SQL & Relational Database Basics

Foundations — what SQL is, how relational databases work, and the language categories you must know.

CONTENTS

DEF What is SQL?

SQL (Structured Query Language) is the standard language for defining, manipulating, and querying data in relational databases. Developed at IBM in the 1970s; standardized by ANSI (1986) and ISO (1987). It is *declarative* — you describe *what* data you want, not *how* to retrieve it.

DEF What is a Relational Database?

A **Relational Database (RDB)** stores data in **tables (relations)** linked by shared columns (keys). Based on E.F. Codd's relational model (1970). Data is accessed via SQL. Examples: MySQL, PostgreSQL, Oracle, SQL Server, SQLite.

vs DBMS vs RDBMS

ASPECT	DBMS	RDBMS
Data model	File / hierarchical / network	Tabular (rows & columns)
Relationships	No support (or hard-coded)	First-class via foreign keys
Query language	Proprietary / none	SQL (ANSI standard)
ACID	Usually not guaranteed	Guaranteed (in OLTP engines)
Examples	XML DBs, old IMS, file systems	MySQL, PostgreSQL, Oracle, SQLite

Database Terminology

TERM	MEANING
Table	A named set of rows & columns (a *relation*)
Row	One record = a *tuple*
Column	A named field = an *attribute*
Schema	Blueprint: tables, types, constraints
Record	Same as row (runtime term)
Tuple	Formal name for a row
Attribute	Formal name for a column
Database	Organized collection of tables

DB Popular Databases

MySQL — Most popular open-source OLTP web DB. Owned by Oracle.

PostgreSQL — Feature-rich open-source ORDBMS. Strong on standards.

SQL Server — Microsoft's enterprise RDBMS (T-SQL dialect).

Oracle — Commercial enterprise leader (PL/SQL dialect).

SQLite — Embedded, file-based, zero-config. Used in mobile & browsers.

MariaDB — MySQL fork; community-driven, binary-compatible.

SQL SQL Command Categories

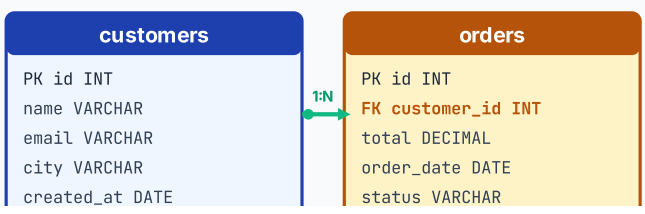
DDL — **Data Definition** — defines structure. `CREATE, ALTER, DROP, TRUNCATE, RENAME`

DML — **Data Manipulation** — changes rows. `INSERT, UPDATE, DELETE, MERGE`

DQL — **Data Query** — reads data. `SELECT` (some texts fold into DML).

DCL — **Data Control** — permissions. `GRANT, REVOKE`

TCL — **Transaction Control**. `COMMIT, ROLLBACK, SAVEPOINT`



! Why It Matters

SQL is **set-based**: operations act on entire sets of rows at once, not row-by-row. This is why SQL beats procedural code for data work — and why thinking in sets (not loops) is the #1 skill to learn.

NOTE Although "SQL" is sometimes pronounced "sequel", the ANSI standard spelling is the letters S-Q-L. `MySQL` and `PostgreSQL`

QUICK TIPS **SQL is case-insensitive** for keywords (but identifiers may be case-sensitive on Linux/PostgreSQL) • **Semicolons** terminate statements — required by most clients when running multiple queries • **Strings use single quotes** `'like this'`; double quotes are for identifiers in PostgreSQL • Think in **sets**, not loops — that's the relational mindset

SQL Syntax & CRUD Operations

The core verbs — define tables, insert, read, modify, and remove data.

+ Database & Table Creation

```sql
CREATE DATABASE shop;
USE shop;  -- MySQL / SQL Server
-- PostgreSQL: \c shop  (psql meta-command)

CREATE TABLE products (
    id          SERIAL       PRIMARY KEY,          -- PG auto-inc
    name        VARCHAR(100) NOT NULL,
    price       DECIMAL(10,2) DEFAULT 0,
    in_stock    BOOLEAN      DEFAULT TRUE,
    created_at  TIMESTAMP    DEFAULT NOW(),
    CONSTRAINT chk_price CHECK (price >= 0)
);
```

DDL

+ INSERT — Create Rows

```sql
-- Single row
INSERT INTO products (name, price)
VALUES ('Laptop', 1299.99);

-- Multi-row
INSERT INTO products (name, price) VALUES
    ('Mouse', 19.99),
    ('Keyboard', 79.50),
    ('Monitor', 349.00);

-- INSERT ... SELECT (copy from another table)
INSERT INTO archive SELECT * FROM products WHERE in_stock = FALSE;
```

DML

↻ UPDATE — Modify Rows

```sql
UPDATE products
SET    price = price * 0.9,
       in_stock = FALSE
WHERE  name = 'Mouse';
```

