Joins, Union, Union All, Index, CTE.
Definition + Query + Problems + Solutions

JOIN:

INNER JOIN: Returns only the rows that have matching values in both tables.

LEFT JOIN (or LEFT OUTER JOIN): Returns all rows from the left table, and the matched rows from the right table. If no match is found, NULL values are returned for columns from the right table.

RIGHT JOIN (or RIGHT OUTER JOIN): Returns all rows from the right table, and the matched rows from the left table. If no match is found, NULL values are returned for columns from the left table.

CROSS JOIN: Returns the Cartesian product of the two tables. It combines each row of the first table with each row of the second table.

SELF JOIN: A regular join but the table is joined with itself.

Indexing:

Indexing in MySQL is a crucial concept for optimizing the performance of database queries. An index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional space and write time. By creating indexes, you can quickly locate data without having to search every row in a database table every time a database table is accessed.

UNION:

Definition: The `UNION` operator combines the result sets of two or more SELECT queries into a single result set, eliminating duplicate rows.
Result: It returns only unique rows from the combined result set.

Performance: It may perform slower than `UNION ALL` due to the need to remove duplicates.

Usage: Use `UNION` when you need a result set without duplicate rows.

Syntex:

```
SELECT column1, column2, …
 FROM table1
UNION
SELECT column1, column2, ...
FROM table2;
```

UNION ALL:

Definition: The `UNION ALL` operator combines the result sets of two or more SELECT queries into a single result set, including all duplicate rows.

Result: It returns all rows from the combined result set, including duplicates.

Performance: It usually performs faster than `UNION` because it does not need to remove duplicates.

Usage: Use `UNION ALL` when you need a result set with all rows, including duplicates.

Syntax:

```
SELECT column1, column2, ...
FROM table1
UNION ALL
SELECT column1, column2, …
 FROM table2;
```

Key Differences:

- **Duplicate Rows**:
  - `UNION`: Eliminates duplicate rows.
  - `UNION ALL`: Includes duplicate rows.

- **Performance**:
  - `UNION`: Slower due to the removal of duplicates.
  - `UNION ALL`: Faster because it doesn't remove duplicates.
- **Usage Context**:
  - `UNION`: Use when you need to ensure all rows in the result set are unique.
  - `UNION ALL`: Use when duplicates are acceptable or desired, and you want better performance.

Summary:

- **UNION**: Combines result sets, removing duplicates, and may be slower.
- **UNION ALL**: Combines result sets, keeping all duplicates, and is faster.

Choose the operator based on whether you need unique results (`UNION`) or if duplicates are acceptable (`UNION ALL`).

Common Table Expression (CTE)

Definition:

A Common Table Expression (CTE) is a temporary result set in SQL that you can reference within a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement. CTEs can simplify complex queries, improve readability, and are especially useful for recursive queries.

The basic syntax for a CTE is:

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

Benefits of Using CTEs:

1. **Readability**: CTEs make complex queries easier to read and understand by breaking them down into simpler, reusable parts.

1. **Modularity**: You can define a CTE once and reference it multiple times within the main query.
2. **Maintenance**: Queries are easier to maintain and debug when broken down into CTEs.
3. **Recursion**: CTEs support recursion, which is essential for querying hierarchical data.

Limitations:

- **Scope**: CTEs are only available within the execution of the single `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement where they are defined.
- **Performance**: While CTEs can simplify query writing, they may not always be the most performant option. It's important to test and compare performance with other methods, such as subqueries or temporary tables.

CTEs are a powerful tool in SQL that enhance query readability and manageability. They are especially useful for complex queries and recursive operations, providing a structured and modular way to build and understand SQL queries.

