Key Differences Union, Intersect & Except

Result Set:

UNION combines and returns all distinct rows from both queries.

EXCEPT returns rows that exist in the first query but not in the second query.

INTERSECT returns common rows between both queries.

Duplicate Rows:

UNION automatically eliminates duplicate rows.

EXCEPT and INTERSECT automatically eliminate duplicate rows as well.

Column Requirements:

In UNION, EXCEPT, and INTERSECT, the number and order of columns in both SELECT statements must be the same, and column names and data types must be compatible.

Performance:

Depending on the specific queries, the performance characteristics of UNION, EXCEPT, and INTERSECT may vary.

The order of execution of SQL queries

It follows a logical sequence of steps. Understanding this order helps in writing and optimizing queries effectively. The typical order of execution for a SQL query is as follows:

FROM Clause:

Specifies the tables from which data will be retrieved.

Defines the source tables for the query.

JOIN Clause:

Combines rows from two or more tables based on specified conditions.

Performs the necessary joins to create a combined result set.

WHERE Clause:

Filters rows based on specified conditions.

Reduces the result set by eliminating rows that do not meet the specified criteria.

GROUP BY Clause:

Groups rows based on one or more columns.

Creates groups of rows that share common values in specified columns.

HAVING Clause:

Filters groups based on aggregate conditions.

Reduces the grouped result set by eliminating groups that do not meet the specified criteria.

SELECT Clause:

Specifies the columns to be included in the result set.

Performs any necessary calculations or expressions on the selected columns.

DISTINCT Keyword:

Removes duplicate rows from the result set.

Considers only unique combinations of column values.

ORDER BY Clause:

Sorts the result set based on specified columns and sorting directions.

Produces the final ordered result set.

LIMIT/OFFSET (Optional):

Limits the number of rows returned by the query.

Used for pagination or to retrieve a specific subset of rows.

Rules and Restrictions to Group and Filter Data in SQL queries

When grouping and filtering data in SQL queries, there are several rules and restrictions to keep in mind to ensure that your queries are syntactically correct and produce the desired results. Here are some important considerations:

GROUP BY Clause:

- Columns in SELECT Clause: Columns in the SELECT clause that are not part of an aggregate function must be included in the GROUP BY clause.
- Aggregate Functions: When using the GROUP BY clause, you can use aggregate functions like COUNT, SUM, AVG, etc., to perform calculations on groups of data.

HAVING Clause:

- Filtering Aggregated Data: The HAVING clause is used to filter the results of aggregate functions in a GROUP BY query.
- Usage After GROUP BY: HAVING must appear after the GROUP BY clause.

WHERE Clause:

- Filtering Individual Rows: The WHERE clause is used to filter individual rows before any grouping or aggregation.
- Usage Before GROUP BY: WHERE is applied before GROUP BY, so it filters individual rows before they are grouped.

General Guidelines:

- Column Aliases: When using column aliases in SELECT, you cannot reference the alias in the WHERE or HAVING clauses. This is because the WHERE and HAVING clauses are evaluated before the SELECT clause.
- Order of Clauses: Understand the order in which SQL processes clauses. Typically, it's FROM, WHERE, GROUP BY, HAVING, SELECT, and ORDER BY.
- Combining Aggregation and Individual Rows: Be cautious when combining aggregated functions with individual rows in the same query. Make sure you understand how the aggregation is affected.

Snowflake & Star Schema

Comparison:

- Snowflake Schema is more normalized, leading to reduced redundancy but increased complexity.
- Star Schema is denormalized, offering simplicity and better query performance at the cost of some redundancy.

- The choice between the two depends on specific use cases, query patterns, and performance requirements. Snowflake Schemas are often chosen when storage efficiency is a significant concern, while Star Schemas are preferred for their simplicity and query performance advantages.