DML

DANGER Always pair `UPDATE` with a `WHERE` clause. **An UPDATE without WHERE updates every row in the table.**

? SELECT — Read Rows

```sql
-- Basic projection & filtering
SELECT name, price
FROM   products
WHERE  price < 100
ORDER BY price DESC
LIMIT 10;

-- All columns
SELECT * FROM products;

-- Aliases & computed columns
SELECT name AS product,
       price * 1.2 AS price_with_tax
FROM   products;
```

DQL

- DELETE — Remove Rows

```sql
DELETE FROM products
WHERE  in_stock = FALSE
  AND  created_at < NOW() - INTERVAL '90 days';
```

DML

⊘ DELETE vs TRUNCATE vs DROP

CMD	SCOPE	WHERE	RESET AUTOINCR	ROLLBACK?
`DELETE`	Row-by-row (DML)	Optional	No	Yes
`TRUNCATE`	Whole table (DDL)	Not allowed	Yes	No (PG: yes in txn)
`DROP`	Table + structure	—	—	No

Δ ALTER & RENAME — Modify Schema

```sql
-- Add column
ALTER TABLE products ADD COLUMN sku VARCHAR(20);

-- Modify type (PG syntax)
ALTER TABLE products ALTER COLUMN price TYPE DECIMAL(12,2);

-- MySQL: MODIFY COLUMN
-- ALTER TABLE products MODIFY COLUMN price DECIMAL(12,2);

-- Drop column
ALTER TABLE products DROP COLUMN sku;

-- Rename table
ALTER TABLE products RENAME TO catalog;
-- Oracle/MySQL: RENAME products TO catalog;
```

DDL

QUICK TIPS **DELETE without WHERE** empties the table but keeps structure; **TRUNCATE** is faster but cannot be rolled back in MySQL • **DROP** removes the table itself — back up first • `SELECT *` is fine for exploration, but **always project explicit columns in production** for stability • Use `RETURNING *` (PostgreSQL) or `OUTPUT` (SQL Server) to see rows affected by INSERT/UPDATE/DELETE

Filtering & Sorting Data

Narrow results with WHERE, pattern-match with LIKE, sort with ORDER BY, and limit rows.

WHERE — Row Filtering

```sql
SELECT name, age, city
FROM   users
WHERE  age >= 18
  AND  city IN ('NYC', 'LA', 'SF')
  AND  email IS NOT NULL
  AND  (status = 'active' OR status = 'trial');
```

WHERE is evaluated per row *before* grouping/aggregation. Use `HAVING` (Page 4) to filter *after* aggregation.

DISTINCT — Unique Rows

```sql
SELECT DISTINCT city, country
FROM users
ORDER BY country, city;
```

ORDER BY / LIMIT / OFFSET

```sql
-- ANSI standard pagination
SELECT name, score
FROM   players
ORDER BY score DESC, name ASC
OFFSET 20 ROWS FETCH FIRST 10 ROWS ONLY;

-- MySQL / PostgreSQL shorthand
SELECT name, score FROM players
ORDER BY score DESC LIMIT 10 OFFSET 20;

-- SQL Server
SELECT TOP 10 name, score FROM players
ORDER BY score DESC;
```

LIKE & Wildcards

```sql
-- Names starting with 'J'
SELECT * FROM users WHERE name LIKE 'J%';

-- 5-letter names ending in 'n'
SELECT * FROM users WHERE name LIKE '____n';

-- Names starting with a-m (SQL Server/PG with regex class)
SELECT * FROM users WHERE name LIKE '[a-m]%';

-- PostgreSQL: SIMILAR TO or ~ (regex)
SELECT * FROM users WHERE name ~ '^J[a-z]+$';
```

WILDCARD	MEANING
`%`	Zero or more characters (any)
`_`	Exactly one character
`[abc]`	Any single char in set (T-SQL only)
`[^abc]`	Any single char NOT in set (T-SQL)
`ESCAPE '\'`	Treat next char literally (e.g. `LIKE '50\%' ESCAPE '\'`)

BETWEEN / IN / IS NULL

```sql
-- Range (inclusive)
WHERE age BETWEEN 18 AND 65

-- Set membership
WHERE country IN ('US', 'CA', 'UK')
WHERE country NOT IN ('US', 'CA')

-- NULL handling — NEVER use = NULL
WHERE email IS NULL
WHERE email IS NOT NULL
```

GOTCHA `WHERE col ≠ 'X'` **excludes NULLs too** — NULL is "unknown", not "not X". Use `(col ≠ 'X' OR col IS NULL)` to keep NULL rows.

