_id	first_name	last_name	last_modified
1	John	Doe	2024-10-21

When an employee record is updated, the last_modified column reflects the current timestamp.

Scenario: Creating a Partitioned Table for Performance Optimization

Question: How can you partition the employees table by department_id to improve performance?

Partitioning splits a large table into smaller parts based on a column, improving query performance and management.

For Example:

```
CREATE TABLE employees (
    employee_id INT,
    first_name VARCHAR(50),
    department_id INT
) PARTITION BY HASH (department_id);

CREATE TABLE employees_part_1 PARTITION OF employees
FOR VALUES WITH (MODULUS 2, REMAINDER 0);

CREATE TABLE employees_part_2 PARTITION OF employees
FOR VALUES WITH (MODULUS 2, REMAINDER 1);
```

Partitioned Tables:

- employees_part_1 will store rows where department_id % 2 = 0
- employees_part_2 will store rows where department_id % 2 = 1

Partitioning helps queries on department_id run faster by scanning only relevant
partitions.

Scenario: Implementing Row-Level Security (RLS)

Question: How can you restrict access to the employees table based on user roles?

Row-Level Security (RLS) ensures that users only see rows they are authorized to view.

For Example:

```
ALTER TABLE employees ENABLE ROW LEVEL SECURITY;

CREATE POLICY hr_policy ON employees
FOR SELECT USING (current_user = 'hr_user');
```

Resulting Behavior:

When logged in as hr_user, only the rows accessible to that user will be visible, while other users are restricted.

Scenario: Creating a Recursive Query with Common Table Expressions (CTE)

Question: How would you use a recursive CTE to find all employees reporting to a specific manager?

A recursive CTE retrieves hierarchical data, such as finding all employees under a specific manager.

For Example:

```
WITH RECURSIVE employee_hierarchy AS (
    SELECT employee_id, first_name, manager_id
    FROM employees
    WHERE manager_id IS NULL
    UNION ALL
    SELECT e.employee_id, e.first_name, e.manager_id
    FROM employees e
    INNER JOIN employee_hierarchy eh
    ON e.manager_id = eh.employee_id
)
SELECT * FROM employee_hierarchy;
```

Resulting Table:

employee_id	first_name	manager_id
1	John	NULL
2	Jane	1

This retrieves all employees under each manager recursively.

Scenario: Creating a Materialized View for Faster Query Results

Question: How can you create a materialized view to store the results of an expensive query on the employees table?

A materialized view stores query results physically, improving performance for repeated access.

For Example:

```
CREATE MATERIALIZED VIEW employee_summary AS
SELECT department_id, COUNT(*) AS employee_count
FROM employees
GROUP BY department_id;
```

Resulting Materialized View:

department_id	employee_count
101	10
102	5

Scenario: Using Indexes for Full-Text Search

Question: How can you create an index on the employees table to support full-text search on the first_name column?

A full-text index improves search performance for text columns.

For Example:

```
CREATE INDEX idx_fulltext_name
ON employees USING gin(to_tsvector('english', first_name));
```

This index optimizes searches on the first_name column.

Scenario: Enforcing Data Consistency with Check Constraints

Question: How can you ensure that the salary column in the employees table is always greater than zero?

A CHECK constraint enforces business rules at the column level.

For Example:

```
ALTER TABLE employees
ADD CONSTRAINT check_salary_positive CHECK (salary > 0);
```

Resulting Table:

employee_id	salary
1	5000

If someone tries to insert a negative salary, the database will reject the operation.

Scenario: Creating a Function to Calculate Bonuses

Question: How can you create a function to calculate a 10% bonus for each employee's salary?

A user-defined function simplifies repeated calculations.

For Example:

```
CREATE OR REPLACE FUNCTION calculate_bonus(salary DECIMAL)
RETURNS DECIMAL AS $$
BEGIN
    RETURN salary * 0.10;
END;
$$ LANGUAGE plpg;

SELECT employee_id, first_name, calculate_bonus(salary) AS bonus
FROM employees;
```

Resulting Table:

employee_id	first_name	bonus
1	John	500

This function calculates a 10% bonus for each employee based on their salary.

Scenario: Creating a Table with Conditional Unique Constraints

Question: How would you ensure that employees within the same department cannot share the same email but allow duplicate emails across different departments?

A conditional unique constraint can be implemented using a composite unique index. This ensures that while employees in the same department must have unique email addresses, employees in different departments can share the same email.

For Example:

```
CREATE UNIQUE INDEX idx_unique_email_department
ON employees (email, department_id);
```

Resulting Data

employee_id	email	department_id
1	john@company.com	101
2	john@company.com	102

Scenario: Implementing Soft Deletes Using a Status Column

Question: How can you implement soft deletes on the employees table by adding a status column?

Soft deletes mark a row as inactive instead of deleting it. This preserves the data for future audits or analysis. A status column helps manage the active or inactive state of each record..

For Example:

```sql
ALTER TABLE employees ADD COLUMN status VARCHAR(10) DEFAULT 'active';

-- Mark an employee as inactive
UPDATE employees SET status = 'inactive' WHERE employee_id = 1;

-- Retrieve only active employees
SELECT * FROM employees WHERE status = 'active';
```

Resulting Table:

employee_id	first_name	status
1	John	inactive
2	Jane	active

Scenario: Using Window Functions to Calculate Running Totals

Question: How can you calculate a running total of salaries for employees ordered by their hire date?

A window function calculates cumulative totals within a specific partition or order. In this case, the running total of salaries is calculated based on hire_date.

For Example:

```
SELECT employee_id, first_name, salary,
       SUM(salary) OVER (ORDER BY hire_date) AS running_total
FROM employees;
```

Resulting Table:

employee_id	first_name	salary	running_total
1	John	5000	5000
2	Jane	4000	9000

Scenario: Creating a Temporary Table

Question: How can you create a temporary table to store intermediate results during a session?

Temporary tables store data only for the duration of the current session. They are useful when you need to store intermediate results or perform calculations without affecting the main database.

For Example:

```
CREATE TEMP TABLE temp_employees AS
SELECT * FROM employees WHERE salary > 3000;
```

Resulting Temporary Table:

employee_id	first_name	salary
1	John	5000
2	Jane	4000

Scenario: Adding Check Constraints on Multiple Columns

Question: How would you ensure that an employee's salary is at least twice their bonus?

A CHECK constraint enforces conditions across multiple columns to maintain data consistency.

For Example:

```
ALTER TABLE employees
ADD CONSTRAINT check_salary_bonus CHECK (salary >= 2 * bonus);
```

Resulting Table:

employee_id	salary	bonus
1	6000	3000

Scenario: Creating an Updatable View

Question: How can you create an updatable view that allows inserting, updating, or deleting rows in the employees table?

An updatable view allows modifications to be made to the underlying table through the view. This is useful when you want to restrict access to certain rows or columns while still allowing updates.

For Example:

```sql
CREATE VIEW active_employees AS
SELECT * FROM employees WHERE status = 'active';

-- Inserting a new active employee through the view
INSERT INTO active_employees (employee_id, first_name, salary)
VALUES (3, 'Alice', 7000);
```

Resulting Table (employees):

employee_id	first_name	salary	status
3	Alice	7000	active

Scenario: Handling Concurrent Data Modifications with Locks

Question: How can you lock the employees table to prevent concurrent modifications during critical operations?

