Alfonso Hidalgo
Data Analyst
SQL Study Guide

Patterns, exercises & solutions for SQL good praxis

Core Decision Framework

Question you're asking	Pattern to use
Need fewer rows (total / count / avg per group)?	GROUP BY + aggregate
Need analytics but keep all rows?	WINDOW FUNCTIONS
Need best / latest / first row per group?	ROW_NUMBER() in subquery
Need to compare with previous row?	LAG() / LEAD()
Need running total / cumulative?	SUM() OVER (ORDER BY ...)
Need rolling average?	AVG() OVER (ROWS BETWEEN ...)
Need to find duplicates?	GROUP BY + HAVING COUNT(*) > 1
Need to combine tables?	JOIN (pick the right type)
Need conditional output?	CASE WHEN
Need to find "no match" rows?	LEFT JOIN + WHERE ... IS NULL

Contents

1. CTEs & Subqueries

CTEs (Common Table Expressions) make complex queries readable and reusable. Use them to break problems into logical steps — each CTE is a named temporary result.

1.1 — Basic CTE with WITH

Problem: Find users whose total spending exceeds the overall average.

```
-- step 1: total spend per user + overall avg via window function
with user_totals as (
    select
        user_id,
        sum(amount) as total_spend,
        avg(sum(amount)) over () as overall_avg  -- avg across all groups
    from transactions
    group by user_id
)

-- step 2: filter — no subquery needed, avg is already a column
select user_id, total_spend
from user_totals
where total_spend > overall_avg
```

> ***Key Takeaway:*** *AVG(...) OVER () with an empty OVER() computes the metric across ALL rows, making it available as a column. This eliminates the need for a subquery referencing the same CTE. You can nest window functions on top of aggregates: avg(sum(col)) over ().*

1.2 — Chained CTEs

Problem: Find the department with the highest average salary, then list all employees in that department.

```
-- step 1: average salary per department
with dept_avg as (
    select dept, avg(salary) as avg_sal
    from employees
    group by dept
),

-- step 2: find the top department
top_dept as (
    select dept
    from dept_avg
    order by avg_sal desc
    limit 1
)

-- step 3: get employees from that department
select e.name, e.salary
from employees e
inner join top_dept t on e.dept = t.dept
order by e.salary desc
```

> ***Key Takeaway:*** *Chain CTEs with commas (no repeated WITH). Each CTE can reference the ones defined before it. This is the cleanest way to solve multi-step problems.*

1.3 — Window Function Instead of Correlated Subquery

Problem: Find employees who earn more than their department average. A correlated subquery would recompute avg per row — use a window function instead.

```
-- compute dept avg on the fly, then filter in outer query
with emp_with_avg as (
    select
        name, dept, salary,
        avg(salary) over (partition by dept) as dept_avg
    from employees
)

select name, dept, salary, dept_avg
```

```
from emp_with_avg
where salary > dept_avg
```

> ***Key Takeaway:*** *AVG(...) OVER (PARTITION BY dept) computes the department average for every row without collapsing them. Wrap in a CTE so you can filter on it in the WHERE clause. This replaces slow correlated subqueries with a single-pass window function.*

2. JOINs

JOINs combine rows from two or more tables based on a related column. Choosing the right JOIN type is critical — the wrong one silently gives wrong results.

2.1 — JOIN Types at a Glance

Type	Keeps	Use when
INNER JOIN	Only matching rows from both	You need data that exists in both tables
LEFT JOIN	All left rows + matching right	You need all left rows, even unmatched ones
RIGHT JOIN	All right rows + matching left	Reverse of LEFT (rarely used, restructure instead)
FULL OUTER JOIN	All rows from both sides	You need everything, matched or not
CROSS JOIN	Every left row × every right row	Generate all combinations (rare)

2.2 — INNER JOIN

```
-- only users who have at least one transaction
select u.user_id, u.name, t.amount
from users u
inner join transactions t
    on u.user_id = t.user_id
```

2.3 — LEFT JOIN (the most important one)

