MySQL for Data Analytics - Beginner's Notes

1. Introduction to MySQL:

- MySQL is an open-source relational database management system (RDBMS) widely used for data storage and retrieval.
- It is particularly relevant for data analytics due to its robust querying capabilities and scalability.

2. Installation and Setup:

- Download and install MySQL Community Server based on your operating system.
- Set up user accounts with appropriate privileges and configure security settings.
- Familiarize yourself with the command-line interface (CLI) or consider using a graphical user interface (GUI) tool like MySQL Workbench.

3. Database Basics:

- Understand the concept of a database, which is a structured collection of data organized into tables.
- Each table consists of rows (records) and columns (attributes) that define the data structure.
- Learn SQL (Structured Query Language), the language used to interact with databases.

4. Creating Databases and Tables:

- Use the **CREATE DATABASE** statement to create a new database.
- Employ the **CREATE TABLE** statement to define tables within the database, specifying column names, data types, and constraints.
- Understand primary keys and foreign keys for maintaining data integrity.

5. Inserting, Updating, and Deleting Data:

- Use the **INSERT INTO** statement to add new records to a table.
- Utilize the **UPDATE** statement to modify existing data.
- Apply the **DELETE** statement to remove unwanted records.

6. Querying Data:

- Employ the **SELECT** statement to retrieve data from one or more tables.
- Understand the basic syntax, including the use of keywords like **FROM**, **WHERE**, **GROUP BY**, **HAVING**, **ORDER BY**, and **LIMIT**.

- Learn about functions and operators to perform calculations and manipulate data within queries.

7. Filtering and Sorting Data:

- Use the **WHERE** clause to filter data based on specified conditions.
- Apply comparison operators like **=**, **<>**, **<**, **>**, **<=**, **>=**, as well as logical operators such as **AND**, **OR**, and **NOT**.
- Sort query results using the **ORDER BY** clause.

8. Aggregating and Grouping Data:

- Employ aggregate functions like **SUM**, **AVG**, **COUNT**, **MIN**, and **MAX** to perform calculations on groups of records.
- Group data using the **GROUP BY** clause to aggregate information by specific columns.
- Filter aggregated results using the **HAVING** clause.

9. Joining Tables:

- Combine data from multiple tables using various types of joins (e.g., **INNER JOIN**, **LEFT JOIN**, **RIGHT JOIN**, **FULL JOIN**).
- Understand the concept of table aliases to simplify query writing.
- Specify join conditions using the **ON** keyword.

10. Advanced Querying:

- Learn about subqueries, which are queries embedded within other queries, to perform complex data retrieval tasks.
- Utilize functions for string manipulation, date/time calculations, and data type conversion.
- Explore the use of views, indexes, and stored procedures to optimize query performance.

11. Data Manipulation and Data Definition Language (DML and DDL):

- Understand the distinction between DML and DDL.
- DML statements (**INSERT**, **UPDATE**, **DELETE**) manipulate data within tables.
- DDL statements (**CREATE**, **ALTER**, **DROP**) define and modify the structure of databases and tables.

12. Data Analytics with MySQL:

- Apply SQL queries to perform basic data analysis tasks such as filtering, sorting, and aggregating data.

- Combine SQL with other data analytics tools (e.g., Python, R) for more advanced analyses and visualization.
- Explore the use of MySQL extensions and plugins designed specifically for analytics purposes.

MySQL Data Types - Notes

1. Numeric Data Types:

- **INT** or **INTEGER**: Used for storing whole numbers (e.g., 10, -5, 0).
- **FLOAT**: Used for storing floating-point numbers with decimal precision.
- **DOUBLE**: Similar to **FLOAT**, but with higher precision for larger numbers.
- **DECIMAL**: Used for precise decimal calculations, suitable for financial data.

2. String Data Types:

- **CHAR**: Fixed-length string with a specified maximum length.
- **VARCHAR**: Variable-length string with a maximum length.
- **TEXT**: Used for storing large amounts of text data (e.g., paragraphs, documents).
- **ENUM**: Defines a list of predefined values that a column can take.

