SQL

SQL Guide: From Basics to Advanced Concepts

Basics

1. Select : Use to select the columns one want to show
    - is use to select all the columns.

```
-- This query selects all columns from the Database.Table_nam
e
Select *
From Database.Table_name;
```

1. Where : Used to pinpoint the data one needs

    Conditions include (>, <, ≥, ≤) and one can also use multiple conditions using And, Or, Not

```
-- This query selects Col1 and Col2 from Db.TN where a specif
ied condition is met
Select Col1,Col2
From Db.TN
Where Condition;
```

1. Group by : Used to group all the data that is in a particular column mentioned.

    One can use this to use aggregate functions, Also one can use multiple column in a Group by.

2. Order by : Used to order the data presented in a particular manner Eg: DESC, ASC (default).

3. Having : Used when one uses group by and wants to order things for aggregate data

```
-- This query calculates the average age for each gender, onl
y including groups where the average age is greater than 40
Select gender,avg(age)
From demographics
Group by gender
Having avg(age) > 40;
```

```
-- This query is similar to the previous one, but it also fil
ters for genders containing 'male' before grouping
Select gender,avg(age)
From demographics
Where gender Like '%male%'
Group by gender
Having avg(age) > 40;
```

1. Limit : This is a very handy and a powerful easy way to order things one want, limit basically means how many rows does one want to show.

   ```
   -- This query selects all columns from demographics, order
   s by age in descending order, and returns the third row (s
   kips first 2, shows 1)
   Select *
   From demograhics
   Order by age Desc
   Limit 2,1;
   ```

   Here in this snippet Limit 2,1 means we skip the first 2 and show One after those two.

2. Aliasing : Used to abbreviate something using 'as' as a keyword for it.

Intermediate

1. Joins : This is a very powerful tool to add up two tables into one.

   There are various ways to join a table such as left, right, full.

```
-- This query demonstrates an inner join between employee_
demographics and employee_salary tables on the employee_id
column
SELECT *
FROM employee_demographics dem
INNER JOIN employee_salary sal
    ON dem.employee_id = sal.employee_id;

-- This query shows a left join between employee_salary an
d employee_demographics tables, including all rows from th
e left table (salary) even if there's no match in the righ
t table (demographics)
SELECT *
FROM employee_salary sal
LEFT JOIN employee_demographics dem
    ON dem.employee_id = sal.employee_id;

-- This query demonstrates a self-join on the employee_sal
ary table to create a "secret santa" pairing
SELECT emp1.employee_id as emp_santa, emp1.first_name as s
anta_first_name, emp1.last_name as santa_last_name, emp2.e
mployee_id, emp2.first_name, emp2.last_name
FROM employee_salary emp1
JOIN employee_salary emp2
    ON emp1.employee_id + 1  = emp2.employee_id;

-- This query shows a multi-table join between employee_de
mographics, employee_salary, and parks_departments
SELECT *
FROM employee_demographics dem
INNER JOIN employee_salary sal
    ON dem.employee_id = sal.employee_id
```

```
LEFT JOIN parks_departments dept
    ON dept.department_id = sal.dept_id;
```

2. Unions : Used to join different rows in a table or from different tables.

```
-- This query demonstrates a UNION between two SELECT stat
ements, combining first and last names from both employee_
demographics and employee_salary tables
SELECT first_name, last_name
FROM employee_demographics
UNION
SELECT first_name, last_name
FROM employee_salary;

-- This query uses UNION ALL to show all values, including
duplicates, when combining data from employee_demographics
and employee_salary
SELECT first_name, last_name
FROM employee_demographics
UNION ALL
SELECT first_name, last_name
FROM employee_salary;

-- This complex query uses multiple UNIONs to categorize e
mployees based on age and salary
SELECT first_name, last_name, 'Old Lady' as Label
FROM employee_demographics
WHERE age > 40 AND gender = 'Female'
UNION
SELECT first_name, last_name, 'Old Man'
FROM employee_demographics
WHERE age > 40 AND gender = 'Male'
UNION
SELECT first_name, last_name, 'Highly Paid Employee'
FROM employee_salary
```

```
WHERE salary >= 70000
ORDER BY first_name;
```

3. String functions

   String functions in SQL are used to manipulate and analyze text data. These functions include operations like extracting substrings, changing case, finding string length, and replacing characters, allowing for powerful text processing within database queries.

```
-- This query demonstrates the LENGTH function to get the
length of each first name
SELECT first_name, LENGTH(first_name)
FROM employee_demographics;

-- This query shows the use of UPPER function to convert f
irst names to uppercase
SELECT first_name, UPPER(first_name)
FROM employee_demographics;

-- This query demonstrates the LEFT function to extract th
e first 4 characters of each first name
SELECT first_name, LEFT(first_name,4)
FROM employee_demographics;

-- This query uses the SUBSTRING function to extract the b
irth year from the birth_date column
SELECT birth_date, SUBSTRING(birth_date,1,4) as birth_year
FROM employee_demographics;

-- This query demonstrates the REPLACE function to replace
'a' with 'z' in first names
SELECT REPLACE(first_name,'a','z')
FROM employee_demographics;

-- This query uses the LOCATE function to find the positio
```

```
n of 'a' in each first name
SELECT first_name, LOCATE('a',first_name)
FROM employee_demographics;

-- This query demonstrates the CONCAT function to combine
first and last names into a full name
SELECT CONCAT(first_name, ' ', last_name) AS full_name
FROM employee_demographics;
```

4. Window Functions :

   Window functions are incredibly powerful and share some similarities with GROUP BY operations. However, unlike GROUP BY, which consolidates results into a single row per group, window functions maintain individual rows in the output. This allows us to examine partitions or groups while preserving the unique characteristics of each row.