When to use: Whenever the question asks for 'all users' or 'including those without...' — you need all rows from the left table regardless of matches.

```
-- all users, including those with no transactions
select u.user_id, u.name, t.amount
from users u
left join transactions t
    on u.user_id = t.user_id
-- users with no txns will have NULL in t.amount
```

2.4 — Anti-Join (Find "No Match" Rows)

Problem: Find users with NO transactions. This is the #1 Data Analyst SQL question pattern.

```
-- approach 1: LEFT JOIN + IS NULL (most common)
select u.user_id
from users u
left join transactions t
    on u.user_id = t.user_id
where t.user_id is null

-- approach 2: NOT EXISTS (often faster)
select u.user_id
from users u
where not exists (
    select 1
    from transactions t
    where t.user_id = u.user_id
)

-- approach 3: NOT IN (careful with NULLs!)
select user_id
from users
where user_id not in (
    select distinct user_id from transactions
)
```

> ***Watch Out:*** *NOT IN fails silently if the subquery returns any NULL values — the entire result becomes empty. Always prefer LEFT JOIN + IS NULL or NOT EXISTS for anti-joins.*

2.5 — Self-Join

Problem: Compare rows within the same table — e.g., find employees who earn more than their manager.

```
-- join employees to itself: e = employee, m = manager
select e.name as employee, e.salary,
       m.name as manager, m.salary as mgr_salary
from employees e
inner join employees m
    on e.manager_id = m.employee_id
where e.salary > m.salary
```

2.6 — Multi-Table JOIN Chain

Problem: Combine 3+ tables — e.g., countries → users → transactions.

```
-- chain joins left to right
select c.country, u.user_id, sum(t.amount) as total_spend
from countries c
inner join users u on c.country_id = u.country_id
inner join transactions t on u.user_id = t.user_id
group by c.country, u.user_id
```

3. GROUP BY & Aggregation

3.1 — Basic Aggregation

```sql
-- total, count, average per group
select
    dept,
    count(*) as headcount,          -- total rows
    count(distinct salary) as unique_salaries, -- unique values
    sum(salary) as total_salary,
    avg(salary) as avg_salary,
    min(salary) as min_salary,
    max(salary) as max_salary
from employees
group by dept
```

> ***Key Takeaway:*** *Every column in SELECT must either be in GROUP BY or inside an aggregate function. COUNT(*) counts all rows; COUNT(column) counts non-NULL values only.*

3.2 — HAVING (Filter After Aggregation)

Problem: Find departments with more than 5 employees and average salary above 80k.

```sql
select dept, count(*) as headcount, avg(salary) as avg_sal
from employees
group by dept
-- HAVING filters AFTER aggregation (WHERE filters BEFORE)
having count(*) > 5
   and avg(salary) > 80000
```

> ***Watch Out:*** *WHERE filters individual rows BEFORE grouping. HAVING filters groups AFTER aggregation. You cannot use aliases in HAVING on MySQL — repeat the aggregate expression.*

3.3 — Detect Duplicates

```sql
-- find emails that appear more than once
select email, count(*) as occurrences
from users
group by email
having count(*) > 1
```

3.4 — Conditional Aggregation

Problem: Count or sum different subsets in a single query without multiple WHERE clauses.

```sql
-- count active vs inactive users in one query
select
    count(case when status = 'active' then 1 end) as active_count,
    count(case when status = 'inactive' then 1 end) as inactive_count,
    sum(case when amount > 100 then amount else 0 end) as high_value_total
from users u
left join transactions t on u.user_id = t.user_id
```

> ***Key Takeaway:*** *CASE inside COUNT counts only matching rows (non-NULL). CASE inside SUM lets you sum conditionally. This avoids multiple passes over the data.*

4. Window Functions

Window functions compute values across rows related to the current row WITHOUT collapsing them into groups. Every row in the result keeps its identity.

