Great! Let's start SQL from scratch 🚀

---

## 🔥 1. What is SQL?

SQL (**Structured Query Language**) is used to manage and query databases. It allows you to **store, retrieve, update, and delete data** from relational databases like **MySQL, PostgreSQL, and SQLite**.

---

### 🔥 2. Basic SQL Commands

SQL has different types of commands:

| Command Type | Examples | Purpose |
|---|---|---|
| **DDL (Data Definition Language)** | CREATE, ALTER, DROP | Defines database structure |
| **DML (Data Manipulation Language)** | INSERT, UPDATE, DELETE | Manipulates data |
| **DQL (Data Query Language)** | SELECT | Retrieves data |
| **DCL (Data Control Language)** | GRANT, REVOKE | Controls access |
| **TCL (Transaction Control Language)** | COMMIT, ROLLBACK | Manages transactions |

---

### 🔥 3. Creating a Database & Table

```
CREATE DATABASE SchoolDB;  -- Create a database

USE SchoolDB;  -- Select the database


CREATE TABLE Students (

   id INT PRIMARY KEY AUTO_INCREMENT,

   name VARCHAR(50),

   age INT,

   grade VARCHAR(10)

);
```

✅ **Creates a table named Students with 4 columns**

---

### 🔥 4. Inserting Data

```
INSERT INTO Students (name, age, grade) VALUES

("Ali", 18, "A"),

("Ayesha", 19, "B"),

("Hassan", 20, "A");
```

✅ **Adds 3 students into the Students table**

---

### 🔥 5. Retrieving Data (SELECT)

```
SELECT * FROM Students;  -- Show all records


SELECT name, age FROM Students WHERE grade = "A";  -- Filter by grade
```

✅ **Fetches all students or only those with grade "A"**

---

### 🔥 6. Updating Data

```
UPDATE Students SET grade = "A+" WHERE name = "Ali";
```

✅ **Changes Ali's grade to A+**

---

### 🔥 7. Deleting Data

```
DELETE FROM Students WHERE age > 19;
```

✅ **Deletes students older than 19**

---

### 🔥 8. Filtering & Sorting Data

```
SELECT * FROM Students WHERE age >= 18 ORDER BY age DESC;
```

✅ **Fetches students aged 18+ in descending order**

---

### 🔥 9. Joins (Combining Data from Multiple Tables)

```
CREATE TABLE Courses (
    course_id INT PRIMARY KEY AUTO_INCREMENT,
    course_name VARCHAR(50)
);


CREATE TABLE Enrollments (
    student_id INT,
    course_id INT,
    FOREIGN KEY (student_id) REFERENCES Students(id),
    FOREIGN KEY (course_id) REFERENCES Courses(course_id)
);


SELECT Students.name, Courses.course_name
FROM Students
```

JOIN Enrollments ON Students.id = Enrollments.student_id

JOIN Courses ON Enrollments.course_id = Courses.course_id;

✅ **Combines students with their enrolled courses**

## 🔥 10. Advanced Joins

Joins allow combining data from multiple tables.

✅ **INNER JOIN (Only Matching Data)**

SELECT Students.name, Courses.course_name

FROM Students

INNER JOIN Enrollments ON Students.id = Enrollments.student_id

INNER JOIN Courses ON Enrollments.course_id = Courses.course_id;

✅ **Returns only students who are enrolled in courses**

✅ **LEFT JOIN (All from Left, Matching from Right)**

SELECT Students.name, Courses.course_name

FROM Students

LEFT JOIN Enrollments ON Students.id = Enrollments.student_id

LEFT JOIN Courses ON Enrollments.course_id = Courses.course_id;

✅ **Returns all students, even those not enrolled**

✅ **RIGHT JOIN (All from Right, Matching from Left)**

SELECT Students.name, Courses.course_name

FROM Students

RIGHT JOIN Enrollments ON Students.id = Enrollments.student_id

RIGHT JOIN Courses ON Enrollments.course_id = Courses.course_id;

✅ **Returns all courses, even if no students are enrolled**

---

## 🔥 11. Aggregation & Grouping

✅ **Using COUNT, SUM, AVG, MAX, MIN**

SELECT grade, COUNT(*) AS total_students

FROM Students

GROUP BY grade;

