SQL / MySQL Cheatsheet

SQL / MySQL Cheatsheet

The 90% you'll use daily

SELECT skeleton
```sql
SELECT [DISTINCT] col AS alias, ...
FROM table t
[JOIN other o ON ...]
[WHERE conditions]
[GROUP BY col, ...]
[HAVING group_condition]
[ORDER BY col [ASC|DESC], ...]
[LIMIT n OFFSET m];   -- MySQL: LIMIT m, n also works
```

Filtering
```sql
WHERE col IS NULL
WHERE col IN (1,2,3)
WHERE col BETWEEN a AND b     -- inclusive
WHERE col LIKE 'abc%'         -- '_' = single char
WHERE col REGEXP '^ab[0-9]+$' -- MySQL regex
```

Joins
```sql
-- inner (match both)
SELECT ... FROM a JOIN b ON a.id = b.a_id;
-- left (keep all from left)
SELECT ... FROM a LEFT JOIN b ON a.id = b.a_id;
-- anti-join (rows in a with no match in b)
SELECT ... FROM a
LEFT JOIN b ON a.id = b.a_id
WHERE b.a_id IS NULL;
```

Aggregates
```sql
SELECT dept, COUNT(*) AS n, AVG(salary) AS avg_sal
FROM emp
GROUP BY dept
HAVING COUNT(*) >= 5;
```

Top N per group (MySQL 8+)
```sql
SELECT * FROM (
  SELECT e.*, ROW_NUMBER() OVER (PARTITION BY dept ORDER BY salary DESC) rn
  FROM emp e
) x WHERE rn <= 3;
```

Upsert (insert or update)
```sql
```

```
INSERT INTO t (id, name, hits)
VALUES (1,'A',1)
ON DUPLICATE KEY UPDATE
  name = VALUES(name),
  hits = hits + 1;
```

Update with join
```sql
UPDATE orders o
JOIN customers c ON o.customer_id = c.id
SET o.flag = 1
WHERE c.vip = 1;
```

Delete with filter
```sql
DELETE FROM t WHERE created_at < NOW() - INTERVAL 30 DAY;
```

Pagination
```sql
SELECT ... FROM t ORDER BY created_at DESC LIMIT 20 OFFSET 0;  -- page 1
-- For large tables prefer keyset pagination:
SELECT ... FROM t
WHERE created_at < :last_seen
ORDER BY created_at DESC
LIMIT 20;
```

Data definition (DDL)

Create table (InnoDB is default)
```sql
CREATE TABLE users (
  id BIGINT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
  email VARCHAR(255) NOT NULL UNIQUE,
  name VARCHAR(100),
  is_active TINYINT(1) NOT NULL DEFAULT 1, -- BOOLEAN alias
  created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP NULL DEFAULT NULL ON UPDATE CURRENT_TIMESTAMP,
  CONSTRAINT email_chk CHECK (email <> '')
) ENGINE=InnoDB;
```

Indexes
```sql
CREATE INDEX idx_user_email ON users(email);
CREATE INDEX idx_orders_user_created ON orders(user_id, created_at); -- composite (leftmost
rule)
-- Prefix index for long strings:
CREATE INDEX idx_title_prefix ON articles(title(50));
```

Foreign keys
```sql
CREATE TABLE orders (
  id BIGINT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
  user_id BIGINT UNSIGNED NOT NULL,
  amount DECIMAL(10,2) NOT NULL,
  CONSTRAINT fk_orders_user
    FOREIGN KEY (user_id) REFERENCES users(id)
    ON UPDATE CASCADE ON DELETE RESTRICT
);
```

Altering
```sql
ALTER TABLE users ADD COLUMN last_login DATETIME NULL;
ALTER TABLE users MODIFY name VARCHAR(150) NOT NULL;
ALTER TABLE users DROP COLUMN is_active;
```

Transactions & locking (InnoDB)
```sql
SET autocommit = 0;
START TRANSACTION;
-- do work
COMMIT;      -- or ROLLBACK;

-- Savepoints
SAVEPOINT s1;
-- ...
ROLLBACK TO s1;
```

Isolation (default: REPEATABLE READ)
```sql
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED; -- per-session
```

Gotchas
- Write sets **row locks**; range predicates can take **gap locks** (phantoms protection).
- Always access rows in the same order across transactions to avoid deadlocks.