Use the LOCK statement to prevent other sessions from modifying a table while critical operations are being performed.

For Example:

```
BEGIN;
LOCK TABLE employees IN EXCLUSIVE MODE;
-- Perform critical operations
COMMIT;
```

During the transaction, no other session can modify the employees table, ensuring
data consistency.

Scenario: Performing Bulk Inserts Efficiently

Question: How can you insert multiple rows into the employees table efficiently?

Bulk inserts minimize database overhead by reducing the number of insert operations, making them more efficient.

For Example:

```
INSERT INTO employees (employee_id, first_name, salary)
VALUES
(4, 'Bob', 5000),
(5, 'Charlie', 4500);
```

Resulting Table:

employee_id	first_name	salary
4	Bob	5000
5	Charlie	4000

Scenario: Using Common Table Expressions (CTE) for Modular Queries

Question: How can you use a CTE to write modular queries in the employees table?

A Common Table Expression (CTE) simplifies complex queries by breaking them into smaller, modular parts.

For Example:

```
WITH high_salary AS (
    SELECT * FROM employees WHERE salary > 4000
)
SELECT * FROM high_salary WHERE first_name LIKE 'J%';
```

Resulting Query Output:

employee_id	first_name	salary
1	John	5000

This query retrieves high-salary employees whose first names start with 'J'.

Scenario: Performing Conditional Updates with Case Statements

Question: How would you update employees' salaries based on their performance rating using a CASE statement?

The CASE statement allows you to apply conditional logic within SQL queries, such as updating salaries based on performance ratings.

For Example:

```
UPDATE employees
SET salary = CASE
    WHEN rating = 'A' THEN salary * 1.10
    WHEN rating = 'B' THEN salary * 1.05
    ELSE salary
END;
```

Resulting Table:

employee_id	salary	rating
1	5000	A
2	4200	B

Scenario: Filtering Employees Based on Matching Department Data

Question: How would you retrieve the names of employees who are part of a valid department using an INNER JOIN?

An INNER JOIN only retrieves rows where there are matching values in both joined tables. In this scenario, we want to ensure that only employees assigned to valid departments are shown.

For Example:

Consider the following tables:

employees

employee_id	name	department_id
1	Alice	101
2	Bob	102
3	Charlie	103

departments

department_id	department_name
101	HR
102	IT

Query:

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

Result:

name	department_name
Alice	HR
Bob	IT

Since Charlie (employee_id 3) is assigned to a department not present in the departments table, that row is excluded.

Scenario: Retrieving All Employees with Their Department Details or NULL

Question: How can you retrieve all employees, even those without a department, using a LEFT JOIN?

A LEFT JOIN retrieves all rows from the left table (employees), even if there are no matching rows in the right table (departments).

For Example:

employees

employee_id	name	department_id
1	Alice	101
2	Bob	102
3	Charlie	NULL

departments

department_id	department_name
101	HR
102	IT

Query:

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

Result:

name	department_name
Alice	HR
Bob	IT
Charlie	NULL

The third row shows NULL in the department_name column because Charlie is not assigned to any department.

Scenario: Identifying Unassigned Departments

Question: How can you retrieve all departments, including those without any employees assigned?

Using a RIGHT JOIN, we ensure all departments are included, even if there are no employees assigned to them.

For Example:

employees

employee_id	name	department_id
1	Alice	101
2	Bob	102

departments

department_id	department_name
101	HR
102	IT
103	Finance

Query:

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

Result:

name	department_name
Alice	HR
Bob	IT
NULL	Finance

The Finance department is listed with NULL for the employee name because it has no assigned employees.

Scenario: Showing Complete Employee and Department Data

Question: How would you display a report with all employees and all departments, even if they don't match?

A FULL OUTER JOIN retrieves all rows from both tables, with NULL values where there are no matches.

For Example:

employees

employee_id	name	department_id
1	Alice	101
2	Bob	104

departments

department_id	department_name
101	HR
102	IT

Query:

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

Result:

name	department_name
Alice	HR
Bob	NULL
NULL	IT

Scenario: Adding Check Constraints on Multiple Columns

Question: How would you ensure that an employee's salary is at least twice their bonus?

A CHECK constraint enforces conditions across multiple columns to maintain data consistency.

For Example:

```
ALTER TABLE employees
ADD CONSTRAINT check_salary_bonus CHECK (salary >= 2 * bonus);
```

Resulting Table:

employee_id	salary	bonus
1	6000	3000

Scenario: Creating an Updatable View

Question: How can you create an updatable view that allows inserting, updating, or deleting rows in the employees table?

An updatable view allows modifications to be made to the underlying table through the view. This is useful when you want to restrict access to certain rows or columns while still allowing updates.

For Example:

```sql
CREATE VIEW active_employees AS
SELECT * FROM employees WHERE status = 'active';

-- Inserting a new active employee through the view
INSERT INTO active_employees (employee_id, first_name, salary)
VALUES (3, 'Alice', 7000);
```

Resulting Table (employees):

employee_id	first_name	salary	status
3	Alice	7000	active

Scenario: Handling Concurrent Data Modifications with Locks

Question: How can you lock the employees table to prevent concurrent modifications during critical operations?

Use the LOCK statement to prevent other sessions from modifying a table while critical operations are being performed.

For Example:

```
BEGIN;
LOCK TABLE employees IN EXCLUSIVE MODE;
-- Perform critical operations
COMMIT;
```

During the transaction, no other session can modify the employees table, ensuring
data consistency.

Scenario: Performing Bulk Inserts Efficiently

Question: How can you insert multiple rows into the employees table efficiently?

Bulk inserts minimize database overhead by reducing the number of insert operations, making them more efficient.

For Example:

```
INSERT INTO employees (employee_id, first_name, salary)
VALUES
(4, 'Bob', 5000),
(5, 'Charlie', 4500);
```

Resulting Table:

employee_id	first_name	salary
4	Bob	5000
5	Charlie	4000

Scenario: Using Common Table Expressions (CTE) for Modular Queries

Question: How can you use a CTE to write modular queries in the employees table?

A Common Table Expression (CTE) simplifies complex queries by breaking them into smaller, modular parts.

For Example:

```
WITH high_salary AS (
    SELECT * FROM employees WHERE salary > 4000
)
SELECT * FROM high_salary WHERE first_name LIKE 'J%';
```

Resulting Query Output:

employee_id	first_name	salary
1	John	5000

This query retrieves high-salary employees whose first names start with 'J'.

Scenario: Performing Conditional Updates with Case Statements

Question: How would you update employees' salaries based on their performance rating using a CASE statement?

The CASE statement allows you to apply conditional logic within SQL queries, such as updating salaries based on performance ratings.

For Example:

```
UPDATE employees
SET salary = CASE
    WHEN rating = 'A' THEN salary * 1.10
    WHEN rating = 'B' THEN salary * 1.05
    ELSE salary
END;
```

Resulting Table:

employee_id	salary	rating
1	5000	A
2	4200	B

Scenario: Listing Managers and Subordinates from the Same Table

Question: How can you list employees along with their managers using a SELF JOIN?