Dataset + problem statements + solutions:

```
create database operation;
use operation;

create table if not exists course (
course_id int,
course_name varchar(30),
course_desc varchar(60),
course_tag varchar(50));

create table if not exists student(
student_id int,
student_name varchar(60),
student_mobile int,
student_course_enroll varchar(30),
student_course_id int);
```

```
insert into course values(101, 'fsda','full stack data analytics','Analytics'),
(102, 'fsds','full stack data science','DS'),
(103, 'big data','full stack big data','BIG DATA'),
(104, 'mern','web dev','WEB DEV'),
(105, 'c','c language','C'),
(106, 'c++','c++ language','C++'),
(107, 'blockchain','full stack blockchain','BC'),
(108, 'java','full stack java','JAVA'),
(109, 'testing','full testing','TESTING'),
(110, 'cybersecurity','full stack data cybersecurity','CYBERSECURITY');

insert into student values(301,'shweta',234598790,'yes',101),
(302,'kamala',234598790,'yes',102),
(303,'rana',567398790,'yes',105),
(304,'rakesh',234598790,'yes',106),
(305,'shyam',222598790,'yes',101),
(306,'radha',777598790,'yes',103),
(307,'raunak',569598790,'yes',105),
(308,'kashish',033598790,'yes',107),
(309,'varma',989598790,'yes',103);
```

```
select * from student;
select * from course;

#Example of Inner Join.
select c.course_id, c.course_name, c.course_desc, s.student_id, s.student_name,s.student_course_id from
course c
inner join student s
on c.course_id = s.student_course_id;

#Example Left Join.
select c.course_id, c.course_name, c.course_desc, s.student_id, s.student_name,s.student_course_id from
course c
left join student s
on c.course_id = s.student_course_id;

#Example of right Join.
select c.course_id, c.course_name, c.course_desc, s.student_id, s.student_name,s.student_course_id from
course c
right join student s
on c.course_id = s.student_course_id;
```

#"Display the courses that have not been purchased by any student."

```
select * from course
left join student on course.course_id = student.student_course_id
where student.student_id is null;
```

#Use of cross Join without condition.

```
select c.course_id, c.course_name, c.course_desc, s.student_id, s.student_name,s.student_course_id from
course c
cross join student s;
```

#Use of cross Join with condition.

```
select c.course_id, c.course_name, c.course_desc, s.student_id, s.student_name,s.student_course_id from
course c
cross join student s
on c.course_id = s.student_course_id;
```

#Use of indexing.
#table for index.
#Single column indexing.

```
create table if not exists course1 (
course_id int,
course_name varchar(30),
course_desc varchar(60),
course_tag varchar(50),
index(course_id));

insert into course1 values(101, 'fsda','full stack data analytics','Analytics'),
(102, 'fsds','full stack data science','DS'),
(103, 'big data','full stack big data','BIG DATA'),
(104, 'mern','web dev','WEB DEV'),
(105, 'c','c language','C'),
(106, 'c++','c++ language','C++'),
(107, 'blockchain','full stack blockchain','BC'),
(108, 'java','full stack java','JAVA'),
(109, 'testing','full testing','TESTING'),
(110, 'cybersecurity','full stack data cybersecurity','CYBERSECURITY');
```

#Multiple column indexing.

```
create table if not exists course2 (
course_id int,
```

```
course_name varchar(30),
course_desc varchar(60),
course_tag varchar(50),
index(course_id,course_name));

insert into course2 values(101, 'fsda','full stack data analytics','Analytics'),
(102, 'fsds','full stack data science','DS'),
(103, 'big data','full stack big data','BIG DATA'),
(104, 'mern','web dev','WEB DEV'),
(105, 'c','c language','C'),
(106, 'c++','c++ language','C++'),
(107, 'blockchain','full stack blockchain','BC'),
(108, 'java','full stack java','JAVA'),
(109, 'testing','full testing','TESTING'),
(110, 'cybersecurity','full stack data cybersecurity','CYBERSECURITY');
```

#Union Operation.
#vertical join Operation.
 # It returns only unique rows from the combined result set.

```
select course_id, course_name from course
union
select student_id, student_name from student;

#Union all Operation.
#It returns all rows from the combined result set, including duplicates.

select course_id, course_name from course
union  all
select student_id, student_name from student;

#Common Table Expression (CTE)
with sample_students as (
select * from course where course_id in(101, 102, 106))
select * from sample_students where course_tag = 'c';


with outcome_cross as (
select c.course_id, c.course_name, c.course_desc, s.student_id, s.student_name,s.student_course_id from
course c
cross join student s )
select course_id, course_name, student_id from outcome_cross where student_id = 301;
```

#create column.

```
with ctetest as (select 1 as col1, 2 as col2
union all
select 3,4)
select col1, col2 from ctetest;
```

#Recursive CTE.

```
with recursive cte(n) as
(select 1 union all select n+1 from cte      where n<5)
select * from cte;
```

```
with recursive cte as
(select 1 as n, 1 as p, -1 as q
union all
select n+1, p+2, q+4 from cte where n<5)
select * from cte;
```

"We should definitely try this query."

```
# simple table create.
select 1 as n, 1 as p, -1 as q;

#simple table create.
select 1 as col1, 2 as col2
union all
select 3,4;

SELECT COLUMN_NAME
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME = 'course';
```

THANK YOU!