# SQL Cheat Sheet

## 1. Core syntax & execution order

Written order ≠ execution order. Numerals = run order.

**Palette:** 1 2 3 4 5 6 7

```
⑤ SELECT   col1, col2, agg_func(col3) AS result_name
① FROM     table_name
② WHERE    some_condition
③ GROUP BY col1, col2
④ HAVING   aggregate_condition
⑥ ORDER BY col1 DESC
⑦ LIMIT    10;
```

Alias from SELECT not usable in WHERE/GROUP BY (run earlier); is usable in ORDER BY.

## 2. Multi-table chain (JOIN hops)

*Q: Column lives N joins away?*

```
③ SELECT c.name AS city, COUNT(*) AS n_rides
① FROM   rides r
      JOIN   requests req    ON r.request_id = req.request_id
      JOIN   neighborhoods n ON req.pickup_nbhd_id = n.nbhd_id
      JOIN   cities c        ON n.city_id = c.city_id
② GROUP BY c.name
④ ORDER BY n_rides DESC;
```

**Read aliases as a path:** r (rides) → req (requests) → n (neighborhoods) → c (cities).

Build it incrementally: run after each hop and check row count before adding the next.

*Q1: Distinct drivers per rider?*

```
③ SELECT req.rider_id, COUNT(DISTINCT r.driver_id) AS n_uniq
① FROM   rides r JOIN requests req ON r.request_id = req.request_id
② GROUP BY req.rider_id
④ ORDER BY n_uniq DESC
⑤ LIMIT 5;
```

Trick: `COUNT(DISTINCT col)` after a JOIN dedupes the multiplied rows.

*Q2: Driver tenure at each ride?*

```
② SELECT r.ride_id, ROUND(julianday(r.started_at)−julianday(d.signup_date)) AS
① FROM   rides r JOIN drivers d ON r.driver_id = d.driver_id
③ LIMIT 5;
```

SQLite date diff: `julianday(a)-julianday(b)` = days. Postgres: `a::date - b::date`.

*Q3: Rider-driver pairs with ≥3 5-star rides?*

```
⑤ SELECT req.rider_id, r.driver_id, COUNT(*) AS n_5star
① FROM   rides r JOIN requests req ON r.request_id = req.request_id
② WHERE  r.rider_rating = 5
③ GROUP BY req.rider_id, r.driver_id
④ HAVING COUNT(*) ≥ 3
⑥ ORDER BY n_5star DESC
⑦ LIMIT 5;
```

**WHERE** = filter rows pre-agg; **HAVING** = filter groups post-agg. Alias `n_5star` can't be used in HAVING (runs before SELECT) — repeat the expression.

*Q4: Previous ride per driver? (self-join)*

```
③ SELECT r1.ride_id, r1.driver_id, r1.started_at, MAX(r2.started_at) AS prev
① FROM   rides r1
   LEFT JOIN rides r2 ON  r1.driver_id = r2.driver_id
                      AND r2.started_at < r1.started_at
② GROUP BY r1.ride_id
④ LIMIT 5;
```

Trick: alias same table twice (r1/r2). Inequality in ON narrows r2 to "earlier same-driver rides"; `MAX` picks the most recent. First ride per driver → `NULL` (LEFT JOIN keeps it). Cleaner via `LAG()` in M5.

## 3. Aggregates & dates

*Q5: Monthly conversion rate?*

```
③ SELECT strftime('%Y-%m', requested_at) AS month,
         ROUND(1.0*SUM(accepted)/COUNT(*),3) AS rate
① FROM   requests
② GROUP BY month
④ ORDER BY month
⑤ LIMIT 6;
```

SQLite month truncation: `strftime('%Y-%m', date)` (Postgres: `date_trunc('month', d)`). Rate from boolean: `1.0*SUM(bool)/COUNT(*)` — the `1.0*` forces float (else integer division = 0).

## 4. Window functions

*Q6: Top-1 per group (highest-fare ride per driver)?*

```
WITH ranked AS (
  SELECT driver_id, ride_id, fare_usd,
         ROW_NUMBER() OVER (PARTITION BY driver_id ORDER BY fare_usd DESC) AS rk
  FROM rides)
③ SELECT driver_id, ride_id, fare_usd
① FROM   ranked
② WHERE  rk = 1
④ ORDER BY fare_usd DESC
⑤ LIMIT 5;
```

Top-N per group recipe: `ROW_NUMBER() OVER (PARTITION BY <grp> ORDER BY <col> DESC)` in a CTE, then filter `rk <= N`. Use `RANK` / `DENSE_RANK` instead to keep ties.