A SELF JOIN joins a table with itself. This is useful when the table contains hierarchical data, like managers and employees.

For Example:

Consider the following tables:

employees

employee_id	name	manager_id
1	Alice	NULL
2	Bob	1
3	Charlie	1

Query:

```sql
SELECT e1.name AS employee, e2.name AS manager
FROM employees e1
LEFT JOIN employees e2
ON e1.manager_id = e2.employee_id;
```

Result:

employee	department_name
Alice	NULL
Bob	IT
Charlie	Alice

Scenario: Generating All Possible Combinations of Employees and Departments

Question: How would you generate every possible combination of employees and departments?

A CROSS JOIN creates a Cartesian product, generating every possible combination of rows from two tables.

For Example:

employees

employee_id	name
1	Alice
2	Bob

departments

department_id	department_name
101	HR
102	IT

Query:

```
SELECT e.name, d.department_name
FROM employees e
CROSS JOIN departments d;
```

Result:

name	department_name
Alice	HR
Alice	IT
Bob	HR
Bob	IT

Scenario: Retrieving Employees in a Specific Department Using a Subquery

Question: How would you find employees working in the 'Sales' department using a subquery?

In this scenario, a single-row subquery returns the department ID for 'Sales' and uses it in the outer query to filter employees.

For Example:

employees

employee_id	name	department_id
1	Alice	101
2	Bob	102
3	Eve	103

departments

department_id	department_name
101	HR
102	IT

Query:

```
SELECT name
FROM employees
WHERE department_id = (SELECT department_id FROM departments WHERE
department_name = 'Sales');
```

Result:

name
Alice

The subquery finds the department ID for 'Sales' (101), and the outer query retrieves
employees belonging to that department.

Scenario: Finding Employees with the Highest Salary in Each Department Using a Correlated Subquery

Question: How would you retrieve the names of the highest-paid employees in each department using a correlated subquery?

A correlated subquery compares each employee's salary with the highest salary within their department.

For Example:

employees

employee_id	name	department_id	salary
1	Alice	101	50000
2	Bob	104	60000
3	Eve	102	70000

Query:

```sql
SELECT name, salary
FROM employees e1
WHERE salary = (SELECT MAX(salary)
                FROM employees e2
                WHERE e1.department_id = e2.department_id);
```

Result:

name	salary
Alice	60000
Bob	70000

Bob and Eve are the highest-paid employees in their respective departments.

Scenario: Combining Employee Data from Two Departments Using UNION

Question: How can you combine the names of employees from two departments without duplicates?

Using the UNION operator ensures the result set contains unique employee names from the specified departments.

For Example:

employees

employee_id	name	department_id
1	Alice	101
2	Bob	102
3	Alice	102

Query:

```
SELECT name FROM employees WHERE department_id = 101
UNION
SELECT name FROM employees WHERE department_id = 102;
```

Result:

name
Alice
Bob
Alice

Even though Alice appears in both departments, she appears only once in the result set due to UNION.

Scenario: Generating All Possible Combinations of Employees and Departments

Question: How would you generate every possible combination of employees and departments?

A CROSS JOIN creates a Cartesian product, generating every possible combination of rows from two tables.

For Example:

Consider the following tables:

employees

employee_id	name
1	Alice
2	Bob

departments

department_id	department_name
101	HR
102	IT

Query:

```
SELECT e.name, d.department_name
FROM employees e
CROSS JOIN departments d;
```

Result:

employee	department_name
Alice	HR
Alice	IT
Bob	HR

Bob	IT

These detailed explanations with tables and results provide a clear understanding of
the scenarios and how SQL queries handle various real-world requirements

Scenario: Retrieving Employees in a Specific Department Using a Subquery

Question: How would you find employees working in the 'Sales' department using a subquery?

In this scenario, a single-row subquery returns the department ID for 'Sales' and uses it in the outer query to filter employees.

For Example:

employees

employee_id	name	department_id
1	Alice	101
2	Bob	102
3	Eve	103

departments

department_id	department_name
101	Sales
102	IT

Query:

```
SELECT name
FROM employees
WHERE department_id = (SELECT department_id FROM departments WHERE
department_name = 'Sales');
```

Result:

name
Alice

The subquery finds the department ID for 'Sales' (101), and the outer query retrieves
employees belonging to that department.

Scenario: Finding Employees with the Highest Salary in Each Department Using a Correlated Subquery

Question: How would you retrieve the names of the highest-paid employees in each department using a correlated subquery?

A correlated subquery compares each employee's salary with the highest salary within their department

For Example:

employees

employee_id	name	department_id	salary
1	Alice	101	50000
2	Bob	102	60000
3	Eve	103	70000

departments

department_id	department_name
101	HR
102	IT

Query:

```
SELECT name, salary
FROM employees e1
WHERE salary = (SELECT MAX(salary)
                FROM employees e2
                WHERE e1.department_id = e2.department_id);
```

Result:

name	salary
Bob	60000
Eve	70000

Scenario: Combining Employee Data from Two Departments Using UNION

Question: How can you combine the names of employees from two departments without duplicates?

Using the UNION operator ensures the result set contains unique employee names from the specified departments.

For Example:

employees

employee_id	name	department_id
1	Alice	101
2	Bob	102
3	Alice	102

Query:

```
SELECT name FROM employees WHERE department_id = 101
UNION
SELECT name FROM employees WHERE department_id = 102;
```

Result:

name
Alice
Bob

Even though Alice appears in both departments, she appears only once in the result
set due to UNION.

Scenario: Including Duplicates When Combining Employee Data Using UNION ALL

Question: How would you include duplicates when combining employee data from two departments?

The UNION ALL operator keeps all duplicates in the combined result.

For Example:

employees

employee_id	name	department_id
1	Alice	101
2	Bob	102
3	Alice	102

Query:

```sql
SELECT name FROM employees WHERE department_id = 101
UNION ALL
SELECT name FROM employees WHERE department_id = 102;
```

Result:

name
Alice
Bob
Alice

Since UNION ALL includes duplicates, Alice appears twice in the result set because
she belongs to both departments.

Scenario: Finding Employees Not Belonging to a Specific Department

Question: How can you retrieve employees who are not in the 'IT' department using a subquery?

You can use a NOT IN clause to filter employees by excluding those who belong to the 'IT' department. This subquery returns the department ID for 'IT', and the outer query retrieves employees not assigned to that department.

For Example:

employees

employee_id	name	department_id
1	Alice	101
2	Bob	102
3	Charlie	103

departments

department_id	department_name
102	IT
103	HR

Query:

```
SELECT name
FROM employees
WHERE department_id NOT IN (SELECT department_id FROM departments WHERE
department_name = 'IT');
```

Result:

name
Alice
Charlie

Scenario: Merging Data from Multiple Departments

Question: How can you combine employees from HR and IT departments, ensuring no duplicates?

You can use the UNION operator to combine employee data from multiple departments. The UNION operator removes duplicates automatically.

For Example:

employees

employee_id	name	department_id
1	Alice	101
2	Bob	102
3	Alice	103

Query:

```sql
SELECT name FROM employees WHERE department_id = 102
UNION
SELECT name FROM employees WHERE department_id = 103;
```

Result:

name
Alice
Bob

Even though Alice is in both departments, she appears only once in the result due to
UNION.

