TOPICS COVERED:

1) SQL VIEWS
2) CASE STATEMENTS
3) COMMON TABLE Expressions (CTE's)
4) SUB QUERIES
5) STORED PROCEDURES
6) DELIMITER
7) DECLARE
8) TRIGGERS
9) Type CASTING
10) WINDOWS FUNCTIONS
11) SQL Hosting
12) SQL INJECTION

SQL Views

In SQL, a View is a virtual table based on the result-set of a SQL statement.

* A View contains rows and columns, just like a real table. The fields in a View are fields from one or more real tables in a database.
* We can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.
* Views allow to encapsulate or hide complexities, or allow limited read access to part of the data.
* A View always shows up-to-date data! The database engine recreates the data, using the view's SQL statement, every time a user queries a view

CREATE VIEW Syntax:

```
CREATE VIEW View-name AS
SELECT column1, column2,....
FROM <table_name>
```

```
WHERE condition;
```

Examples:

* A view to show only products with category = 1

```
CREATE VIEW cat_1_product AS
SELECT *
FROM PRODUCTS
WHERE category = 1;
```

* If we want to query this view we can say

```
SELECT *
FROM cat_1_product;
```

* A view to limit read access only to certain columns

```
CREATE VIEW basic_cat_prod AS
SELECT name, category, price
FROM products;
```

from this view we can query

```
SELECT *
FROM basic_cat_prod
WHERE category = 1;
```

UPDATING A VIEW:

CREATE OR REPLACE VIEW Syntax:

```
CREATE OR REPLACE VIEW View_name AS
SELECT Column1, Column2,....
FROM <table_name>
WHERE condition;
```

Example: If we want to add quanlity to basic_cat_prod view:

```
CREATE OR REPLACE VIEW basic_cat_prod AS
SELECT name, category, price, quanlity
FROM products
WHERE quanlity > 10;
```

DROP A VIEW:

We can delete a view with DROP VIEW command.

DROP VIEW Syntax:

```
DROP VIEW View_name;
```

CASE statements

CASE statements in MySQL allow us to perform conditional logic within our query.

* They are useful when we need to perform different actions or calculations based on certain conditions.

1) Simple CASE Statement:

The simple CASE statement compares an expression to a set of conditions and returns a result based on the first condition that evaluates to true.

Syntax:

```
CASE expression
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    ELSE else_result
END
```

Example: Consider a table called 'employees' with columns 'employee_id', 'first_name', and

salary. If we want to categorize Employees based on salary range

```
SELECT employee_id, first_name, salary
    CASE
        WHEN Salary < 5000 THEN 'Low'
        WHEN Salary >= 5000 AND
            Salary < 10000 THEN 'Medium'
        ELSE 'High'
    END AS salary_category
FROM Employees;
```

* In simple case statement condition is checked on only one field or column ex:- Salary in the above example.

2) Searched CASE statement:

In searched CASE each condition could be combination of multiple conditions using logical operators or relationship operators and such.

Syntax:

```
CASE expression
```

```
        WHEN condition1 THEN result1
        WHEN condition2 THEN result2
          .
          .
          .
        ELSE else_result
    END
```

* Even though the syntax looks same as above the condition here evaluates multiple expressions and can be on different fields. See below example.

Example:

```
SELECT employee_name, salary,
    CASE
        WHEN salary > 5000 AND
            department = 'IT' THEN 'High IT'
        WHEN salary > 5000 AND
            department = 'HR' THEN 'High HR'
        WHEN salary > 3000 AND
            department = 'IT' THEN 'Medium IT'
        WHEN salary > 3000 AND
            department = 'HR' THEN 'medium HR'
        ELSE 'low'
    END AS Salary_Category
```

```
FROM employees;
```

* Here both salary and department are evaluated with logical operator 'AND' between them for each condition inside WHEN.

Common Table Expressions (CTEs)

CTEs in MySQL allow us to define temporary named result sets that can be used within a query.

* CTEs provide a way to breakdown complex queries into smaller, more manageable parts.

