Master SQL from Basics to Advanced Using Admission Database

This documentation provides a complete, end-to-end learning journey for mastering **SQL concepts** using a real-world **Admission Management Database**.

It helps you to understand how SQL works in real-world scenarios involving **students, courses, departments, and admissions**, from database creation to advanced analytics.

Database & Table

Step 1: Create Database

```
CREATE DATABASE IF NOT EXISTS admission_db;
-- Optional: Drop if you want to reset
-- DROP DATABASE admission_db;
```

Step 2: Select Database

```
USE admission_db;
```

Step 3: Create Departments Table

```
CREATE TABLE IF NOT EXISTS departments (
    department_id INT PRIMARY KEY AUTO_INCREMENT,
    department_name VARCHAR(100) NOT NULL UNIQUE
);
```

Step 4: Insert Data into Departments

```
INSERT INTO departments (department_name) VALUES
('Computer Science'),
('Information Technology'),
('Electronics'),
('Mechanical'),
('Civil');
```

Step 5: Create Students Table with Constraints

```
CREATE TABLE students (
    student_id INT PRIMARY KEY AUTO_INCREMENT,
    full_name VARCHAR(100) NOT NULL,
    gender VARCHAR(10) NOT NULL CHECK (gender IN
('Male', 'Female')),
    dob DATE NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,
    phone_number VARCHAR(15) NOT NULL UNIQUE,
    percentage DECIMAL(5,2) NOT NULL CHECK (percentage
BETWEEN 0 AND 100),
    total_fees DECIMAL(10,2) NOT NULL CHECK (total_fees
>= 0),
    fees_paid DECIMAL(10,2) NOT NULL CHECK (fees_paid
>= 0),
    department_id INT NOT NULL,
    FOREIGN KEY (department_id) REFERENCES
departments(department_id)
);
```

Step 6: Insert Data into Students

```
INSERT INTO students
(full_name, gender, dob, email, phone_number,
percentage, total_fees, fees_paid,
department_id)
VALUES
('Deepraj Patil', 'Male', '1998-09-07',
'deepraj@gmail.com', '7689065437', 90.78,
100000.00, 95000.00, 1),
('Vivek Patil', 'Male', '2019-09-09',
```

```
'vivek@gmail.com', '8756789534', 90.67,
90000.00, 85000.00, 2),
('Rahul Sunchu', 'Male', '2018-06-04',
'rahul@gmail.com', '7856789504', 98.67,
95000.00, 95000.00, 1),
('Shivraj Patil', 'Male', '2017-05-01',
'shivraj@gmail.com', '9076789534', 80.67,
88000.00, 80000.00, 3),
('Shruthi Deshmukh', 'Female', '2017-03-01',
'shruthi@gmail.com', '7864678950', 98.67,
87000.00, 87000.00, 2),
('Vaishnavi Kulkarni', 'Female', '2019-06-01',
'vaishnavi@gmail.com', '9090789534', 80.67,
88000.00, 85000.00, 4),
('Ankit Sharma', 'Male', '1997-12-12',
'ankit@gmail.com', '9876543210', 75.45,
85000.00, 80000.00, 1),
('Pooja Nair', 'Female', '1999-07-23',
'pooja@gmail.com', '9123456789', 92.12,
92000.00, 92000.00, 2),
('Karan Mehta', 'Male', '2000-01-15',
'karan@gmail.com', '9345678123', 68.32,
78000.00, 70000.00, 3),
('Neha Singh', 'Female', '1998-05-10',
'neha@gmail.com', '9456123789', 88.56, 89000.00,
87000.00, 4),
('Rohan Joshi', 'Male', '2001-09-30',
```

```
'rohan@gmail.com', '9988776655', 95.24,
97000.00, 95000.00, 2),
('Sneha Patil', 'Female', '2002-03-17',
'sneha@gmail.com', '9786543210', 78.45,
80000.00, 78000.00, 1),
('Amit Verma', 'Male', '1996-11-05',
'amit@gmail.com', '9654321876', 85.67, 86000.00,
85000.00, 3),
('Kavya Rao', 'Female', '1997-04-08',
'kavya@gmail.com', '9543218765', 91.33,
91000.00, 90000.00, 4),
('Manish Gupta', 'Male', '1995-10-22',
'manish@gmail.com', '9123987654', 73.25,
75000.00, 70000.00, 2),
('Ritu Sharma', 'Female', '2000-02-14',
'ritu@gmail.com', '9345612789', 97.42, 98000.00,
97000.00, 1),
('Aditya Desai', 'Male', '2001-07-09',
'aditya@gmail.com', '9567812345', 82.64,
84000.00, 80000.00, 3),
('Meera Iyer', 'Female', '1999-11-29',
'meera@gmail.com', '9234567810', 89.78,
90000.00, 88000.00, 4),
('Saurabh Yadav', 'Male', '1998-08-16',
'saurabh@gmail.com', '9678123456', 66.54,
82000.00, 75000.00, 2),
('Priya Kulkarni', 'Female', '1997-06-21',
```

```
'priya@gmail.com', '9345678901', 93.11,
94000.00, 94000.00, 1);
```

Step 7: Check Tables

```
SHOW TABLES;
```

Step 8: Verify Data

```
SELECT * FROM departments;

SELECT * FROM students;
```

1. SELECT & FROM (Basics of Data Retrieval)

- Concept: `SELECT` is used to fetch columns, `FROM` specifies table.

Q1. Show all student records.

```
SELECT * FROM students;
```

Q2. Show only the names and percentages of all students.

```
SELECT full_name, percentage FROM students;
```

Q3. Show only genders of all students (with duplicates).

```
SELECT gender FROM students;
```

Q4. Remove duplicate genders.

```
SELECT DISTINCT gender FROM students;
```

2. WHERE Clause (Filtering Data)

- Concept: `WHERE` filters rows based on conditions.

Q1. Show all male students.

```
SELECT * FROM students WHERE gender = 'Male';
```

Q2. Show all students who belong to the "Civil" department.

```
SELECT * FROM students
WHERE department_id = (SELECT department_id FROM departments WHERE
department_name = 'Civil');
```

Q3. Show all students whose names start with "S".

```
SELECT * FROM students WHERE full_name LIKE 'S%';
```

Q4. Show all students whose names contain the letter "a".

```
SELECT * FROM students WHERE full_name LIKE '%a%';
```

Q5. Show students with a percentage between 80 and 90.

```
SELECT * FROM students WHERE percentage BETWEEN 80 AND 90;
```

Q6. Show department_id, gender, year of birth, and fees_paid of students with percentage > 70.

```
SELECT department_id, gender, YEAR(dob) AS birth_year, fees_paid
FROM students
WHERE percentage > 70;
```

3. GROUP BY + Aggregations

- Concept: `GROUP BY` groups rows; used with aggregate functions.

Q1. Show how many students are in each department.

```
SELECT department_id, COUNT(*) AS total_students
FROM students
GROUP BY department_id;
```

Q2. Show how many students are in each gender category.

```
SELECT gender, COUNT(*) AS total_students
FROM students
GROUP BY gender;
```

Q3. Show the average percentage of male and female students.

```
SELECT gender, AVG(percentage) AS avg_percentage
FROM students
GROUP BY gender;
```

Q4. Show the minimum and maximum percentage of students grouped by birth year.

```
SELECT YEAR(dob) AS birth_year, MIN(percentage), MAX(percentage)
FROM students
GROUP BY YEAR(dob);
```

Q5. Show the average fees paid grouped by department, gender, and birth year.

```
SELECT department_id, gender, YEAR(dob) AS birth_year, AVG(fees_paid)
AS avg_fees
FROM students
GROUP BY department_id, gender, YEAR(dob);
```

4. HAVING Clause

- Concept: `HAVING` filters groups (works with aggregates).

Q1. Show departments where the average percentage of students is greater than 75.

```
SELECT department_id, AVG(percentage) AS avg_percentage
FROM students
GROUP BY department_id
HAVING AVG(percentage) > 75;
```

Q2. Show gender-wise total percentage only if the total > 900.

```
SELECT gender, SUM(percentage) AS total_percentage
FROM students
GROUP BY gender
HAVING SUM(percentage) > 900;
```

5. ORDER BY & LIMIT

- Concept: `ORDER BY` sorts rows, `LIMIT` restricts number of results.

Q1. Show top 3 students by percentage.

```
SELECT full_name, percentage
FROM students
ORDER BY percentage DESC
LIMIT 3;
```

Q2. Show first 2 departments by department_id.

```
SELECT * FROM departments
ORDER BY department_id ASC
LIMIT 2;
```

Q3. Show first 3 students with percentage > 80.

```
SELECT * FROM students
WHERE percentage > 80
LIMIT 3;
```

6. JOINS

- Concept: Combine tables using relations.