Scenario: Finding Common Employees Across Multiple Departments

Question: How would you find employees who belong to both the HR and IT departments?

The INTERSECT operator returns only the rows that appear in both result sets, identifying employees belonging to multiple departments.

For Example:

employees

employee_id	name	department_id
1	Alice	101
2	Bob	102
3	Alice	103

Query:

```
SELECT name FROM employees WHERE department_id = 102
INTERSECT
SELECT name FROM employees WHERE department_id = 103;
```

Result:

name
Alice

Alice appears because she is in both departments.

Scenario: Removing Data Matching a Certain Condition

Question: How can you list employees who belong to HR but not IT using the EXCEPT operator?

The EXCEPT operator returns the rows from the first query that do not appear in the second query, helping you exclude specific matches.

For Example:

employees

employee_id	name	department_id
1	Alice	102
2	Bob	103
3	Alice	103

Query:

```
SELECT name FROM employees WHERE department_id = 103
EXCEPT
SELECT name FROM employees WHERE department_id = 102;
```

Result:

name
Bob

Bob appears because he belongs only to HR and not IT.

Scenario: Creating a Non-Recursive CTE for Simplifying Queries

Question: How can you use a CTE to simplify a query for finding employees with high salaries?

A non-recursive CTE defines a temporary result set, which makes complex queries easier to read and reuse.

For Example:

employees

employee_id	name	salary
1	Alice	50000
2	Bob	70000
3	Eve	80000

Query:

```sql
WITH HighSalary AS (
  SELECT name, salary FROM employees WHERE salary > 60000
)
SELECT * FROM HighSalary;
```

Result:

name	salary
Bob	70000
Eve	80000

Since UNION ALL includes duplicates, Alice appears twice in the result set because
she belongs to both departments.

Scenario: Using a Recursive CTE to Generate Hierarchical Data

Question: How would you display employee hierarchies using a recursive CTE?

A recursive CTE helps you retrieve hierarchical data, such as managers and subordinates.

For Example:

employees

employee_id	name	manager_id
1	Alice	NULL
2	Bob	1
3	Eve	1
4	Dave	2

Query:

```
WITH EmployeeHierarchy AS (
  SELECT employee_id, name, manager_id FROM employees WHERE manager_id IS
NULL
  UNION ALL
  SELECT e.employee_id, e.name, e.manager_id
  FROM employees e
  JOIN EmployeeHierarchy h ON e.manager_id = h.employee_id
)
SELECT * FROM EmployeeHierarchy;
```

Result:

employee_id	name	manager_id
1	Alice	NULL
2	Bob	1
3	Eve	1
4	Dave	2

Scenario: Using a Temporary Table for Intermediate Data Storage

Question: How can you use a temporary table to store and manipulate intermediate data?

Temporary tables are useful for storing intermediate results during a session.

For Example:

employees

employee_id	name	salary
1	Alice	50000
2	Bob	70000
3	Eve	80000

Query:

```
CREATE TEMPORARY TABLE TempEmployees AS
SELECT * FROM employees WHERE salary > 60000;

SELECT * FROM TempEmployees;
```

Result:

employee_id	name	manager_id
2	Bob	70000
3	Eve	80000

Scenario: Generating Subtotals Using ROLLUP

Question: How would you calculate total salaries per department and the grand total using ROLLUP?

The ROLLUP operator generates subtotals and grand totals within a grouped result.

For Example:

employees

department_id	salary
101	50000
101	60000
102	80000

Query:

```
SELECT department_id, SUM(salary) AS total_salary
FROM employees
GROUP BY ROLLUP(department_id);
```

Result:

department_id	total_salary
2	110000
3	80000
NULL	190000

Scenario: Ranking Employees Within Departments

Question: How can you rank employees by salary within their departments?

The RANK() function assigns ranks to employees within each department based on their salary.

For Example:

employees

name	department_id	salary
Bob	101	70000
Alice	101	50000
Eve	102	80000

Query:

```
SELECT name, department_id, salary,
       RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS
rank
FROM employees;
```

Result:

name	department_id	total_salary	rank
Bob	2	70000	1
Alice	3	50000	2
Eve	NULL	80000	1

Scenario: Using the MERGE Statement for Data Synchronization

Question: How can you synchronize data between two employee tables using the MERGE statement?

The MERGE statement allows conditional inserts, updates, and deletes in a single query.

For Example:

employees

name	employee_id	salary
Bob	1	70000

new_employees

name	employee_id	salary
Bob	1	80000
Alice	2	60000

Query:

```
MERGE INTO employees AS target
USING new_employees AS source
ON target.employee_id = source.employee_id
WHEN MATCHED THEN
  UPDATE SET target.salary = source.salary
WHEN NOT MATCHED THEN
  INSERT (employee_id, name, salary)
  VALUES (source.employee_id, source.name, source.salary);
```

Result:

name	employee_id	salary
Bob	1	80000
Alice	2	60000

The query updates Bob’s salary and inserts Alice as a new employee.

Scenario: Identifying and Handling Duplicate Data with ROW_NUMBER()

Question: How would you identify and remove duplicate employee records based on the employee's name using the ROW_NUMBER() function?

The ROW_NUMBER() function assigns a unique sequential number to each row within a partition, ordered by a specific column. When duplicate records exist, this function helps keep only the first occurrence while identifying and deleting the rest.

For Example:

employees

employee_id	name	salary
1	Alice	50000
2	Bob	50000
3	Charlie	70000

Query:

```
WITH RankedEmployees AS (
  SELECT employee_id, name, salary,
         ROW_NUMBER() OVER (PARTITION BY name ORDER BY employee_id) AS
row_num
  FROM employees
)
DELETE FROM employees
WHERE employee_id IN (
  SELECT employee_id FROM RankedEmployees WHERE row_num > 1
```

Result:

employee_id	name	salary
1	Alice	50000
3	Bob	70000

The ROW_NUMBER() function assigns a unique rank to each duplicate entry based on the employee name. The DELETE statement keeps only the first occurrence (where row_num = 1) and removes the rest.

Scenario: Calculating Running Totals Using SUM() with a Window Function

Question: How can you calculate the cumulative salary for employees in the order of their hiring date?

The SUM() function with the OVER clause calculates a cumulative sum across rows while keeping individual rows intact. This is known as a running total.

For Example:
employees

employee_id	name	salary	hire_date
1	Alice	50000	2023-01-01
2	Bob	70000	2023-03-01
3	Alice	80000	2023-05-01

Query:

```
SELECT name, salary,
       SUM(salary) OVER (ORDER BY hire_date) AS running_total
FROM employees;
```

Result:

name	salary	running_total
Alice	50000	50000
Bob	70000	120000
Alice	80000	200000

The OVER clause ensures that the sum is calculated incrementally, row by row, in the order of hire_date.

Scenario: Implementing Recursive Queries for Organizational Hierarchies

Question: How can you retrieve the full reporting hierarchy of employees using a recursive CTE?

A recursive CTE is ideal for hierarchical data, such as employee-manager relationships. It repeatedly calls itself until all levels of the hierarchy are fetched.