Syntax:

```
WITH cte_name (column1, column2,...) AS (
    --- Query that defines the CTE
    SELECT ....
    FROM ....
    WHERE...
)
--- Main Query that uses CTE
SELECT ....
FROM cte_name,...
.....
```

Explanation:

1) Define the CTE:

* Start with the 'WITH' keyword followed by name we want to assign to the CTE

(Cte_name).

* Optionally, specify the column names (column1, column2,...) for CTE.

* Use keyword AS followed by paranthesis '( )' to enclose the query that defines the CTE.

* This query can include filtering, joining, aggregation, and any other SQL operations.

2) Use the CTE:

* After defining the CTE, we can refer to it as a table in the subsequent query.

* Here we can do any operations on the result set of CTE that we could do on a table using SQL queries.

* After the paranthesis enclosed CTE query there is no semi-colon. The result of the CTE can be used in the main query until it ends with a semi-colon.

Example: Consider a database with two tables: 'employees' and 'departments'. The 'employees' table has columns 'employee_id', 'first_name',

'last_name' and 'department_id', while the 'departments' table has columns 'department_id' and 'department_name'.

```
WITH employee_department AS (
    SELECT e.employee_id, e.first_name, e.last_name,
        d.department_name
    FROM employee e
    JOIN departments d
        ON e.department_id = d.department_id
)
SELECT *
FROM employee_department ;
```

default JOIN is INNER JOIN

* In this example, a CTE named 'employee_department' is defined, which joins the 'employee' and 'departments' tables based on 'department_Id' column. The CTE selects necessary columns from both tables. The main table then selects all the columns from the CTE, effectively retreiving the employee information along with their department names.

* CTE's are especially useful when dealing with complex queries involving multiple joins, aggregations, or recursive queries. They help improve query readability, maintainability, and performance by breaking down logic into smaller, logical units.

** CTE's are supported in MYSQL 8.0 and above.

Subqueries in MySQL:

Subqueries in MySQL allow us to nest one query (inner query) inside another query (outer query). The result of inner query is used by the outer query to perform further operations.

Syntax:

```
SELECT column1, column2, ....
FROM table1
WHERE column1 IN (SELECT column1
                  FROM table2
                  WHERE ....)
```

Explanation:

1) Define the subquery:

* The subquery is enclosed with paranthesis `( )`.

  It can be used in various parts of the outer query, such as the 'SELECT', 'FROM', 'WHERE', or 'HAVING' clauses.

2) Use the subquery:

* The result of the subquery is treated as temporary table or dataset.
* It can be used in conjunction with operators

like 'IN', 'NOT IN', 'EXISTS', 'NOT EXISTS', or comparision operators ('=', '<', '>', etc) to filter or join data in outer query.

Example: Consider a database with two tables 'customers' and 'orders'. The 'customers' table has columns 'customer_id', 'customer_name' and 'country', while the 'orders' table has columns 'order_id', 'customer_id', and 'order_date'. We want to retreive a list of customers who have placed an order in the year 2022.

```
SELECT Customer_name
FROM Customers
WHERE Customer_id IN (SELECT Customer_id
        FROM orders
        WHERE YEAR(order_date) = 2022);
```

* In this example, a subquery is used to filter the 'orders' table and retreive the 'customer_id' values for orders placed in 2022. The outer query then uses the 'IN' operator to select the 'customer_name' from the

'Customers' table for those specific customer_ids.

* Subqueries can be used in various scenarios, such as
  * Filtering based on a condition: Using a subquery in the 'WHERE' clause to filter data based on a specific condition.
  * Joining tables: Using a subquery in the 'FROM' clause to join tables based on a common column.
  * Calculating aggregate values: Using a subquery in the 'SELECT' or 'HAVING' clause to calculate aggregate values like counts, sums, averages, etc.

* Subqueries impact performance, so it's important to optimize and ensure that indexes are appropriately applied to improve execution time.

MySQL stored procedures:

Stored procedures are a set of SQL statements that are stored in the database and can be executed repeatedly.