Q1. Show students along with their department names.

```
SELECT s.full_name, s.percentage, d.department_name
FROM students s
JOIN departments d ON s.department_id = d.department_id;
```

Q2. Count students per department with department name.

```
SELECT d.department_name, COUNT(s.student_id) AS total_students
FROM students s
JOIN departments d ON s.department_id = d.department_id
GROUP BY d.department_name;
```

7. Subqueries

- Concept: Query inside another query.

Q1. Students above average percentage.

```
SELECT * FROM students
WHERE percentage > (SELECT AVG(percentage) FROM students);
```

Q2. Students with maximum total fees.

```
SELECT * FROM students
WHERE total_fees = (SELECT MAX(total_fees) FROM students);
```

Q3. Students in CS or IT departments.

```
SELECT * FROM students
WHERE department_id IN (
  SELECT department_id FROM departments
  WHERE department_name IN ('Computer Science','Information
Technology')
);
```

Q4. Correlated Subquery → Students in departments where avg fees > 90,000.

```
SELECT * FROM students s
WHERE EXISTS (
  SELECT 1 FROM students s2
  WHERE s.department_id = s2.department_id
  GROUP BY s2.department_id
  HAVING AVG(s2.total_fees) > 90000
);
```

8. CTE (Common Table Expression)

- Concept: Temporary named query for reusability.

Q1. Simple CTE.

```
WITH student_details AS (
  SELECT full_name, percentage, email
  FROM students
)
SELECT * FROM student_details;
```

Q2. CTE with condition.

```
WITH student_details AS (
  SELECT full_name, percentage
  FROM students
)
SELECT * FROM student_details WHERE percentage > 90;
```

Q3. CTE with Join.

```
WITH student_dept AS (
  SELECT s.full_name, s.percentage, d.department_name
  FROM students s
  JOIN departments d ON s.department_id = d.department_id
)
SELECT * FROM student_dept;
```

Q4. CTE with Group By.

```sql
WITH student_dept AS (
  SELECT s.gender, s.percentage
  FROM students s
)
SELECT gender, AVG(percentage) AS avg_percentage
FROM student_dept
GROUP BY gender;
```

9. Views

- Concept: Virtual table (saved query).

Q1. Create a view of students scoring > 90.

```sql
CREATE VIEW high_achievers AS
SELECT full_name, percentage, department_id
FROM students
WHERE percentage > 90;

SELECT * FROM high_achievers;
```

10. Stored Procedure

- Concept: Predefined query logic for reuse.

Q1. Create a procedure to get top N students by percentage.

```sql
DELIMITER //
CREATE PROCEDURE GetTopStudents(IN top_n INT)
BEGIN
  SELECT full_name, percentage
  FROM students
  ORDER BY percentage DESC
  LIMIT top_n;
END //
DELIMITER ;

-- Execute
CALL GetTopStudents(5);
```

11. Window Functions

- Concept: Analytics functions across rows.

Q1. Rank students by percentage.

```
SELECT full_name, percentage,
       RANK() OVER (ORDER BY percentage DESC) AS rank_position
FROM students;
```

Q2. Running total of fees_paid.

```
SELECT full_name, fees_paid,
       SUM(fees_paid) OVER (ORDER BY student_id) AS running_total
FROM students;
```

12. String Functions

Q1. Convert names to uppercase.

```
SELECT UPPER(full_name) FROM students;
```

Q2. Replace gmail.com with innomatics.in.

```
SELECT REPLACE(email, 'gmail.com', 'innomatics.in') FROM students;
```

13. Date Functions

Q1. Show year of birth.

```
SELECT full_name, YEAR(dob) AS birth_year FROM students;
```

Q2. Calculate age.

```
SELECT full_name, TIMESTAMPDIFF(YEAR, dob, CURDATE()) AS age FROM
students;
```

14. Transactions

- Concept: Ensuring atomic operations.

```
START TRANSACTION;
UPDATE students SET fees_paid = fees_paid + 5000 WHERE student_id = 1;
UPDATE students SET fees_paid = fees_paid - 5000 WHERE student_id = 2;
COMMIT;
```