Sql_queries_solutions

May 25, 2024

0.0.1 Select All Columns From a Table

Query:

```
SELECT * FROM employees;
```

0.0.2 Select Specific Columns

Query:

```
SELECT first_name, last_name, salary FROM employees;
```

0.0.3 Distinct Values

Query:

```
SELECT DISTINCT job_title FROM employees;
```

0.0.4 Where Clause

Query:

```
SELECT * FROM employees WHERE salary > 50000;
```

0.0.5 AND, OR Clauses

Query:

```
SELECT * FROM employees WHERE salary > 50000 AND department_id = (SELECT department_id FROM depa
```

0.0.6 Order By

Query:

```
SELECT * FROM employees ORDER BY hire_date DESC;
```

0.0.7 Limit Clause

Query:

```
SELECT * FROM employees ORDER BY salary DESC LIMIT 5;
```

0.0.8 Between Clause

Query:

```
SELECT * FROM employees WHERE hire_date BETWEEN '2020-01-01' AND '2021-01-01';
```

0.0.9 IN Clause

Query:

```
SELECT * FROM employees WHERE department_id IN (SELECT department_id FROM departments WHERE depa
```

0.0.10 LIKE Clause

Query:

```
SELECT * FROM employees WHERE first_name LIKE 'A%';
```

These queries should work with the provided table structures and sample data. Each query demonstrates a different basic SQL concept, allowing you to retrieve and manipulate data from the `employees` table effectively.

Sure! Here are the answers for queries 11-50, marked down as requested, along with the SQL code for each query.

0.0.11 Aggregate Functions

11. COUNT Function Count the number of employees in the employees table.

```
SELECT COUNT(*) AS employee_count FROM employees;
```

12. SUM Function Calculate the total salary of all employees.

```
SELECT SUM(salary) AS total_salary FROM employees;
```

13. AVG Function Calculate the average salary of all employees.

```
SELECT AVG(salary) AS average_salary FROM employees;
```

14. MIN Function Find the minimum salary in the employees table.

```
SELECT MIN(salary) AS minimum_salary FROM employees;
```

15. MAX Function Find the maximum salary in the employees table.

```
SELECT MAX(salary) AS maximum_salary FROM employees;
```

16. GROUP BY Group employees by department and count the number of employees in each department.

```
SELECT department_id, COUNT(*) AS employee_count
FROM employees
GROUP BY department_id;
```

17. HAVING Clause Find departments with more than 10 employees.

```
SELECT department_id, COUNT(*) AS employee_count
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 10;
```

18. GROUP BY with Aggregate Functions Find the average salary for each department.

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

0.0.12 Joins

19. Inner Join Select all employees and their respective department names.

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

20. Left Join Select all employees and their respective department names, including those without a department.

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

21. Right Join Select all departments and their respective employees, including those without employees.

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

22. Full Outer Join Select all employees and departments, including those without matches.

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

23. Self Join Find pairs of employees who have the same manager.

```sql
SELECT e1.first_name AS employee1, e2.first_name AS employee2, e1.manager_id
FROM employees e1
JOIN employees e2
ON e1.manager_id = e2.manager_id
WHERE e1.employee_id != e2.employee_id;
```

24. Cross Join Generate all possible combinations of employees and departments.

```sql
SELECT e.*, d.*
FROM employees e
CROSS JOIN departments d;
```

25. Join on Multiple Conditions Select all employees and their projects, including the project start date and end date.

```sql
SELECT e.*, p.project_name, p.start_date, p.end_date
FROM employees e
JOIN projects p
ON e.employee_id = p.employee_id;
```

0.0.13 Subqueries

26. Subquery in SELECT Select the name of each employee and their department's total salary.

```sql
SELECT e.first_name, e.last_name,
       (SELECT SUM(salary) FROM employees WHERE department_id = e.department_id) AS total_salary
FROM employees e;
```

27. Subquery in FROM Select the average salary of departments with an average salary greater than 60,000.

```sql
SELECT department_id, avg_salary
FROM (SELECT department_id, AVG(salary) AS avg_salary FROM employees GROUP BY department_id) AS
WHERE avg_salary > 60000;
```

28. Subquery in WHERE Select all employees who have a salary greater than the average salary of their department.

```sql
SELECT *
FROM employees e
WHERE salary > (SELECT AVG(salary) FROM employees WHERE department_id = e.department_id);
```

29. Correlated Subquery Select all employees who have the highest salary in their department.

```sql
SELECT *
FROM employees e
WHERE salary = (SELECT MAX(salary) FROM employees WHERE department_id = e.department_id);
```

30. Subquery with EXISTS Select all employees who have been assigned at least one project.

```sql
SELECT *
FROM employees e
WHERE EXISTS (SELECT 1 FROM projects p WHERE p.employee_id = e.employee_id);
```

0.0.14 String Functions

31. Concatenate Concatenate the first name and last name of employees.

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM employees;
```

32. Substring Select the first three characters of the name column for all employees.