* They provide a way to encapsulate and reuse SQL logic.
* They can accept input parameters and return output parameters.

Creating a stored procedure:

* Use the 'CREATE PROCEDURE' statement to create a stored procedure.
* Set the delimiter to something other than a semi-colon to avoid conflicts.
* Define the procedure name, input/output parameters, and the procedure body.
* Use the 'BEGIN' and 'END' keywords to enclose the procedure statements.
* Finally, set the delimiter back to semicolon.

Syntax:

```
DELIMITER $$
```

→ this can be anything other than ; that is used inside

```
CREATE PROCEDURE procedure_name([parameter_list])
    [characteristics....]
    BEGIN
        --- procedure body
    END $$
DELIMITER ;
```

Example:

```
DELIMITER //
CREATE PROCEDURE get_customer_details (
        IN customer_id INT)
    BEGIN
        SELECT *
        FROM Customers
        WHERE  id = customer_id
    END //
DELIMITER ;
```

Calling a stored procedure:

* Use the 'CALL' statement to execute a stored procedure.
* Provide the necessary arguments for input

Parameters.

Example:

```
CALL get_customer_details(5);
```

this is id parameter given a value 5.

Stored procedure parameters:

* Stored procedures can have input, output, or input/output parameters.
* Input parameters are used to pass values into the procedure.
* Output parameters are used to return values from the procedure.
* Input/output Parameters can be used for both passing and returning values.

Example:

```
DELIMITER //
CREATE PROCEDURE Calculate_total (IN price INT,
    IN quantity INT, OUT total INT)
    BEGIN
        SET total = price * quantity;
    END //
DELIMITER ;
```

Conditional statements and Loops:

Stored procedures can include conditional statements like IF, CASE, etc.

They can also include loops such as WHILE or REPEAT

Example:

```
DELIMITER //
CREATE PROCEDURE check_grade (
                IN score INT)
  BEGIN
    DECLARE grade CHAR(1);
    IF score >= 90 THEN
       SET grade = 'A';
    ELSE IF score >= 80 THEN
       SET grade = 'B';
    ELSE IF score >= 70 THEN
       SET grade = 'C';
    ELSE
       SET grade = 'D';
    END IF;

    SELECT grade;
```

```
END //
DELIMITER ;
```

MYSQL DELIMITER:

In MySQL, the 'DELIMITER' statement is used to change the default delimiter used in SQL statements. It is particularly useful when defining stored procedures, triggers, or functions that contain multiple SQL statements.

By default MYSQL uses semicolon (;) as the statement delimiter, but when with complex routines, it becomes necessary to change the delimiter to avoid conflicts.

Syntax:

```
DELIMITER new_delimiter;
```

* 'new_delimiter' is the new-delimiter to be set.

Example: SQL stored procedure with multiple SQL statements with out DELIMITER.

```
CREATE PROCEDURE example_procedure()
BEGIN
    SELECT * from table1;
    Update table2
    SET column1 = Value1;
    DELETE FROM table3
```

```
    WHERE cond1;
    . . .
END;
```

* Above example gives syntax error. In above example, we have a stored procedure 'example-procedure' that contains multiple SQL statements. By default, MySQL uses the semi-colon (';') as the delimiter to seperate the statements. However, when executing this code directly, MySQL interprets each semicolon as the end of the entire procedure, resulting in a syntax error.

* To avoid this issue we need to change the delimiter using the 'DELIMITER' statement:

```
DELIMITER //
CREATE PROCEDURE example_procedure()
  BEGIN
    SELECT * from table1;
    Update table2
    SET column1 = value1;
    DELETE FROM table3
    WHERE cond1;
```

. . .

END //

DELIMITER ;

* In the above example, we set the new delimiter to '//' (can be anything like $$, @@...) using the 'DELIMITER //' statement before defining the stored procedure. This allows us to use the semicolon (';') within the procedure without conflicting with the statement delimiter.
* Finally we end the procedure definition with 'END //' (this symbol has to be same as we used when defining the delimiter). Then we reset the delimiter back to semicolon (;) using 'DELIMITER ;'.

