SQL NOTES

CRUD Operation using Python and SQL Queries: GitHub Repository

Create Database :

```
CREATE DATABASE IF NOT EXISTS sql_practice;
```

Create Table :

```
CREATE TABLE campusx (
    StudentID INT AUTO_INCREMENT PRIMARY KEY, -- Unique identifier for each student
    Name VARCHAR(100) NOT NULL,                -- Student's name (mandatory)
    Email VARCHAR(100) UNIQUE NOT NULL,        -- Student's email (must be unique and mandatory)
    Age INT,                                   -- Student's age
    Course VARCHAR(50),                        -- Course the student is enrolled in
    EnrollmentDate DATE                        -- Date of enrollment
);
```

Insert Values into the Table:

```
INSERT INTO campusx (Name, Email, Age, Course, EnrollmentDate) VALUES
('John Doe', 'john.doe@example.com', 20, 'Computer Science', '2025-01-01'),
('Jane Smith', 'jane.smith@example.com', 22, 'Biology', '2025-02-15'),
('Sam Brown', 'sam.brown@example.com', 21, 'Mathematics', '2025-03-10');
```

Empty the Table : (Both the keys and Values)

```
TRUNCATE TABLE campusx;
```

Delete Table :

```
DROP TABLE campusx;
```

Constrains:

1. **NOT NULL**

```
CREATE TABLE students (
    StudentID INT AUTO_INCREMENT PRIMARY KEY,
    Name VARCHAR(100) NOT NULL,  -- Name cannot be NULL
    Age INT
);
```

2. **UNIQUE**

```
CREATE TABLE students (
    StudentID INT AUTO_INCREMENT PRIMARY KEY,
    Email VARCHAR(100) UNIQUE,  -- Email must be unique
    Name VARCHAR(100)
);
```

3. **PRIMARY KEY**

```
CREATE TABLE students (
    StudentID INT AUTO_INCREMENT PRIMARY KEY,  -- Primary Key
    Name VARCHAR(100) NOT NULL,
    Email VARCHAR(100) UNIQUE
);
```

4. **FOREIGN KEY**

```
CREATE TABLE courses (
    CourseID INT AUTO_INCREMENT PRIMARY KEY,
    CourseName VARCHAR(100) NOT NULL
);
```

```
CREATE TABLE enrollments (
   EnrollmentID INT AUTO_INCREMENT PRIMARY KEY,
   StudentID INT,
   CourseID INT,
   FOREIGN KEY (StudentID) REFERENCES students(StudentID),  -- Foreign Key referencing students table
   FOREIGN KEY (CourseID) REFERENCES courses(CourseID)      -- Foreign Key referencing courses table
);
```

5. **CHECK**

```
CREATE TABLE students (
   StudentID INT AUTO_INCREMENT PRIMARY KEY,
   Name VARCHAR(100) NOT NULL,
   Age INT CHECK (Age >= 18)  -- Age must be 18 or older
);
```

6. **DEFAULT**

```
CREATE TABLE students (
   StudentID INT AUTO_INCREMENT PRIMARY KEY,
   Name VARCHAR(100) NOT NULL,
   EnrollmentDate DATE DEFAULT CURRENT_DATE  -- Default value is the current date
);
```

7. **AUTO_INCREMENT**

```
CREATE TABLE students (
   StudentID INT AUTO_INCREMENT PRIMARY KEY,  -- Auto-incrementing primary key
   Name VARCHAR(100) NOT NULL
);
```

Alter Table:

1. **Add Column:**

```
ALTER TABLE students
ADD COLUMN university VARCHAR(50) NOT NULL;
```

2. **Delete Column:**

```
ALTER TABLE students
DROP COLUMN university;
```

3. **Modify Column:**

```
ALTER TABLE students
ADD COLUMN address VARCHAR(50) NOT NULL;
-- Modify the 'address' column to change its data type to TEXT and allow NULL values
ALTER TABLE students
MODIFY COLUMN address TEXT;
```

4. **Add/Remove Constraint**

```
-- Add a unique constraint named 'Unique_StudentEmail' to the 'email' column
ALTER TABLE students
ADD CONSTRAINT Unique_StudentEmail UNIQUE (email);
-- Drop the unique constraint named 'Unique_StudentEmail'
ALTER TABLE students
DROP INDEX Unique_StudentEmail;
```

Retrieve Data from the Table:

1. **All columns:** SELECT * FROM students;

2. **Particular columns:** SELECT Name, Email FROM students;