3. Date and Time Data Types:

- **DATE**: Stores a date in the format 'YYYY-MM-DD'.
- **TIME**: Represents a time value in the format 'HH:MM:SS'.
- **DATETIME**: Combines date and time in the format 'YYYY-MM-DD HH:MM:SS'.
- **TIMESTAMP**: Stores a timestamp representing a specific point in time.

4. Boolean Data Types:

- **BOOL** or **BOOLEAN**: Used to represent boolean values (true or false).

5. Binary Data Types:

- **BINARY**: Fixed-length binary data.
- **VARBINARY**: Variable-length binary data.
- **BLOB**: Used for storing large binary objects (e.g., images, files).

6. Other Data Types:

- **SET**: Stores a set of values chosen from a predefined list.

- **JSON**: Introduced in MySQL 5.7, used for storing and manipulating JSON (JavaScript Object Notation) data.

7. Spatial Data Types:

- **GEOMETRY**, **POINT**, **LINESTRING**, **POLYGON**, etc.: Used for storing and querying spatial data (e.g., coordinates, shapes).

MySQL Data Types and Size - Notes

1. Numeric Data Types:

- **TINYINT**: 1 byte (values from -128 to 127 or 0 to 255)
- **SMALLINT**: 2 bytes (values from -32,768 to 32,767 or 0 to 65,535)
- **INT** or **INTEGER**: 4 bytes (values from -2,147,483,648 to 2,147,483,647 or 0 to 4,294,967,295)
- **BIGINT**: 8 bytes (values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 or 0 to 18,446,744,073,709,551,615)
- **FLOAT**: 4 bytes (single-precision floating-point number)
- **DOUBLE**: 8 bytes (double-precision floating-point number)
- **DECIMAL**: Variable size, specified as **DECIMAL(precision, scale)**

2. String Data Types:

- **CHAR**: 0 to 255 bytes (fixed-length)
- **VARCHAR**: 0 to 65,535 bytes (variable-length)
- **TEXT**: 0 to 65,535 bytes or 16MB (depending on the configuration)
- **ENUM**: 1 or 2 bytes (depending on the number of values)

3. Date and Time Data Types:

- **DATE**: 3 bytes (range from '1000-01-01' to '9999-12-31')
- **TIME**: 3 bytes or 6 bytes (range from '-838:59:59' to '838:59:59')
- **DATETIME**: 8 bytes (range from '1000-01-01 00:00:00' to '9999-12-31 23:59:59')
- **TIMESTAMP**: 4 bytes (range from '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC)

4. Boolean Data Types:

- **BOOL** or **BOOLEAN**: 1 byte (values are 0 or 1)

5. Binary Data Types:

- **BINARY**: 1 to 255 bytes (fixed-length)
- **VARBINARY**: 1 to 65,535 bytes (variable-length)
- **BLOB**: 0 to 65,535 bytes or 16MB (depending on the configuration)

6. Other Data Types:

- **SET**: 1, 2, 3, 4, or 8 bytes (depending on the number of set members)
- **JSON**: Variable size (depending on the JSON data being stored)

7. Spatial Data Types:

- Sizes vary depending on the specific spatial type and the configuration.

Operators Used in MySQL - Notes

Operators in MySQL allow you to perform various operations on data, such as mathematical calculations, logical comparisons, and string manipulations. Here are the key operators used in MySQL:

1. Arithmetic Operators:

- **+**: Addition
- **-**: Subtraction
- *****: Multiplication
- **/**: Division
- **%**: Modulo (remainder of division)

2. Comparison Operators:

- **=**: Equal to
- **<>** or **!=**: Not equal to
- **<**: Less than
- **>**: Greater than
- **<=**: Less than or equal to
- **>=**: Greater than or equal to

3. Logical Operators:

- **AND**: Logical AND operator
- **OR**: Logical OR operator
- **NOT**: Logical NOT operator

4. String Operators:

- **CONCAT**: Concatenates two or more strings together
- **LIKE**: Pattern matching operator (supports wildcard characters **%** and **_**)

5. Assignment Operators:

- **=**: Assigns a value to a variable or column
- **+=**, **-=**, ***=**, **/=**, **%=**: Perform an arithmetic operation and assign the result back to the variable or column

6. Null-Safe Equality Operator:

- **<=>**: Performs a comparison that is true even if one or both operands are NULL