Notes:

1) The 'DELIMITER' statement is not an SQL statement itself. It is a command used to change the delimiter used for parsing SQL statements.
2) Changing the delimiter is necessary when

working with complex routines that contain multiple statements.

3) The new delimiter can be any valid character or string that is not part of the SQL statements with in the routine.

4) After changing the delimiter, the new delimiter is used to seperate the statements with in the routine.

5) once the routine definition is complete, it is essential to reset the delimiter back to the default semicolon (';') using 'DELIMITER ;'.

* The DELIMITER tool in MYSQL is a helpful tool for managing complex stored procedures, triggers or functions that involve multiple SQL statements. By changing the delimiter, we can ensure that the individual statements within the routine are correctly interpreted by MySQL.

* DECLARE Statement:

In MySQL, the 'DECLARE' statement is used with in the stored procedures to declare and define variables. It allows you to create variables that can be used to store and manipulate data during the execution of the stored procedure.

Syntax:

```
DECLARE variable_name <datatype> <DEFAULT
        default_value>;
```

* variable_name is the name of the variable to be declared.
* datatype is the datatype of the variable, such as INT, VARCHAR, DATE, etc.
* DEFAULT default_value (optional) specifies the default value for the variable if it is not explicity assigned.

Example:

```
DELIMITER $$
CREATE PROCEDURE Calculate_tax ( IN
    invoice_amount DECIMAL (10,2))
    BEGIN
      DECLARE tax_rate DECIMAL(5,2);
      DECLARE tax_amount DECIMAL(10,2);
      SET tax_rate = 0.15;
      SET tax_amount = invoice_amount *
                          tax_rate;
      SELECT tax_amount;
    END $$
DELIMILER ;
```

* In this example, we define a stored procedure named 'calculated_tax' that takes an input parameter 'invoice_amount'. Within the procedure, we declare two variables: 'tax_rate' of type DECIMAL(5,2) and 'tax_amount' of type DECIMAL(10,2).
* We then assign a value of 0.15 to the 'tax_rate' variable using the 'SET' statement. The 'tax_amount' variable is calculated

by multiplying the 'invoice_amount' with the 'tax_rate'.

* Finally we select and display 'tax_amount'.

Notes:

1) The 'DECLARE' statement is used to define variables within a stored procedure.

2) Variables declared using 'DECLARE' are local to the stored procedure and cannot be accessed outside of it.

3) Each variable must have a unique name within the scope of the stored procedure.

4) Variables can be assigned default values using 'DEFAULT' clause.

5) Variables can be used to store and manipulate data during the execution of the stored procedure, enabling calculations, comparisions and other operations.

6) MySQL supports various data types for variables, including numeric types, string types, data and time types and more.

* The 'DECLARE' statement is a fundamental aspect of working with variables within MySQL stored procedure. It allows us to create and use variables to hold and manipulate data, enhancing flexibility and functionality of our stored procedures.

Example with default_value:

```
DELIMITER $$
CREATE PROCEDURE calculate_discount(IN
    product_price DECIMAL(10,2))
  BEGIN
    DECLARE discount_rate DECIMAL(5,2)
                DEFAULT 0.1;
    DECLARE discount_amount DECIMAL(10,2)
            DEFAULT product_price * discount_rate;
    SELECT discount_amount;
  END $$
DELIMILER ;
```

Triggers in MySQL:

Triggers in MySQL are database objects that are associated with a table and automatically executed when a specific event occurs. They are useful for enforcing business rules, maintaining data integrity, performing auditing, or automating certain actions in response to data changes.

Notes:

1) Triggers are defined using SQL statements and are attached to tables.
2) They are executed in response to specific events such as INSERT, UPDATE, DELETE, or a combination of these events.
3) Triggers are defined at the database level and operate on a per-row basis, meaning they are executed for each affected row.

Types of Triggers:

1) BEFORE Triggers:

* These triggers are executed both before the specified event occurs.