*Q7: Rolling 7-day average?*

```
WITH daily AS (SELECT date(started_at) AS day, COUNT(*) AS n
               FROM rides GROUP BY day)
② SELECT day, ROUND(AVG(n) OVER (ORDER BY day
                 ROWS BETWEEN 6 PRECEDING AND CURRENT ROW), 1) AS r7
① FROM   daily
③ ORDER BY day
④ LIMIT 7;
```

Pre-aggregate (CTE) to 1 row per day so the window's `ROWS BETWEEN N PRECEDING AND CURRENT ROW` = N+1 days. SQLite supports `ROWS` but not `RANGE BETWEEN INTERVAL`.

*Q8: Bottom 10% per city by accept rate?*

```
WITH deciled AS (
  SELECT *, NTILE(10) OVER (PARTITION BY city_id ORDER BY accept_rate) AS d
  FROM driver_city)
③ SELECT *
① FROM   deciled
② WHERE  d = 1
④ LIMIT 5;
```

`NTILE(k)` splits each partition into $k$ equal-size buckets numbered 1..k. Bottom decile = bucket 1; top decile = bucket 10. One-liner alternative to a correlated subquery.

## 5. Subqueries & CTEs

*Q9: Riders active in ≥N distinct weeks?*

```
WITH active AS (
  SELECT DISTINCT rider_id, strftime('%Y-%W', requested_at) AS wk
  FROM requests
  WHERE date(requested_at) BETWEEN '2025-12-01' AND '2025-12-31')
④ SELECT rider_id, COUNT(DISTINCT wk) AS n_wk
① FROM   active
② GROUP BY rider_id
③ HAVING COUNT(DISTINCT wk) ≥ 4
⑤ LIMIT 3;
```

CTE deduplicates to one row per (rider, week); outer HAVING filters groups whose distinct-week count clears the bar. Repeat the expression in HAVING (alias `n_wk` not yet defined when HAVING runs).

*Q10: Conversion funnel via CTE chain (Dec requests → rides → 5-star)?*

```
WITH dec_req AS (
  SELECT request_id FROM requests
  WHERE date(requested_at) BETWEEN '2025-12-01' AND '2025-12-31'),
     dec_rides AS (
  SELECT dr.*, r.ride_id, r.rider_rating
  FROM dec_req dr LEFT JOIN rides r ON dr.request_id = r.request_id)
② SELECT COUNT(*) AS n_req,
          SUM(ride_id IS NOT NULL) AS n_done,
          SUM(rider_rating = 5) AS n_5star
① FROM   dec_rides;
```

Each CTE = one named step. `LEFT JOIN` keeps unmatched requests (NULL ride_id → counted as "not completed"). `SUM(bool)` = count of true rows (SQLite/Postgres treat booleans as 0/1).

## 6. Capstone — City Dashboard (M1+M2+M3+M4+M5)

**Computes:** per-city ride count, avg fare, and surge-ride share for cities with ≥500 rides since 2025-06-01 — ranked by avg fare descending. Same query as M5 #16; one CTE, every clause earns its keep.

```
WITH city_stats AS (
  M4:  CTE wraps the prep
  ⑤ SELECT c.name AS city, COUNT(*) AS n_rides,
  M1:  project + COUNT aggregate
    ROUND(AVG(r.fare_usd), 2) AS avg_fare,
  M1:  AVG aggregate
    SUM(CASE WHEN r.surge_mult > 1 THEN 1 ELSE 0 END) AS n_surged
  M3:  CASE WHEN conditional count
  ① FROM rides r
  M2:  alias rides
  JOIN requests req ON r.request_id = req.request_id
  M2:  chain hop 1
  JOIN neighborhoods n ON req.pickup_nbhd_id = n.nbhd_id
  M2:  chain hop 2
  JOIN cities c ON n.city_id = c.city_id
  M2:  chain hop 3
  ② WHERE date(r.started_at) ≥ '2025-06-01'
  M1:  filter rows (pre-agg)
  ③ GROUP BY c.name
  M3:  1 row per city
  ④ HAVING COUNT(*) > 500
  M3:  filter groups (post-agg)
)
② SELECT city, n_rides, avg_fare, n_surged,
  outer:  project CTE columns
  RANK() OVER (ORDER BY avg_fare DESC) AS fare_rank,
  M5:  window rank
  ROUND(1.0 * n_surged / n_rides, 3) AS surge_pct
  M1:  rate (1.0* forces float div)
① FROM city_stats
  M4:  read the CTE
③ ORDER BY fare_rank;
  M1:  final sort
```