Comparison Operators

OP	MEANING	EXAMPLE
`=`	Equal	`age = 30`
`<>` / `≠`	Not equal (`<>` is ANSI)	`status <> 'deleted'`
`<`, `>`	Less / greater than	`price < 100`
`⩽`, `⩾`	Less/greater or equal	`qty ⩾ 5`
`<> ALL`	Compare to all (subquery)	`price < ALL (SELECT ...)`

Logical Operators

OP	MEANING	NOTES
`AND`	Both true	Higher precedence than OR
`OR`	Either true	Use parentheses to group
`NOT`	Negates	`NOT IN`, `NOT LIKE`, `NOT BETWEEN`
`ALL`	True if comparison holds for all rows of subquery	`> ALL (subquery)`
`ANY` / `SOME`	True if comparison holds for at least one	`= ANY (subquery)` ≡ `IN`

QUICK TIPS `WHERE` runs **before** GROUP BY; `HAVING` runs **after** • Use `FETCH FIRST n ROWS ONLY` for ANSI-SQL pagination; `LIMIT/OFFSET` for MySQL/PG; `TOP n` for SQL Server • **Never use** `= NULL` — always `IS NULL` / `IS NOT NULL` • Big OFFSET is slow on large tables — prefer **keyset pagination** (`WHERE id > last_seen_id`)

Functions & Aggregation

Compute across rows with aggregates, group data, branch logic with CASE, and handle NULLs gracefully.

Σ Aggregate Functions

FN	RETURNS	NOTES
`COUNT(*)`	Total rows (incl. NULLs)	Most common
`COUNT(col)`	Non-NULL values of col	Skips NULLs
`COUNT(DISTINCT col)`	Unique non-NULL values	Slower; no index help
`SUM(col)`	Sum of numeric values	NULLs ignored; NULL if all NULL
`AVG(col)`	Average (ignores NULLs)	= SUM(col)/COUNT(col), not COUNT(*)
`MIN(col)` / `MAX(col)`	Smallest / largest value	Works on dates & strings too
`GROUP_CONCAT`	Concatenated values	MySQL; PG `STRING_AGG`; SQL Server `STRING_AGG`

```sql
SELECT dept,
       COUNT(*)            AS headcount,
       ROUND(AVG(salary),2) AS avg_sal,
       MIN(salary)         AS min_sal,
       MAX(salary)         AS max_sal
FROM   employees
GROUP BY dept;
```

⊞ GROUP BY & HAVING

```sql
SELECT customer_id, COUNT(*) AS orders
FROM   orders
WHERE  status = 'paid'      -- pre-group filter
GROUP BY customer_id
HAVING  COUNT(*) > 5       -- post-group filter
ORDER BY orders DESC;
```

CLAUSE	WHEN	CAN USE AGGREGATES?
`WHERE`	Before grouping (per row)	No
`HAVING`	After grouping (per group)	Yes

⇆ CASE — Conditional Logic

```sql
-- Searched CASE (more flexible)
SELECT name
```

S Common String Functions

FUNCTION	DESCRIPTION
`UPPER(s)` / `LOWER(s)`	Uppercase / lowercase
`LENGTH(s)`	Char count (`CHAR_LENGTH` ANSI)
`SUBSTRING(s, start, len)`	Substring; ANSI 1-indexed
`CONCAT(a, b, ...)`	Concatenate; `||` in ANSI/PG
`TRIM(s)`	Strip leading/trailing whitespace
`REPLACE(s, from, to)`	Substring replacement
`POSITION(sub IN s)`	Index of sub in s (1-indexed, 0 if not found)
`LEFT(s, n)` / `RIGHT(s, n)`	First/last n characters
`LPAD(s, n, c)` / `RPAD`	Pad to length n with char c

D Date / Time Functions

FUNCTION	DESCRIPTION / DIALECT
`NOW()` / `CURRENT_TIMESTAMP`	Current timestamp (ANSI: `CURRENT_TIMESTAMP`)
`CURRENT_DATE` / `CURDATE()`	Today (ANSI: `CURRENT_DATE` ; `CURDATE` is MySQL)
`EXTRACT(YEAR FROM d)`	Get part of date (ANSI)
`DATEADD(d, n)`	Add interval (PG: `d + INTERVAL '7 days'` ; MySQL: `DATE_ADD(d, INTERVAL 7 DAY)`)
`DATEDIFF(a, b)`	Days between (PG: `a - b` returns interval)
`DATE_TRUNC('month', d)`	Truncate to start of unit (PG); MySQL: `DATE_FORMAT`
`TO_CHAR(d, 'YYYY-MM-DD')`	Format date (PG/Oracle); MySQL: `DATE_FORMAT`

N Numeric Functions

- `ROUND(x, n)` — round to n decimals
- `CEIL(x)` / `FLOOR(x)` — round up/down
- `ABS(x)` — absolute value
- `POWER(x, y)` — x^y
- `MOD(a, b)` — remainder (or `a % b`)
- `SQRT(x)` — square root
- `RANDOM()` — PG; MySQL `RAND()`
- `GREATEST(a,b,c)` / `LEAST(...)`