* They are commonly used to modify the data being inserted, updated, or deleted, or perform validations.
* Useful for enforcing data integrity rules or performing calculations before the actual change happens.

Syntax: (BEFORE INSERT Trigger)

```
CREATE TRIGGER before_insert_trigger
BEFORE INSERT ON table_name
FOR EACH ROW
BEGIN
    --- logic steps
END;
```

Example:

```
CREATE TRIGGER before_insert_trigger
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
    SET NEW.created_at = NOW()
END;
```

* This trigger is executed before inserting a row in to the 'employee' table. It sets the 'created_at' column to the current timestamp.

2) AFTER Triggers:

* These triggers are executed after the specified event occurs.
* They are used for tasks such as logging, generating reports, updating related tables, or sending notifications.
* Useful for performing actions based on the changes made to the data.

Syntax: (AFTER UPDATE Trigger)

```
CREATE TRIGGER after_update_trigger
AFTER UPDATE ON table_name
FOR EACH ROW
BEGIN
    --- logic steps
END;
```

Example:

```
CREATE TRIGGER after_update_trigger
AFTER UPDATE ON orders
FOR EACH ROW
 BEGIN
    INSERT INTO order_logs (order_id, action, updated_at)
    VALUES (New.id, 'updated', NOW());
 END;
```

* This trigger is executed after updating a row in the 'orders' table. It logs the update action in to the 'order_logs' table.

3) INSTEAD OF Triggers:

* These triggers are executed instead of the default action associated with the event.
* They are primarily used with views to enable performing operations on views that involve multiple underlying tables.
* Useful for implementing complex view modifications or custom handling of data changes.

Syntax: (INSTEAD OF INSERT Trigger)

```
CREATE TRIGGER instead_of_insert_trigger
```

```
INSTEAD OF INSERT on view_name
FOR EACH ROW
BEGIN
    -- logic goes here
END;
```

Example:

```
CREATE TRIGGER instead_of_insert_trigger
INSTEAD OF INSERT on view_sales
FOR EACH ROW
BEGIN
   INSERT INTO sales (product_id, quantity)
   VALUES (NEW.product_id, NEW.quantity);
END;
```

* This trigger is executed instead of the default insert action on the 'view_sales' view. It redirects the insert operation to the 'sales' table.

4) **Compound Triggers:**

* Compound triggers combine BEFORE, AFTER, or INSTEAD OF triggers to define multiple

trigger actions for the same event.

* They allow you to perform different actions at different stages of the event execution.
* Useful for implementing complex business rules or performing multiple operations based on the event.

Syntax: (COMPOUND TRIGGER BEFORE AND AFTER INSERT)

```
CREATE TRIGGER compound_insert_trigger
BEFORE INSERT ON table_name
FOR EACH ROW
BEGIN
    -- logic goes here
END;
AFTER INSERT ON table_name
FOR EACH ROW
BEGIN
    -- logic goes here
END;
```

Example:

```
CREATE TRIGGER compound_insert_trigger
BEFORE INSERT ON Customers
FOR EACH ROW
 BEGIN
    SET NEW.created_at = NOW()
 END;
 AFTER INSERT ON
 FOR EACH ROW
 BEGIN
   INSERT INTO Customer_logs(Customer_id, action, updated_at)
   VALUES (New.id, '          ', NOW());
 END;
```

* This compound trigger consists of a BEFORE INSERT and an AFTER INSERT trigger for the 'customers' table. It sets the 'created_at' timestamp before insertion and logs the insertion action into the 'customer_logs' table after insertion.

When to use triggers:

* Use triggers to enforce data integrity constraints such as validating data before

insertion or update.

* Use triggers for auditing purposes, such as logging changes made to specific tables.
* Use triggers to automate certain actions or calculations based on data changes.
* Use triggers to maintain consistency across related tables or views.

Use triggers when we need to perform complex operations involving multiple tables or views.

Type Casting in MySQL:

Type casting in MySQL allows us to convert values from one data type to another. It is useful when we need to ensure data compatibility, perform calculations involving different datatypes, or format data in a specific way.

