SQL: Complete Notes — Basics to Advanced

---

1. What SQL Is

SQL (Structured Query Language) is the standard language for talking to **relational databases** — data organized into tables with rows and columns, where tables can be linked to each other via keys (e.g., MySQL, PostgreSQL, SQL Server, Oracle, SQLite).

SQL commands fall into categories:

Category	Stands for	Does	Examples
DDL	Data Definition Language	Defines structure	`CREATE`, `ALTER`, `DROP`, `TRUNCATE`
DML	Data Manipulation Language	Changes data	`INSERT`, `UPDATE`, `DELETE`
DQL	Data Query Language	Reads data	`SELECT`
DCL	Data Control Language	Permissions	`GRANT`, `REVOKE`
TCL	Transaction Control Language	Manages transactions	`COMMIT`, `ROLLBACK`, `SAVEPOINT`

---

2. Database & Table Basics

```
CREATE DATABASE company;
USE company;


DROP DATABASE company;          -- deletes the whole database
```

A **table** is a grid of rows (records) and columns (fields/attributes).

```
employees
+----+----------+------------+--------+
| id | name     | department | salary |
+----+----------+------------+--------+
```

```
| 1  | Priya    | HR          | 55000  |
| 2  | Rahul    | Engineering| 72000  |
+----+----------+------------+--------+
```

---

3. Data Types

Type	Use for
`INT` / `BIGINT`	Whole numbers
`DECIMAL(p,s)` / `NUMERIC`	Exact decimals (money!)
`FLOAT` / `DOUBLE`	Approximate decimals
`VARCHAR(n)`	Variable-length text, max n chars
`CHAR(n)`	Fixed-length text
`TEXT`	Long text
`DATE`	`YYYY-MM-DD`
`DATETIME` / `TIMESTAMP`	Date + time
`BOOLEAN`	True/false
`ENUM(...)`	Fixed set of allowed values (MySQL)

> **Never use FLOAT for money.** Use `DECIMAL` — floats introduce rounding errors.

---

4. Creating & Modifying Tables (DDL)

```
CREATE TABLE employees (
    id          INT ,
    name        VARCHAR(100),
    department  VARCHAR(50),
    salary      DECIMAL(10,2),
    hire_date   DATE DEFAULT CURRENT_DATE,
);
```

```
ALTER TABLE employees ADD COLUMN email VARCHAR(100);
ALTER TABLE employees DROP COLUMN email;
ALTER TABLE employees MODIFY COLUMN salary DECIMAL(12,2);   -- MySQL
ALTER TABLE employees RENAME COLUMN name TO full_name;

DROP TABLE employees;            -- deletes table + data + structure
TRUNCATE TABLE employees;        -- deletes all rows, keeps structure (faster than DELETE, can't be rolled back in some DBs)
```

---

5. Inserting, Updating, Deleting Data (DML)

```
-- Insert
INSERT INTO employees (name, department, salary)
VALUES ('Ananya', 'Marketing', 60000);

INSERT INTO employees (name, department, salary) VALUES
('Vikram', 'Sales', 50000),
('Neha', 'HR', 58000);           -- multi-row insert

-- Update
UPDATE employees
SET salary = 65000
WHERE name = 'Ananya';

-- ⚠️ ALWAYS use WHERE with UPDATE/DELETE, or it applies to EVERY row.

-- Delete
DELETE FROM employees
WHERE department = 'Sales';
```

---

6. Querying Data — SELECT Basics

```
SELECT * FROM employees;                    -- all columns
SELECT name, salary FROM employees;         -- specific columns
SELECT DISTINCT department FROM employees;  -- unique values only
SELECT name AS employee_name FROM employees;   -- alias (rename in output)
SELECT name, salary * 12 AS annual_salary FROM employees;   -- computed column
```

Logical order SQL actually executes in (different from how you *write* it — this trips people up constantly):

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

You *write* `SELECT` first, but it runs almost last. This is why you can't use a column alias (defined in `SELECT`) inside a `WHERE` clause.

