

## sql//field-manual

QUERY REFERENCE

by @contenintech

### THE SELECT STATEMENT

- 01 · Anatomy of a Query
- 02 · Filtering with WHERE
- 03 · Sorting & Limiting

### AGGREGATION

- 04 · Aggregate Functions
- 05 · GROUP BY & HAVING

### COMBINING TABLES

- 06 · All Join Types
- 07 · Set Operations

### ADVANCED QUERYING

- 08 · Subqueries
- 09 · CTEs & Recursive Queries
- 10 · Window Functions

### EXPRESSIONS & FUNCTIONS

- 11 · CASE Expressions
- 12 · NULL Handling
- 13 · String, Date & Math Functions

### WRITING DATA

- 14 · INSERT, UPDATE, DELETE Deep Dive

### PRACTICE

- 15 · Reading Query Plans
- 16 · Common Query Patterns

```
~/sql (psql) $ SELECT * FROM queries
```

# The SQL Field Manual

Every clause, function, and query shape an SDE writes day to day — filtering, aggregating, joining, subqueries, CTEs, window functions, and the patterns that come up in real interviews and real codebases. Each entry explains what it does and why, with a working example.

## 01 Anatomy of a Query

The clauses of a SELECT statement, and the order they actually execute in.

## Full clause order

SYNTAX

```
SELECT  columns / expressions
FROM    table
JOIN    other_table ON condition
WHERE   row filter (before grouping)
GROUP BY grouping columns
HAVING  group filter (after grouping)
ORDER BY sort order
LIMIT  row cap
```

## Written order vs. logical execution order

CONCEPT

SQL is famously not executed in the order you type it — understanding the real order explains many "why can't I do that here" moments (like why you can't use a column alias in `WHERE`).

```
graph LR
    A[FROM / JOIN] --> B[WHERE]
    B --> C[GROUP BY]
    C --> D[HAVING]
    D --> E[SELECT]
    E --> F[ORDER BY]
    F --> G[LIMIT]
```

Because `SELECT` (where column aliases are defined) runs *after* `WHERE` and `GROUP BY`, you can't reference a `SELECT` alias in `WHERE` — but you usually can in `ORDER BY`, since that runs afterward.

## 02 Filtering with WHERE

Narrowing down rows before any grouping happens.

## Comparison & logical operators

SYNTAX

```
SELECT * FROM orders WHERE total > 100;
SELECT * FROM orders WHERE status != 'cancelled';
SELECT * FROM orders WHERE total > 100 AND status = 'paid';
SELECT * FROM orders WHERE status = 'paid' OR status = 'shipped';
SELECT * FROM orders WHERE NOT status = 'cancelled';
```

## IN, BETWEEN, LIKE

SYNTAX

```
SELECT * FROM orders WHERE status IN ('paid', 'shipped', 'delivered');
SELECT * FROM orders WHERE total BETWEEN 50 AND 200;    -- inclusive both ends
SELECT * FROM users  WHERE email LIKE '%@gmail.com';    -- % = any sequence of characters
SELECT * FROM users  WHERE name  LIKE 'A__';            -- _ = exactly one character
SELECT * FROM users  WHERE name  ILIKE 'aditi%';         -- case-insensitive LIKE (Postgres)
```

**Performance note:** a leading `%` (like `'%gmail.com'`) can't use a standard B-tree index efficiently, since the engine can't jump to a starting point — it forces a full scan.

```
SELECT * FROM users WHERE phone IS NULL;
SELECT * FROM users WHERE phone IS NOT NULL;
-- WHERE phone = NULL never matches anything – NULL isn't "equal" to anything, including itself
```

03Sorting & Limiting

Controlling row order and how many rows come back.

```
SELECT * FROM orders ORDER BY created_at DESC;
SELECT * FROM orders ORDER BY status ASC, total DESC;  -- multiple sort keys
SELECT * FROM orders ORDER BY created_at DESC LIMIT 10;      -- top 10 most recent
SELECT * FROM orders ORDER BY created_at DESC LIMIT 10 OFFSET 20;  -- page 3, 10 per page
```

**Keyset pagination** (e.g. WHERE created\_at < :last\_seen ORDER BY created\_at DESC LIMIT 10) scales far better than OFFSET on large tables, since OFFSET still has to scan and discard every skipped row.

04Aggregate Functions

Collapsing many rows into a single summary value.

| FUNCTION            | RETURNS                                  |
|---------------------|------------------------------------------|
| COUNT(*)            | Number of rows (including NULLs)         |
| COUNT(col)          | Number of non-NULL values in that column |
| COUNT(DISTINCT col) | Number of distinct non-NULL values       |
| SUM(col)            | Total of all values                      |
| AVG(col)            | Mean of all non-NULL values              |
| MIN(col) / MAX(col) | Smallest / largest value                 |

```
SELECT COUNT(*) AS total_orders,
       SUM(total) AS revenue,
       AVG(total) AS avg_order_value,
       MAX(total) AS biggest_order
FROM orders
WHERE status = 'paid';
```

Aggregate functions ignore NULLs (except COUNT(\*)) — AVG divides by the count of non-NULL rows, not the total row count, which can be surprising with sparse data.