⌀ NULL Handling: COALESCE & NULLIF

```sql
-- COALESCE: first non-NULL value
SELECT COALESCE(nickname, real_name, 'anonymous')
```

QUICK TIPS `COUNT(*)` includes NULLs; `COUNT(col)` ignores them — choose deliberately • Every non-aggregated column in SELECT must appear in **GROUP BY** (SQL standard; MySQL `ONLY_FULL_GROUP_BY` enforces this) • Use **NULLIF(x, 0)** to safely divide and avoid "division by zero" errors • Positional GROUP BY (`GROUP BY 1, 2`) works in PG/MySQL but is not portable to SQL Server

SQL Joins — Combine Rows from Multiple Tables

Six join types with Venn diagrams, syntax, and sample outputs.

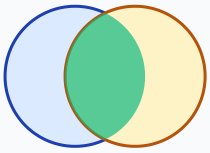
Sample Table: `employees`

EMP_ID	NAME	DEPT_ID
1	Alice	10
2	Bob	20
3	Cara	NULL
4	Dan	30

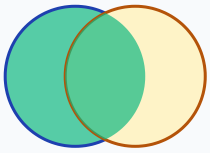
Sample Table: `departments`

DEPT_ID	DEPT_NAME
10	Engineering
20	Sales
40	Marketing

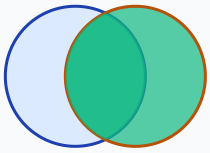
Visual Join Types — Venn Diagrams



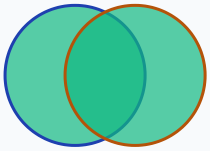
Only matching rows from both tables



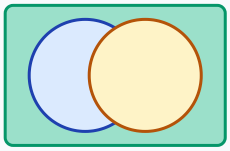
All left rows + matches from right



All right rows + matches from left



All rows from both tables



Cartesian product: every × every

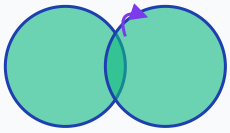


Table joined to itself via aliases

JOIN Syntax — ANSI-92 vs ANSI-89

ANSI-92 (PREFERRED)

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

ANSI-89 (LEGACY)

```sql
SELECT e.name, d.dept_name
FROM employees e, departments d
WHERE e.dept_id = d.dept_id;
```

AVOID ANSI-89 Forgetting the WHERE in ANSI-89 silently produces a **Cartesian product**. ANSI-92 makes the intent explicit and is required for OUTER joins.

Comparison Table

JOIN	ROWS RETURNED	NULLS?
`INNER JOIN`	Only matches	No
`LEFT [OUTER] JOIN`	All left + matches	Right cols NULL when no match
`RIGHT [OUTER] JOIN`	All right + matches	Left cols NULL when no match
`FULL [OUTER] JOIN`	All from both	Yes, on the unmatched side
`CROSS JOIN`	m × n rows	No (no ON clause)
`SELF JOIN`	Same table, twice via alias	Depends on join type used

NOTE **MySQL** does not support `FULL OUTER JOIN` directly — emulate with `LEFT JOIN UNION RIGHT JOIN`. **SQLite** also lacks FULL OUTER (pre-3.39).

Worked Examples

LEFT JOIN

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

SELF JOIN

```sql
-- Find employees and their managers
SELECT e.name AS employee,
       m.name AS manager
FROM   employees e
```

QUICK TIPS **INNER JOIN** = intersection; **LEFT JOIN** = preserve left; **FULL JOIN** = preserve both • To preserve LEFT-join behavior, filter the right table inside `ON`, not `WHERE` • **CROSS JOIN** is intentional; an *unintended* Cartesian product is usually a missing `ON` clause • Always **qualify columns** with table aliases — easier to read and prevents ambiguity errors

Relational Database Concepts

Keys, constraints, and cardinality — the rules that keep relational data consistent.

K Keys — Identifier Hierarchy

KEY	DEFINITION	EXAMPLE
Super Key	Any column set that uniquely identifies a row	{id}, {id, email}, {email, phone}
Candidate Key	A *minimal* super key (no redundant columns)	{id}, {email}
Primary Key (PK)	The chosen candidate key — non-NULL, unique, immutable	`id INT PRIMARY KEY`
Alternate Key	A candidate key that wasn't picked as PK	`email UNIQUE`
Foreign Key (FK)	References a PK in another table	`dept_id REFERENCES departments(id)`
Composite Key	A key made of ≥2 columns	`(order_id, product_id)` in a junction table
Unique Key	Constraint enforcing uniqueness (allows NULLs, unlike PK)	`email VARCHAR UNIQUE`
Surrogate Key	System-generated, no business meaning (e.g. `SERIAL`)	Auto-increment `id`
Natural Key	A key with business meaning (e.g. SSN, ISBN)	ISBN for books