For Example:

employees

employee_id	name	manager_id
1	Alice	NULL
2	Bob	1
3	Alice	2

Query:

```
WITH EmployeeHierarchy AS (
  SELECT employee_id, name, manager_id
  FROM employees
  WHERE manager_id IS NULL
  UNION ALL
  SELECT e.employee_id, e.name, e.manager_id
  FROM employees e
  JOIN EmployeeHierarchy h ON e.manager_id = h.employee_id
)
SELECT * FROM EmployeeHierarchy;
```

Result:

employee_id	name	manager_id
1	Alice	NULL
2	Bob	1
3	Charlie	2

The first part retrieves employees with no manager.
The recursive part joins employees with their managers, continuing until all relationships are found.

Scenario: Partitioning Data for Performance Optimization

Question: How can you create a partitioned table to store large datasets by year?

Partitioning splits large tables into smaller, more manageable segments based on a specified column, improving query performance by scanning only relevant partitions.

For Example:

```
CREATE TABLE Orders (
  order_id INT,
  order_date DATE,
  customer_id INT
) PARTITION BY RANGE (YEAR(order_date));
```

This table is partitioned by the year of order_date. Queries for a specific year will
only scan the corresponding partition, speeding up performance.

Scenario: Creating a Materialized View for Fast Query Results

Question: How would you use a materialized view to store aggregated data for fast access?

A materialized view stores the result of a query physically, improving performance for frequently accessed complex queries.

For Example:

employees

employee_id	name	department_id	salary
1	Alice	101	50000
2	Bob	101	70000

Query:

```
CREATE MATERIALIZED VIEW DepartmentSalary AS
SELECT department_id, SUM(salary) AS total_salary
FROM employees
GROUP BY department_id;
```

Result:

department_id	total_salary
101	120000

This view pre-aggregates salary data, enabling faster access for queries.

Scenario: Using a Recursive CTE to Generate Hierarchical Data

Question: How would you display employee hierarchies using a recursive CTE?

A recursive CTE helps you retrieve hierarchical data, such as managers and subordinates.

For Example:

employees

employee_id	name	manager_id
1	Alice	NULL
2	Bob	1
3	Eve	1
4	Dave	2

Query:

```
WITH EmployeeHierarchy AS (
  SELECT employee_id, name, manager_id FROM employees WHERE manager_id IS
NULL
  UNION ALL
  SELECT e.employee_id, e.name, e.manager_id
  FROM employees e
  JOIN EmployeeHierarchy h ON e.manager_id = h.employee_id
)
SELECT * FROM EmployeeHierarchy;
```

Result:

employee_id	name	manager_id
1	Alice	NULL
2	Bob	1
3	Eve	1
4	Dave	2

Scenario: Using a Temporary Table for Intermediate Data Storage

Question: How can you use a temporary table to store and manipulate intermediate data?

Temporary tables are useful for storing intermediate results during a session.

For Example:

employees

employee_id	name	salary
1	Alice	50000
2	Bob	70000
3	Eve	80000

Query:

```
CREATE TEMPORARY TABLE TempEmployees AS
SELECT * FROM employees WHERE salary > 60000;

SELECT * FROM TempEmployees;
```

Result:

employee_id	name	manager_id
2	Bob	70000
3	Eve	80000

Scenario: Generating Subtotals Using ROLLUP

Question: How would you calculate total salaries per department and the grand total using ROLLUP?

The ROLLUP operator generates subtotals and grand totals within a grouped result.

For Example:

employees

department_id	salary
101	50000
101	60000
102	80000

Query:

```
SELECT department_id, SUM(salary) AS total_salary
FROM employees
GROUP BY ROLLUP(department_id);
```

Result:

department_id	total_salary
2	110000
3	80000
NULL	190000

Scenario: Ranking Employees Within Departments

Question: How can you rank employees by salary within their departments?

The RANK() function assigns ranks to employees within each department based on their salary.

For Example:

employees

name	department_id	salary
Bob	101	70000
Alice	101	50000
Eve	102	80000

Query:

```
SELECT name, department_id, salary,
       RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS
rank
FROM employees;
```

Result:

name	department_id	total_salary	rank
Bob	2	70000	1
Alice	3	50000	2
Eve	NULL	80000	1

Scenario: Using the MERGE Statement for Data Synchronization

Question: How can you synchronize data between two employee tables using the MERGE statement?

The MERGE statement allows conditional inserts, updates, and deletes in a single query.

For Example:

employees

name	employee_id	salary
Bob	1	70000

new_employees

name	employee_id	salary
Bob	1	80000
Alice	2	60000

Query:

```
MERGE INTO employees AS target
USING new_employees AS source
ON target.employee_id = source.employee_id
WHEN MATCHED THEN
  UPDATE SET target.salary = source.salary
WHEN NOT MATCHED THEN
  INSERT (employee_id, name, salary)
  VALUES (source.employee_id, source.name, source.salary);
```

Result:

name	employee_id	salary
Bob	1	80000
Alice	2	60000

The query updates Bob’s salary and inserts Alice as a new employee.

Scenario: Identifying and Handling Duplicate Data with ROW_NUMBER()

Question: How would you identify and remove duplicate employee records based on the employee's name using the ROW_NUMBER() function?

The ROW_NUMBER() function assigns a unique sequential number to each row within a partition, ordered by a specific column. When duplicate records exist, this function helps keep only the first occurrence while identifying and deleting the rest.

For Example:

employees

employee_id	name	salary
1	Alice	50000
2	Bob	50000
3	Charlie	70000

Query:

```
WITH RankedEmployees AS (
  SELECT employee_id, name, salary,
         ROW_NUMBER() OVER (PARTITION BY name ORDER BY employee_id) AS
row_num
  FROM employees
)
DELETE FROM employees
WHERE employee_id IN (
  SELECT employee_id FROM RankedEmployees WHERE row_num > 1
```

Result:

employee_id	name	salary
1	Alice	50000
3	Bob	70000

The ROW_NUMBER() function assigns a unique rank to each duplicate entry based on the employee name. The DELETE statement keeps only the first occurrence (where row_num = 1) and removes the rest.

Scenario: Calculating Running Totals Using SUM() with a Window Function

Question: How can you calculate the cumulative salary for employees in the order of their hiring date?

The SUM() function with the OVER clause calculates a cumulative sum across rows while keeping individual rows intact. This is known as a running total.

For Example:
employees

employee_id	name	salary	hire_date
1	Alice	50000	2023-01-01
2	Bob	70000	2023-03-01
3	Alice	80000	2023-05-01

Query:

```
SELECT name, salary,
       SUM(salary) OVER (ORDER BY hire_date) AS running_total
FROM employees;
```

Result:

name	salary	running_total
Alice	50000	50000
Bob	70000	120000
Alice	80000	200000

The OVER clause ensures that the sum is calculated incrementally, row by row, in the order of hire_date.

Scenario: Implementing Recursive Queries for Organizational Hierarchies

Question: How can you retrieve the full reporting hierarchy of employees using a recursive CTE?

A recursive CTE is ideal for hierarchical data, such as employee-manager relationships. It repeatedly calls itself until all levels of the hierarchy are fetched.