7. IN Operator:

- Used to specify multiple values in a WHERE clause
- Example: **SELECT column FROM table WHERE column IN (value1, value2, ...)**

8. BETWEEN Operator:

- Used to specify a range of values in a WHERE clause
- Example: **SELECT column FROM table WHERE column BETWEEN value1 AND value2**

9. IS NULL / IS NOT NULL Operators:

- Used to check for NULL values in a WHERE clause
- Example: **SELECT column FROM table WHERE column IS NULL**

10. EXISTS Operator:

- Used to check for the existence of rows in a subquery
- Example: **SELECT column FROM table1 WHERE EXISTS (SELECT column FROM table2 WHERE condition)**

Querying Data in MySQL - Detailed Notes

Querying data is one of the fundamental tasks in MySQL. It allows you to retrieve, filter, sort, and aggregate data from one or more tables. Here are the key concepts and syntax to effectively query data:

1. The SELECT Statement:

- The SELECT statement is used to retrieve data from one or more tables.
- Basic syntax: **SELECT column1, column2, ... FROM table_name;**
- You can specify specific columns to retrieve or use ***** to retrieve all columns.

- Example: **SELECT name, age FROM customers;**

2. Filtering Data with WHERE:

- The WHERE clause filters data based on specified conditions.
- Basic syntax: **SELECT column1, column2, ... FROM table_name WHERE condition;**
- Conditions can use comparison operators (e.g., **=**, **<>**, **<**, **>**) and logical operators (**AND**, **OR**, **NOT**).
- Example: **SELECT name, age FROM customers WHERE age > 25;**

3. Sorting Data with ORDER BY:

- The ORDER BY clause sorts query results based on one or more columns.
- Basic syntax: **SELECT column1, column2, ... FROM table_name ORDER BY column_name [ASC|DESC];**
- ASC (ascending) is the default sorting order, and DESC (descending) sorts in reverse order.
- Example: **SELECT name, age FROM customers ORDER BY age DESC;**

4. Aggregating Data with GROUP BY:

- The GROUP BY clause groups query results based on one or more columns.
- Basic syntax: **SELECT column1, column2, ... FROM table_name GROUP BY column_name1, column_name2, ...;**
- Used in combination with aggregate functions like COUNT, SUM, AVG, MIN, MAX, etc.
- Example: **SELECT country, COUNT(*) FROM customers GROUP BY country;**

5. Filtering Grouped Data with HAVING:

- The HAVING clause filters groups based on conditions after the GROUP BY operation.
- Basic syntax: **SELECT column1, column2, ... FROM table_name GROUP BY column_name HAVING condition;**
- Allows filtering based on aggregate function results.
- Example: **SELECT country, COUNT(*) FROM customers GROUP BY country HAVING COUNT(*) > 10;**

6. Limiting the Number of Rows with LIMIT:

- The LIMIT clause restricts the number of rows returned by a query.
- Basic syntax: **SELECT column1, column2, ... FROM table_name LIMIT number_of_rows;**
- Example: **SELECT name, age FROM customers LIMIT 10;**

LIKE Operator in MySQL - Notes

The LIKE operator in MySQL is used to perform pattern matching on string values. It allows you to search for rows that match a specific pattern or contain a particular substring. Here are the key aspects of the LIKE operator:

1. Basic Syntax:

- The LIKE operator is used in the WHERE clause of a SELECT statement.
- Syntax: **SELECT columns FROM table WHERE column LIKE pattern;**

2. Wildcard Characters:

- The LIKE operator uses two wildcard characters for pattern matching:
    - **%** (percent sign): Matches any sequence of characters (including zero characters).
    - **_** (underscore): Matches any single character.
- Example: **SELECT * FROM customers WHERE customer_name LIKE 'J%';**

3. Matching Patterns:

- You can use the wildcard characters with the LIKE operator to create different pattern matching scenarios:
    - **LIKE 'J%'**: Matches any value that starts with 'J'.
    - **LIKE '%son'**: Matches any value that ends with 'son'.
    - **LIKE '%do%'**: Matches any value that contains 'do' anywhere.
    - **LIKE '_on%'**: Matches any value that has 'on' as the second and third characters.
- Example: **SELECT * FROM products WHERE product_name LIKE '%apple%';**