C Constraints

CONSTRAINT	EFFECT & SYNTAX
`PRIMARY KEY`	Unique + not null + indexed. `id INT PRIMARY KEY`
`FOREIGN KEY`	Enforces referential integrity. `FOREIGN KEY (dept_id) REFERENCES departments(id) ON DELETE CASCADE`
`UNIQUE`	No duplicates (NULLs allowed in MySQL/PG; multiple NULLs allowed in MySQL, one in standard SQL).
`CHECK`	Boolean expression must be true. `CHECK (price ⩾ 0)`
`NOT NULL`	Column cannot be NULL.
`DEFAULT`	Value used when not supplied. `DEFAULT 0`

FK ACTIONS `ON DELETE` / `ON UPDATE` can be: `CASCADE` (propagate), `SET NULL`, `SET DEFAULT`, `RESTRICT` (block), `NO ACTION` (default; same as RESTRICT in most engines).

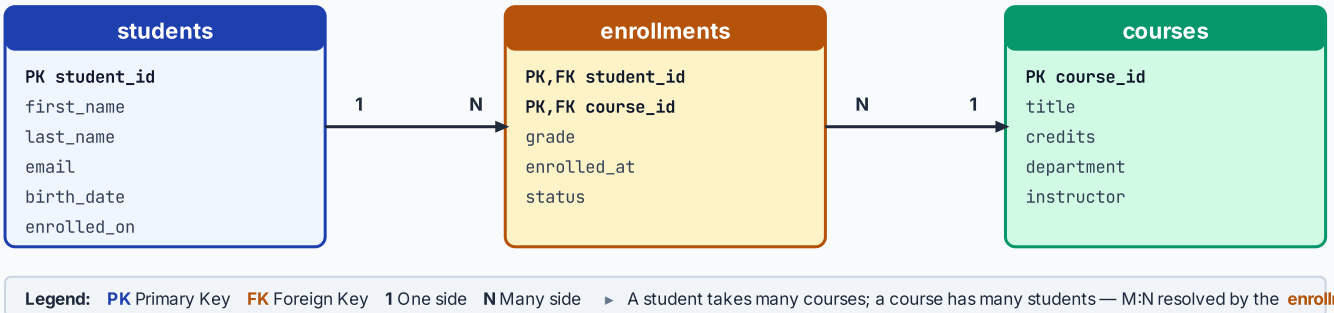
R Relationships & Cardinality

1:1 One-to-One
Each row in A maps to ≤1 row in B (e.g. `users ↔ user_profiles`). Put FK on either side + UNIQUE constraint.

1:N One-to-Many
Most common. One row in A maps to many in B. Put FK on the "many" side. E.g. `customers → orders`.

M:N Many-to-Many
Requires a **junction table** (a.k.a. associative entity) with composite PK referencing both sides. E.g. `students ↔ courses` via `enrollments`.

ER ER Diagram Example — Student-Course Enrollment



Legend: **PK** Primary Key **FK** Foreign Key **1** One side **N** Many side ▸ A student takes many courses; a course has many students — M:N resolved by the enrollmen

QUICK TIPS **Primary Key** = unique + not null + chosen; **Candidate Key** = minimal unique; **Surrogate** = meaningless id • Always index FK columns — most engines do NOT auto-index them (PostgreSQL does NOT; InnoDB does) • Use `ON DELETE RESTRICT` for parents you never want to lose; `CASCADE` only for true children • Resolve M:N with a junction table whose PK is the composite of both FKs — this also prevents duplicate pairs

Advanced SQL

Subqueries, set operations, views, indexes, CTEs, and the powerful window-function family.

[] Subqueries

```sql
-- Scalar (single value)
SELECT * FROM products
WHERE price > (SELECT AVG(price) FROM products);

-- Column (returns multiple rows, one col)
SELECT * FROM orders
WHERE customer_id IN (
    SELECT id FROM customers WHERE country='US'
);

-- Table (returns multiple cols/rows) — derived table
SELECT d.dept_name, t.avg_sal
FROM departments d
JOIN (
    SELECT dept_id, AVG(salary) AS avg_sal
    FROM employees GROUP BY dept_id
) t ON d.dept_id = t.dept_id;
```

⇄ Correlated vs Non-Correlated

TYPE	BEHAVIOR
Non-correlated	Inner query runs *once*; result reused. E.g. `WHERE x > (SELECT AVG(x) FROM t)`
Correlated	Inner query references outer row; re-runs per outer row. Slower but expressive. E.g. find each employee earning above their *department* average.

```sql
SELECT e.name, e.salary, e.dept_id
FROM employees e
WHERE e.salary > (
    SELECT AVG(e2.salary)
    FROM employees e2
    WHERE e2.dept_id = e.dept_id
);
```