4.1 — ROW_NUMBER (Top N per Group)

Problem: Find the top 3 earners per department.

```
-- MySQL/PostgreSQL do not support QUALIFY
-- use a subquery or CTE instead
with ranked as (
    select
        dept,
        name,
        salary,
        row_number() over (
            partition by dept     -- reset numbering per dept
            order by salary desc  -- highest salary = 1
        ) as rn
    from employees
)

select dept, name, salary
from ranked
where rn <= 3  -- top 3 per department
```

> ***Platform Note:*** *QUALIFY is Snowflake/BigQuery syntax — NOT supported on MySQL/PostgreSQL. Always use a CTE + WHERE rn = 1 pattern instead.*

4.2 — ROW_NUMBER vs RANK vs DENSE_RANK

Function	Ties	Gaps	Example (salaries: 100, 100, 90)
ROW_NUMBER()	Each gets unique #	N/A	1, 2, 3
RANK()	Same rank for ties	Gaps after ties	1, 1, 3
DENSE_RANK()	Same rank for ties	No gaps	1, 1, 2

> ***Key Takeaway:*** *Use ROW_NUMBER when you need exactly N rows per group (even with ties). Use DENSE_RANK when ties should share the same rank without gaps. Use RANK when gaps after ties are acceptable.*

4.3 — LAG / LEAD (Compare With Previous/Next Row)

Problem: Calculate day-over-day price change.

```
select
    date,
    price,
    -- LAG: get the previous row's value
    lag(price) over (order by date) as prev_price,
    -- compute the change
    price - lag(price) over (order by date) as daily_change,
    -- LEAD: get the next row's value
    lead(price) over (order by date) as next_price
from prices
```

LAG(col, n, default) looks n rows BACK (default 1). LEAD(col, n, default) looks n rows FORWARD. The optional default avoids NULL for the first/last row.

4.4 — Running Total / Cumulative Sum

```
select
    date,
    revenue,
    -- running total: sum of all rows up to and including current
    sum(revenue) over (order by date) as cumulative_revenue,
    -- running count
    count(*) over (order by date) as running_count
```

```
from sales
```

4.5 — Rolling Average

```
select
    date,
    price,
    -- 3-day moving average: current row + 2 preceding
    avg(price) over (
        order by date
        rows between 2 preceding and current row
    ) as ma_3
from prices
```

Window frame options: ROWS BETWEEN x PRECEDING AND y FOLLOWING, ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW (default for running totals), ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING (centered window).

4.6 — Percent of Total / Percent Rank

```
-- each employee's salary as % of department total
select
    name, dept, salary,
    round(
        salary * 100.0 /
        sum(salary) over (partition by dept),
        2
    ) as pct_of_dept
from employees
```

5. CASE WHEN

5.1 — Conditional Column

Problem: Categorize employees into salary bands.

```
select
    name,
    salary,
    case
        when salary >= 100000 then 'Senior'
        when salary >= 70000  then 'Mid'
        else 'Junior'
    end as band  -- evaluated top-to-bottom, first match wins
from employees
```

> ***Watch Out:*** *CASE evaluates conditions top-to-bottom and stops at the first TRUE. Put the most specific/restrictive condition first to avoid incorrect matches.*

5.2 — Pivot via CASE

Problem: Turn row values into columns — e.g., show revenue per quarter as separate columns.

```
-- manual pivot: each CASE creates one column
select
    product,
    sum(case when quarter = 'Q1' then revenue else 0 end) as q1,
    sum(case when quarter = 'Q2' then revenue else 0 end) as q2,
    sum(case when quarter = 'Q3' then revenue else 0 end) as q3,
    sum(case when quarter = 'Q4' then revenue else 0 end) as q4
from sales
group by product
```

> ***Key Takeaway:*** *SQL has no built-in PIVOT on MySQL/PostgreSQL. Use SUM(CASE WHEN ... END) to pivot manually. This pattern appears frequently in SQL exercises.*