String Operators in MySQL - Detailed Notes

MySQL provides various string operators that allow you to manipulate and compare strings. Understanding these operators can help you perform powerful string operations in your queries. Here are the key string operators in MySQL:

1. CONCAT Operator:

- The CONCAT operator is used to concatenate two or more strings together.
- Basic syntax: **SELECT CONCAT(string1, string2, ...) FROM table_name;**
- Example: **SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM customers;**

2. LIKE Operator:

- The LIKE operator is used for pattern matching in string comparisons.
- It supports the use of wildcard characters:
    - **%** represents any sequence of characters (including an empty sequence).
    - **_** represents any single character.
- Basic syntax: **SELECT column FROM table_name WHERE column LIKE pattern;**
- Example: **SELECT product_name FROM products WHERE product_name LIKE 'Apple%';**

3. SUBSTRING Operator:

- The SUBSTRING operator extracts a substring from a string.
- Basic syntax: **SELECT SUBSTRING(string, start_position, length) FROM table_name;**
- Example: **SELECT SUBSTRING(description, 1, 10) AS shortened_description FROM products;**

4. LENGTH Operator:

- The LENGTH operator returns the length (number of characters) of a string.
- Basic syntax: **SELECT LENGTH(string) FROM table_name;**
- Example: **SELECT LENGTH(product_name) AS name_length FROM products;**

5. UPPER and LOWER Operators:

- The UPPER operator converts a string to uppercase.
- The LOWER operator converts a string to lowercase.
- Basic syntax: **SELECT UPPER(string) FROM table_name;** or **SELECT LOWER(string) FROM table_name;**
- Example: **SELECT UPPER(last_name) AS last_name_upper FROM customers;**

6. REPLACE Operator:

- The REPLACE operator replaces occurrences of a specified string within another string.
- Basic syntax: **SELECT REPLACE(string, search_string, replacement_string) FROM table_name;**
- Example: **SELECT REPLACE(description, 'old', 'new') AS updated_description FROM products;**

7. TRIM Operator:

- The TRIM operator removes leading and trailing spaces (or specified characters) from a string.
- Basic syntax: **SELECT TRIM(string) FROM table_name;**
- Example: **SELECT TRIM(product_name) AS trimmed_name FROM products;**

8. CONCAT_WS Operator:

- The CONCAT_WS operator concatenates strings with a specified separator.
- The first argument is the separator, followed by the strings to be concatenated.
- Basic syntax: **SELECT CONCAT_WS(separator, string1, string2, ...) FROM table_name;**
- Example: **SELECT CONCAT_WS(', ', first_name, last_name) AS full_name FROM customers;**

Date Time Functions

- SELECT NOW();
- SELECT CURDATE();
- SELECT CURTIME();
- SELECT DATE('2023-05-19 14:30:45');
- SELECT TIME('2023-05-19 14:30:45');
- SELECT YEAR('2023-05-19');
- SELECT MONTH('2023-05-19');
- SELECT DAY('2023-05-19');
- SELECT HOUR('14:30:45');
- SELECT MINUTE('14:30:45');
- SELECT SECOND('14:30:45');
- SELECT TIMESTAMP('2023-05-19 14:30:45');
- SELECT DATE_FORMAT('2023-05-19', '%Y-%m-%d');
- SELECT ADDDATE('2023-05-19', INTERVAL 7 DAY);
- SELECT SUBDATE('2023-05-19', INTERVAL 7 DAY);
- SELECT DATEDIFF('2023-05-19', '2023-05-12');
- SELECT DATE_ADD('2023-05-19', INTERVAL 1 MONTH);
- SELECT DATE_SUB('2023-05-19', INTERVAL 1 WEEK);

Aggregating Keywords in MySQL - Detailed Notes

Aggregating keywords in MySQL allow you to perform calculations and summarize data across multiple rows. These keywords are essential for generating meaningful insights from your data. Here are the key aggregating keywords in MySQL:

1. COUNT:

- The COUNT keyword calculates the number of rows or non-null values in a column.

- Basic syntax: **SELECT COUNT(column) FROM table_name;**
- Example: **SELECT COUNT(*) FROM orders;**

2. SUM:

- The SUM keyword calculates the sum of values in a numeric column.
- Basic syntax: **SELECT SUM(column) FROM table_name;**
- Example: **SELECT SUM(total_price) FROM orders;**

3. AVG:

- The AVG keyword calculates the average value of a numeric column.
- Basic syntax: **SELECT AVG(column) FROM table_name;**
- Example: **SELECT AVG(order_amount) FROM orders;**

4. MIN:

- The MIN keyword retrieves the minimum value from a column.
- Basic syntax: **SELECT MIN(column) FROM table_name;**
- Example: **SELECT MIN(order_date) FROM orders;**

5. MAX:

- The MAX keyword retrieves the maximum value from a column.
- Basic syntax: **SELECT MAX(column) FROM table_name;**
- Example: **SELECT MAX(order_amount) FROM orders;**

6. GROUP_CONCAT:

- The GROUP_CONCAT keyword concatenates strings from multiple rows into a single string.
- Basic syntax: **SELECT GROUP_CONCAT(column) FROM table_name GROUP BY grouping_column;**
- Example: **SELECT GROUP_CONCAT(product_name) FROM order_items GROUP BY order_id;**

7. GROUP BY:

- The GROUP BY clause groups query results based on one or more columns.
- Used in conjunction with aggregate functions like COUNT, SUM, AVG, MIN, MAX, etc.
- Basic syntax: **SELECT column1, aggregate_function(column2) FROM table_name GROUP BY column1;**
- Example: **SELECT category, SUM(sales) FROM products GROUP BY category;**

8. HAVING:

- The HAVING clause filters groups created by GROUP BY based on specified conditions.
- Similar to the WHERE clause but operates on groups rather than individual rows.
- Basic syntax: **SELECT column1, aggregate_function(column2) FROM table_name GROUP BY column1 HAVING condition;**
- Example: **SELECT category, SUM(sales) FROM products GROUP BY category HAVING SUM(sales) > 1000;**

Statistical Functions in MySQL - Notes

MySQL provides several statistical functions that allow you to perform calculations and analyze data. These functions help you derive insights and draw conclusions from your data. Here are the key statistical functions in MySQL:

1. AVG():
   - Calculates the average (mean) value of a column.
   - Syntax: **AVG(column)**
2. SUM():
   - Calculates the sum of values in a column.
   - Syntax: **SUM(column)**
3. COUNT():
   - Counts the number of rows or non-null values in a column.
   - Syntax: **COUNT(column)** or **COUNT(*)** (counts all rows)
   - Can be used with DISTINCT to count unique values: **COUNT(DISTINCT column)**
4. MIN():
   - Retrieves the minimum value from a column.
   - Syntax: **MIN(column)**
5. MAX():
   - Retrieves the maximum value from a column.
   - Syntax: **MAX(column)**
6. STD():
   - Calculates the population standard deviation of a column.

- Syntax: **STD(column)**

7. STDDEV():
    - Calculates the sample standard deviation of a column.
    - Syntax: **STDDEV(column)**
8. VARIANCE():
    - Calculates the population variance of a column.
    - Syntax: **VARIANCE(column)**
9. STDDEV_POP():
    - Calculates the population standard deviation of a column.
    - Syntax: **STDDEV_POP(column)**
10. STDDEV_SAMP():
    - Calculates the sample standard deviation of a column.
    - Syntax: **STDDEV_SAMP(column)**
11. VAR_POP():
    - Calculates the population variance of a column.
    - Syntax: **VAR_POP(column)**
12. VAR_SAMP():
    - Calculates the sample variance of a column.
    - Syntax: **VAR_SAMP(column)**
13. CORR():
    - Calculates the correlation coefficient between two columns.
    - Syntax: **CORR(column1, column2)**
14. COVAR_POP():
    - Calculates the population covariance between two columns.
    - Syntax: **COVAR_POP(column1, column2)**
15. COVAR_SAMP():
    - Calculates the sample covariance between two columns.
    - Syntax: **COVAR_SAMP(column1, column2)**

Subqueries in MySQL - Notes

Subqueries, also known as nested queries or inner queries, allow you to use the result of one query within another query. They are enclosed in parentheses and can be used in various parts of a SQL statement. Here are the key aspects of subqueries in MySQL:

1. Syntax:

- Subquery in SELECT clause: **SELECT column1, (SELECT column2 FROM table2 WHERE condition) FROM table1;**
- Subquery in FROM clause: **SELECT column1 FROM (SELECT column2 FROM table2 WHERE condition) AS subquery_table;**
- Subquery in WHERE clause: **SELECT column1 FROM table1 WHERE column2 = (SELECT column2 FROM table2 WHERE condition);**

1. Subquery in SELECT Clause:
    - Get the total number of products and the count of orders for each product:

```
SELECT product_id, product_name, (SELECT COUNT(*) FROM orders WHERE orders.product_id = products.product_id) AS order_count FROM products;
```

2. Subquery in FROM Clause:
    - Get the average order amount for each customer by using a subquery as a derived table:

```
SELECT customers.customer_id, customers.customer_name, avg_order.amount_avg FROM customers INNER JOIN (SELECT customer_id, AVG(order_amount) AS amount_avg FROM orders GROUP BY customer_id) AS avg_order ON customers.customer_id = avg_order.customer_id;
```

3. Subquery in WHERE Clause:
    - Get all customers who have placed an order in the last 30 days:

```
SELECT customer_id, customer_name FROM customers WHERE customer_id IN (SELECT DISTINCT customer_id FROM orders WHERE order_date >= CURDATE() - INTERVAL 30 DAY);
```

4. Subquery with Comparison Operators:
    - Get all products whose price is higher than the average price of all products:

```
SELECT product_id, product_name, price FROM products WHERE price > (SELECT AVG(price) FROM products);
```

5. Subquery with EXISTS Operator:
    - Get all customers who have placed at least one order:

```
SELECT customer_id, customer_name FROM customers WHERE EXISTS (SELECT * FROM orders WHERE orders.customer_id = customers.customer_id);
```

6. Subquery with ANY/SOME Operator:
    - Get all orders with order amounts greater than any order placed by customer ID 1:

```
SELECT order_id, order_amount FROM orders WHERE order_amount > ANY (SELECT order_amount FROM orders WHERE customer_id = 1);
```

These examples demonstrate different scenarios where subqueries can be applied to enhance data retrieval and analysis in MySQL. By leveraging subqueries effectively, you can perform complex queries and obtain specific information based on the results of inner queries.

Retrieving and Saving Data in a New Table - Notes

To retrieve data from an existing table and save it in a new table in MySQL, you can follow these steps:

1. Create the new table:
    - Use the **CREATE TABLE** statement to define the structure of the new table, including column names, data types, and constraints.
2. Retrieve data from the existing table:
    - Use the **SELECT** statement to specify the data you want to retrieve from the existing table.
    - Customize the query using filtering conditions, sorting, grouping, and aggregate functions as needed.
3. Insert the retrieved data into the new table:
    - Use the **INSERT INTO** statement to insert the retrieved data into the new table.
    - Specify the columns into which the data will be inserted and the source of the data (the **SELECT** statement).
4. Execute the query:

- Run the query in the MySQL environment, such as the command-line interface or a GUI tool like MySQL Workbench.

Example:

Suppose you have an existing table named **customers** with columns **customer_id**, **name**, **email**, and **phone**. You want to retrieve the data for customers who have made a purchase and save it in a new table named **purchasing_customers**.

Here's an example query to achieve this:

```
CREATE TABLE purchasing_customers ( customer_id INT, name VARCHAR(255), email VARCHAR(255), phone VARCHAR(20) );

INSERT INTO purchasing_customers (customer_id, name, email, phone) SELECT customer_id, name, email, phone FROM customers WHERE purchase_amount > 0;
```

In this example, the **CREATE TABLE** statement defines the structure of the **purchasing_customers** table. The **INSERT INTO** statement inserts the retrieved data from the **customers** table into the new table, filtering out customers who haven't made a purchase.

Remember to adapt the column names, data types, and conditions to match your specific scenario.