3. **Aliasing Columns (Change Column Name):**

```
SELECT Name AS StudentName, Email AS StudentEmail FROM students;
```

4. **Aliasing Table (Change Table Name):**
   SELECT s.Name, s.Email FROM students AS s;
   Practical Use in Joins:
   SELECT st.Name, cr.CourseName FROM students AS st
   JOIN enrollments AS en ON st.StudentID = en.StudentID
   JOIN courses AS cr ON en.CourseID = cr.CourseID;

5. **Expressions (Retrieve specific columns with expressions, using aliasing, and including logical and arithmetic expressions):**
   SELECT
   StudentID,
   Name AS StudentName,
   Email AS StudentEmail,
   Major AS StudyField,
   Address,
   university AS University,
   CONCAT(Name, ' (', Major, ')') AS StudentInfo, **-- String expression to concatenate Name and Major**
   LENGTH(Name) AS NameLength, **-- String expression to get the length of the Name**
   Age + 1 AS AgeNextYear, **-- Arithmetic expression to calculate age next year**
   CASE
   WHEN Age >= 18 THEN 'Adult'
   ELSE 'Minor'
   END AS AgeGroup, **-- Logical expression to determine age group**
   YEAR(EnrollmentDate) AS EnrollmentYear **-- Date expression to extract the year from EnrollmentDate**
   FROM students
   WHERE Major = 'Software Engineering' AND Age IS NOT NULL; **-- Logical expression to filter by major and ensure age is not null**

6. **Constant:**
   SELECT
   Name AS StudentName,
   'Active' AS Status, **-- String constant**
   100 AS FixedNumber, **-- Numeric constant**
   '2025-01-01' AS StartDate, **-- Date constant**
   Age + 1 AS AgeNextYear **-- Arithmetic expression**
   FROM students
   WHERE Major = 'Software Engineering';

7. **Distinct Column:**
   SELECT DISTINCT Major FROM students;
   SELECT DISTINCT Name, Email, Major FROM students; **– Multiple Columns**

8. **Comparison Operator:**
   SELECT * FROM students WHERE Major = 'Computer Science'; **-- Equal to**
   SELECT * FROM students WHERE Major <> 'Computer Science'; **-- Not equal to**
   SELECT * FROM students WHERE Age > 20; **-- Greater than**
   SELECT * FROM students WHERE Age < 20; **-- Less than**
   SELECT * FROM students WHERE Age >= 20; **-- Greater than or equal to**
   SELECT * FROM students WHERE Age <= 20; **-- Less than or equal to**

9. **AND, OR, BETWEEN:**
   SELECT * FROM students WHERE
   Major = 'Computer Science' AND (Age BETWEEN 18 AND 25) OR Major = 'Software Engineering';

10. **IN, NOT IN:**
    SELECT * FROM students
    WHERE Major IN ('Computer Science', 'Software Engineering'); **-- IN operator**
    SELECT * FROM students
    WHERE Major NOT IN ('Biology', 'Mathematics'); **-- NOT IN operator**

11. **Wildcards with LIKE:**

-- Using Percent (%) Wildcard
SELECT * FROM students WHERE Name LIKE 'J%'; **-- Finds names starting with 'J'**
SELECT * FROM students WHERE Email LIKE '%@example.com'; **-- Finds emails ending with '@example.com'**
SELECT * FROM students WHERE Name LIKE '%a%'; **-- Finds names containing the letter 'a'**

-- Using Underscore (_) Wildcard
SELECT * FROM students WHERE Name LIKE 'J_n'; **-- Finds names like 'Jon' or 'Jan'**
SELECT * FROM students WHERE Name LIKE '_a__'; **-- Finds names with 'a' as the second character, such as 'Mary'**

-- Update Example
UPDATE students SET Address = 'Unknown' WHERE Address LIKE '__%'; **-- Update addresses with at least two characters to 'Unknown'**
UPDATE students SET Email = 'updated@example.com' WHERE Email LIKE '_%example.com'; **-- Update emails that contain 'example.com'**
UPDATE students SET Major = 'Engineering' WHERE Major LIKE '%Software%'; **-- Update 'Software' majors to 'Engineering'**

-- Combined Wildcards
SELECT * FROM students WHERE Name LIKE 'J%n'; **-- Finds names starting with 'J' and ending with 'n'**
SELECT * FROM students WHERE Name LIKE '_a_%'; **-- Finds names with 'a' as the second character and at least three characters long**

Order of Query Execution:

FROM→ JOIN →WHERE→GROUP BY→HAVING→SELECT→DISTINCT→ORDER BY→LIMIT

Functions:

-- ABS Function
SELECT ABS(-10) AS AbsoluteValue; **-- Result: 10**

-- ROUND Function
SELECT ROUND(123.456, 2) AS RoundedValue; **-- Result: 123.46**

-- CEIL Function
SELECT CEIL(123.456) AS CeilingValue; **-- Result: 124**

-- FLOOR Function
SELECT FLOOR(123.456) AS FloorValue; **-- Result: 123**

-- UPPER and LOWER Functions
SELECT UPPER('hello world') AS UpperCaseValue; **-- Result: HELLO WORLD**
SELECT LOWER('HELLO WORLD') AS LowerCaseValue; **-- Result: hello world**

-- CONCAT Function
SELECT CONCAT('Hello', ' ', 'World') AS ConcatenatedString; **-- Result: Hello World**

-- LENGTH Function
SELECT LENGTH('Hello World') AS StringLength; **-- Result: 11**

-- SUBSTR Function (String, Start, Length)
SELECT SUBSTR('Hello World', 1, 5) AS Substring; **-- Result: Hello**

Aggregate Functions:

-- MIN and MAX
SELECT MIN(Age) AS MinimumAge, MAX(Age) AS MaximumAge FROM students;

-- SUM
SELECT SUM(Score) AS TotalScore FROM grades;

-- AVG
SELECT AVG(Age) AS AverageAge FROM students;

-- COUNT
SELECT COUNT(*) AS TotalStudents FROM students;

-- COUNT WITH DISTINCT
SELECT COUNT(DISTINCT Major) AS UniqueMajors FROM students;

-- COUNT WITH CONDITION
SELECT COUNT(*) AS StudentsInCS FROM students WHERE Major = 'Computer Science';

Sorting Data:

-- Ascending Order (Default)
SELECT * FROM students
ORDER BY Age ASC; **-- Sorts by age in ascending order**

-- Descending Order
SELECT * FROM students
ORDER BY Age DESC; **-- Sorts by age in descending order**

-- Multiple Columns
SELECT * FROM students
ORDER BY Major ASC, Age DESC; **-- Sorts by major in ascending order and by age in descending order within each major**

-- Sorting with LIMIT
SELECT * FROM students
ORDER BY Age DESC
LIMIT 5; **-- Limits the result to the top 5 oldest students**

– Combined Query: Concise Complex Sorting Query
SELECT
StudentID, Name, Email, Major, Age, GPA, EnrollmentDate
FROM students
ORDER BY
Major ASC, **-- Sorts by major in ascending order**
GPA DESC, **-- Then sorts by GPA in descending order**
Age DESC **-- Finally, sorts by age in descending order**
LIMIT 5; **-- Limits the result to the top 5 records based on the sorting criteria**

Grouping Data:

-- Basic Grouping
SELECT Major, COUNT(*) AS NumberOfStudents
FROM students
GROUP BY Major;

-- Grouping with Aggregate Functions
SELECT Major, AVG(GPA) AS AverageGPA
FROM students
GROUP BY Major;

-- Grouping with Multiple Columns
SELECT Major, YEAR(EnrollmentDate) AS EnrollmentYear, COUNT(*) AS NumberOfStudents
FROM students
GROUP BY Major, YEAR(EnrollmentDate);

-- Grouping with HAVING Clause
SELECT Major, AVG(GPA) AS AverageGPA
FROM students
GROUP BY Major
HAVING AVG(GPA) > 3.0;

– Combining Grouping, Sorting, and Limiting

```
SELECT
   Major, AVG(GPA) AS AverageGPA, COUNT(*) AS NumberOfStudents
FROM students
GROUP BY
   Major
HAVING
   AVG(GPA) > 3.0
ORDER BY
   AverageGPA DESC,  NumberOfStudents ASC LIMIT 5;
```

Having Data(Filtering):

-- Grouping by Major with HAVING Clause

```
SELECT
   Major, AVG(GPA) AS AverageGPA, COUNT(*) AS NumberOfStudents
FROM students
GROUP BY
   Major
HAVING
   AVG(GPA) > 3.0  -- Filters groups to include only those with an average GPA greater than 3.0
ORDER BY
   AverageGPA DESC,  -- Sorts the result set by average GPA in descending order
   NumberOfStudents ASC;  -- Then sorts by the number of students in ascending order
```

CASE(Conditional Statement):