6. NULL Handling

```
-- ■■ IS NULL / IS NOT NULL: test for missing values ■■
select * from users where email is null
select * from users where email is not null

-- ■■ COALESCE: return first non-NULL value ■■
-- fallback chain: try col1, then col2, then default
select coalesce(nickname, first_name, 'Unknown') as display_name
from users

-- ■■ COALESCE in aggregation: replace NULL with 0 ■■
select
    u.user_id,
    coalesce(sum(t.amount), 0) as total_spend
from users u
left join transactions t on u.user_id = t.user_id
group by u.user_id

-- ■■ NULLIF: return NULL if two values are equal ■■
-- useful to avoid division by zero
select revenue / nullif(cost, 0) as margin
from sales
-- if cost = 0, nullif returns NULL instead of error

-- ■■ IFNULL (MySQL) / COALESCE (standard) ■■
select ifnull(email, 'no email') from users  -- MySQL only
select coalesce(email, 'no email') from users -- works everywhere
```

> ***Key Takeaway:*** *COALESCE is the universal NULL handler — works on all platforms. Use it after LEFT JOINs to replace NULL aggregation results with 0. NULLIF prevents division-by-zero errors.*

7. String & Date Functions

7.1 — String Functions

```
-- concatenation
select concat(first_name, ' ', last_name) as full_name  -- MySQL
select first_name || ' ' || last_name as full_name       -- PostgreSQL

-- substring
select substring(name, 1, 3) as first_3_chars  -- 'Ana' from 'Anastasia'

-- length
select length(name) as name_len

-- case conversion
select upper(name), lower(name)

-- trim whitespace
select trim(name), ltrim(name), rtrim(name)

-- pattern matching
select * from users where name like 'A%'   -- starts with A
select * from users where name like '%son' -- ends with 'son'
select * from users where name like '_a%'  -- 2nd char is 'a'
```

7.2 — Date Functions

```
-- ■■ Current date/time ■■
select current_date, current_timestamp, now()

-- ■■ Extract parts ■■
select extract(year from order_date) as yr   -- PostgreSQL
select year(order_date) as yr                 -- MySQL
select extract(month from order_date) as mo
select extract(dow from order_date) as day_of_week  -- PG: 0=Sun
```

```sql
-- ■■ Date difference ■■
select datediff(end_date, start_date) as days_diff   -- MySQL
select end_date - start_date as days_diff             -- PostgreSQL

-- ■■ Add/subtract intervals ■■
select order_date + interval '7 days'        -- PostgreSQL
select date_add(order_date, interval 7 day)  -- MySQL

-- ■■ Truncate to period ■■
select date_trunc('month', order_date) as month_start  -- PostgreSQL
```

> ***Platform Note:*** *Your SQL environment may use MySQL or PostgreSQL. If unsure, test both syntaxes. Key differences: CONCAT vs ||, YEAR() vs EXTRACT(), DATEDIFF() vs subtraction.*

8. Set Operations

```
-- ■■ UNION: combine results, remove duplicates ■■
select user_id from premium_users
union
select user_id from trial_users

-- ■■ UNION ALL: combine results, keep duplicates (faster) ■■
select user_id from premium_users
union all
select user_id from trial_users

-- ■■ INTERSECT: rows in BOTH queries ■■
select user_id from premium_users
intersect
select user_id from active_users

-- ■■ EXCEPT: rows in first query but NOT in second ■■
select user_id from all_users
except
select user_id from banned_users
```

> ***Key Takeaway:*** *All set operations require the same number of columns with compatible types. UNION removes duplicates (slower); UNION ALL keeps them (faster — use when safe). EXCEPT is an alternative to anti-joins.*

9. Full Practice Exercises

These exercises use a schema common in data analysis questions: `countries(country_id, country)`, `users(user_id, country_id)`, `transactions(txn_id, user_id, amount)`.