MySQL-specific quirks (know these)
- No `FULL OUTER JOIN` → emulate with `LEFT JOIN … WHERE right IS NULL` + `UNION` `RIGHT JOIN … WHERE left IS NULL`.
- `LIMIT` supports `LIMIT offset, count`.
- `BOOL/BOOLEAN` is `TINYINT(1)` under the hood.
- `DATETIME` vs `TIMESTAMP`: `TIMESTAMP` is timezone-aware & auto-updating, smaller range; `DATETIME` is wider range, no TZ auto-magic.
- SQL modes matter (`ONLY_FULL_GROUP_BY`, `STRICT_TRANS_TABLES`). With `ONLY_FULL_GROUP_BY`, every selected non-aggregated column must be in `GROUP BY`.

- Backticks `like_this` escape identifiers.
- Version features:
  - MySQL 8.0+: **CTEs**, **recursive CTEs**, **window functions**, **common JSON functions**, `EXPLAIN ANALYZE`.
  - MySQL 5.7: JSON type exists, no windows/CTEs.

Readability & performance checklist
- □ Select explicit columns (avoid `SELECT *`).
- □ Make predicates **sargable** (no functions on indexed columns in WHERE/ON).
- □ Build composite indexes that match your **WHERE** and **ORDER BY** (leftmost prefix).
- □ Use covering indexes when possible (all needed columns from index).
- □ Filter early (WHERE) and group late; avoid unnecessary `DISTINCT`.
- □ Inspect plans: `EXPLAIN SELECT ...;` (use `EXPLAIN ANALYZE` on 8.0+ to see actual timings).
- □ Use proper types (no `VARCHAR(255)` everywhere; use `INT`/`BIGINT`, `DATE`, `ENUM` when appropriate).
- □ For large imports: `LOAD DATA [LOCAL] INFILE ...` (much faster than many INSERTs).

Built-ins you'll reach for

Dates
```sql
SELECT NOW(), CURDATE(), DATE(created_at),
       DATE_ADD(NOW(), INTERVAL 7 DAY),
       TIMESTAMPDIFF(DAY, start_at, end_at),
       DATEDIFF(CURDATE(), joined_on);
```

Strings
```sql
SELECT CONCAT(first,' ',last),
       SUBSTRING(title,1,50), TRIM(name),
       LOWER(email), REPLACE(txt,'foo','bar');
```

Conditionals & NULL
```sql
SELECT IF(score >= 50, 'pass','fail'),
       CASE WHEN x IS NULL THEN 0 ELSE x END,
       COALESCE(numeric_col, 0) AS value;
```

JSON (5.7+)
```sql
SELECT JSON_EXTRACT(attrs, '$.color') AS color,
       attrs->'$.color' AS color2,      -- same as above
       attrs->>'$.color' AS color_text, -- unquoted
       JSON_SET(attrs, '$.size', 'M')   -- returns new JSON
FROM products;
-- Index JSON via generated columns:
ALTER TABLE products
  ADD COLUMN color VARCHAR(20) GENERATED ALWAYS AS (JSON_UNQUOTE(attrs->'$.color')) STORED,
```

```
  ADD INDEX idx_color (color);
```

Common patterns (copy/paste)

Find duplicates
```sql
SELECT email, COUNT(*) c
FROM users
GROUP BY email
HAVING c > 1;
```

Delete duplicates (keep smallest id)
```sql
DELETE u FROM users u
JOIN (
  SELECT MIN(id) keep_id, email FROM users GROUP BY email
) k ON u.email = k.email AND u.id <> k.keep_id;
```

Running total (8.0+)
```sql
SELECT order_id, created_at, amount,
       SUM(amount) OVER (ORDER BY created_at) AS running_total
FROM orders;
```

Percentiles (approximate)
```sql
SELECT PERCENTILE_CONT(0.9) WITHIN GROUP (ORDER BY amount) OVER () AS p90
FROM orders;  -- 8.0. Not in older versions.
```

Pivot-lite with conditional aggregates
```sql
SELECT
  SUM(status='paid')   AS paid,
  SUM(status='failed') AS failed,
  SUM(status='refunded') AS refunded
FROM payments;
```

Rolling 30-day active users
```sql
SELECT
  DATE(event_time) AS d,
  COUNT(DISTINCT user_id) AS dau,
  COUNT(DISTINCT CASE WHEN event_time >= NOW() - INTERVAL 30 DAY THEN user_id END) AS
mau_rolling
FROM events
GROUP BY d;
```

Security & admin basics

Users & grants
```sql
CREATE USER 'app'@'%' IDENTIFIED BY 'strongpassword';
GRANT SELECT, INSERT, UPDATE, DELETE ON mydb.* TO 'app'@'%';
FLUSH PRIVILEGES;  -- not always needed but safe
```

Backups
```bash
mysqldump -u root -p --single-transaction mydb > mydb.sql
# Restore:
mysql -u root -p mydb < mydb.sql
```

Peek at what’s happening
```sql
SHOW PROCESSLIST;
SHOW ENGINE INNODB STATUS\G
SHOW GLOBAL STATUS LIKE 'Threads_connected';
SELECT * FROM information_schema.tables WHERE table_schema='mydb' ORDER BY data_length DESC;
```

Normalization (quick)
- 1NF: atomic columns; no arrays/CSV in a cell.
- 2NF: every non-key depends on the whole key.
- 3NF: no transitive dependencies (non-key → non-key).

