SQL:

THE COMPLETE GUIDE

Mastering Relational Databases
from Basics to Advanced Techniques

Tejas Talole

What is SQL?

SQL (Structured Query Language) is the language used to communicate with databases. It lets you create, read, update, and delete data in databases.

Basic SQL Commands

1. **SELECT** — Get data from a table
2. **INSERT** — Add new data to a table
3. **UPDATE** — Change existing data
4. **DELETE** — Remove data from a table
5. **CREATE** — Make a new table or database
6. **DROP** — Delete a table or database

Step 1: SELECT — Get Data from Table

Imagine you have a table called `Students`:

ID	Name	Age	Grade
1	Tejas	22	A
2	Priya	21	B
3	Rahul	23	A

To get all data from this table:

```
SELECT * FROM Students;
```

- `SELECT` means get data
- `*` means all columns
- `FROM Students` means from the table named Students

Result:

ID	Name	Age	Grade
1	Tejas	22	A
2	Priya	21	B
3	Rahul	23	A

Try this: What query would you write to get only the names of students?

Step 2: SELECT Specific Columns

Instead of getting all columns (`*`), you can get only the columns you want.

Example: To get only the **Name** and **Age** of students from the `Students` table:

```
SELECT Name, Age FROM Students;
```

Result:

Name	Age
Tejas	22
Priya	21
Rahul	23

Step 3: Filtering Data with WHERE

What if you want only students with Grade 'A'?

You use the **WHERE** clause:

```
SELECT * FROM Students
WHERE Grade = 'A';
```

Result:

ID	Name	Age	Grade
1	Tejas	22	A
3	Rahul	23	A

Step 4: Using Comparison Operators

Some common comparison operators:

- `=` equal to
- `<>` or `!=` not equal to
- `>` greater than
- `<` less than

- `>=` greater than or equal
- `<=` less than or equal

Example: Get students older than 21:

```
SELECT * FROM Students
WHERE Age > 21;
```

Your turn: Write a query to get names of students whose grade is NOT 'B'.

Step 5: Sorting Data with ORDER BY

You can sort your results by one or more columns using **ORDER BY**.

Syntax:

```
SELECT columns
FROM table
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC];
```

- `ASC` means ascending (smallest to largest) — default
- `DESC` means descending (largest to smallest)

Example: Get all students ordered by Age from youngest to oldest:

```
SELECT * FROM Students
ORDER BY Age ASC;
```

Or, to get students ordered by Grade descending (Z to A):

```
SELECT * FROM Students
ORDER BY Grade DESC;
```

Step 6: Limiting Results with LIMIT

If you want only a few results, use **LIMIT**:

```
SELECT * FROM Students
ORDER BY Age DESC
LIMIT 2;
```

This gives you only the top 2 oldest students.

Step 7: Combining WHERE, ORDER BY, and LIMIT

Example: Get the top 2 students with Grade 'A' sorted by Age descending:

```sql
SELECT * FROM Students
WHERE Grade = 'A'
ORDER BY Age DESC
LIMIT 2;
```

Try this: Write a query to get the names of students younger than 23, ordered by Name ascending.

Step 8: INSERT — Add New Data

Syntax to insert one row into a table:

```
INSERT INTO table_name (column1, column2, column3)
VALUES (value1, value2, value3);
```

For our `Students` table, example:

```
INSERT INTO Students (ID, Name, Age, Grade)
VALUES (4, 'Anita', 20, 'B');
```

This adds a new student Anita with ID 4, Age 20, Grade B.

Important Notes:

- The columns and values order must match.
- You don't always need to specify all columns if some have defaults or allow NULL.
- If you want to insert values into **all columns**, you can skip column names:

```
INSERT INTO Students
VALUES (5, 'Suresh', 24, 'A');
```

Try This:

Write an INSERT query to add a student named **Mira**, age 22, grade 'C', with ID 6.

