Day 31 – SQL Practical

Structured Query Language (SQL) is the standard language used to communicate with relational databases.

In this practical, I covered the essential SQL commands for creating databases, tables, inserting, updating, deleting data, filtering results, sorting, using aggregate functions, and performing joins between tables.

---

1. Create a New Database

```
CREATE DATABASE newdb;
SHOW DATABASES;
```

---

2. Use the Database

```
USE newdb;
```

- `CREATE DATABASE` – Creates a new database.
- `USE` – Switches the current working database.
- `SHOW DATABASES` – Lists all databases.

Why Important?

Before creating tables or inserting data, you must select the correct database to avoid working in the wrong one.

---

3. Create a Table: `student`

- `CREATE TABLE` – Defines the table's structure.
- Constraints used:
  - PRIMARY KEY: Ensures uniqueness and prevents duplicate values.
  - NOT NULL: Makes sure a column cannot have NULL values.

Create a student table with the following columns:

- `name` – Student's name
- `id` – Student ID (NOT NULL, PRIMARY KEY)
- `address` – Address of student
- `marks` – Marks obtained

```
CREATE TABLE student (
    name VARCHAR(30),
    id INT NOT NULL PRIMARY KEY,
    address VARCHAR(50),
    marks INT
);
```

To view all tables:

```
SHOW TABLES;
```

To describe the table:

```
DESC student;
```

---

4. Insert Data into the Table

- **Two ways:** Single row insertion and multiple rows in one query.
- Using multiple inserts reduces execution time.

  Important Tip: Always match the number of values with the number of columns.

We can insert values in two ways – secure way & insecure way.

Example: Insert single row

```
INSERT INTO student VALUES('Jake', 15, 'Hyd', 70);
```

Insert multiple rows

```
INSERT INTO student
VALUES
('Alex', 18, 'Delhi', 75),
('Raj', 20, 'Mum', 79),
('Phil', 25, 'Hyd', 60),
('Rose', 30, 'Delhi', 65);
```

---

5. View Data from the Table

To view all records:

`SELECT *` – Fetches all columns.

```
SELECT * FROM student;
```

To select specific column:

Selecting specific columns improves performance when you don't need all data.

```
SELECT name FROM student;
```

To select multiple specific columns:

```
SELECT name, id FROM student;
```

---

6. Filtering Data with WHERE

To fetch a record with a specific condition:

Filtering using `WHERE` allows fetching targeted records.

```
SELECT * FROM student WHERE id = 15;
```

---

7. Primary Key Constraint

Since `id` is a PRIMARY KEY, duplicate values are not allowed:

```
INSERT INTO student VALUES('Sam', 15, 'Hyd', 55);
-- This will throw an error due to duplicate ID.
```

8. Update Records

To update the address for a specific ID:

- `UPDATE` modifies existing data.
- Always use `WHERE` to avoid updating all rows unintentionally.

```
UPDATE student SET address='Chennai' WHERE id=25;
```

9. Alter Table – Add Column

- `ALTER TABLE` is used to:
- Add a new column (`ADD`).
- Modify column datatype (`MODIFY`).
- Delete a column (`DROP`).

  Why Important? Schema changes happen when requirements change.

Add a new column `phoneNo` :

```
ALTER TABLE student ADD phoneNo INT;
```

Set the same phone number for all:

```
UPDATE student SET phoneNo=123;
```

Update a specific phone number:

```
UPDATE student SET phoneNo=456 WHERE id=25;
```

10. Modify Column Data Type

```
ALTER TABLE student MODIFY COLUMN name VARCHAR(60);
```

11. Drop a Column

```
ALTER TABLE student DROP COLUMN phoneNo;
```

12. Delete a Row

```
DELETE FROM student WHERE name='Alex';
```

13. Wildcard Usage in WHERE

- `%` → Matches any number of characters.
- `_` → Matches a single character.