For Example:

employees

employee_id	name	manager_id
1	Alice	NULL
2	Bob	1
3	Alice	2

Query:

```
WITH EmployeeHierarchy AS (
  SELECT employee_id, name, manager_id
  FROM employees
  WHERE manager_id IS NULL
  UNION ALL
  SELECT e.employee_id, e.name, e.manager_id
  FROM employees e
  JOIN EmployeeHierarchy h ON e.manager_id = h.employee_id
)
SELECT * FROM EmployeeHierarchy;
```

Result:

employee_id	name	manager_id
1	Alice	NULL
2	Bob	1
3	Charlie	2

The first part retrieves employees with no manager.
The recursive part joins employees with their managers, continuing until all relationships are found.

Scenario: Partitioning Data for Performance Optimization

Question: How can you create a partitioned table to store large datasets by year?

Partitioning splits large tables into smaller, more manageable segments based on a specified column, improving query performance by scanning only relevant partitions.

For Example:

```
CREATE TABLE Orders (
  order_id INT,
  order_date DATE,
  customer_id INT
) PARTITION BY RANGE (YEAR(order_date));
```

This table is partitioned by the year of order_date. Queries for a specific year will
only scan the corresponding partition, speeding up performance.

Scenario: Creating a Materialized View for Fast Query Results

Question: How would you use a materialized view to store aggregated data for fast access?

A materialized view stores the result of a query physically, improving performance for frequently accessed complex queries.

For Example:

employees

employee_id	name	department_id	salary
1	Alice	101	50000
2	Bob	101	70000

Query:

```
CREATE MATERIALIZED VIEW DepartmentSalary AS
SELECT department_id, SUM(salary) AS total_salary
FROM employees
GROUP BY department_id;
```

Result:

department_id	total_salary
101	120000

This view pre-aggregates salary data, enabling faster access for queries.

Scenario: Synchronizing Two Tables Using MERGE

Question: How can you synchronize employee data between two tables using the MERGE statement?

The MERGE statement combines INSERT, UPDATE, and DELETE operations in one query, synchronizing data between two tables.

For Example:
employees

employee_id	name	salary
1	Bob	70000

new employees

employee_id	name	salary
1	Bob	80000
2	Alice	60000

Query:

```
MERGE INTO employees AS target
USING new_employees AS source
ON target.employee_id = source.employee_id
WHEN MATCHED THEN
  UPDATE SET target.salary = source.salary
WHEN NOT MATCHED THEN
  INSERT (employee_id, name, salary)
  VALUES (source.employee_id, source.name, source.salary);
```

Result:

employee_id	name	salary
1	Bob	80000
2	Alice	60000

Scenario: Handling Deadlocks Using Transaction Isolation Levels

Question: How can you avoid deadlocks by using appropriate isolation levels?

Deadlocks occur when transactions block each other. Setting the transaction isolation level to READ COMMITTED ensures that only committed data is read, reducing lock contention.

For Example:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

BEGIN TRANSACTION;
UPDATE accounts SET balance = balance - 500 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 500 WHERE account_id = 2;
COMMIT;
```

This setup reduces the chance of deadlocks.

Scenario: Creating Indexes for Faster Joins

Question: How can indexing improve the performance of join operations?

Creating an index on join columns speeds up the matching process by reducing the number of rows the database scans.

For Example:

```
CREATE INDEX idx_emp_dept ON employees(department_id);
```

This index optimizes join queries involving department_id.

Scenario: Using Window Functions for Row Numbering and Ranking

Question: How would you use ROW_NUMBER() and RANK() to rank employees by salary within departments?

The ROW_NUMBER() and RANK() functions assign row numbers and ranks based on salary, but they differ in handling ties.

For Example:

employees

employee_id	name	department_id	salary
1	Bob	101	70000
2	Alice	101	50000
3	Eve	102	80000

Query:

```
SELECT name, department_id, salary,
       RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS
rank,
       ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary DESC)
AS row_num
FROM employees;
```

Result:

name	department_id	salary	rank	row_num
Bob	101	70000	1	1
Alice	101	50000	2	2
Eve	102	80000	1	1

Scenario: Updating Employee Bonuses Using a CASE Statement

Question: How can you update employee bonuses based on their salary using a CASE statement?

The CASE statement in SQL allows you to apply conditional logic inside a query. It evaluates conditions and returns specific values based on the result of those conditions. You can use it in UPDATE queries to assign different bonuses based on salary ranges.

For Example:

employees

name	employee_id	salary	bonus
Bob	1	50000	NULL
Alice	2	70000	NULL
Eve	3	80000	NULL

Query:

```sql
UPDATE employees
SET bonus =
  CASE
    WHEN salary > 70000 THEN 1000
    WHEN salary BETWEEN 50000 AND 70000 THEN 500
    ELSE 0
  END;
```

Explanation:

- If an employee's salary is greater than 70,000, they receive a bonus of 1000.
- If the salary falls between 50,000 and 70,000, the bonus is 500.
- Otherwise, the bonus is 0.

Result:

name	employee_id	salary	bonus
Bob	1	50000	500
Alice	2	70000	500
Eve	3	80000	1000

Scenario: Calculating Percentage Contribution Using Window Functions

Question: How can you calculate the percentage contribution of each employee’s salary to the total salary within their department?

The SUM() function with the OVER clause calculates totals without collapsing rows. By dividing an employee's salary by the department's total salary, you can compute the percentage contribution of each employee.

For Example:

employees

employee_id	name	department_id	salary
1	Alice	101	50000
2	Bob	101	70000
3	Charlie	102	80000

Query:

```
SELECT name, department_id, salary,
       ROUND((salary * 100.0) / SUM(salary) OVER (PARTITION BY
department_id), 2) AS percentage
FROM employees;
```

Explanation:

- SUM() with OVER (PARTITION BY department_id) calculates the total salary
within each department.
- The individual salary is divided by the department total to determine the
percentage contribution.
- ROUND() limits the percentage to two decimal places.

Result:

department_id	name	salary	percentage
101	Alice	50000	41.67
101	Bob	70000	58.33
102	Eve	80000	100.00

Scenario: Identifying Missing Order IDs Using Window Functions

Question: How can you identify missing order IDs in a sequence using a query?

You can use the LEAD() window function to find gaps between consecutive order IDs. This helps identify if there are missing entries in a sequential dataset.

For Example:
orders

order_id	order_date
1	2023-01-01
2	2023-01-03
4	2023-01-05

Query:

```sql
SELECT order_id,
       LEAD(order_id) OVER (ORDER BY order_id) AS next_order_id
FROM orders
WHERE LEAD(order_id) OVER (ORDER BY order_id) - order_id > 1;
```

Result:

order_id	next_order_id
2	4

The result indicates that order ID 3 is missing.

Scenario: Detecting the First and Last Orders by Each Customer

Question: How can you find the first and last orders placed by each customer?

Use ROW_NUMBER() to assign row numbers to orders, ordered by date, and identify the first and last orders for each customer.

For Example:

employees

order_id	customer_id	order_date
1	1	2023-01-01
2	1	2023-01-05
3	2	2023-01-03

Query:

```
WITH RankedOrders AS (
  SELECT order_id, customer_id, order_date,
         ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date
ASC) AS first_order,
         ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date
DESC) AS last_order
  FROM orders
)
SELECT order_id, customer_id, order_date
FROM RankedOrders
WHERE first_order = 1 OR last_order = 1;
```

Result:

order_id	customer_id	order_date
1	1	2023-01-01
2	1	2023-01-05
3	2	2023-01-03

Scenario: Aggregating Data with GROUPING SETS

Question: How can you aggregate data by both department and location in a single query?

GROUPING SETS allows multiple levels of aggregation in one query, without writing multiple GROUP BY queries.

For Example:

department_id	name	salary	location
101	Alice	50000	NY
101	Bob	70000	NY
102	Eve	80000	SF

Query:

```
SELECT department_id, location, SUM(salary) AS total_salary
FROM employees
GROUP BY GROUPING SETS ((department_id), (location));
```

Result:

department_id	location	total_salary
101	NULL	120000
102	NULL	80000
NULL	NY	120000
NULL	SF	80000

This query calculates total salaries by department and by location separately.

Scenario: Handling Null Values with COALESCE()

Question: How can you replace NULL values with default text in a join?

Use COALESCE() to return the first non-null value in a list.

For Example:

employees

employee_id	name	department_id
1	Alice	101
2	Bob	NULL

departments

department_id	department_name
101	HR

Query:

```
SELECT e.name,
       COALESCE(d.department_name, 'Not Assigned') AS department_name
FROM employees e
LEFT JOIN departments d ON e.department_id = d.department_id;
```

Result:

name	department_name
Alice	HR
Bob	Not Assigned

Scenario: Using Recursive CTEs for Cumulative Sales

Question: How can you calculate cumulative sales using a recursive CTE?

The MERGE statement combines INSERT, UPDATE, and DELETE operations in one query, synchronizing data between two tables.

For Example:
sales

month	sales
Jan	100
Feb	200
Mar	150

Query:

```sql
WITH MonthlySales AS (
  SELECT month, sales,
         ROW_NUMBER() OVER (ORDER BY month) AS rn
  FROM sales
),
RecursiveSales AS (
  SELECT month, sales, rn, sales AS cumulative_sales
  FROM MonthlySales WHERE rn = 1
  UNION ALL
  SELECT ms.month, ms.sales, ms.rn, rs.cumulative_sales + ms.sales
  FROM MonthlySales ms

  JOIN RecursiveSales rs ON ms.rn = rs.rn + 1
)
SELECT month, cumulative_sales FROM RecursiveSales;
```

Result:

month	cumulative_sales
Jan	100
Feb	300
Mar	450

Scenario: Using EXCEPT to Find Missing Data

Question: How can you find customers who placed an order in 2022 but did not place any orders in 2023?

The EXCEPT operator compares the results of two queries and returns rows that appear in the first query but not in the second. This is useful for identifying records that exist in one dataset but are missing in another.

For Example:

order_id	customer_id	order_year
1	1	2022
2	1	2022
3	2	2023

This setup reduces the chance of deadlocks.

Query:

```sql
SELECT customer_id FROM orders WHERE order_year = 2022
EXCEPT
SELECT customer_id FROM orders WHERE order_year = 2023;
```

Result:

customer_id
2

Customer 2 placed an order in 2022 but did not place any orders in 2023.

Scenario: Creating a Pivot Table to Display Employee Count by Department and Location

Question: How can you generate a pivot table showing the number of employees by department and location?

A pivot table transforms row data into columns, summarizing the data for easier analysis. SQL's PIVOT operator can help achieve this transformation.

For Example:

department_id	name	employee_id	location
101	Alice	1	NY
101	Bob	2	SF
102	Eve	3	NY
102	John	4	SF

Query:

```
SELECT *
FROM (
  SELECT department_id, location, employee_id
  FROM employees
) AS SourceTable
PIVOT (
  COUNT(employee_id)
  FOR location IN ([NY], [SF])
) AS PivotTable;
```

Result:

department_id	NY	SF
101	1	1
102	1	1

The inner query selects relevant columns for the pivot.The PIVOT clause counts employees per department by location (NY and SF).

Scenario: Dynamically Generating Columns in a Pivot Table

Question: How can you dynamically generate pivot table columns for locations using SQL?

When you don't know the exact values (such as location names) in advance, dynamic SQL can be used to generate pivot columns at runtime.

For Example:

employees

department_id	name	employee_id	location
101	Alice	1	NY
101	Bob	2	SF
102	Eve	3	NY
102	John	4	SF

Query:

```
DECLARE @cols NVARCHAR(MAX);
DECLARE @query NVARCHAR(MAX);

-- Get unique locations dynamically
SELECT @cols = STRING_AGG(QUOTENAME(location), ', ')
FROM (SELECT DISTINCT location FROM employees) AS Locations;

-- Construct the dynamic SQL query
SET @query = '
SELECT *
FROM (
    SELECT department_id, location, employee_id
    FROM employees
) AS SourceTable
PIVOT (
    COUNT(employee_id)
    FOR location IN (' + @cols + ')
) AS PivotTable;';

-- Execute the dynamic query
EXEC sp_execute @query;
```

Result:

department_id	NY	SF
101	1	1
102	1	1

STRING_AGG() generates a comma-separated list of locations dynamically.
sp_execute executes the dynamically constructed SQL query.

Scenario: Using CASE to Classify Employees Based on Salary

Question: How can you classify employees as 'Low', 'Medium', or 'High' income earners based on their salary?

The CASE statement allows you to apply conditional logic to classify employees based on their salary.

For Example:

employees

employee_id	name	department_id
1	Alice	50000
2	Bob	70000
3	Eve	100000

Query:

```sql
SELECT name, salary,
       CASE
           WHEN salary < 60000 THEN 'Low'
           WHEN salary BETWEEN 60000 AND 90000 THEN 'Medium'
           ELSE 'High'
       END AS income_category
FROM employees;
```

Result:

name	salary	income_category
Alice	50000	Low
Bob	70000	Medium
Eve	100000	High

- If the salary is below 60,000, the employee is classified as 'Low'.
- If the salary is between 60,000 and 90,000, they are 'Medium'.
- Salaries above 90,000 are categorized as 'High'.

Scenario: Assign unique row numbers to employees within each department based on their salaries.

Question: How can you assign unique row numbers to employees in each department using the ROW_NUMBER() function?

The ROW_NUMBER() function generates a unique sequential number for each row in a result set. Using PARTITION BY allows resetting the row number for each partition (in this case, each department). This is particularly useful when ranking employees or generating identifiers within a department, especially for ordered datasets such as salaries.

For Example:

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

Sample Data:

department_id	employee_id	salary
1	101	60000

1	102	55000
2	103	70000
2	104	60000

Resulting Table:

department_id	employee_id	salary	row_num
1	101	60000	1
1	102	55000	2
2	103	70000	1
2	104	60000	2

In this output, ROW_NUMBER() restarts for each department, ranking employees
within their departments based on their salaries.

Scenario: Rank employees across the organization based on their salaries.

Question: How can you use the RANK() function to assign ranks to employees across the entire organization based on salary?

The RANK() function assigns identical ranks to rows with equal values. If there are ties, it leaves gaps in the sequence. This is useful for identifying overall leaders in performance or earnings, accounting for ties without disrupting the rank logic.

For Example:

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

Sample Data:

employee_id	salary
101	60000
102	60000
103	55000

Resulting Table::

employee_id	salary	salary_rank
101	60000	1
102	60000	1
103	55000	3

In this result, two employees with the same salary share rank 1. The next employee
receives rank 3, leaving a gap due to the tie.

Scenario: Identify the top 3 earners in each department.

Question: How can you use RANK() to identify the top 3 earners in each department?

The RANK() function with PARTITION BY allows you to calculate ranks within a group (in this case, departments). You can then filter the top 3 earners using a WHERE clause.

For Example:

```
WITH RankedEmployees AS (
    SELECT department_id, employee_id, salary,
           RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS
rank
    FROM employees
)
SELECT department_id, employee_id, salary
FROM RankedEmployees
WHERE rank <= 3;
```

Sample Data:

department_id	employee_id	salary
1	101	60000

1	102	50000
1	103	70000
2	201	80000
2	202	75000

Result:

department_id	employee_id	salary	rank
1	101	60000	1
1	102	55000	2
2	103	50000	3
2	104	80000	1
2	105	75000	2

Scenario: Assign dense ranks to employees within a department based on performance scores.

Question: How can you assign dense ranks to employees based on their performance within each department?

The DENSE_RANK() function assigns ranks without leaving gaps between them, even if multiple rows have the same value. This makes it ideal for scenarios where a continuous ranking is required, such as ranking employees by performance.

For Example:

```
SELECT department_id, employee_id, performance_score,
       DENSE_RANK() OVER (PARTITION BY department_id ORDER BY
performance_score DESC) AS dense_rank
FROM employees;
```

Sample Data:

department_id	employee_id	performance_score
1	101	90

1	102	90
1	103	85
2	201	88
2	202	85

Resulting Table:

department_id	employee_id	salary	dense_rank
1	101	90	1
1	102	90	1
2	103	85	2
2	104	88	1
2	105	85	2

Scenario: Divide employees into 4 equal groups based on their hire date.

Question: How can you use the NTILE() function to divide employees into 4 equal groups based on hire date?

The NTILE() function distributes rows into a specified number of buckets. If the rows don't divide evenly, earlier buckets will contain an extra row.

For Example:

```
SELECT employee_id, hire_date,
       NTILE(4) OVER (ORDER BY hire_date) AS hire_group
FROM employees;
```

Sample Data:

employee_id	hire_date
101	2021-01-01
102	2021-02-15

103	2021-03-10
104	2021-04-05
105	2021-05-20

Result:

employee_id	hire_date	hire_group
101	2021-01-01	1
102	2021-02-15	1
103	2021-03-10	2
104	2021-04-05	3
105	2021-05-20	4

Scenario: Using Recursive CTEs for Cumulative Sales

Question: How can you calculate cumulative sales using a recursive CTE?

The MERGE statement combines INSERT, UPDATE, and DELETE operations in one query, synchronizing data between two tables.

For Example:
sales

month	sales
Jan	100
Feb	200
Mar	150

Query:

```sql
WITH MonthlySales AS (
  SELECT month, sales,
         ROW_NUMBER() OVER (ORDER BY month) AS rn
  FROM sales
),
RecursiveSales AS (
  SELECT month, sales, rn, sales AS cumulative_sales
  FROM MonthlySales WHERE rn = 1
  UNION ALL
  SELECT ms.month, ms.sales, ms.rn, rs.cumulative_sales + ms.sales
  FROM MonthlySales ms

  JOIN RecursiveSales rs ON ms.rn = rs.rn + 1
)
SELECT month, cumulative_sales FROM RecursiveSales;
```

Result:

month	cumulative_sales
Jan	100
Feb	300
Mar	450

Scenario: Using EXCEPT to Find Missing Data

Question: How can you find customers who placed an order in 2022 but did not place any orders in 2023?

The EXCEPT operator compares the results of two queries and returns rows that appear in the first query but not in the second. This is useful for identifying records that exist in one dataset but are missing in another.

For Example:

order_id	customer_id	order_year
1	1	2022
2	1	2022
3	2	2023

This setup reduces the chance of deadlocks.

Query:

```sql
SELECT customer_id FROM orders WHERE order_year = 2022
EXCEPT
SELECT customer_id FROM orders WHERE order_year = 2023;
```

Result:

customer_id
2

Customer 2 placed an order in 2022 but did not place any orders in 2023.

Scenario: Creating a Pivot Table to Display Employee Count by Department and Location

Question: How can you generate a pivot table showing the number of employees by department and location?

A pivot table transforms row data into columns, summarizing the data for easier analysis. SQL's PIVOT operator can help achieve this transformation.

For Example:

department_id	name	employee_id	location
101	Alice	1	NY
101	Bob	2	SF
102	Eve	3	NY
102	John	4	SF

Query:

```
SELECT *
FROM (
  SELECT department_id, location, employee_id
  FROM employees
) AS SourceTable
PIVOT (
  COUNT(employee_id)
  FOR location IN ([NY], [SF])
) AS PivotTable;
```

Result:

department_id	NY	SF
101	1	1
102	1	1

The inner query selects relevant columns for the pivot.The PIVOT clause counts employees per department by location (NY and SF).

Scenario: Dynamically Generating Columns in a Pivot Table

Question: How can you dynamically generate pivot table columns for locations using SQL?

When you don't know the exact values (such as location names) in advance, dynamic SQL can be used to generate pivot columns at runtime.

For Example:

employees

department_id	name	employee_id	location
101	Alice	1	NY
101	Bob	2	SF
102	Eve	3	NY
102	John	4	SF

Query:

```
DECLARE @cols NVARCHAR(MAX);
DECLARE @query NVARCHAR(MAX);

-- Get unique locations dynamically
SELECT @cols = STRING_AGG(QUOTENAME(location), ', ')
FROM (SELECT DISTINCT location FROM employees) AS Locations;

-- Construct the dynamic SQL query
SET @query = '
SELECT *
FROM (
    SELECT department_id, location, employee_id
    FROM employees
) AS SourceTable
PIVOT (
    COUNT(employee_id)
    FOR location IN (' + @cols + ')
) AS PivotTable;';

-- Execute the dynamic query
EXEC sp_execute @query;
```

Result:

department_id	NY	SF
101	1	1
102	1	1

STRING_AGG() generates a comma-separated list of locations dynamically.
sp_execute executes the dynamically constructed SQL query.

Scenario: Using CASE to Classify Employees Based on Salary

Question: How can you classify employees as 'Low', 'Medium', or 'High' income earners based on their salary?

The CASE statement allows you to apply conditional logic to classify employees based on their salary.

For Example:

employees

employee_id	name	department_id
1	Alice	50000
2	Bob	70000
3	Eve	100000

Query:

```sql
SELECT name, salary,
       CASE
           WHEN salary < 60000 THEN 'Low'
           WHEN salary BETWEEN 60000 AND 90000 THEN 'Medium'
           ELSE 'High'
       END AS income_category
FROM employees;
```

Result:

name	salary	income_category
Alice	50000	Low
Bob	70000	Medium
Eve	100000	High

- If the salary is below 60,000, the employee is classified as 'Low'.
- If the salary is between 60,000 and 90,000, they are 'Medium'.
- Salaries above 90,000 are categorized as 'High'.