Step 9: UPDATE — Change Existing Data

Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

- `SET` is used to specify the columns and their new values.
- `WHERE` specifies which rows to update. Without `WHERE`, **all rows will be updated!**

Example 1: Change Grade of student with ID 2

```
UPDATE Students
SET Grade = 'A'
WHERE ID = 2;
```

This changes Priya's grade from B to A.

Example 2: Increase Age by 1 for all students

```
UPDATE Students
SET Age = Age + 1;
```

Since there is no `WHERE`, age of all students increases by 1.

Try this:
Write a query to change Mira's grade to 'B' where her name is 'Mira'.

Step 10: DELETE — Remove Data

Syntax:

```
DELETE FROM table_name
WHERE condition;
```

- `WHERE` is **very important** here.
- If you omit `WHERE`, **all rows in the table will be deleted!**

Example: Delete the student with ID 6 (Mira)

```
DELETE FROM Students
WHERE ID = 6;
```

This deletes the row where the student's ID is 6.

Warning:

Be very careful with `DELETE` queries. Always use a `WHERE` clause unless you want to delete everything.

Try this: Write a query to delete the student whose name is 'Anita'.

Step 11: CREATE TABLE — Make a New Table

Syntax:

```
CREATE TABLE table_name (
    column1 datatype constraints,
    column2 datatype constraints,
    column3 datatype constraints,
    ...
);
```

Example: Create a `Students` table

```
CREATE TABLE Students (
    ID INT PRIMARY KEY,
    Name VARCHAR(50) NOT NULL,
    Age INT,
    Grade CHAR(1)
);
```

Explanation:

- `ID INT PRIMARY KEY` — ID is an integer and uniquely identifies each row.
- `Name VARCHAR(50) NOT NULL` — Name is a string up to 50 characters and cannot be empty.
- `Age INT` — Age is an integer (can be NULL).
- `Grade CHAR(1)` — Grade is a single character (like 'A', 'B', etc.).

Common Data Types

Data Type	Description
INT	Integer numbers
VARCHAR(n)	Variable-length string up to n characters
CHAR(n)	Fixed-length string
DATE	Date values
FLOAT	Floating-point numbers (decimals)

Try this: Write a query to create a table named `Courses` with columns:

- CourseID (integer, primary key)

- CourseName (varchar 100, not null)
- Duration (integer, duration in days)

Step 12: ALTER TABLE — Modify Table Structure

You can use **ALTER TABLE** to:

- Add new columns
- Modify existing columns
- Drop (remove) columns
- Rename table or columns (depending on the database)

---

1. Add a New Column

```
ALTER TABLE table_name
ADD column_name datatype;
```

Example: Add a column `Email` to the `Students` table:

```
ALTER TABLE Students
ADD Email VARCHAR(100);
```

---

2. Modify a Column (Change datatype or size)

```
ALTER TABLE table_name
MODIFY column_name new_datatype;
```

Example: Change `Name` column size to 100:

```
ALTER TABLE Students
MODIFY Name VARCHAR(100);
```

(Note: Syntax may vary slightly depending on your SQL database like MySQL, SQL Server, PostgreSQL.)

---

3. Drop a Column

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

Example: Remove the `Email` column:

```sql
ALTER TABLE Students
DROP COLUMN Email;
```

Try this: Write a query to add a `StartDate` column (type DATE) to the `Courses` table.

Step 13: DROP TABLE — Delete a Table

Syntax:

```
DROP TABLE table_name;
```

Example: To delete the `Courses` table completely:

```
DROP TABLE Courses;
```

Warning: This permanently deletes the table **and all its data**. Use carefully.

Next: Let's learn about **JOINs** — combining data from multiple tables.

Would you like to start with:

- INNER JOIN (common matching rows)
- LEFT JOIN (all rows from left table + matched rows from right)
- RIGHT JOIN (all rows from right table + matched rows from left)
- FULL JOIN (all rows from both tables)

Step 14: Understanding JOINs

JOINs let you combine rows from **two or more tables** based on related columns.