  Example: `'r%'` → Names starting with 'r' `'%e'` → Names ending with 'e'

```
SELECT * FROM student WHERE name LIKE 'r%';   -- Starts with 'r'
SELECT * FROM student WHERE name LIKE 'c%';   -- Starts with 'c'
SELECT * FROM student WHERE name LIKE '%e';   -- Ends with 'e'
SELECT * FROM student WHERE name LIKE '_a%';  -- Second letter is 'a'
SELECT * FROM student WHERE name LIKE '%i_';  -- Second last letter is 'i'
```

---

14. Sorting Data

- `ORDER BY` allows sorting by ascending (ASC) or descending (DESC) order.
- Helps in ranking results or organizing reports.

Ascending order:

```
SELECT * FROM student ORDER BY marks;
```

Descending order:

```
SELECT * FROM student ORDER BY marks DESC;
```

---

15. DISTINCT Keyword

Retrieve unique values from a column.

```
-- Unique addresses from student table
SELECT DISTINCT address FROM student;
```

---

16. BETWEEN Operator

Select values within a range (inclusive).

```
-- Students with marks between 65 and 80
SELECT * FROM student
WHERE marks BETWEEN 65 AND 80;
```

---

17. IN Operator

Select rows that match any value in a list.

```
-- Students from specific cities
SELECT * FROM student
WHERE address IN ('Delhi', 'Hyd');
```

---

18. NOT IN Operator

Exclude rows that match any value in a list.

```
-- Students not from Delhi or Hyd
SELECT * FROM student
WHERE address NOT IN ('Delhi', 'Hyd');
```

---

19. LIMIT Clause

Limit the number of rows returned.

```
-- Top 3 students by marks
SELECT * FROM student
ORDER BY marks DESC
LIMIT 3;
```

---

20. SQL Aggregate Functions

- SUM() → Total of numeric column.
- AVG() → Average value.
- COUNT() → Number of records.
- MAX() → Highest value.
- MIN() → Lowest value.

  These are useful for reporting and analysis.

```
SELECT SUM(marks) FROM student;    -- Total marks
SELECT AVG(marks) FROM student;    -- Average marks
SELECT COUNT(name) FROM student;   -- Count of students
SELECT MAX(marks) FROM student;    -- Highest marks
SELECT MIN(marks) FROM student;    -- Lowest marks
```

---

21. GROUP BY with Aggregate Functions

Group data based on a column and apply aggregate functions to each group.

```
-- Total marks by address
SELECT address, SUM(marks) AS total_marks
FROM student
GROUP BY address;

-- Average marks by address
SELECT address, AVG(marks) AS avg_marks
FROM student
GROUP BY address;
```

---

22. HAVING Clause

Filter results after aggregation.

Unlike WHERE, which filters rows before aggregation, HAVING works with grouped data.

```
-- Addresses with average marks greater than 70
SELECT address, AVG(marks) AS avg_marks
FROM student
GROUP BY address
HAVING AVG(marks) > 70;
```

---

23. Create Another Table for Joins

```
CREATE TABLE emp (
    id INT NOT NULL PRIMARY KEY,
    salary INT,
    empcode INT,
    name VARCHAR(30)
);

INSERT INTO emp VALUES
(10, 20000, 102, 'aman'),
(23, 60000, 104, 'arup'),
(30, 30000, 105, 'max'),
(40, 25000, 103, 'ram'),
(35, 90000, 106, 'sam');
```

---

24. SQL Joins

Joins are used to combine data from two or more tables:

- **Inner Join** → Returns rows where matches exist in both tables.

```
SELECT * FROM student
INNER JOIN emp
ON student.id = emp.id;
```

- **Left Join** → Returns all rows from left table + matched rows from right table.

```
SELECT * FROM student
LEFT JOIN emp
ON student.id = emp.id;
```

- **Right Join** → Returns all rows from right table + matched rows from left table.

```
SELECT * FROM student
RIGHT JOIN emp
ON student.id = emp.id;
```

- **Cross Join** → Returns all possible combinations of both tables.

```
SELECT * FROM student
CROSS JOIN emp;
```

Summary

In this practical, I learned how to:

- Create and manage databases
- Insert, update, and delete data
- Filter and sort results
- Use aggregate functions for analysis
- Join multiple tables to combine data These are the fundamental building blocks of SQL for data analysis and application development.