De-normalize *deliberately* for read performance once you know the access patterns.

Joins: CROSS JOIN vs INNER/LEFT and ON vs WHERE

Here’s the clean, practical breakdown.

CROSS JOIN vs (INNER/LEFT) JOIN

* **CROSS JOIN**
  Cartesian product: every row of A with every row of B. No join condition.

  ```sql
  SELECT * FROM A CROSS JOIN B;   -- n(A) * n(B) rows
  -- same as:
  -- SELECT * FROM A, B;
  ```

* **INNER JOIN**
  Only rows that **match** the ON condition.

  ```sql
  SELECT *
  FROM A
  JOIN B ON A.key = B.key;
  ```

* **LEFT JOIN**
  Keep **all rows from A**, match B when possible; if no match, B’s columns are NULL.

  ```sql
  SELECT *
  FROM A
  LEFT JOIN B ON A.key = B.key;
  ```

When to use CROSS JOIN?
When you *want* all combinations, e.g., building a grid like **Students × Subjects** before counting exams.

> Note: In MySQL, writing `CROSS JOIN ... ON ...` is parsed like an `INNER JOIN`. Stick to: `CROSS JOIN` (no ON) for cartesian product, and `INNER/LEFT JOIN ... ON ...` for matches.

ON vs WHERE (the rules that matter)

1) INNER JOIN: ON vs WHERE are equivalent
You can put the join predicate in `ON` **or** in `WHERE` (it yields the same result).

```sql
-- A
SELECT * FROM A JOIN B ON A.key = B.key;

-- B (equivalent for INNER JOIN)
SELECT * FROM A CROSS JOIN B
WHERE A.key = B.key;
```

2) LEFT JOIN: ON vs WHERE are **not** equivalent
Putting filters on the right table in the **WHERE** clause can turn a LEFT JOIN into an INNER JOIN by eliminating the NULL-extended rows.

Correct (keeps A rows with no match in B):
```sql
SELECT *
FROM A
LEFT JOIN B
  ON A.key = B.key          -- define the match here
 AND B.status = 'active'    -- right-table filter belongs here for LEFT JOIN
WHERE A.created_at >= '2025-01-01';   -- final result filter (left-side ok)
```

Wrong (accidentally becomes INNER JOIN):
```sql
SELECT *
```

```
FROM A
LEFT JOIN B ON A.key = B.key
WHERE B.status = 'active';  -- this removes NULL B rows -> no longer left join
```

3) Anti-join pattern (find rows in A with **no** match in B)
```sql
SELECT A.*
FROM A
LEFT JOIN B ON A.key = B.key
WHERE B.key IS NULL;        -- keep only A rows with no matching B
```
(Any additional right-side matching criteria belong in the **ON**, not WHERE.)

Quick rules of thumb
* Use **CROSS JOIN** only when you explicitly need **all combinations**.
* Use **INNER JOIN** to keep only matching pairs.
* Use **LEFT JOIN** to keep all left rows (even when unmatched).
* For **LEFT JOIN**:
  * Put right-table conditions in **ON**.
  * Put final-result filters (usually on left table) in **WHERE**.
* For **INNER JOIN**, `ON` vs `WHERE` doesn’t matter for correctness, but keep **join predicates in ON** and **post-join filters in WHERE** for clarity.

Mini join diagrams (conceptual)

```
CROSS JOIN (Cartesian product)
A × B = every combination
[A1] [A2]
  |    |
 [B1] [B2]
Pairs: (A1,B1), (A1,B2), (A2,B1), (A2,B2), ...

INNER JOIN (A ∩ B)
[   A   (overlap)   B   ]
Only rows where keys match on both sides.

LEFT JOIN (A ▷ B)
[   A   (overlap)   B   ]
Keeps all A rows. If no match in B, B columns are NULL.

LEFT ANTI-JOIN (A \ B) via LEFT JOIN + WHERE B.key IS NULL
[   A   (   )   B   ]
Keeps only A rows that have no matching B.
```