* MySQL provides various functions and techniques for type casting.

1) CAST() function:

The CAST() function is used to explicitly convert a value to a specified data type.

Syntax: CAST (value AS datatype)

Example:

```
SELECT CAST ('42' AS INT)
```

Converts the string '42' to an integer.

2) CONVERT() function:

The CONVERT() function is another way to convert a value to a specified data type. Its syntax is similar to CAST().

Syntax: CONVERT (value AS datatype)

Example:

```
SELECT CONVERT('3.14', DECIMAL(5,2))
```

* Converts the string '3.14' to a decimal with precision 5 and scale 2.

3) **Numeric Conversion functions:**

MySQL provides various functions for numeric conversion, such as ROUND(), CEIL(), FLOOR(), ABS(), etc. These functions allow us to manipulate and convert numeric values as needed.

Example:

```
SELECT ROUND(3.7);
```

* Converts decimal value 3.7 to the nearest integer 4.

4) **Date and Time Conversion Functions:**

MySQL offers functions like DATE_FORMAT(), DATE_ADD(), DATE_SUB(), etc., which can be used to convert or manipulate date and time values.

Example:

```
SELECT DATE_FORMAT('2022-12-31', '%Y/%M/%d');
```

* Converts the date '2022-12-31' to format '2022/12/31'

5) **Implicit Type Casting:**

MySQL also performs implicit type casting in some cases. For example, when we perform arithmetic operations involving different data-types. MySQL, automatically converts them to a common datatype based on a set of rules known as 'type coercion'.

Example:

```
SELECT 5+'10';
```

* Implicitly converts the string '10' to an integer and performs the addition operation.

Notes:

* It is important to be aware of the datatypes involved and the potential implications of type casting.
* Improper use of type casting can lead to data loss, unexpected results, or performance issues. make sure to understand the characteristics

and limitations of different datatypes in MySQL.

Windows function in MySQL:

Windows functions, also known as windowing or analytic functions, are a powerful feature in MySQL that allow us to perform calculations on a specific "window" or subset of rows within a result set.

* These functions operate on a group of rows and return a result for each row based on the values of other rows within the same window.

* windows functions are often used for tasks such as ranking, aggregation, and moving averages.

Syntax: (General Syntax)

```
function_name(expression) OVER (
    [PARTITION BY partition_expression]
    [ORDER BY order_expression [ASC|DESC]]
    [frame_specification]
)
```

Explanation:

* 'function_name' is the name of the windows

function we want to use, such as 'ROW_NUMBER', 'RANK', 'DENSE_RANK', 'LEAD', 'LAG' etc.

* 'expression' is the column or expression on which function will be applied.
* 'PARTITION BY': optional clause that defines the partitioning of the result set in to subset based on one or more columns. The function is applied seperately to each partition.
* 'ORDER BY': optional clause that specifies the ordering of the rows within each partition. The function will be calculated based on this order.
* 'frame_specification': optional clause that defines the window frame or range of rows within the partition to include in the calculation. It determines which rows are considered when performing the function.

Few functions:

1) ROW_NUMBER():

Returns the sequential number of a row within a partitioned result set, based on the specific order.

Syntax:

```
SELECT
    ROW_NUMBER() OVER (ORDER BY column_name)
        AS row_number, column_name
FROM <table_name>;
```

Example:

```
SELECT
    ROW_NUMBER() OVER (ORDER BY salary DESC)
        AS row_number, employee_name, salary
FROM employees;
```

* Assigns a unique sequential order to the result set based on the salary column in descending order.

2) **RANK():**

Assigns a unique rank to each row within a partitioned result set, based on specific order. Ties receive the same rank and the next rank is skipped.

Syntax:

```
SELECT
  RANK() OVER (ORDER BY column_name)
    AS rank, column_name
FROM <table_name>;
```

Example:

```
SELECT
  RANK() OVER (ORDER BY score DESC)
    AS rank, student_name, score
FROM students;
```

3) **DENSE_RANK():**

Assigns a unique rank to each row within a partitioned result set, based on the specified order. Ties receive the same rank and the next rank is not skipped.