```
SELECT SUBSTRING(first_name, 1, 3) AS short_name FROM employees;
```

33. Length Find the length of each employee's name.

```
SELECT first_name, LENGTH(first_name) AS name_length FROM employees;
```

34. UPPER and LOWER Convert all employee names to uppercase.

```
SELECT UPPER(first_name) AS upper_name FROM employees;
```

35. TRIM Trim leading and trailing spaces from employee names.

```
SELECT TRIM(first_name) AS trimmed_name FROM employees;
```

36. REPLACE Replace all occurrences of 'Manager' with 'Team Lead' in the job title column.

```
SELECT REPLACE(job_title, 'Manager', 'Team Lead') AS new_job_title FROM employees;
```

0.0.15 Date Functions

37. Current Date Select the current date.

```
SELECT CURRENT_DATE;
```

38. Date Difference Calculate the number of days each employee has been with the company.

```
SELECT first_name, DATEDIFF(CURRENT_DATE, hire_date) AS days_with_company FROM employees;
```

39. Extract Year Extract the year from the hire_date of each employee.

```
SELECT first_name, EXTRACT(YEAR FROM hire_date) AS hire_year FROM employees;
```

40. Extract Month Extract the month from the hire_date of each employee.

```
SELECT first_name, EXTRACT(MONTH FROM hire_date) AS hire_month FROM employees;
```

41. Date Add Add 1 year to the hire_date of each employee.

```
SELECT first_name, hire_date, DATE_ADD(hire_date, INTERVAL 1 YEAR) AS new_hire_date FROM employe
```

42. Date Subtract Subtract 1 year from the hire_date of each employee.

```
SELECT first_name, hire_date, DATE_SUB(hire_date, INTERVAL 1 YEAR) AS new_hire_date FROM employe
```

0.0.16 Advanced SQL Queries

43. CASE Statement Categorize employees into salary ranges (e.g., 'Low', 'Medium', 'High').

```sql
SELECT first_name, salary,
       CASE
           WHEN salary < 50000 THEN 'Low'
           WHEN salary BETWEEN 50000 AND 80000 THEN 'Medium'
           ELSE 'High'
       END AS salary_range
FROM employees;
```

44. Pivot Pivot the data to show departments as columns and count of employees in each department.

```sql
SELECT department_name,
       COUNT(CASE WHEN department_name = 'HR' THEN 1 END) AS HR,
       COUNT(CASE WHEN department_name = 'Finance' THEN 1 END) AS Finance,
       COUNT(CASE WHEN department_name = 'IT' THEN 1 END) AS IT,
       COUNT(CASE WHEN department_name = 'Marketing' THEN 1 END) AS Marketing,
       COUNT(CASE WHEN department_name = 'Sales' THEN 1 END) AS Sales
FROM employees e
JOIN departments d ON e.department_id = d.department_id;
```

45. Unpivot Unpivot the data to convert columns into rows.

```sql
SELECT employee_id, attribute, value
FROM (
    SELECT employee_id, first_name, last_name, salary
    FROM employees
) AS e
UNPIVOT (
    value FOR attribute IN (first_name, last_name, salary)
) AS unpvt;
```

46. Recursive CTE Generate a list of all employees in a hierarchy with their managers.

```sql
WITH RECURSIVE EmployeeCTE 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 EmployeeCTE ec
    ON e.manager_id = ec.employee_id
)
SELECT * FROM EmployeeCTE;
```

47. Window Functions Rank employees based on their salary within each department.

```sql
SELECT employee_id, first_name, department_id, salary,
       RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS salary_rank
FROM employees;
```

48. Common Table Expressions (CTE) Use CTE to simplify complex queries for calculating the

average salary in each department.

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

49. Rank Rank employees by salary within their department.

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

50. Row Number Assign a unique row number to each employee within their department ordered by hire date.

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

These SQL queries should cover the various tasks and scenarios specified, using the table structures created earlier.

Sure! Here are the SQL queries for tasks 51-100 based on the earlier list of problems.

0.0.17 Data Modification

51. Insert Insert a new employee into the `employees` table.