Example Setup

Table: Students

ID	Name	Age	Grade
1	Tejas	22	A
2	Priya	21	B
3	Rahul	23	A

Table: Courses

StudentID	CourseName
1	Python
2	SQL
3	JavaScript

---

1. INNER JOIN — Only matching rows in both tables

```
SELECT Students.Name, Courses.CourseName
FROM Students
INNER JOIN Courses
ON Students.ID = Courses.StudentID;
```

Result:

Name	CourseName
Tejas	Python
Priya	SQL
Rahul	JavaScript

◈ Matches only where `Students.ID` = `Courses.StudentID`.

---

2. LEFT JOIN — All rows from the left table + matches from right

```sql
SELECT Students.Name, Courses.CourseName
FROM Students
LEFT JOIN Courses
ON Students.ID = Courses.StudentID;
```

- If a student has no course, `CourseName` will show as `NULL`.

---

3. RIGHT JOIN — All rows from the right table + matches from left

```sql
SELECT Students.Name, Courses.CourseName
FROM Students
RIGHT JOIN Courses
ON Students.ID = Courses.StudentID;
```

---

4. FULL JOIN — All rows from both tables (MySQL uses `UNION` to simulate)

```sql
SELECT Students.Name, Courses.CourseName
FROM Students
LEFT JOIN Courses ON Students.ID = Courses.StudentID
UNION
SELECT Students.Name, Courses.CourseName
FROM Students
RIGHT JOIN Courses ON Students.ID = Courses.StudentID;
```

✅ **Tip:**

- Use INNER JOIN when you need only matching data.
- Use LEFT JOIN when the left table is your main focus.
- Use RIGHT JOIN when the right table is your main focus.
- Use FULL JOIN when you want everything.

Step 15: GROUP BY — Group Data for Calculations

GROUP BY is used with **aggregate functions** like:

- `COUNT()` → counts rows
- `SUM()` → adds numbers
- `AVG()` → finds average
- `MIN()` → smallest value
- `MAX()` → largest value

Example Table: `Students`

ID	Name	Age	Grade
1	Tejas	22	A
2	Priya	21	B
3	Rahul	23	A
4	Anita	20	B

---

1. COUNT students in each grade

```
SELECT Grade, COUNT(*) AS TotalStudents
FROM Students
GROUP BY Grade;
```

Result:

Grade	TotalStudents
A	2
B	2

---

2. Average age by grade

```
SELECT Grade, AVG(Age) AS AverageAge
FROM Students
GROUP BY Grade;
```

---

3. Total age sum by grade

```sql
SELECT Grade, SUM(Age) AS TotalAge
FROM Students
GROUP BY Grade;
```

✅ **Tip:**

- Every column in `SELECT` that's not inside an aggregate function must be in `GROUP BY`.
- You can combine **GROUP BY** with **ORDER BY** to sort results.

Step 16: HAVING — Filter Groups After Aggregation

When you use `GROUP BY`, you can't use `WHERE` to filter aggregated results. Instead, you use **HAVING**.

Example: Students Table

ID	Name	Age	Grade
1	Tejas	22	A
2	Priya	21	B
3	Rahul	23	A
4	Anita	20	B
5	Mira	24	A

---

1. Show only grades with more than 2 students

```
SELECT Grade, COUNT(*) AS TotalStudents
FROM Students
GROUP BY Grade
HAVING COUNT(*) > 2;
```

---

2. Average age above 22 by grade

```
SELECT Grade, AVG(Age) AS AverageAge
FROM Students
GROUP BY Grade
HAVING AVG(Age) > 22;
```

✅ **Key Difference:**

- `WHERE` → filters **rows before grouping**
- `HAVING` → filters **groups after grouping**

Step 17: Subqueries — A Query Inside a Query

A **subquery** is a SQL query nested inside another SQL statement.