   Window functions provide the ability to perform calculations across a set of rows that are related to the current row, offering a more flexible and detailed analysis compared to traditional aggregate functions.

```
-- This query demonstrates a window function to calculate
the average salary across all employees
SELECT dem.employee_id, dem.first_name, gender, salary,
AVG(salary) OVER()
FROM employee_demographics dem
JOIN employee_salary sal
    ON dem.employee_id = sal.employee_id;

-- This query uses a window function with PARTITION BY to
calculate the average salary by gender
SELECT dem.employee_id, dem.first_name, gender, salary,
AVG(salary) OVER(PARTITION BY gender)
FROM employee_demographics dem
JOIN employee_salary sal
    ON dem.employee_id = sal.employee_id;
```

```
-- This query demonstrates ROW_NUMBER, RANK, and DENSE_RAN
K window functions to rank employees by salary within each
gender
SELECT dem.employee_id, dem.first_name, gender, salary,
ROW_NUMBER() OVER(PARTITION BY gender ORDER BY salary des
c) row_num,
Rank() OVER(PARTITION BY gender ORDER BY salary desc) rank
_1,
dense_rank() OVER(PARTITION BY gender ORDER BY salary des
c) dense_rank_2
FROM employee_demographics dem
JOIN employee_salary sal
    ON dem.employee_id = sal.employee_id;
```

5. Case Statements :

   Case statements in SQL are used to add conditional logic within a query. They allow you to specify different results based on conditions, similar to if-then-else statements in programming languages. Case statements are particularly useful for categorizing data or performing conditional aggregations.

   The basic syntax of a CASE statement is as follows:

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    ELSE result
END
```

   Here's an example of using a CASE statement to categorize employees based on their salary:

```
SELECT
    employee_id,
```

```
    first_name,
    last_name,
    salary,
    CASE
        WHEN salary < 30000 THEN 'Low'
        WHEN salary BETWEEN 30000 AND 50000 THEN 'Medium'
        WHEN salary > 50000 THEN 'High'
        ELSE 'Not Categorized'
    END AS salary_category
FROM
    employee_salary;
```

Case statements can also be used in aggregate functions or in WHERE clauses to filter data based on complex conditions. They provide a powerful way to transform and analyze data within SQL queries.

6. Subqueries : Subqueries, also known as nested queries or inner queries, are SQL queries embedded within another query. They allow you to use the results of one query as part of another query. Subqueries can be used in various parts of SQL statements, including SELECT, FROM, WHERE, and HAVING clauses. Here are some key points about subqueries:
    - Uses of Subqueries:
        - Filtering data in WHERE clauses
        - Providing values for IN clauses
        - Creating derived tables in FROM clauses
        - Performing calculations in SELECT statements

    Here's an example of a subquery used in a WHERE clause:

```
SELECT employee_id, first_name, last_name
FROM employee_demographics
WHERE employee_id IN (
    SELECT employee_id
```

```
    FROM employee_salary
    WHERE salary > 50000
);
```

This query selects employees from the employee_demographics table whose employee_id matches those in the employee_salary table with a salary greater than 50000.

Subqueries can significantly enhance the power and flexibility of your SQL queries, allowing for more complex data retrieval and manipulation operations.

Advanced

1. CTE : CTE stands for Common Table Expression in SQL. It is a temporary named result set that you can reference within a SELECT, INSERT, UPDATE, DELETE, or MERGE statement. CTEs are defined using a WITH clause and are particularly useful for:

   - Simplifying complex queries by breaking them down into more manageable parts
   - Creating recursive queries
   - Improving the readability and maintainability of SQL code
   - Referencing the same subquery multiple times in a single statement

   CTEs are similar to derived tables or subqueries, but they are more readable and can be referenced multiple times within the query. They exist only for the duration of the query and are not stored as objects in the database schema.

   The basic syntax for a CTE is:

```
WITH cte_name AS (
    SELECT column1, column2, ...
    FROM table
    WHERE condition
)
SELECT * FROM cte_name;
```

CTEs can significantly simplify complex queries, especially those involving multiple subqueries or self-joins, making the SQL code more organized and easier to understand.

```
-- This query demonstrates a basic CTE to calculate salary
statistics by gender
WITH CTE_Example AS
(
SELECT gender, SUM(salary), MIN(salary), MAX(salary), COUN
T(salary), AVG(salary)
FROM employee_demographics dem
JOIN employee_salary sal
    ON dem.employee_id = sal.employee_id
GROUP BY gender
)
SELECT *
FROM CTE_Example;

-- This query shows how to use multiple CTEs in a single q
uery
WITH CTE_Example AS
(
SELECT employee_id, gender, birth_date
FROM employee_demographics dem
WHERE birth_date > '1985-01-01'
),
CTE_Example2 AS
(
SELECT employee_id, salary
FROM parks_and_recreation.employee_salary
WHERE salary >= 50000
)
SELECT *
FROM CTE_Example cte1
```

```
LEFT JOIN CTE_Example2 cte2
    ON cte1.employee_id = cte2.employee_id;
```

2. Stored Procedure：A stored procedure in MySQL is a reusable unit of SQL code that can be saved and executed repeatedly. It's like a function in programming languages, allowing you to encapsulate a set of SQL statements into a named, parameterized routine. Here are some key points about stored procedures in MySQL:

   - Definition: Stored procedures are created using the CREATE PROCEDURE statement and can contain multiple SQL statements.
   - Parameters: They can accept input parameters and return output parameters, making them flexible for various use cases.
   - Reusability: Once created, stored procedures can be called from other SQL statements, applications, or other stored procedures.
   - Performance: They can improve performance by reducing network traffic, as multiple SQL statements are executed in a single call.
   - Security: Stored procedures can enhance database security by controlling access to underlying tables.
   - Maintenance: They centralize business logic, making it easier to maintain and update database operations.

   Here's a basic example of creating and using a stored procedure in MySQL:

```
-- Creating a stored procedure
DELIMITER //
CREATE PROCEDURE GetEmployeeCount()
BEGIN
    SELECT COUNT(*) FROM employees;
END //
DELIMITER ;

-- Calling the stored procedure
CALL GetEmployeeCount();
```

Stored procedures are powerful tools for database management and can significantly improve the efficiency and organization of your database operations.

More Example :

```
-- This code creates a stored procedure named 'large_salari
es2' that contains two SELECT statements
DELIMITER $$
CREATE PROCEDURE large_salaries2()
BEGIN
    SELECT *
    FROM employee_salary
    WHERE salary >= 60000;
    SELECT *
    FROM employee_salary
    WHERE salary >= 50000;
END $$
DELIMITER ;

-- This code creates a stored procedure with a parameter to
filter results based on employee_id
DELIMITER $$
CREATE PROCEDURE large_salaries3(employee_id_param INT)
BEGIN
    SELECT *
    FROM employee_salary
    WHERE salary >= 60000
    AND employee_id_param = employee_id;
END $$
DELIMITER ;
```

3. Events and Triggers : Events and Triggers in MySQL are powerful database objects that automate certain actions based on specific occurrences in the database. Here's an explanation of each:

Triggers are database objects that automatically execute in response to certain events on a particular table or view. Key points about triggers:

- They are associated with a specific table and are automatically fired when a defined action (INSERT, UPDATE, or DELETE) occurs on that table.
- Triggers can execute before or after the triggering event.
- They are useful for maintaining data integrity, auditing changes, or automatically updating related tables.

Events in MySQL are tasks that run according to a schedule. They are sometimes referred to as "temporal triggers" or "scheduled tasks". Key points about events:

- They are used to schedule and automate database maintenance tasks or other routine operations.
- Events can be set to run once at a specific time or repeatedly at regular intervals.
- They are particularly useful for tasks like data archiving, generating reports, or cleaning up temporary data.

Both triggers and events contribute to automating database operations, improving efficiency and consistency in database management.

```
-- This code creates a trigger that automatically inserts d
ata into employee_demographics when a new row is inserted i
nto employee_salary
DELIMITER $$
CREATE TRIGGER employee_insert2
    AFTER INSERT ON employee_salary
    FOR EACH ROW
BEGIN
    INSERT INTO employee_demographics (employee_id, first_n
ame, last_name)
    VALUES (NEW.employee_id, NEW.first_name, NEW.last_nam
```

```
e);
END $$
DELIMITER ;

-- This code creates an event that runs every 30 seconds to
delete employees aged 60 or older from the employee_demogra
phics table
DELIMITER $$
CREATE EVENT delete_retirees
ON SCHEDULE EVERY 30 SECOND
DO BEGIN
    DELETE
    FROM parks_and_recreation.employee_demographics
    WHERE age >= 60;
END $$
DELIMITER ;
```

Summary

SQL Basics

- SELECT is used to choose columns for display, with * selecting all columns
- WHERE clause filters data based on specified conditions
- GROUP BY aggregates data by specified columns
- HAVING filters grouped data, often used with aggregate functions

SQL Intermediate Concepts

- JOINS combine tables based on related columns, with various types like inner, left, and right joins
- UNION combines rows from different tables or queries
- String functions manipulate text data, including LENGTH, UPPER, LOWER, and SUBSTRING

- CASE statements add conditional logic to SELECT statements

SQL Advanced Topics

- Common Table Expressions (CTEs) create named subqueries for complex operations
- Stored Procedures are reusable blocks of SQL code that can be called and executed
- Triggers automatically execute when specified events occur in the database