Joins in MySQL - Notes

Joins in MySQL allow you to combine data from multiple tables based on related columns. They are cr ucial for retrieving data that spans across different tables. Here are the key aspects of joins in MySQL:

1. INNER JOIN:
    - The INNER JOIN returns only the matching rows from both tables.
    - Syntax: **SELECT columns FROM table1 INNER JOIN table2 ON table1.column = table2.column;**
    - Example: **SELECT customers.customer_id, orders.order_id FROM customers INNER JOIN orders ON customers.customer_id = orders.customer_id;**
2. LEFT JOIN (or LEFT OUTER JOIN):
    - The LEFT JOIN returns all rows from the left table and the matching rows from the right table.
    - If there is no match, NULL values are returned for the right table.

- Syntax: **SELECT columns FROM table1 LEFT JOIN table2 ON table1.column = table2.column;**
- Example: **SELECT customers.customer_id, orders.order_id FROM customers LEFT JOIN orders ON customers.customer_id = orders.customer_id;**

3. RIGHT JOIN (or RIGHT OUTER JOIN):
   - The RIGHT JOIN returns all rows from the right table and the matching rows from the left table.
   - If there is no match, NULL values are returned for the left table.
   - Syntax: **SELECT columns FROM table1 RIGHT JOIN table2 ON table1.column = table2.column;**
   - Example: **SELECT customers.customer_id, orders.order_id FROM customers RIGHT JOIN orders ON customers.customer_id = orders.customer_id;**
4. FULL JOIN (or FULL OUTER JOIN):
   - The FULL JOIN returns all rows from both tables and combines the result.
   - If there is no match, NULL values are returned for the non-matching table.
   - Syntax: **SELECT columns FROM table1 FULL JOIN table2 ON table1.column = table2.column;**
   - Example: **SELECT customers.customer_id, orders.order_id FROM customers FULL JOIN orders ON customers.customer_id = orders.customer_id;**
5. CROSS JOIN:
   - The CROSS JOIN returns the Cartesian product of both tables.
   - It combines each row from the first table with each row from the second table.
   - Syntax: **SELECT columns FROM table1 CROSS JOIN table2;**
   - Example: **SELECT customers.customer_id, products.product_id FROM customers CROSS JOIN products;**
6. Self-Join:
   - A self-join is used when a table needs to be joined with itself.
   - It requires the use of table aliases to differentiate between the two instances of the same table.
   - Syntax: **SELECT columns FROM table1 AS t1 INNER JOIN table1 AS t2 ON t1.column = t2.column;**

- Example: **SELECT e.employee_name, m.employee_name AS manager_name FROM employees AS e INNER JOIN employees AS m ON e.manager_id = m.employee_id;**

Window Functions in MySQL - Notes

Window functions, also known as analytical functions, are a powerful feature in MySQL that allow you to perform calculations on a set of rows within a specific window or partition. Here are the key aspects of window functions in MySQL:

1. Syntax:

- The syntax for window functions is:

function_name(expression) OVER (PARTITION BY partition_columns ORDER BY order_columns [window_frame])

2. Partitioning:

- The PARTITION BY clause divides the rows into partitions or groups based on specified columns.
- Each partition is treated separately for the application of window functions.
- Example: **SUM(sales) OVER (PARTITION BY category)**

3. Ordering:

- The ORDER BY clause specifies the order of rows within each partition.
- It determines the sequence in which the window function operates.
- Example: **ROW_NUMBER() OVER (ORDER BY date)**

4. Commonly Used Window Functions:

1. ROW_NUMBER():
   - Assigns a unique sequential number to each row within a partition.
   - Useful for generating row identifiers or ranking rows based on a specific order.
   - Example: **ROW_NUMBER() OVER (ORDER BY sales DESC)**
2. RANK() and DENSE_RANK():
   - RANK() calculates the rank of each row within a partition based on specified criteria, with gaps in rankings for ties.
   - DENSE_RANK() calculates the rank of each row within a partition, but with no gaps for ties.
   - Example: **RANK() OVER (PARTITION BY category ORDER BY sales DESC)**
3. LAG() and LEAD():
   - LAG() allows you to access the value of a column from the previous row within a partition.