Example Table: `Students`

ID	Name	Age	Grade
1	Tejas	22	A
2	Priya	21	B
3	Rahul	23	A
4	Anita	20	B
5	Mira	24	A

---

1. Find students older than the average age

```sql
SELECT Name, Age
FROM Students
WHERE Age > (SELECT AVG(Age) FROM Students);
```

Here:

- `(SELECT AVG(Age) FROM Students)` is the subquery.
- The main query uses its result to filter rows.

---

2. Find students in the same grade as 'Tejas'

```sql
SELECT Name
FROM Students
WHERE Grade = (SELECT Grade FROM Students WHERE Name = 'Tejas');
```

---

3. Subquery in `FROM` (as a temporary table)

```sql
SELECT Grade, AvgAge
FROM (
    SELECT Grade, AVG(Age) AS AvgAge
    FROM Students
    GROUP BY Grade
) AS GradeStats
WHERE AvgAge > 21;
```

✅ **Tips:**

- A subquery can be in `SELECT`, `FROM`, or `WHERE`.
- Use aliases when subquery acts like a table.

Step 18: SQL Constraints

Constraints are rules you set on table columns to control the type of data that can be stored.

Common Constraints

Constraint	Purpose
PRIMARY KEY	Uniquely identifies each row (cannot be NULL)
FOREIGN KEY	Links data between tables
UNIQUE	Ensures all values in a column are different
NOT NULL	Column must have a value (no empty cells)
DEFAULT	Sets a default value if no value is given
CHECK	Ensures data meets a condition

---

1. PRIMARY KEY

```sql
CREATE TABLE Students (
    ID INT PRIMARY KEY,
    Name VARCHAR(50)
);
```

- `ID` must be unique and not NULL.

---

2. FOREIGN KEY

```sql
CREATE TABLE Courses (
    CourseID INT PRIMARY KEY,
    StudentID INT,
    FOREIGN KEY (StudentID) REFERENCES Students(ID)
);
```

- Ensures `StudentID` in `Courses` exists in `Students`.

---

3. UNIQUE

```sql
ALTER TABLE Students
ADD CONSTRAINT unique_email UNIQUE (Email);
```

- No two students can have the same email.

---

4. NOT NULL

```
ALTER TABLE Students
MODIFY Name VARCHAR(50) NOT NULL;
```

- Name cannot be empty.

---

5. DEFAULT

```
ALTER TABLE Students
ADD Grade CHAR(1) DEFAULT 'C';
```

- If no grade is given, it will be `C`.

---

6. CHECK

```
ALTER TABLE Students
ADD CONSTRAINT check_age CHECK (Age >= 18);
```

- Ensures Age is at least 18.

Step 19: SQL Indexes

An **index** is like the index at the back of a book — it helps you find data faster without scanning the whole table.

Why use indexes?

- They **speed up SELECT queries** (especially on large tables).
- They make searching, filtering, and sorting faster.
- **Downside**: They take extra storage and slow down `INSERT`, `UPDATE`, and `DELETE` because the index must also be updated.

---

1. Create an Index

```sql
CREATE INDEX idx_student_name
ON Students(Name);
```

- Creates an index called `idx_student_name` on the `Name` column.

---

2. Create a Unique Index

```sql
CREATE UNIQUE INDEX idx_unique_email
ON Students(Email);
```

- Ensures values in `Email` are unique.

---

3. Drop an Index (MySQL)

```sql
DROP INDEX idx_student_name
ON Students;
```

---

4. When to use indexes

✅ Use on:

- Columns you frequently search (`WHERE Name = 'Tejas'`)
- Columns used in sorting (`ORDER BY Age`)
- Columns used in joins (`ON Students.ID = Courses.StudentID`)

❌ Avoid on:

- Very small tables (indexes won’t help much)
- Columns frequently updated (can slow performance)

Step 20: SQL Views

A **view** is like a virtual table created from a saved query. It doesn't store data itself — it just stores the SQL query and shows results when you use it.

Why use views?