Syntax:

```
SELECT
  DENSE_RANK() OVER (ORDER BY column_name)
    AS rank, column_name
FROM <table_name>;
```

Example:

```
SELECT
    DENSE_RANK() OVER (ORDER BY price ASC)
      AS rank, product_name, price
FROM products;
```

4) **SUM():**

Calculates the sum of a column within a window defined by the partition and order clauses.

Syntax:

```
SELECT
    column_name, SUM(column_name) OVER(
    PARTITION BY partition_column ORDER BY
    order_column) AS sum_value
FROM <table_name>;
```

Example:

```
SELECT Order_id, order_date, order_total,
    SUM(order_total) OVER (PARTITION BY
    order_date) As daily_total
```

```
FROM orders;
```

5) AVG():

Calculates the average of a column within a window defined by the Partition and order clauses.

Syntax:

```
SELECT
    column_name, AVG(column_name) OVER(
PARTITION BY partition_column ORDER BY
    order_column) AS avg_value
FROM <table_name>;
```

Example:

```
SELECT  product_id, product_name, product_price,
    AVG(product_price) OVER (PARTITION BY
    category_id) AS category_avg
FROM products;
```

6) LEAD():

Retreives the value of a column from the next row within the window defined by the

order clause.

Syntax:

```
SELECT column_name, LEAD(column_name)
    OVER(ORDER BY order_column) AS next_value
FROM <table_name>;
```

Example:

```
SELECT employee_name, salary, LEAD(salary)
    OVER (ORDER BY salary DESC) AS
    next_highest_salary
FROM employees;
```

* Useful for calculating differences or comparing adjacent values.

7) **LAG():**

Retreives the value of a column from the previous row within the window defined by the order clause.

Syntax:

```
SELECT column_name, LAG(column_name)
  OVER(ORDER BY order_column) AS previous_value
FROM <table_name>;
```

Example:

```
SELECT product_name, price, LEAD(price)
  OVER (ORDER BY price DESC) AS
  previous_price
FROM products;
```

* Useful for calculating differences or comparing adjacent values.

When to use window functions:

* Calculating running totals, averages, or aggregates within specific partitions or groups.
* Obtaining row numbers or rankings based on certain criteria.
* Analyzing trends or patterns in data by comparing current and previous/next values.
* Performing complex calculations that require access to multiple rows within a window.

** windows functions are particularly useful when we need to perform calculation on subsets of data within a result without resorting to complex subqueries or temporary tables. They provide a concise and efficient way to handle such scenarios.

SQL Hosting:

SQL hosting refers to the practice of hosting a MySQL database on a remote server or a hosting provider's infrastructure.

* It allows users to store their database and access it from anywhere with an internet connection.

When to use SQL Hosting:

1) **Web Applications:** SQL hosting is commonly used for web applications that require a reliable and accessible database. Hosting the database on a specific server ensures scalability, performance, and ease of management.

2) **Collaboration:** SQL hosting enables multiple users or teams to collaborate on a shared database. It allows them to access, modify, and retrieve data concurrently, promoting efficient teamwork.

3) **Data Security:** Hosting the database on a secure server provided by a reputable hosting provider ensures data security. Hosting providers typically employ various security measures, including firewalls, encryption, and backup systems, to protect

the database.

4) **Scalability:** SQL hosting allows for easy scalability as the application's data storage needs grow. Hosting providers offer flexible plans and resources, allowing users to scale up or down based on their requirements.

* If we want the web application to connect to a remote MySQL database we need to specify the host, port, database name, username and password.

Benefits of SQL Hosting:

* **Accessibility:** SQL hosting allows access to the database from anywhere with an internet connection, enabling remote work and collabration.
* **Reliability:** Hosting providers ensure high uptime and reliability, mimizing the risk of data loss or service interruption.
* **Scalability:** Hosting providers offer scalable solutions, allowing users to easily expand their database resources are needed.