Window Function Example:

Command Prompt - mysql -u root -p

```
6 rows in set (0.00 sec)

mysql> -- Using a window function to calculate the average age for each row
mysql> SELECT
    ->     id,
    ->     name,
    ->     age,
    ->     email,
    ->     AVG(age) OVER () AS average_age
    -> FROM
    ->     sample_table;
+----+-------------+------+-------------------------+--------------------+
| id | name        | age  | email                   | average_age        |
+----+-------------+------+-------------------------+--------------------+
|  1 | John Doe    | 25   | john.doe@example.com    | 21.666666666666668 |
|  2 | Jane Smith  | 0    | jane.smith@example.com  | 21.666666666666668 |
|  3 | Bob Johnson | 30   | default@example.com     | 21.666666666666668 |
|  4 | Alice Brown | 22   | alice.brown@example.com | 21.666666666666668 |
|  5 | Chris Lee   | 28   | chris.lee@example.com   | 21.666666666666668 |
|  6 | Jane Smith  | 25   | jane.smith@example.com  | 21.666666666666668 |
+----+-------------+------+-------------------------+--------------------+
6 rows in set (0.03 sec)
```

Common Table Expression (CTE) Example:

```
mysql> WITH AvgAgeCTE AS (
    ->     SELECT AVG(age) AS avg_age FROM sample_table
    -> )
    -> SELECT
    ->     id,
    ->     name,
    ->     age,
    ->     email
    -> FROM
    ->     sample_table
    -> WHERE
    ->     age > (SELECT avg_age FROM AvgAgeCTE);
+----+-------------+------+-------------------------+
| id | name        | age  | email                   |
+----+-------------+------+-------------------------+
|  1 | John Doe    | 25   | john.doe@example.com    |
|  3 | Bob Johnson | 30   | default@example.com     |
|  4 | Alice Brown | 22   | alice.brown@example.com |
|  5 | Chris Lee   | 28   | chris.lee@example.com   |
|  6 | Jane Smith  | 25   | jane.smith@example.com  |
+----+-------------+------+-------------------------+
5 rows in set (0.00 sec)

mysql>
```

Regular Expression Examples:

Command Prompt - mysql -u root -p

```
mysql> use etl_db;
Database changed
mysql> -- Display rows where the email addresses contain 'example'
mysql> SELECT * FROM sample_table WHERE email REGEXP 'example';
+----+-------------+------+-------------------------+
| id | name        | age  | email                   |
+----+-------------+------+-------------------------+
|  1 | John Doe    | 25   | john.doe@example.com    |
|  2 | Jane Smith  | 0    | jane.smith@example.com  |
|  3 | Bob Johnson | 30   | default@example.com     |
|  4 | Alice Brown | 22   | alice.brown@example.com |
|  5 | Chris Lee   | 28   | chris.lee@example.com   |
|  6 | Jane Smith  | 25   | jane.smith@example.com  |
+----+-------------+------+-------------------------+
6 rows in set (0.05 sec)

mysql> -- Display rows where the email addresses start with 'j'
mysql> SELECT * FROM sample_table WHERE email REGEXP '^j';
+----+------------+------+------------------------+
| id | name       | age  | email                  |
+----+------------+------+------------------------+
|  1 | John Doe   | 25   | john.doe@example.com   |
|  2 | Jane Smith | 0    | jane.smith@example.com |
|  6 | Jane Smith | 25   | jane.smith@example.com |
+----+------------+------+------------------------+
3 rows in set (0.00 sec)

mysql> -- Display rows where the email addresses end with 'com'
mysql> SELECT * FROM sample_table WHERE email REGEXP 'com$';
+----+-------------+------+-------------------------+
| id | name        | age  | email                   |
+----+-------------+------+-------------------------+
|  1 | John Doe    | 25   | john.doe@example.com    |
|  2 | Jane Smith  | 0    | jane.smith@example.com  |
|  3 | Bob Johnson | 30   | default@example.com     |
|  4 | Alice Brown | 22   | alice.brown@example.com |
|  5 | Chris Lee   | 28   | chris.lee@example.com   |
|  6 | Jane Smith  | 25   | jane.smith@example.com  |
+----+-------------+------+-------------------------+
6 rows in set (0.00 sec)

mysql> -- Display rows where the email addresses contain digits
mysql> SELECT * FROM sample_table WHERE email REGEXP '[0-9]';
Empty set (0.00 sec)

mysql>
```

View Tables Example:

Command Prompt - mysql -u root -p

```
mysql> -- Display rows where the email addresses contain digits
mysql> SELECT * FROM sample_table WHERE email REGEXP '[0-9]';
Empty set (0.00 sec)

mysql> -- Create a view
mysql> CREATE VIEW sample_view AS
    -> SELECT id, name, age, email
    -> FROM sample_table
    -> WHERE age IS NOT NULL;
Query OK, 0 rows affected (0.02 sec)

mysql> -- Query the view
mysql> SELECT * FROM sample_view;
+----+-------------+------+-------------------------+
| id | name        | age  | email                   |
+----+-------------+------+-------------------------+
|  1 | John Doe    | 25   | john.doe@example.com    |
|  2 | Jane Smith  | 0    | jane.smith@example.com  |
|  3 | Bob Johnson | 30   | default@example.com     |
|  4 | Alice Brown | 22   | alice.brown@example.com |
|  5 | Chris Lee   | 28   | chris.lee@example.com   |
|  6 | Jane Smith  | 25   | jane.smith@example.com  |
+----+-------------+------+-------------------------+
6 rows in set (0.03 sec)

mysql>
```

Materialized View Example:

Command Prompt - mysql -u root -p

```
mysql> -- Create a materialized view-like table
mysql> CREATE TABLE materialized_view_table AS
    -> SELECT id, name, age, email
    -> FROM sample_table
    -> WHERE age IS NOT NULL;
Query OK, 6 rows affected (0.05 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> -- Create a trigger to refresh the materialized view-like table when the source table changes
mysql> DELIMITER //
mysql> CREATE TRIGGER refresh_materialized_view
    -> AFTER INSERT ON sample_table
    -> FOR EACH ROW
    -> BEGIN
    ->     DELETE FROM materialized_view_table WHERE id = NEW.id;
    ->     INSERT INTO materialized_view_table SELECT * FROM sample_table WHERE id = NEW.id;
    -> END;
    -> //
Query OK, 0 rows affected (0.04 sec)

mysql> DELIMITER ;
mysql> -- Query the materialized view-like table
mysql> SELECT * FROM materialized_view_table;
+----+-------------+------+-------------------------+
| id | name        | age  | email                   |
+----+-------------+------+-------------------------+
|  1 | John Doe    | 25   | john.doe@example.com    |
|  2 | Jane Smith  | 0    | jane.smith@example.com  |
|  3 | Bob Johnson | 30   | default@example.com     |
|  4 | Alice Brown | 22   | alice.brown@example.com |
|  5 | Chris Lee   | 28   | chris.lee@example.com   |
|  6 | Jane Smith  | 25   | jane.smith@example.com  |
+----+-------------+------+-------------------------+
6 rows in set (0.00 sec)

mysql>
```

Using OVER and PARTITION BY for Total Aggregation:

Example 1: Calculating the total amount for each category using OVER and PARTITION BY:

Command Prompt - mysql -u root -p

```
mysql> SELECT
    ->     product,
    ->     category,
    ->     amount,
    ->     SUM(amount) OVER (PARTITION BY category) AS total_category_amount
    -> FROM
    ->     sales;
+----------+-----------+--------+-----------------------+
| product  | category  | amount | total_category_amount |
+----------+-----------+--------+-----------------------+
| Product1 | Category1 | 100.00 |                430.00 |
| Product2 | Category1 | 150.00 |                430.00 |
| Product5 | Category1 | 180.00 |                430.00 |
| Product3 | Category2 | 120.00 |                320.00 |
| Product4 | Category2 | 200.00 |                320.00 |
+----------+-----------+--------+-----------------------+
5 rows in set (0.00 sec)
```

Example 2: Calculating the total amount for each product within its category:

```
mysql> SELECT
    ->     product,
    ->     category,
    ->     amount,
    ->     SUM(amount) OVER (PARTITION BY category) AS total_category_amount,
    ->     SUM(amount) OVER (PARTITION BY category, product) AS total_product_amount
    -> FROM
    ->     sales;
+----------+-----------+--------+-----------------------+----------------------+
| product  | category  | amount | total_category_amount | total_product_amount |
+----------+-----------+--------+-----------------------+----------------------+
| Product1 | Category1 | 100.00 |                430.00 |               100.00 |
| Product2 | Category1 | 150.00 |                430.00 |               150.00 |
| Product5 | Category1 | 180.00 |                430.00 |               180.00 |
| Product3 | Category2 | 120.00 |                320.00 |               120.00 |
| Product4 | Category2 | 200.00 |                320.00 |               200.00 |
+----------+-----------+--------+-----------------------+----------------------+
5 rows in set (0.00 sec)

mysql>
```