- To simplify complex queries
- To hide certain columns for security
- To make code reusable

---

1. Create a View

```
CREATE VIEW GradeAStudents AS
SELECT Name, Age
FROM Students
WHERE Grade = 'A';
```

- Now, `GradeAStudents` acts like a table.

---

2. Use the View

```
SELECT * FROM GradeAStudents;
```

This shows only students with Grade 'A'.

---

3. Update a View

```
CREATE OR REPLACE VIEW GradeAStudents AS
SELECT Name, Age, Grade
FROM Students
WHERE Grade = 'A';
```

---

4. Drop a View

```
DROP VIEW GradeAStudents;
```

✅ **Tip:**

- Views are great for **hiding sensitive data**.
- They can join multiple tables and still be queried like a single table.

Step 21: Stored Procedures

A **Stored Procedure** is a saved block of SQL code that you can run whenever needed. It can take inputs, perform queries, and even include logic like IF statements.

---

1. Create a Stored Procedure

```
DELIMITER $$

CREATE PROCEDURE GetStudentsByGrade(IN gradeLetter CHAR(1))
BEGIN
    SELECT Name, Age
    FROM Students
    WHERE Grade = gradeLetter;
END $$

DELIMITER ;
```

- `IN gradeLetter` → input parameter
- You run it like this:

```
CALL GetStudentsByGrade('A');
```

---

2. Advantages

- Reusable code
- Improves performance (less query parsing)
- Can contain complex logic

Step 22: Functions

A **Function** is like a procedure but **must return a value** and is often used inside queries.

Example: Create a Function

```
DELIMITER $$

CREATE FUNCTION TotalStudents()
RETURNS INT
DETERMINISTIC
BEGIN
    DECLARE total INT;
```

```
    SELECT COUNT(*) INTO total FROM Students;
    RETURN total;
END $$

DELIMITER ;
```

You can use it like:

```
SELECT TotalStudents();
```

✅ Key Differences

Feature	Stored Procedure	Function
Returns value?	No (unless using OUT params)	Yes
Called by	`CALL`	Inside queries
Use in SELECT?	No	Yes

Step 23: SQL Transactions

A **transaction** is a set of SQL statements that are executed as a single unit. If something goes wrong, you can undo the whole transaction.

Key Commands:

- **START TRANSACTION** → Begin a transaction
- **COMMIT** → Save all changes
- **ROLLBACK** → Undo changes since the last `START TRANSACTION`
- **SAVEPOINT** → Mark a point in a transaction to roll back to

---

Example:

```sql
START TRANSACTION;

UPDATE Students
SET Grade = 'B'
WHERE Name = 'Tejas';

DELETE FROM Students
WHERE Name = 'Anita';

COMMIT;
```

- Both queries will only be saved if `COMMIT` runs successfully.

Example with ROLLBACK:

```sql
START TRANSACTION;

UPDATE Students
SET Grade = 'A'
WHERE Name = 'Priya';

ROLLBACK;
```

- This will undo the update and keep Priya's old grade.

Example with SAVEPOINT:

```sql
START TRANSACTION;

DELETE FROM Students WHERE Name = 'Rahul';
SAVEPOINT before_mira;
DELETE FROM Students WHERE Name = 'Mira';

ROLLBACK TO before_mira;
COMMIT;
```

- Deletes Rahul
- Sets a savepoint
- Deletes Mira
- Rolls back only the Mira delete
- Commits the rest

✅ **Use transactions when:**

- You're inserting/updating/deleting multiple related records
- You want to ensure **all or nothing** changes (like bank transfers)

Step 24: CTE (Common Table Expressions)

A **CTE** is like a temporary named result set that you can use in your query. It makes complex queries easier to read.

Syntax:

```
WITH cte_name AS (
    SELECT Grade, COUNT(*) AS TotalStudents
    FROM Students
    GROUP BY Grade
)
SELECT * FROM cte_name
WHERE TotalStudents > 2;
```

- `WITH` defines the CTE.
- It can be used multiple times in the main query.