```
INSERT INTO employees (first_name, last_name, email, phone_number, hire_date, job_title, salary,
VALUES ('John', 'Doe', 'john.doe@example.com', '555-9999', '2023-05-24', 'Developer', 70000, 3,
```

52. Update Update the salary of all employees in the 'HR' department by 10%.

```
UPDATE employees
SET salary = salary * 1.10
WHERE department_id = (SELECT department_id FROM departments WHERE department_name = 'HR');
```

53. Delete Delete all employees who have not been assigned a department.

```
DELETE FROM employees WHERE department_id IS NULL;
```

54. Upsert (Insert or Update) Insert a new employee or update the existing employee if they already exist.

```
INSERT INTO employees (first_name, last_name, email, phone_number, hire_date, job_title, salary,
VALUES ('Jane', 'Smith', 'jane.smith@example.com', '555-8888', '2023-05-24', 'Developer', 80000,
ON DUPLICATE KEY UPDATE
first_name = VALUES(first_name), last_name = VALUES(last_name), phone_number = VALUES(phone_numb
```

0.0.18 Set Operations

55. UNION Combine the results of two queries to list all employees and managers.

```
SELECT first_name, last_name FROM employees
UNION
SELECT first_name, last_name FROM employees WHERE manager_id IS NOT NULL;
```

56. UNION ALL Combine the results of two queries including duplicates.

```
SELECT first_name, last_name FROM employees
UNION ALL
SELECT first_name, last_name FROM employees WHERE manager_id IS NOT NULL;
```

57. INTERSECT Find common employees who are both in the `employees` and `managers` tables.

```
SELECT first_name, last_name FROM employees
INTERSECT
SELECT first_name, last_name FROM employees WHERE manager_id IS NOT NULL;
```

58. EXCEPT Find employees who are in the `employees` table but not in the `managers` table.

```
SELECT first_name, last_name FROM employees
EXCEPT
SELECT first_name, last_name FROM employees WHERE manager_id IS NOT NULL;
```

0.0.19 Indexes and Performance

59. Create Index Create an index on the `name` column of the `employees` table.

```
CREATE INDEX idx_name ON employees(first_name, last_name);
```

60. Drop Index Drop the index on the `name` column of the `employees` table.

```
DROP INDEX idx_name ON employees;
```

61. Query Optimization Optimize a query to select employees with a salary greater than 60,000.

```
EXPLAIN SELECT * FROM employees WHERE salary > 60000;
```

0.0.20 Data Types and Constraints

62. Check Constraint Add a check constraint to ensure that employee salaries are always greater than 30,000.

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

63. Default Value Add a default value for the `hire_date` column to be the current date.

```
ALTER TABLE employees MODIFY hire_date DATE DEFAULT CURRENT_DATE;
```

64. Not Null Constraint Ensure that the `name` column in the `employees` table cannot be null.

```sql
ALTER TABLE employees MODIFY first_name VARCHAR(255) NOT NULL;
ALTER TABLE employees MODIFY last_name VARCHAR(255) NOT NULL;
```

65. Unique Constraint Add a unique constraint on the `email` column in the `employees` table.

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

0.0.21 Complex Queries and Analysis

66. Top-N Analysis Select the top 3 highest paid employees in each department.

```sql
SELECT employee_id, first_name, last_name, department_id, salary
FROM (
    SELECT employee_id, first_name, last_name, department_id, salary,
           DENSE_RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS rank
    FROM employees
) ranked_employees
WHERE rank <= 3;
```

67. Duplicate Records Find duplicate records in the `employees` table based on the `email` column.

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

68. Nth Highest Salary Find the 5th highest salary in the `employees` table.

```sql
SELECT DISTINCT salary
FROM employees
ORDER BY salary DESC
LIMIT 1 OFFSET 4;
```

69. Gaps and Islands Identify continuous periods of employment in the `employees` table.

```sql
SELECT employee_id, first_name, last_name, start_date, end_date,
       DATEDIFF(end_date, start_date) AS duration
FROM employees
WHERE end_date IS NOT NULL;
```

70. Moving Average Calculate the moving average of salaries over a 3-month period.

```sql
SELECT employee_id, first_name, salary,
       AVG(salary) OVER (ORDER BY hire_date ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS moving_
FROM employees;
```

0.0.22 Transaction and Locking

71. Begin Transaction Start a transaction to update employee salaries and commit the changes.

```sql
START TRANSACTION;
UPDATE employees SET salary = salary * 1.05;
COMMIT;
```

72. Rollback Transaction Start a transaction to update employee salaries and rollback the changes if an error occurs.