05GROUP BY & HAVING

Computing an aggregate per group instead of over the whole table.

Every non-aggregated column in SELECT must appear in GROUP BY — the database needs to know how to collapse every other column down to one value per group.

```
SELECT customer_id, COUNT(*) AS order_count, SUM(total) AS lifetime_value
FROM orders
GROUP BY customer_id;

-- group by multiple columns
SELECT customer_id, status, COUNT(*) AS n
FROM orders
GROUP BY customer_id, status;
```

WHERE filters individual rows before grouping; HAVING filters entire groups after aggregation — which is why HAVING can reference aggregate functions like SUM() and WHERE cannot.

```
SELECT customer_id, SUM(total) AS lifetime_value
FROM orders
WHERE status = 'paid'           -- filters rows first
GROUP BY customer_id
HAVING SUM(total) > 1000;      -- then filters the resulting groups
```

06All Join Types

Combining rows from multiple tables based on a related column.

| JOIN            | RETURNS                                                               |
|-----------------|-----------------------------------------------------------------------|
| INNER JOIN      | Only rows that match in both tables.                                  |
| LEFT JOIN       | All left-table rows, matched right-table columns or NULL.             |
| RIGHT JOIN      | All right-table rows, matched left-table columns or NULL.             |
| FULL OUTER JOIN | All rows from both sides, NULL wherever there's no match.             |
| CROSS JOIN      | Every combination of rows from both tables (cartesian product).       |
| SELF JOIN       | A table joined to itself, using aliases to distinguish the two roles. |

```
-- INNER JOIN: customers who have placed at least one order
SELECT u.name, o.id, o.total
FROM users u
INNER JOIN orders o ON o.customer_id = u.id;

-- LEFT JOIN: every customer, with NULL order columns if they have none
SELECT u.name, o.id, o.total
FROM users u
LEFT JOIN orders o ON o.customer_id = u.id;

-- find customers with NO orders at all (anti-join pattern)
SELECT u.name
FROM users u
LEFT JOIN orders o ON o.customer_id = u.id
WHERE o.id IS NULL;

-- SELF JOIN: employees and their manager's name
SELECT e.name AS employee, m.name AS manager
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.id;
```

The **anti-join pattern** (LEFT JOIN + WHERE right.id IS NULL) is the standard way to find rows in one table with no match in another — "customers with no orders," "products never sold."

## 07 Set Operations

Combining the results of two separate queries, stacked rather than joined side-by-side.

| OPERATOR  | RETURNS                                                               |
|-----------|-----------------------------------------------------------------------|
| UNION     | Rows from either query, duplicates removed.                           |
| UNION ALL | Rows from either query, duplicates kept — faster since no dedup pass. |
| INTERSECT | Only rows appearing in both queries.                                  |
| EXCEPT    | Rows in the first query that don't appear in the second.              |

```
SELECT email FROM newsletter_subscribers
UNION
SELECT email FROM customers;
-- combined, deduplicated list of every email in either group

SELECT customer_id FROM orders WHERE year = 2025
EXCEPT
SELECT customer_id FROM orders WHERE year = 2026;
-- customers who ordered in 2025 but not 2026
```

Both queries in a set operation must have the same number of columns, in compatible types — column names come from the first query.

## 08 Subqueries

A query nested inside another query.

## Scalar & IN subqueries

SYNTAX

```
-- scalar subquery: returns a single value, usable like a column
SELECT name, (SELECT COUNT(*) FROM orders WHERE customer_id = u.id) AS order_count
FROM users u;

-- subquery in WHERE with IN
SELECT * FROM products
WHERE category_id IN (SELECT id FROM categories WHERE name = 'Electronics');
```

## Correlated subqueries

CONCEPT

A correlated subquery references a column from the outer query, so it re-runs once per outer row — powerful, but can be slow on large tables since it can't simply run once.

```
-- customers whose most recent order was over $500
SELECT * FROM users u
WHERE 500 < (
    SELECT total FROM orders o
    WHERE o.customer_id = u.id
    ORDER BY created_at DESC
    LIMIT 1
);
```

## EXISTS

SYNTAX

EXISTS only checks whether the subquery returns any rows at all — often faster than IN for large subquery results, since the engine can stop at the first match.

```
SELECT * FROM users u
WHERE EXISTS (
    SELECT 1 FROM orders o WHERE o.customer_id = u.id
);

SELECT * FROM users u
WHERE NOT EXISTS (
    SELECT 1 FROM orders o WHERE o.customer_id = u.id
); -- users with zero orders – an alternative to the anti-join pattern
```

## 09 CTEs & Recursive Queries

Naming a subquery so it reads like a series of steps instead of nested parentheses.