* **Data security:** Reputable hosting providers implement robust security measures, such as encryption, firewalls, and regular backups, to protect data from unauthorized access and ensure its integrity.

** It is important to choose a reliable and secure hosting provider that meets your applications requirements for performance, scalability, and data security.

SQL Injection:

SQL injection is a common security vulnerability that occurs when an attacker is able to manipulate user input in an application that interacts with a MySQL database.

* The attacker injects malicious SQL code into the applications input fields, exploiting vulnerabilities in the applications handling of user input. When the application executes the SQL query, it unintentionally executes the injected code as well, leading to unauthorized access, data theft, or other malicious actions.

How SQL Injection works:

User input: SQL injection occurs when an application allows user input to be directly concatenated with SQL queries without proper validation or sanitization.

Malicious SQL code: An attacker can input specially crafted strings that contain SQL code fragments into the applications input fields.

Concatenation: The application combines the user input with the SQL query, treating the injected SQL code as a legitimate part of the query.

Unauthorized Actions: When the query is executed, the injected code is executed along with the original query, allowing the attacker to perform unauthorized actions on the database.

Types of SQL Injection:

1) **Union-Based SQL Injection:** The attacker exploits the UNION operator to combine the results of a malicious query with the original query's results.

Example:

```
SELECT username, password
FROM users
WHERE username = 'admin' UNION ALL
SELECT table_name, column_name
FROM information_schema.columns
```

* In this example, the attacker appends a

UNION ALL statement to retreive information from the 'information_schema.columns' table

2) **Boolean-Based SQL Injection:** The attacker exploits boolean expressions to infer information about the database.

Example:

```
SELECT product_name, price
FROM products
WHERE product_id = 1 AND 1=1
UNION ALL
SELECT username, password
FROM users
WHERE 'a' = 'a'
```

* On this example, the attacker injects a condition that always evaluates to true to retreive data from the 'users' table.

3) **Time-Based SQL Injection:** The attacker uses time delays in SQL queries to extract information based on the application's response time.

Example:

```
SELECT product_name, price
FROM products
WHERE product_id = 1;
IF (1=1, SLEEP(5), '')
```

* In this example, the attacker injects a sleep function to delay the query execution and infer information based on the response time

4) **Error-Based SQL Injection:** The attacker triggers specific errors in SQL queries to extract information from error messages.

Example:

```
SELECT product_name, price
FROM products
WHERE product_id = 1;
SELECT 1/0
```

* In this example, the attacker injects a division by zero operation to generate an error and retreive information from the error message.

5) **Blind SQL Injection:** The attacker exploits boolean-based or time-based techniques to extract information without receiving explicit results.

Example:

```
SELECT product_name, price
FROM products
WHERE product_id = 1 AND 1=1;
SELECT SLEEP(5)
```

* On this example the attacker injects a sleep function to delay the query execution, inferring information based on the applications response time.

Impact of SQL Injection:

* Unauthorized Data Access: Attackers can retreive sensitive information, such as usernames, passwords, credit card details, or other confidential data.
* Data manipulation: Attackers can modify, delete, or insert data in to the database, altering the application's behavior or compro-

mising the integrity of the data.

* Remote Code Execution: In severe cases, attackers can execute arbitary code on the server, gaining complete control over the application and underlying system.

Preventing SQL Injection:

* **Prepared statements:** use parameterized queries or prepared statements with placeholders to seperate SQL code from user input.
* **Input validation and sanitization:** Implement strong input validation and sanitization techniques to filter out or escape special characters in user input.
* **Least Privilege principle:** Assign the minimum required privileges to the application's database user to limit the potential impact of an SQL injection attack.
* **Regular Security Audits:** conduct regular security audits and penetration testing to identify and address any vulnerabilities in the application.

** It is crucial to be aware of SQL injection vulnerabilities and implement proper security measures to prevent them. By validating and sanitizing the user input, using parameterized queries, and following secure coding practices, the risk of SQL injection can be significantly reduced.