---

7. Filtering — WHERE

```
SELECT * FROM employees WHERE department = 'HR';
SELECT * FROM employees WHERE salary > 60000;
SELECT * FROM employees WHERE salary BETWEEN 50000 AND 70000;
SELECT * FROM employees WHERE department IN ('HR', 'Sales');
SELECT * FROM employees WHERE name LIKE 'A%';        -- starts with A
SELECT * FROM employees WHERE name LIKE '%a';        -- ends with a
SELECT * FROM employees WHERE name LIKE '%an%';      -- contains "an"
SELECT * FROM employees WHERE email IS NULL;
SELECT * FROM employees WHERE email IS NOT NULL;
SELECT * FROM employees WHERE department = 'HR' AND salary > 55000;
SELECT * FROM employees WHERE department = 'HR' OR department = 'Sales';
SELECT * FROM employees WHERE NOT department = 'HR';
```

`LIKE` wildcards: `%` = any number of characters, `_` = exactly one character.

---

8. Sorting & Limiting

```
SELECT * FROM employees ORDER BY salary DESC;
SELECT * FROM employees ORDER BY department ASC, salary DESC;   -- multi-level sort

SELECT * FROM employees LIMIT 5;                -- MySQL/PostgreSQL/SQLite
SELECT * FROM employees LIMIT 5 OFFSET 10;      -- skip 10, take next 5 (pagination)
SELECT TOP 5 * FROM employees;                  -- SQL Server syntax instead of LIMIT
```

---

9. Aggregate Functions & GROUP BY

```
SELECT COUNT(*) FROM employees;
SELECT SUM(salary) FROM employees;
SELECT AVG(salary) FROM employees;
SELECT MIN(salary), MAX(salary) FROM employees;

-- Group by department
SELECT department, COUNT(*) AS headcount, AVG(salary) AS avg_salary
FROM employees
GROUP BY department;
```

```
-- Filter groups with HAVING (WHERE can't be used on aggregates)
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department
HAVING AVG(salary) > 60000;
```

> **WHERE filters rows before grouping. HAVING filters groups after aggregation.** This is one of the most commonly confused pairs in SQL.

---

10. Joins — The Most Important Concept

Joins combine rows from two or more tables based on a related column.

```
-- Sample second table
CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50)
);
```

Join type	Returns
`INNER JOIN`	Only rows that match in both tables
`LEFT JOIN`	All rows from left table + matches from right (NULL if no match)
`RIGHT JOIN`	All rows from right table + matches from left
`FULL OUTER JOIN`	All rows from both, matched where possible
`CROSS JOIN`	Every combination of both tables (cartesian product)
`SELF JOIN`	A table joined with itself

```
SELECT e.name, d.dept_name
FROM employees e
INNER JOIN departments d ON e.department_id = d.dept_id;

SELECT e.name, d.dept_name
FROM employees e
LEFT JOIN departments d ON e.department_id = d.dept_id;
-- includes employees with no department assigned

-- Self join example: find employees who share a manager
SELECT a.name AS employee, b.name AS colleague
```

```
FROM employees a
JOIN employees b ON a.manager_id = b.manager_id AND a.id != b.id;
```

Visual mental model:

INNER JOIN: A ∩ B (only overlap)
LEFT JOIN: all of A, matched with B where possible
FULL OUTER: everything from A and B

---

11. Subqueries

A query nested inside another query.

```
-- Subquery in WHERE
SELECT name FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);

-- Subquery in FROM (derived table)
SELECT department, avg_sal FROM (
    SELECT department, AVG(salary) AS avg_sal
    FROM employees
    GROUP BY department
) AS dept_averages
WHERE avg_sal > 60000;

-- Correlated subquery (references outer query — runs once per row, slower)
SELECT name, salary FROM employees e1
WHERE salary > (
    SELECT AVG(salary) FROM employees e2
    WHERE e2.department = e1.department
);

-- EXISTS
SELECT name FROM employees e
WHERE EXISTS (SELECT 1 FROM departments d WHERE d.dept_id = e.department_id);
```

---

12. Set Operations

```
SELECT name FROM employees
UNION
SELECT name FROM contractors;          -- combines, removes duplicates
```

```
SELECT name FROM employees
UNION ALL
SELECT name FROM contractors;        -- combines, KEEPS duplicates (faster)

SELECT name FROM employees
INTERSECT
SELECT name FROM contractors;        -- only names in both

SELECT name FROM employees
EXCEPT                               -- MINUS in Oracle
SELECT name FROM contractors;        -- in employees but not contractors
```

> Columns in each `SELECT` must match in number and compatible data type.