```sql
START TRANSACTION;
UPDATE employees SET salary = salary * 1.05;
ROLLBACK;
```

73. Deadlock Analysis Analyze and resolve a deadlock situation in the database.

```sql
-- This typically involves reviewing logs and queries, but here is an example of resolving a dea
BEGIN;
-- Transaction 1
UPDATE employees SET salary = salary * 1.05 WHERE employee_id = 1;
-- Transaction 2
UPDATE employees SET salary = salary * 1.05 WHERE employee_id = 2;
COMMIT;

-- If deadlock occurs, the application should catch the error and retry the transaction
```

74. Isolation Levels Set the transaction isolation level to `READ COMMITTED`.

```sql
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

0.0.23 JSON and XML

75. JSON Functions Extract data from a JSON column in the `employees` table.

```sql
SELECT JSON_EXTRACT(json_data, '$.key') AS extracted_value FROM employees;
```

76. XML Functions Extract data from an XML column in the `employees` table.

```sql
SELECT EXTRACTVALUE(xml_data, '/data') AS extracted_value FROM employees;
```

0.0.24 Miscellaneous

77. Dynamic SQL Write a dynamic SQL query to select columns based on user input.

```sql
SET @columns = 'first_name, last_name, salary';
SET @sql = CONCAT('SELECT ', @columns, ' FROM employees');
PREPARE stmt FROM @sql;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
```

78. Stored Procedures Create a stored procedure to insert a new employee.

```sql
DELIMITER //
CREATE PROCEDURE InsertEmployee(
    IN first_name VARCHAR(255),
```

```
    IN last_name VARCHAR(255),
    IN email VARCHAR(255),
    IN phone_number VARCHAR(20),
    IN hire_date DATE,
    IN job_title VARCHAR(255),
    IN salary DECIMAL(10, 2),
    IN department_id INT,
    IN manager_id INT,
    IN birth_date DATE,
    IN address VARCHAR(255),
    IN city VARCHAR(255),
    IN state VARCHAR(255),
    IN country VARCHAR(255),
    IN postal_code VARCHAR(20),
    IN start_date DATE,
    IN end_date DATE,
    IN json_data JSON,
    IN xml_data TEXT)
BEGIN
    INSERT INTO employees (first_name, last_name, email, phone_number, hire_date, job_title, sal

_id, manager_id, birth_date, address, city, state, country, postal_code, start_date, end_date, j
    VALUES (first_name, last_name, email, phone_number, hire_date, job_title, salary, department
END //
DELIMITER ;
```

79. Triggers Create a trigger to log changes to the `employees` table.

```
DELIMITER //
CREATE TRIGGER employee_changes
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO employee_changes_log (employee_id, change_time, old_salary, new_salary)
    VALUES (OLD.employee_id, NOW(), OLD.salary, NEW.salary);
END //
DELIMITER ;
```

80. Views Create a view to simplify access to the employee details.

```
CREATE VIEW employee_details AS
SELECT e.employee_id, e.first_name, e.last_name, e.email, e.phone_number, e.hire_date, e.job_tit
FROM employees e
LEFT JOIN departments d ON e.department_id = d.department_id
LEFT JOIN employees m ON e.manager_id = m.employee_id;
```

81. User Defined Functions Create a user-defined function to calculate the annual salary of an employee.

```
DELIMITER //
CREATE FUNCTION CalculateAnnualSalary(monthly_salary DECIMAL(10, 2))
```

```
RETURNS DECIMAL(10, 2)
DETERMINISTIC
BEGIN
    RETURN monthly_salary * 12;
END //
DELIMITER ;
```

82. Partitioning Partition the `employees` table by department.

```
ALTER TABLE employees
PARTITION BY HASH(department_id)
PARTITIONS 5;
```

83. Foreign Key Constraint Add a foreign key constraint between `employees` and `departments` tables.

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

84. Cascade Delete Implement a cascade delete between `employees` and `departments`.

```
ALTER TABLE employees
ADD CONSTRAINT fk_department
FOREIGN KEY (department_id) REFERENCES departments(department_id)
ON DELETE CASCADE;
```

85. Self-Referencing Foreign Key Add a self-referencing foreign key to track employee-manager relationships.

```
ALTER TABLE employees
ADD CONSTRAINT fk_manager
FOREIGN KEY (manager_id) REFERENCES employees(employee_id);
```

0.0.25 Analytical Queries

86. Percentile Calculate the 90th percentile salary in the `employees` table.

```
SELECT salary
FROM employees
ORDER BY salary
LIMIT 1 OFFSET (SELECT ROUND(0.9 * COUNT(*)) FROM employees) - 1;
```