Step 25: Window Functions

Window functions perform calculations **across a set of rows** related to the current row without grouping the results.

Example: Ranking students by age

```
SELECT Name, Age,
       RANK() OVER (ORDER BY Age DESC) AS AgeRank
FROM Students;
```

Common Window Functions:

- **ROW_NUMBER()** → assigns a unique number per row
- **RANK()** → assigns rank, ties get same rank
- **DENSE_RANK()** → similar to RANK but no gaps
- **LEAD() / LAG()** → get next/previous row's value

Step 26: Normalization

Normalization is the process of organizing data to reduce duplication and improve consistency.

Levels:

1. **1NF (First Normal Form)** — No repeating columns, each cell has a single value
2. **2NF (Second Normal Form)** — Must be in 1NF + all non-key columns depend on the whole primary key
3. **3NF (Third Normal Form)** — Must be in 2NF + no transitive dependency (columns depend only on primary key)
4. **BCNF (Boyce-Codd Normal Form)** — Stricter version of 3NF

✅ Example:

- Instead of storing a student's course name directly in the `Students` table, store a `CourseID` and link it to a `Courses` table via a **foreign key**.

Step 27: UNION vs UNION ALL

Both are used to combine results from two queries.

Example:

```
SELECT Name FROM Students WHERE Grade = 'A'
UNION
SELECT Name FROM Students WHERE Age > 22;
```

- **UNION** → removes duplicate rows
- **UNION ALL** → keeps duplicates (faster)

Key Difference:

Feature	UNION	UNION ALL
Duplicates	Removed	Kept
Performance	Slower	Faster

Step 28: EXISTS & NOT EXISTS

Used to check if a subquery returns any rows.

Example: Students who have at least one course

```
SELECT Name
FROM Students s
WHERE EXISTS (
    SELECT 1
    FROM Courses c
    WHERE c.StudentID = s.ID
);
```

NOT EXISTS — Students without any course

```
SELECT Name
FROM Students s
WHERE NOT EXISTS (
    SELECT 1
    FROM Courses c
```

```
    WHERE c.StudentID = s.ID
);
```

Step 29: OFFSET for Pagination

Used to skip a certain number of rows.

Example: Get 5 students, skipping the first 10

```
SELECT * FROM Students
ORDER BY Name
LIMIT 5 OFFSET 10;
```

- Useful for **pages** in apps: Page 1 → `LIMIT 5 OFFSET 0` Page 2 → `LIMIT 5 OFFSET 5`

✅ You now know:

- Basics (SELECT, INSERT, UPDATE, DELETE)
- Filtering (WHERE, ORDER BY, LIMIT)
- Joins, GROUP BY, HAVING
- Advanced features (Subqueries, CTE, Window Functions)
- Constraints, Indexes, Views, Stored Procedures
- Transactions & Advanced querying tools

Step 30: `EXPLAIN` — See How SQL Executes Your Query

`EXPLAIN` shows the **execution plan** — how the database will run your query. This helps you optimize slow queries.

Example:

```
EXPLAIN SELECT * FROM Students WHERE Name = 'Tejas';
```

It returns details like:

- **id** → Query step number
- **select_type** → Type of select (simple, subquery, etc.)
- **table** → Which table is accessed
- **type** → Join type (ALL, index, range)
- **possible_keys** → Which indexes could be used
- **key** → Which index was actually used
- **rows** → Number of rows it will scan
- **Extra** → Additional info like `Using where`, `Using index`

✅ **Use EXPLAIN when:**

- Your query is slow
- You want to check if indexes are being used
- You're joining large tables

🎯 Congratulations!

You've now learned **SQL from zero to advanced level**:

- CRUD operations
- Filtering, sorting, limiting
- Joins & aggregations
- Constraints & indexes
- Views, procedures, functions
- Transactions & subqueries
- CTE, window functions, normalization
- UNION, EXISTS, pagination
- Performance tuning with EXPLAIN