---

13. Window Functions

Unlike `GROUP BY` (which collapses rows), window functions calculate across a set of rows **while still returning every row**.

```
SELECT name, department, salary,
   RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS dept_rank,
   ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS
row_num,
   AVG(salary) OVER (PARTITION BY department) AS dept_avg,
   SUM(salary) OVER (ORDER BY hire_date) AS running_total
FROM employees;
```

Function	Does
`ROW_NUMBER()`	Unique sequential number per row
`RANK()`	Same rank for ties, then skips numbers
`DENSE_RANK()`	Same rank for ties, no skipping
`LAG()` / `LEAD()`	Value from previous/next row
`NTILE(n)`	Splits rows into n buckets

Extremely common real-world use: "top 3 highest paid per department," "month-over-month change," "running totals."

---

14. Common Table Expressions (CTEs)

A named temporary result set, defined with `WITH` — makes complex queries readable.

```
WITH high_earners AS (
    SELECT * FROM employees WHERE salary > 60000
)
SELECT department, COUNT(*)
FROM high_earners
GROUP BY department;

-- Recursive CTE (e.g., org chart / hierarchy traversal)
WITH RECURSIVE org_chart AS (
    SELECT id, name, manager_id, 1 AS level
    FROM employees WHERE manager_id IS NULL
    UNION ALL
    SELECT e.id, e.name, e.manager_id, oc.level + 1
    FROM employees e
    JOIN org_chart oc ON e.manager_id = oc.id
)
SELECT * FROM org_chart;
```

---

15. Constraints & Keys

```
CREATE TABLE departments (
    dept_id     INT PRIMARY KEY,
    dept_name   VARCHAR(50) NOT NULL UNIQUE
);

CREATE TABLE employees (
    id          INT PRIMARY KEY AUTO_INCREMENT,
    name        VARCHAR(100) NOT NULL,
    department_id INT,
    salary      DECIMAL(10,2) CHECK (salary > 0),
    FOREIGN KEY (department_id) REFERENCES departments(dept_id)
);
```

Constraint	Meaning

`PRIMARY KEY`	Uniquely identifies each row (implies `NOT NULL` + `UNIQUE`)
`FOREIGN KEY`	Links to another table's primary key, enforces referential integrity
`NOT NULL`	Column can't be empty
`UNIQUE`	No duplicate values allowed
`CHECK`	Custom validation rule
`DEFAULT`	Value used if none provided

Composite key = primary key made of multiple columns together.

---

16. Indexes

An index speeds up lookups on a column (like a book's index) — at the cost of slightly slower writes and extra storage.

```
CREATE INDEX idx_department ON employees(department_id);
CREATE UNIQUE INDEX idx_email ON employees(email);
DROP INDEX idx_department ON employees;
```

Use indexes on columns you frequently `WHERE`, `JOIN`, or `ORDER BY` on — not on every column (over-indexing slows down inserts/updates).

---

17. Transactions (TCL)

A transaction groups multiple statements so they **all succeed or all fail together** — critical for things like money transfers.

```
START TRANSACTION;              -- or: BEGIN;

UPDATE accounts SET balance = balance - 500 WHERE id = 1;
UPDATE accounts SET balance = balance + 500 WHERE id = 2;

COMMIT;      -- save both changes permanently
-- OR
ROLLBACK;    -- undo everything since START TRANSACTION
```

ACID properties transactions guarantee:

- **Atomicity** — all or nothing
- **Consistency** — database moves from one valid state to another
- **Isolation** — concurrent transactions don't interfere with each other
- **Durability** — once committed, it survives a crash

```
SAVEPOINT before_update;
UPDATE employees SET salary = salary * 1.1;
ROLLBACK TO before_update;        -- undo just to this point, not the whole transaction
```

---

18. Views

A **view** is a saved, reusable query that behaves like a virtual table.

```
CREATE VIEW high_earners AS
SELECT name, department, salary
FROM employees
WHERE salary > 60000;

SELECT * FROM high_earners;       -- query it like a normal table

DROP VIEW high_earners;
```

Useful for simplifying complex/repeated queries and for restricting what certain users can see (expose a view, not the underlying sensitive table).

---

19. Permissions (DCL)

```
GRANT SELECT, INSERT ON employees TO 'analyst_user';
REVOKE INSERT ON employees FROM 'analyst_user';
```

---

20. Normalization (Database Design)

Organizing tables to reduce redundancy and avoid update anomalies.

Form	Rule
1NF	Each column holds atomic (indivisible) values, no repeating groups

2NF	1NF + no partial dependency (non-key columns depend on the *whole* primary key)
3NF	2NF + no transitive dependency (non-key columns depend only on the key, not on other non-key columns)

Example fix: instead of storing `department_name` directly in the `employees` table (duplicated across every employee in that department), store `department_id` and keep `department_name` in a separate `departments` table. Update the name once, not on every row.

Denormalization — deliberately breaking these rules for read performance (common in reporting/analytics tables, data warehouses) — a legitimate trade-off, not a mistake, when done intentionally.

---

21. Common Mistakes & Gotchas

Mistake	Why it's a problem
`DELETE`/`UPDATE` without `WHERE`	Wipes/changes the entire table
Comparing `NULL` with `=`	`WHERE column = NULL` never matches — use `IS NULL`
Using `WHERE` to filter an aggregate	Use `HAVING` instead — `WHERE` runs before grouping
Forgetting `GROUP BY` covers all non-aggregated `SELECT` columns	Some DBs error, others silently give wrong results
`SELECT *` in production code	Breaks if columns change; also slower — name columns explicitly
Not indexing foreign key / join columns	Joins get slow on large tables
Confusing `UNION` and `UNION ALL`	`UNION` dedupes and is slower; use `UNION ALL` if duplicates are fine/expected
Implicit type conversion in `WHERE`	Can silently prevent index usage and cause wrong matches
Not using transactions for multi-step writes	Partial failure leaves data inconsistent

---

22. Quick Reference Cheat Sheet

```
-- Structure
CREATE TABLE t (id INT PRIMARY KEY, col VARCHAR(50));
ALTER TABLE t ADD COLUMN new_col INT;
DROP TABLE t;

-- Data
INSERT INTO t (col) VALUES ('x');
UPDATE t SET col = 'y' WHERE id = 1;
DELETE FROM t WHERE id = 1;

-- Query
SELECT col FROM t WHERE col = 'x' ORDER BY col LIMIT 10;
SELECT dept, COUNT(*) FROM t GROUP BY dept HAVING COUNT(*) > 1;

-- Joins
SELECT * FROM a JOIN b ON a.id = b.a_id;
SELECT * FROM a LEFT JOIN b ON a.id = b.a_id;

-- Window function
SELECT col, RANK() OVER (PARTITION BY dept ORDER BY col DESC) FROM t;

-- CTE
WITH temp AS (SELECT * FROM t WHERE col > 10)
SELECT * FROM temp;

-- Transaction
START TRANSACTION;
UPDATE t SET col = 'z' WHERE id = 1;
COMMIT;  -- or ROLLBACK;
```

The one-paragraph summary

Data lives in tables linked by keys. You define structure with DDL, change data with DML, and read it with `SELECT` — filtered by `WHERE`, grouped with `GROUP BY`/`HAVING`, sorted with `ORDER BY`, and combined across tables with `JOIN`. For anything more advanced — ranking, running totals, hierarchies, or readable complex logic — reach for window functions and CTEs. Wrap multi-step writes in transactions so they can't half-fail.