✅ **Counts students in each grade**

✅ **Filtering Groups using HAVING**

SELECT grade, COUNT(*) AS total_students

FROM Students

```
GROUP BY grade
HAVING COUNT(*) > 1;
```

✅ **Shows grades where more than 1 student exists**

---

## 🔥 12. Subqueries (Nested Queries)

✅ **Finding the Youngest Student**

```
SELECT * FROM Students
WHERE age = (SELECT MIN(age) FROM Students);
```

✅ **Fetches the youngest student**

---

## 🔥 13. Indexing (Improving Query Speed)

```
CREATE INDEX idx_student_name ON Students(name);
```

✅ **Speeds up searches on the name column**

---

## 🔥 14. Stored Procedures (Reusable SQL Code)

```
DELIMITER //
CREATE PROCEDURE GetStudentsByGrade(IN gradeValue VARCHAR(10))
BEGIN
  SELECT * FROM Students WHERE grade = gradeValue;
END //
DELIMITER ;
```

✅ **Creates a procedure to fetch students by grade**

---

## 🔥 15. Triggers (Automatic Actions)

```
CREATE TRIGGER BeforeInsertStudent
BEFORE INSERT ON Students
FOR EACH ROW
SET NEW.name = UPPER(NEW.name);
```

✅ **Automatically converts names to uppercase before inserting**

## 🔥 16. Transactions (Ensuring Data Integrity)

A **transaction** is a group of SQL operations that must all succeed or be rolled back if any fail.

✅ **Using Transactions (COMMIT, ROLLBACK)**

```
START TRANSACTION;
```

INSERT INTO Students (name, age, grade) VALUES ("Zain", 21, "B");

UPDATE Students SET grade = "A" WHERE name = "Hassan";

COMMIT;  -- Saves the changes

✅ **Ensures both queries run successfully before saving changes**

✅ **Rolling Back Changes (ROLLBACK)**

START TRANSACTION;

DELETE FROM Students WHERE age > 22;

ROLLBACK;  -- Cancels the deletion if something goes wrong

✅ **Prevents accidental data loss**

---

## 🔥 17. SQL Optimization Techniques

Improving query performance is crucial for large datasets.

✅ **1. Using Indexes for Faster Searches**

CREATE INDEX idx_student_name ON Students(name);

✅ **Speeds up searches on the name column**

---

✅ **2. Avoid SELECT *, Specify Columns**

❌ **Bad Practice**

SELECT * FROM Students;

✅ **Better Practice**

SELECT name, age FROM Students;

✅ **Faster because it fetches only needed columns**

---

✅ **3. Using EXPLAIN to Analyze Queries**

EXPLAIN SELECT * FROM Students WHERE age > 18;

✅ **Shows how MySQL processes the query (helps in optimization)**

---

✅ **4. Using LIMIT to Reduce Load**

SELECT * FROM Students LIMIT 10;

✅ **Fetches only 10 records, reducing database load**

---

### ✅ 5. Normalization (Avoiding Redundant Data)

❌ **Bad Practice** (All data in one table)

```
CREATE TABLE Employees (
   emp_id INT PRIMARY KEY,
   name VARCHAR(50),
   department VARCHAR(50),
   department_location VARCHAR(50)
);
```

✅ **Better Practice (Separate Tables & Use Foreign Keys)**

```
CREATE TABLE Departments (
   dept_id INT PRIMARY KEY,
   name VARCHAR(50),
   location VARCHAR(50)
);

CREATE TABLE Employees (
   emp_id INT PRIMARY KEY,
   name VARCHAR(50),
   dept_id INT,
   FOREIGN KEY (dept_id) REFERENCES Departments(dept_id)
);
```

✅ **Reduces redundancy & improves efficiency**

---

### 🔥 18. Views (Saved Queries for Faster Access)

```
CREATE VIEW TopStudents AS
SELECT name, grade FROM Students WHERE grade = "A";
```

✅ **Creates a virtual table to quickly access top students**

---

### 🔥 19. Stored Procedures for Efficiency

Instead of writing the same query repeatedly, use **Stored Procedures**.

```
DELIMITER //
CREATE PROCEDURE GetStudentsByGrade(IN gradeValue VARCHAR(10))
```

```
BEGIN
    SELECT * FROM Students WHERE grade = gradeValue;
END //
DELIMITER ;
```

✅ **Reusable function to fetch students by grade**