## Basic CTE (WITH clause)

SYNTAX

```
WITH high_value AS (  
  SELECT customer_id, SUM(total) AS spent  
  FROM orders  
  GROUP BY customer_id  
  HAVING SUM(total) > 1000  
)  
SELECT u.name, hv.spent  
FROM users u  
JOIN high_value hv ON hv.customer_id = u.id  
ORDER BY hv.spent DESC;
```

Multiple CTEs can be chained, each referencing the ones defined before it:

```
WITH paid_orders AS (  
  SELECT * FROM orders WHERE status = 'paid'  
),  
customer_totals AS (  
  SELECT customer_id, SUM(total) AS spent FROM paid_orders GROUP BY customer_id  
)  
SELECT * FROM customer_totals WHERE spent > 500;
```

## Recursive CTEs

SYNTAX

A recursive CTE repeatedly references itself to walk a hierarchy — the classic use case is an org chart or category tree of unknown depth.

```
WITH RECURSIVE org_chart AS (  
  -- base case: the top-level employee (no manager)  
  SELECT id, name, manager_id, 1 AS level  
  FROM employees  
  WHERE manager_id IS NULL  
  
  UNION ALL  
  
  -- recursive case: join each next level to the previous result  
  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 ORDER BY level;
```

The recursion stops automatically once the recursive term returns no new rows — always make sure the join condition eventually runs out of matches, or it loops until a driver-enforced recursion limit.

## 10 Window Functions

Compute a value across a set of related rows without collapsing them like GROUP BY does.

PARTITION BY groups rows for the window (like GROUP BY, but without collapsing them); ORDER BY inside OVER() defines the order the window function processes rows in.

```
SELECT
  customer_id,
  order_date,
  total,
  SUM(total) OVER (PARTITION BY customer_id ORDER BY order_date) AS running_total
FROM orders;
-- each row keeps its identity, but gains a running total per customer
```

| FUNCTION                 | RETURNS                                                                           |
|--------------------------|-----------------------------------------------------------------------------------|
| ROW_NUMBER()             | A unique, sequential number per row within its partition (1, 2, 3, ...).          |
| RANK()                   | Rank within partition; ties share a rank, and the next rank skips (1, 1, 3).      |
| DENSE_RANK()             | Like RANK, but no gaps after ties (1, 1, 2).                                      |
| NTILE(n)                 | Splits the partition into n roughly equal buckets (e.g. quartiles with NTILE(4)). |
| LAG(col, n)              | The value of col from n rows before the current one.                              |
| LEAD(col, n)             | The value of col from n rows after the current one.                               |
| FIRST_VALUE / LAST_VALUE | The first/last value in the window frame.                                         |

```
-- rank orders by value within each customer
SELECT customer_id, total,
       RANK() OVER (PARTITION BY customer_id ORDER BY total DESC) AS rank_in_customer
FROM orders;

-- month-over-month comparison using LAG
SELECT month, revenue,
       LAG(revenue) OVER (ORDER BY month) AS prev_month_revenue,
       revenue - LAG(revenue) OVER (ORDER BY month) AS change
FROM monthly_revenue;

-- top 3 highest-paid employees per department
SELECT * FROM (
  SELECT name, department, salary,
         ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS rn
  FROM employees
) ranked
WHERE rn <= 3;
```

A window function can't be used directly in WHERE (it runs after WHERE/GROUP BY in logical order) — wrap the query in a subquery or CTE and filter on the outside, as shown above.

# 11CASE Expressions

SQL's inline if/else — usable anywhere a value is expected.

```
SELECT
    name,
    total,
    CASE
        WHEN total >= 1000 THEN 'VIP'
        WHEN total >= 100 THEN 'Regular'
        ELSE 'New'
    END AS customer_tier
FROM orders;

-- CASE inside an aggregate: conditional counting ("pivot" pattern)
SELECT
    SUM(CASE WHEN status = 'paid' THEN 1 ELSE 0 END) AS paid_count,
    SUM(CASE WHEN status = 'cancelled' THEN 1 ELSE 0 END) AS cancelled_count
FROM orders;
```

Combining CASE with SUM/COUNT is the standard way to "pivot" rows into columns without a database-specific PIVOT extension.