∃ EXISTS / ANY / ALL

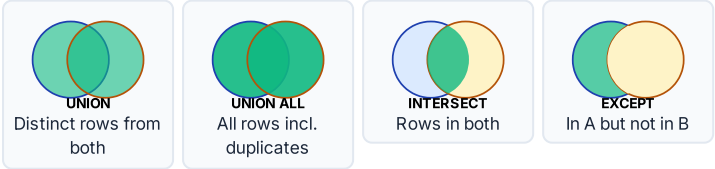
```sql
-- EXISTS: true if subquery returns any rows
SELECT name FROM customers c
WHERE EXISTS (
    SELECT 1 FROM orders o
    WHERE o.customer_id = c.id
);

-- ANY: true if comparison holds for at least one
SELECT * FROM products
WHERE price > ANY (
    SELECT price FROM products WHERE
category='Books'
);

-- ALL: true if comparison holds for every row
SELECT * FROM products
WHERE price > ALL (
    SELECT price FROM products WHERE
category='Books'
);
```

PERFORMANCE For large datasets, `EXISTS` often beats `IN` — it short-circuits on the first match, while `IN` materializes the full list. `NOT EXISTS` beats `NOT IN` especially when NULLs are present.

∪ Set Operations




```sql
SELECT city FROM customers
UNION           -- distinct cities
SELECT city FROM suppliers;

-- MySQL uses EXCEPT (8.0+) / INTERSECT (8.0+)
-- Older MySQL: emulate EXCEPT with LEFT JOIN ... IS NULL
-- Emulate INTERSECT with INNER JOIN + DISTINCT

-- Rules: same # of cols, compatible types,
--        ORDER BY only at the very end
```

QUICK TIPS Prefer **CTE** over nested subqueries — readable, can be referenced multiple times, and recursive • **UNION ALL** is faster than **UNION** — only use UNION when you must dedupe • Window functions compute *over* rows but don't collapse them — perfect for rankings, running totals, time-series shifts • Indexes speed up `WHERE` / `JOIN` / `ORDER BY` but slow down writes — measure, don't guess

Normalization & Transactions

Design clean schemas with normal forms, then protect data integrity with ACID transactions.

N Normalization — Step by Step

1NF Atomic Values

Rule: Each cell holds a single value; no repeating groups.
Bad: `tags = "a,b,c"`
Good: separate `post_tags` table.

2NF No Partial Dependency

Rule: 1NF + every non-key column depends on the *whole* PK (not a part of composite PK).
Bad: `(order_id, product_id) → product_name` (depends only on product_id).

3NF No Transitive Dependency

Rule: 2NF + non-key columns don't depend on other non-key columns.
Bad: `zip → city` in a customers table.
Good: move to `zipcodes` table.

BCNF Boyce-Codd

Rule: 3NF + every determinant is a candidate key. Stricter than 3NF for tables with overlapping candidate keys.

Resolves edge cases 3NF misses.

± Denormalization — When & Why

Denormalization **intentionally** violates normal forms for read performance. Normalized schemas require many JOINs; denormalized ones duplicate data to avoid them. It's a trade-off: faster reads vs. larger storage + harder consistency. Use when reads vastly outnumber writes (analytics, reporting, dashboards) and JOINs become the bottleneck. Re-normalize for OLTP workloads where consistency is paramount.

NORMALIZED	DENORMALIZED
Less duplication	Controlled duplication
More JOINs	Fewer JOINs, faster reads
Easy to update	Updates touch multiple places
Best for OLTP	Best for OLAP / warehousing

A Advantages of Normalization

- **Eliminates data redundancy** — same fact stored in one place only
- **Improves consistency** — no anomaly when one row updates & another doesn't
- **Smaller tables** — less disk + cache pressure
- **Easier schema evolution** — add a column without touching many tables
- **Clearer design** — each table = one entity type

TRADE-OFF Heavy normalization can hurt read-heavy analytical queries. The modern practice: **normalize OLTP, denormalize (or use star schema) for OLAP.**

T Transactions — BEGIN / COMMIT / ROLLBACK / SAVEPOINT

```tcl
BEGIN;  -- start transaction (ANSI: START TRANSACTION)

UPDATE accounts SET balance = balance - 100
  WHERE id = 1;

SAVEPOINT after_debit;

UPDATE accounts SET balance = balance + 100
  WHERE id = 2;

-- oops, undo only the credit:
ROLLBACK TO after_debit;

-- try again, then commit:
UPDATE accounts SET balance = balance + 100
  WHERE id = 2;

COMMIT;   -- or ROLLBACK to undo everything
```

A ACID Properties

A Atomicity

All-or-nothing. If any statement fails, the whole transaction rolls back.

C Consistency

Transaction moves DB from one valid state to another — constraints, triggers, cascades all hold.

I Isolation

Concurrent transactions don't interfere. Visible level depends on isolation level.

D Durability

Once committed, changes survive crashes (written to WAL / redo log on disk).