--Basic Usage: Basic CASE Statement to Categorize GPA

```
SELECT
   StudentID, Name, GPA,
   CASE
      WHEN GPA >= 3.5 THEN 'High'
      WHEN GPA >= 2.0 THEN 'Medium'
      ELSE 'Low'
   END AS GPA_Category
FROM students;
```

-- Combining CASE with Aggregation: Using CASE with Aggregate Functions to Count Students by GPA Category

```
SELECT
   CASE
      WHEN GPA >= 3.5 THEN 'High'
      WHEN GPA >= 2.0 THEN 'Medium'
      ELSE 'Low'
   END AS GPA_Category,
   COUNT(*) AS NumberOfStudents
FROM students
GROUP BY
   CASE
      WHEN GPA >= 3.5 THEN 'High'
      WHEN GPA >= 2.0 THEN 'Medium'
      ELSE 'Low'
   END;
```

--Multiple Conditions: Using CASE Statement with Multiple Conditions

```
SELECT
   StudentID, Name, Age, Major,
   CASE
      WHEN Major = 'Computer Science' AND Age >= 20 THEN 'CS Senior'
      WHEN Major = 'Computer Science' AND Age < 20 THEN 'CS Junior'
      WHEN Major = 'Software Engineering' AND Age >= 20 THEN 'SE Senior'
      WHEN Major = 'Software Engineering' AND Age < 20 THEN 'SE Junior'
      ELSE 'Other'
   END AS Classification
FROM  students;
```

JOIN Tables:

1) **Cartesian Product (Cross Join)**: A Cartesian product is the result of combining all rows from two or more tables without any condition. It returns the product of the number of rows in the tables.

   SELECT * FROM table1, table2;

2) **Union:** The UNION operator is used to combine the result sets of two or more SELECT statements. Each SELECT statement within the UNION **must have the same number of columns in the result sets with similar data types.**

   SELECT column_name(s) FROM table1
   UNION
   SELECT column_name(s) FROM table2;

3) **UNION ALL:** This combines the result sets of the two SELECT statements, including all duplicate rows.

   SELECT column_name1, column_name2
   FROM table1
   UNION ALL
   SELECT column_name1, column_name2
   FROM table2;

4) **Inner Join(Default Join)**: An inner join returns rows when there is a match in both tables.

   SELECT * FROM table1
   INNER JOIN table2 ON table1.common_column = table2.common_column;

5) **Self Join**: A self join is a regular join, but the table is joined with itself.

   SELECT A.column1, B.column2 FROM table1 A, table1 B
   WHERE A.common_column = B.common_column;

6) **Outer Join**: Outer joins return rows even when there are no matches in one of the tables. There are three types: left outer, right outer, and full outer.

   - ➔ **Left Outer Join**: Returns all rows from the left table and matched rows from the right table. If no match is found, NULLs are returned for columns from the right table.

     SELECT
       table1.*, table2.*
     FROM
       table1
     LEFT JOIN
       table2 ON table1.common_column = table2.common_column

   - ➔ **Right Outer Join**: Returns all rows from the right table and matched rows from the left table. If no match is found, NULLs are returned for columns from the left table.

     SELECT table1.*, table2.*
     FROM
       table1
     RIGHT JOIN
       table2 ON table1.common_column = table2.common_column

   - ➔ **Full Outer Join**: In SQL, a FULL OUTER JOIN returns all rows when there is a match in one of the tables. If there is no match, the result is NULL on the side that does not have a match. However, not all SQL databases support the FULL OUTER JOIN syntax directly. In those cases, you can achieve the same result using UNION to combine LEFT JOIN and RIGHT JOIN.

     SELECT table1.*, table2.*
     FROM
       table1
     LEFT JOIN
       table2 ON table1.common_column = table2.common_column

```
UNION
SELECT
   table1.*, table2.*
FROM
   table1
RIGHT JOIN
   table2 ON table1.common_column = table2.common_column;
```

7) **Subquery**: A subquery is a query nested inside another query. There are independent and dependent subqueries.

8) **Independent (Uncorrelated) Subquery:** An independent or uncorrelated subquery can be executed independently of the outer query. It is self-contained and does not rely on columns from the outer query.

```
SELECT Name, GPA FROM students
WHERE GPA = (SELECT MAX(GPA) FROM students);
```

9) **Correlated Subquery:** A correlated subquery depends on the outer query for its values. It is evaluated once for each row processed by the outer query.

```
SELECT Name, Major, GPA FROM students s1
WHERE GPA > (SELECT AVG(GPA) FROM students s2 WHERE s1.Major = s2.Major);
```