- LEAD() allows you to access the value of a column from the next row within a partition.
- Useful for performing calculations involving values from adjacent rows.
- Example: **LAG(sales) OVER (PARTITION BY category ORDER BY date)**

4. SUM(), AVG(), COUNT(), MIN(), MAX():
   - These are aggregate functions that can be used as window functions.
   - When used with the OVER clause, they calculate the sum, average, count, minimum, or maximum over a specified window of rows within a partition.
   - Example: **SUM(sales) OVER (PARTITION BY category ORDER BY date ROWS BETWEEN 3 PRECEDING AND CURRENT ROW)**
5. FIRST_VALUE() and LAST_VALUE():
   - FIRST_VALUE() retrieves the first value within a window of rows.
   - LAST_VALUE() retrieves the last value within a window of rows.
   - Useful for retrieving the earliest or latest value based on a specific order within a partition.
   - Example: **FIRST_VALUE(date) OVER (PARTITION BY category ORDER BY date)**
6. NTILE():
   - Divides the rows within a partition into the specified number of buckets or groups.
   - Assigns a bucket number to each row, distributing them equally as much as possible.
   - Example: **NTILE(4) OVER (PARTITION BY category ORDER BY sales DESC)**
7. CUME_DIST():
   - Calculates the cumulative distribution of a value within a partition.
   - Returns a value between 0 and 1 representing the fraction of rows with a value less than or equal to the current row's value.
   - Example: **CUME_DIST() OVER (PARTITION BY category ORDER BY sales)**

Views in MySQL - Notes

Views in MySQL are virtual tables created from the result of a query. They provide a way to store complex queries as a named object, allowing you to simplify data access and enhance security. Here are the key aspects of views in MySQL:

1. Creation Syntax:
   - Syntax: **CREATE VIEW view_name AS SELECT statement;**
   - Example: **CREATE VIEW customer_view AS SELECT customer_name, email FROM customers WHERE status = 'active';**
2. Purpose and Benefits of Views:
   - Simplify Complex Queries: Views allow you to encapsulate complex queries into a single, easily readable and reusable object.

- Data Abstraction: Views provide a layer of abstraction, presenting a customized view of the data to the users, hiding the underlying complexity.
- Security and Permissions: Views can restrict access to specific columns or rows, ensuring data confidentiality and controlling user permissions.
- Data Consistency: Views can help enforce data consistency by applying filters, calculations, or aggregations to the underlying data.
- Modular Approach: Views promote a modular approach to database design, breaking down complex queries into smaller, manageable components.

3. Updating Views:
   - Views can be updatable, allowing you to modify the underlying data through the view.
   - The updatability of views depends on several factors, such as the complexity of the query and the presence of certain operations in the view definition.
   - Views created with the **WITH CHECK OPTION** clause ensure that any modifications made through the view satisfy the view's filter condition.
4. Altering and Dropping Views:
   - Views can be altered using the **ALTER VIEW** statement to modify the view's definition.
   - Syntax: **ALTER VIEW view_name AS SELECT statement;**
   - Views can be dropped using the **DROP VIEW** statement to remove the view from the database.
   - Syntax: **DROP VIEW view_name;**
5. Querying Views:
   - Views can be queried just like regular tables in MySQL.
   - Syntax: **SELECT * FROM view_name;**
6. Nesting and Joining Views:
   - Views can be nested, where one view is built upon another view.
   - Views can be joined with other views or tables in complex queries, providing a powerful way to combine data from multiple sources.

Stored Procedures in MySQL - Notes

Stored procedures in MySQL are pre-compiled, named database objects that encapsulate a set of SQL statements. They allow you to group SQL statements together and execute them as a single unit. Here are the key aspects of stored procedures in MySQL:

1. Creation Syntax:

- Syntax:

CREATE PROCEDURE procedure_name ([parameter_list]) BEGIN -- SQL statements END;

- Example:

CREATE PROCEDURE GetCustomerOrders(IN customer_id INT) BEGIN SELECT * FROM orders WHERE customer_id = customer_id; END;

2. Parameters:
    - Stored procedures can have input, output, or input/output parameters.
    - Input parameters are used to pass values into the stored procedure.
    - Output parameters are used to return values from the stored procedure.
    - Input/output parameters can be used for both passing values into the procedure and returning values.
    - Parameters are declared in the parameter_list section of the CREATE PROCEDURE statement.