I Isolation Levels

T Transaction Flow

QUICK TIPS **Normalization mnemonic:** "1NF atomic, 2NF full-key, 3NF only-key, BCNF nothing-but-key" (Kent's rule) • **Default isolation:** MySQL = REPEATABLE READ, PostgreSQL = READ COMMITTED — know your default! • Keep transactions **short** — long-running ones hold locks, block others, and reduce concurrency • `AUTOCOMMIT` is ON by default in most clients — every statement becomes its own transaction unless you `BEGIN`

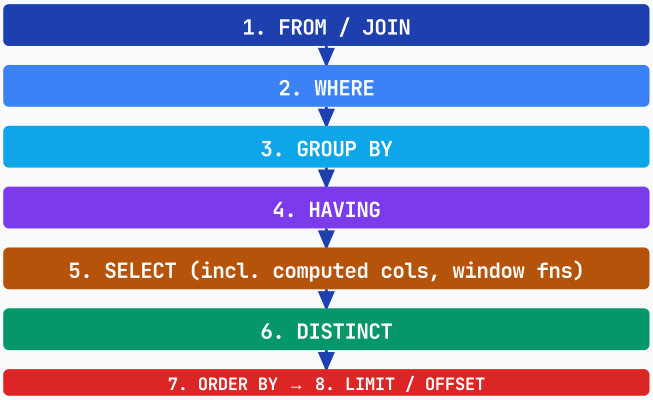
SQL Interview & Cheat Tables

Dense reference: 50 commands, execution order, type comparisons, performance tips, and interview Q&A.

50 Most-Used SQL Commands

COMMAND	PURPOSE
`SELECT`	Read rows
`FROM`	Source table
`WHERE`	Row filter
`GROUP BY`	Aggregate grouping
`HAVING`	Group filter
`ORDER BY`	Sort rows
`LIMIT` / `OFFSET`	Pagination
`DISTINCT`	Unique rows
`INSERT INTO`	Add rows
`UPDATE`	Modify rows
`DELETE FROM`	Remove rows
`CREATE DATABASE`	New database
`CREATE TABLE`	New table
`ALTER TABLE`	Modify schema
`DROP TABLE`	Delete table
`TRUNCATE TABLE`	Empty table fast
`RENAME`	Rename table
`CREATE INDEX`	Make index
`DROP INDEX`	Remove index
`CREATE VIEW`	Virtual table
`JOIN / INNER JOIN`	Match rows
`LEFT / RIGHT JOIN`	Outer join
`FULL OUTER JOIN`	Both sides
`CROSS JOIN`	Cartesian product
`UNION / UNION ALL`	Combine result sets
`INTERSECT / EXCEPT`	Set intersection/diff

SQL Logical Execution Order



WHY Aliases defined in `SELECT` are NOT available to `WHERE` or `GROUP BY` (they run first) — but ARE available to `ORDER BY` (runs last).

Operator Precedence (high → low)

RANK	OPERATORS
1	`()` parentheses
2	`~` (PG regex), unary `-`
3	`*` `/` `%`
4	`+` `-` (binary)
5	`||` string concat
6	comparison: `=` `<>` `<` `>` `IN` `LIKE` `BETWEEN` `IS`

JOIN Quick Compare

JOIN	USE WHEN...
`INNER`	Only matched rows
`LEFT`	All left, even if no match
`RIGHT`	All right (rare; rewrite as LEFT)
`FULL`	Both sides, NULLs for missing
`CROSS`	Pair every row (m×n)

Data Type Comparison

CONCEPT	MYSQL	PG	SQL SRV
Auto-increment	AUTO_INCREMENT	SERIAL / IDENTITY	IDENTITY
Bool	TINYINT(1)	BOOLEAN	BIT
Text unlimited	TEXT/LONGTEXT	TEXT	VARCHAR(MAX)
UUID	CHAR(36)	UUID	UNIQUEIDENTIFIER
Timestamptz	TIMESTAMP	TIMESTAMPTZ	DATETIMEOFFSET
JSON	JSON	JSONB	NVARCHAR(MAX)
Array	—	`int[]`	—

QUICK TIPS **Execution order mnemonic:** "Fat Goats Have Selected Old Lambs" (From, Group, Having, Select, Order, Limit) • Always run `EXPLAIN` before optimizing — measure, never guess • Interview favorite: **"Find 2nd highest X"** — practice with both `LIMIT/OFFSET` and `DENSE_RANK()` approaches • Know your engine's default isolation level — MySQL = RR, PG/Oracle = RC

Ultimate One-Page Revision Sheet

Everything you need in one glance — print this page and stick it on your wall.