Q1 — Users with NO transactions

```
-- direct anti-join: no CTE needed, just left join + is null
select count(*) as users_without_txns
from users u
left join transactions t
    on u.user_id = t.user_id
where t.user_id is null
```

Q2 — Users with total spend below $10

```
-- group + having: filter aggregated result directly, no CTE needed
select count(*) as users_below_10
from (
    select user_id
    from transactions
    group by user_id
    having sum(amount) < 10
) as low_spenders
```

Q3 — Countries with NO users

```
-- direct anti-join on countries -> users
select count(*) as countries_without_users
from countries c
left join users u
    on c.country_id = u.country_id
where u.country_id is null
```

Q4 — Total revenue per country

```
-- chain 3 tables: countries -> users -> transactions
select
    c.country,
    coalesce(sum(t.amount), 0) as total_revenue
from countries c
left join users u on c.country_id = u.country_id
```

```
left join transactions t on u.user_id = t.user_id
group by c.country
order by total_revenue desc
```

Q5 — Top spender per country

```
-- combine chained joins + window function
with user_spend as (
    -- total spend per user with their country
    select
        c.country,
        u.user_id,
        sum(t.amount) as total_spend
    from countries c
    inner join users u on c.country_id = u.country_id
    inner join transactions t on u.user_id = t.user_id
    group by c.country, u.user_id
),

ranked as (
    -- rank users within each country by spend
    select
        country, user_id, total_spend,
        row_number() over (
            partition by country
            order by total_spend desc
        ) as rn
    from user_spend
)

-- pick the top spender per country
select country, user_id, total_spend
from ranked
where rn = 1
```

> ***Key Takeaway:*** *This combines CTEs, JOINs, GROUP BY, and ROW_NUMBER — the full toolkit. Practice this pattern until it's automatic. Almost every Data Analyst SQL question is a variation of it.*

Bonus: Keyword Map, Execution Order & Strategy

Keyword → Pattern Map

Keyword in problem	SQL pattern
total / count / average per ...	GROUP BY + aggregate
latest / first / best / most recent	ROW_NUMBER() OVER (... ORDER BY ... DESC) in CTE
previous / prior / day-over-day	LAG()
running / cumulative / balance	SUM() OVER (ORDER BY ...)
rolling / moving average	AVG() OVER (ROWS BETWEEN N PRECEDING AND CURRENT ROW)
duplicates / repeated	GROUP BY + HAVING COUNT(*) > 1
per group ranking / top N per	ROW_NUMBER() OVER (PARTITION BY ... ORDER BY ...)
no transactions / no users / without	LEFT JOIN + WHERE ... IS NULL
categorize / bucket / label	CASE WHEN ... THEN ... END
combine / together from two tables	UNION or JOIN
percentage / share of total	value / SUM() OVER (PARTITION BY ...) * 100
if / conditional sum or count	SUM(CASE WHEN ...) or COUNT(CASE WHEN ...)

SQL Execution Order

SQL doesn't execute in the order you write it. Knowing the real execution order explains why you can't use aliases in WHERE, and why HAVING comes after GROUP BY.

Step	Clause	What it does
1	FROM / JOIN	Picks tables and combines rows
2	WHERE	Filters individual rows
3	GROUP BY	Collapses rows into groups
4	HAVING	Filters groups (after aggregation)
5	SELECT	Computes output columns and aliases
6	DISTINCT	Removes duplicate result rows
7	ORDER BY	Sorts the final output
8	LIMIT / OFFSET	Returns a subset of rows

Fast Problem-Solving Checklist

#	Step
1	Read the question — underline keywords (total, latest, previous, per group, no...)
2	Identify tables needed and how they connect (which column is the join key?)
3	Decide: does the result collapse rows? → GROUP BY. Keep rows? → WINDOW FUNCTION
4	Write the FROM + JOINs first (get the data foundation right)
5	Add WHERE to filter rows, GROUP BY + HAVING to filter groups
6	Add SELECT columns last (aggregates, CASE, window functions)
7	Wrap in a CTE if you need to filter on a window function result
8	Test: does the output match the expected format? Check column names and ORDER BY

Golden Rule: *The Data Analyst SQL pattern is almost always: CTE for prep → JOIN tables → GROUP BY or WINDOW → filter result. Master the anti-join (LEFT JOIN + IS NULL) and the ROW_NUMBER-in-CTE pattern — they cover 80% of questions.*