12

## NULL Handling

NULL means "unknown," which makes it behave differently from any actual value.

| FUNCTION              | BEHAVIOR                                                                          |
|-----------------------|-----------------------------------------------------------------------------------|
| COALESCE(a, b, ...)   | Returns the first non-NULL argument — the standard way to supply a default.       |
| NULLIF(a, b)          | Returns NULL if a = b, otherwise returns a — useful to avoid divide-by-zero.      |
| IS NULL / IS NOT NULL | The only correct way to test for NULL — = NULL always evaluates to unknown/false. |

```
SELECT COALESCE(nickname, first_name, 'Anonymous') AS display_name FROM users;

-- avoid a divide-by-zero by turning 0 into NULL first
SELECT total / NULLIF(quantity, 0) AS price_per_unit FROM order_items;
```

NULL propagates through most expressions: NULL + 5, NULL = NULL, and NULL AND true are all NULL/unknown, not true or false — this is why filtering logic with NULLs needs care.

13

## String, Date & Math Functions

The built-in functions that show up in almost every real query.

String functions

REFERENCE

```
CONCAT(first_name, ' ', last_name)      -- "Aditi Sharma"
UPPER(name) / LOWER(name)
TRIM('  hello  ')                       -- "hello"
LENGTH(name)                           -- character count
SUBSTRING(name FROM 1 FOR 3)             -- first 3 characters
REPLACE(email, '@old.com', '@new.com')
```

## Date functions

[REFERENCE](#)

```
NOW()                -- current timestamp
CURRENT_DATE         -- today's date
DATE_TRUNC('month', created_at) -- rounds down to the 1st of the month
EXTRACT(YEAR FROM created_at)  -- pull out a date part
created_at + INTERVAL '7 days' -- date arithmetic
AGE(NOW(), created_at)        -- how much time has elapsed
```

## Math functions

[REFERENCE](#)

```
ROUND(price, 2)      -- round to 2 decimal places
CEIL(x) / FLOOR(x)
ABS(x)
POWER(x, 2)
x % 3                -- modulo
```

## 14 INSERT, UPDATE, DELETE Deep Dive

Writing data, beyond the basics.

### INSERT ... SELECT

[SYNTAX](#)

Populates a table directly from a query's results instead of literal values — common for copying/archiving data.

```
INSERT INTO archived_orders (id, customer_id, total)
SELECT id, customer_id, total
FROM orders
WHERE created_at < NOW() - INTERVAL '1 year';
```

### UPDATE with a join

[SYNTAX](#)

```
UPDATE orders o
SET status = 'flagged'
FROM users u
WHERE o.customer_id = u.id
AND u.is_suspicious = true;
```

RETURNING clause

SYNTAX

Gets back the affected rows from an INSERT/UPDATE/DELETE in the same round trip — avoids a separate SELECT afterward.

```
INSERT INTO users (name, email) VALUES ('Aditi', 'aditi@co.com')
RETURNING id, created_at;

DELETE FROM sessions WHERE expires_at < NOW()
RETURNING id;
```

15Reading Query Plans

Understanding what the database actually does with a query.

EXPLAIN ANALYZE

DIAGNOSE

```
EXPLAIN ANALYZE
SELECT * FROM orders WHERE customer_id = 42 ORDER BY created_at DESC;
```

| PLAN   | NODE      | MEANING                                                                                     |
|--------|-----------|---------------------------------------------------------------------------------------------|
| Seq    | Scan      | Reads every row in the table — fine for small tables, a red flag on large ones.             |
| Index  | Scan      | Uses an index to jump straight to matching rows, then fetches the full row.                 |
| Index  | Only Scan | Answers entirely from the index, without touching the table at all — fastest.               |
| Nested | Loop      | For each row on one side of a join, scans for matches on the other — fine for small inputs. |
| Hash   | Join      | Builds a hash table from one side, then probes it — efficient for larger joins.             |
| Sort   |           | An explicit sort step — can be expensive on large result sets without a supporting index.   |

Compare the planner's row estimate to the actual row count in EXPLAIN ANALYZE output — a big mismatch usually means outdated table statistics, which can lead the planner to pick a bad plan.

16Common Query Patterns

Shapes of queries that come up constantly in real work and interviews.

Find duplicates

PATTERN

```
SELECT email, COUNT(*) AS n
FROM users
GROUP BY email
HAVING COUNT(*) > 1;
```

## Second-highest value

PATTERN

```
-- using DENSE_RANK to handle ties correctly
SELECT salary FROM (
    SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS rnk
    FROM employees
) t
WHERE rnk = 2;
```

## Running total & moving average

PATTERN

```
SELECT order_date, total,
    SUM(total) OVER (ORDER BY order_date) AS running_total,
    AVG(total) OVER (ORDER BY order_date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS moving_avg_7d
FROM orders;
```

## Delete duplicate rows, keep one

PATTERN

```
DELETE FROM users
WHERE id NOT IN (
    SELECT MIN(id) FROM users GROUP BY email
);
```

## Percentage of total

PATTERN

```
SELECT category, SUM(revenue) AS category_revenue,
    SUM(revenue) * 100.0 / SUM(SUM(revenue)) OVER () AS pct_of_total
FROM sales
GROUP BY category;
```