CRUD Summary

```
CREATE TABLE t (col type ...);
INSERT INTO t (cols) VALUES (...);
SELECT cols FROM t
  WHERE ...
  ORDER BY ... LIMIT n;
UPDATE t SET col=val WHERE ...;
DELETE FROM t WHERE ...;
DROP TABLE t;  TRUNCATE t;
```

JOIN Summary

INNER	matches only
LEFT	all left + match
RIGHT	all right + match
FULL	all of both
CROSS	m × n
SELF	same table ×2

Aggregate Summary

`COUNT(*)`	rows incl NULL
`COUNT(col)`	non-NULL only
`SUM/AVG`	numeric totals
`MIN/MAX`	bounds
`GROUP BY`	pre-aggregate
`HAVING`	post-aggregate filter

Keys Summary

Super	any unique col set
Candidate	minimal super key
Primary	chosen candidate
Alternate	other candidates
Foreign	refs another PK
Composite	≥2 cols as key
Surrogate	auto-gen id
Natural	business key

Constraints Summary

`PRIMARY KEY`	u + nn
`FOREIGN KEY`	ref integrity
`UNIQUE`	no dup
`CHECK`	expr true
`NOT NULL`	required
`DEFAULT`	fallback

Normalization

1NF	atomic cells
2NF	full-key dep
3NF	only-key dep
BCNF	determinant = candidate

MNEMONIC "Every non-key attr must depend on the key, the whole key, and nothing but the key — so help me Codd."

ACID Summary

A	Atomicity — all or nothing
C	Consistency — valid states
I	Isolation — concurrent safe
D	Durability — survives crash

Window Functions

```
fn() OVER (
  PARTITION BY col
  ORDER BY col
  ROWS BETWEEN ...
)
```

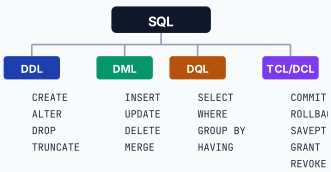
`ROW_NUMBER`	1,2,3 (unique)
`RANK`	1,1,3 (gaps)
`DENSE_RANK`	1,1,2 (no gap)
`LAG/LEAD`	prev/next value
`FIRST_VALUE`	first in frame
`NTILE(n)`	bucket rows

Transactions

```
BEGIN;
  ...SQL ops...
SAVEPOINT sp1;
  ...more ops...
ROLLBACK TO sp1;  -- undo to sp
COMMIT;           -- save all
ROLLBACK;         -- undo all
```

Isolation: RU > RC > RR > S
(more isolation = less concurrency)

Command Hierarchy



Common Mistakes

- UPDATE/DELETE without WHERE
- `= NULL` instead of `IS NULL`
- LEFT JOIN filter in WHERE not ON
- Forgetting ON → Cartesian product
- Wrap col in fn in WHERE → kills index
- SELECT * in production code
- Big OFFSET pagination
- Using FLOAT for money
- Not committing long transactions

Best Practices

- **Be explicit** — name columns in INSERTs, never rely on column order
- **Use transactions** for multi-statement consistency
- **Index strategically** — measure with EXPLAIN, don't guess
- **Use constraints** — let the DB enforce integrity, not app code
- **Prefer ANSI joins** (INNER JOIN ... ON) — clearer than comma syntax
- **Alias everything** in multi-table queries
- **Validate with EXPLAIN ANALYZE** before shipping
- **Back up before** DDL on production tables
- **Use DECIMAL** for money, never FLOAT
- **Keep transactions short** to minimize lock contention

Memory Tricks & Mnemonics

- **ACID** = "All Crashes Isolate Durably" — Atomicity, Consistency, Isolation, Durability
- **SQL execution order**: "Fat Goats Have Selected Old Lambs" → FROM, GROUP BY, HAVING, SELECT, ORDER BY, LIMIT
- **Normalization Kent's rule**: "non-key attrs depend on the key, the whole key, and nothing but the key"
- **JOIN order matters**: "DRY" — Drive (FROM), Restrict (WHERE), Yield (SELECT)
- **NULL arithmetic**: "NULL eats everything" — any op with NULL yields NULL
- **LEFT vs RIGHT**: "Left keeps left, right keeps right — FULL keeps both"
- **UNION vs UNION ALL**: "ALL = All the rows, including dups"
- **RANK vs DENSE_RANK**: "Dense packs them tight (no gaps)"
- **CHAR vs VARCHAR**: "CHAR is Constant, VARCHAR is Variable"
- **WHERE vs HAVING**: "WHERE works on rows, HAVING works on groups"

EXAM DAY Read the question twice. Identify the **tables involved**, the **required output columns**, the **filter conditions**, and the **aggregation level**. Then write FROM first, SELECT last — that's how the engine actually runs it. When stuck, remember: **think in sets, not loops**.

QUICK TIPS This page is the **only one you need** on the morning of an exam or interview • Practice writing queries **by hand** — syntax errors are the #1 interview fail • Re-read pages 5 (Joins) and 7 (Window functions) the night before — they have the most "gotcha" questions • For interviews: **think aloud**, name your tables/aliases clearly, and always test edge cases (NULL, empty, duplicate rows)