87. Cumulative Sum Calculate the cumulative sum of salaries ordered by `hire_date`.

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

88. Lag Function Use the `LAG` function to compare each employee's salary with the previous one.

```
SELECT employee_id, first_name, last_name, salary,
       LAG(salary, 1) OVER (ORDER BY hire_date) AS previous_salary
```

```
FROM employees;
```

89. Lead Function Use the `LEAD` function to compare each employee's salary with the next one.

```
SELECT employee_id, first_name, last_name, salary,
       LEAD(salary, 1) OVER (ORDER BY hire_date) AS next_salary
FROM employees;
```

0.0.26 Data Cleanup and Transformation

90. Remove Duplicates Remove duplicate records from the `employees` table.

```
DELETE e1
FROM employees e1
INNER JOIN employees e2
WHERE e1.employee_id < e2.employee_id AND e1.email = e2.email;
```

91. Normalize Data Normalize the `employees` table to the 3rd normal form.

```
-- Assuming the `departments` and `managers` tables already exist
-- Normalize address into a separate table
CREATE TABLE addresses (
    address_id INT PRIMARY KEY AUTO_INCREMENT,
    address VARCHAR(255),
    city VARCHAR(255),
    state VARCHAR(255),
    country VARCHAR(255),
    postal_code VARCHAR(20)
);

-- Add a foreign key reference to the addresses table in employees
ALTER TABLE employees
ADD address_id INT,
ADD FOREIGN KEY (address_id) REFERENCES addresses(address_id);

-- Normalize phone numbers into a separate table
CREATE TABLE phone_numbers (
    phone_id INT PRIMARY KEY AUTO_INCREMENT,
    employee_id INT,
    phone_number VARCHAR(20),
    FOREIGN KEY (employee_id) REFERENCES employees(employee_id)
);
```

92. Denormalize Data Denormalize the `employees` table for faster querying.

```
CREATE TABLE denormalized_employees AS
SELECT e.*, d.department_name, a.address, a.city, a.state, a.country, a.postal_code, p.phone_num
FROM employees e
LEFT JOIN departments d ON e.department_id = d.department_id
```

```
LEFT JOIN addresses a ON e.address_id = a.address_id
LEFT JOIN phone_numbers p ON e.employee_id = p.employee_id;
```

93. Data Masking Mask sensitive information in the `employees` table.

```
SELECT employee_id, first_name, last_name, email,
       CONCAT(SUBSTRING(phone_number, 1, 2), '*****', SUBSTRING(phone_number, -2)) AS masked_pho
       CONCAT(LEFT(email, 2), '*****', RIGHT(email, LOCATE('@', email) - 3), '@', SUBSTRING(emai
FROM employees;
```

0.0.27 Security and Permissions

94. Grant Permissions Grant read permissions on the `employees` table to a specific user.

```
GRANT SELECT ON employees TO 'username'@'host';
```

95. Revoke Permissions Revoke all permissions on the `employees` table from a specific user.

```
REVOKE ALL PRIVILEGES ON employees FROM 'username'@'host';
```

96. Role-Based Access Control Implement role-based access control for the `employees` table.

```
CREATE ROLE read_only;
GRANT SELECT ON employees TO read_only;
GRANT read_only TO 'username'@'host';
```

0.0.28 Data Import and Export

97. Import Data Import data from a CSV file into the `employees` table.

```
LOAD DATA INFILE '/path/to/employees.csv'
INTO TABLE employees
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```

98. Export Data Export data from the `employees` table to a CSV file.

```
SELECT * INTO OUTFILE '/path/to/employees.csv'
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
FROM employees;
```

0.0.29 Advanced Analytics

99. Time Series Analysis Perform a time series analysis on employee hire dates.

```sql
SELECT hire_date, COUNT(*) AS hires
FROM employees
GROUP BY hire_date
ORDER BY hire_date;
```

100. Geospatial Analysis Perform a geospatial analysis to find the distance between employee locations.

```sql
-- Assuming the employees table has latitude and longitude columns
SELECT e1.employee_id AS emp1, e2.employee_id AS emp2,
       ST_Distance_Sphere(
           POINT(e1.longitude, e1.latitude),
           POINT(e2.longitude, e2.latitude)
       ) AS distance
FROM employees e1
CROSS JOIN employees e2
WHERE e1.employee_id != e2.employee_id;
```

These SQL queries cover a wide range of tasks and scenarios, providing you with a comprehensive toolkit for working with the `employees` and related tables.

[ ]: