

SQL Reference

A developer-focused guide from basic queries to advanced patterns.

Last reviewed 29 Sep 2026 ·

Prerequisites: [DE Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/)

Related: [Data Modeling](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/) · [Snowflake](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/snowflake-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/snowflake-reference/) · [dbt](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Practice: [Lab 01 — SQL Analytics](https://github.com/sarangambekar1997/de-workspace/tree/main/labs/01-sql-analytics) (https://github.com/sarangambekar1997/de-workspace/tree/main/labs/01-sql-analytics)

Overview

Challenge: Business data is stored in tables — orders, customers, events — often billions of rows distributed across many machines. Answering a question such as "revenue by region last quarter" should not require writing a program that iterates over every row.

Solution: SQL is declarative: you describe the result, and the engine determines how to produce it — which indexes to use, the join order, and how to parallelize the work. The same core syntax works across relational databases, cloud warehouses, lakehouse engines, and embedded engines.

You write:	The engine
decides:	
SELECT region, SUM(amount)	scan only 2
columns (columnar)	
FROM orders	skip
partitions outside Q3 (pruning)	
WHERE order_date >= '2024-07-01'	aggregate
in parallel on 32 nodes	

```
GROUP BY region;  
partial sums → 5 rows back
```

merge

Relevance to data engineering: SQL is the primary language of warehouses, transformation layers, and data quality checks, and it is central to technical interviews. Joins, aggregation, and window functions deserve particular depth.



On this page - [What is SQL?](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#what-is-sql) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#what-is-sql>) - [Data Types](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#data-types) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#data-types>) - [SELECT](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#select) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#select>) - [Filtering](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#filtering) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#filtering>) - [Sorting & Limiting](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#sorting--limiting) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#sorting--limiting>) - [Aggregates](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#aggregates) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#aggregates>) - [Joins](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#joins) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#joins>) - [Subqueries](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#subqueries) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#subqueries>) - [INSERT / UPDATE / DELETE](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#insert--update--delete) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#insert--update--delete>) - [CTEs](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#ctes--common-table-expressions) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#ctes--common-table-expressions>) - [Window Functions](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#window-functions) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#window-functions>) - [Indexes & Performance](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#indexes--performance) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#indexes--performance>) - [Transactions](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#transactions--acid) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#transactions--acid>) - [Views](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#views) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#views>)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#common-pitfalls) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#common-pitfalls>) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#cheat-sheet) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#cheat-sheet>) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#interview-questions) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#interview-questions>) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#further-reading) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/#further-reading>)

What is SQL?

SQL (Structured Query Language) is the standard language for querying and manipulating relational databases. You describe **what** data you want — the database engine decides **how** to retrieve it.

Data lives in **tables** — a grid of rows (records) and columns (fields). Related tables connect via **foreign keys**. A collection of tables is a **schema**, and a schema lives inside a **database**.

Which SQL? This reference targets standard SQL (ANSI) with notes on Postgres. MySQL, SQLite, and SQL Server follow the same core syntax with minor differences in functions and types.

Mental model — query execution order

```
-- Logical order the engine processes clauses:
-- 1. FROM      — which table(s)?
-- 2. JOIN      — combine with other tables?
-- 3. WHERE     — filter rows
-- 4. GROUP BY  — collapse rows into groups
-- 5. HAVING    — filter groups
-- 6. SELECT    — pick/compute columns
-- 7. ORDER BY  — sort results
-- 8. LIMIT     — cap row count
```

```
SELECT department, COUNT(*) AS headcount
FROM   employees
WHERE  active = true
GROUP BY department
HAVING COUNT(*) > 3
ORDER BY headcount DESC
LIMIT 10;
```

Data Types

Type	What it holds	Example
INTEGER / INT	Whole numbers	42
BIGINT	Large whole numbers (up to $\sim 9.2 \times 10^{18}$)	9876543210
NUMERIC(p,s)	Exact decimal — use for money	NUMERIC(10,2) → 1234567.89
FLOAT / REAL	Approximate decimal (IEEE 754)	3.14159

Type	What it holds	Example
<code>VARCHAR(n)</code>	Variable-length text up to n chars	<code>'Alice'</code>
<code>TEXT</code>	Unlimited text (Postgres/MySQL)	Long descriptions, JSON strings
<code>BOOLEAN</code>	true / false	<code>true</code>
<code>DATE</code>	Calendar date, no time	<code>'2024-03-15'</code>
<code>TIMESTAMP</code>	Date + time	<code>'2024-03-15 09:00:00'</code>
<code>TIMESTAMPTZ</code>	Timestamp with timezone (Postgres)	<code>'2024-03-15 09:00:00+05:30'</code>
<code>UUID</code>	128-bit universally unique identifier	<code>'550e8400-e29b-41d4-a716...'</code>
<code>JSON / JSONB</code>	JSON document (JSONB = binary, indexable)	<code>'{"role": "admin"}'</code>

Use **NUMERIC** for money, never **FLOAT**. Float arithmetic is approximate — `0.1 + 0.2` can return `0.30000000000000004`. `NUMERIC(12,2)` is exact.

SELECT

The most common statement. Retrieves rows from one or more tables.

```
-- All columns
SELECT * FROM employees;

-- Specific columns
SELECT id, name, salary FROM employees;

-- Computed column
SELECT name, salary * 12 AS annual_salary FROM employees;

-- Alias a table
SELECT e.name, e.department
FROM employees AS e;
```

DISTINCT — deduplicate rows

```
SELECT DISTINCT department FROM employees;
-- Returns each department name once, even if 50 employees
share it.
```

Common column expressions

```
SELECT
  UPPER(name)                AS name_upper,
  LOWER(email)               AS email_lower,
  LENGTH(name)               AS name_len,
  CONCAT(first_name, ' ', last_name) AS full_name,
  COALESCE(phone, 'N/A')     AS phone,    -- first
non-null value
  ROUND(salary / 1000.0, 1)   AS k_salary,
  CURRENT_DATE               AS today,
  AGE(CURRENT_DATE, hire_date) AS tenure   --
Postgres
FROM employees;
```

COALESCE is your null-guard. `COALESCE(a, b, c)` returns the first argument that is not NULL.

Filtering

The `WHERE` clause filters rows before aggregation. Only rows where the condition is **true** pass through.

Operators: `=` `!=` `<>` `<` `<=` `>` `>=` `AND` `OR` `NOT` `IN` `NOT IN` `BETWEEN` `LIKE` `IS NULL` `IS NOT NULL`

```
-- Equality and comparison
SELECT * FROM employees WHERE department = 'Engineering';
SELECT * FROM employees WHERE salary >= 90000;
SELECT * FROM employees WHERE hire_date < '2022-01-01';

-- AND / OR
SELECT * FROM employees
WHERE department = 'Engineering' AND salary > 100000;
```

```

SELECT * FROM employees
WHERE department = 'Design' OR department = 'Product';

-- IN — cleaner than many ORs
SELECT * FROM employees
WHERE department IN ('Design', 'Product', 'Marketing');

-- BETWEEN (inclusive on both ends)
SELECT * FROM employees
WHERE salary BETWEEN 80000 AND 120000;

-- LIKE — pattern matching
-- % matches any sequence, _ matches one character
SELECT * FROM employees WHERE name LIKE 'A%';      --
starts with A
SELECT * FROM employees WHERE email LIKE '%@acme.com';
SELECT * FROM employees WHERE name LIKE '_lic_';    -- e.g.
"Alice"

-- NULL checks — always use IS NULL, never = NULL
SELECT * FROM employees WHERE manager_id IS NULL;  -- top-
level managers
SELECT * FROM employees WHERE phone IS NOT NULL;

```

NULL is not a value — it's the absence of one. `NULL = NULL` evaluates to NULL (unknown), not true. Always use `IS NULL` / `IS NOT NULL`.

Sorting & Limiting

```

-- ORDER BY (ASC is default)
SELECT name, salary FROM employees ORDER BY salary DESC;
SELECT name, hire_date FROM employees ORDER BY hire_date
ASC;

-- Multiple sort columns
SELECT name, department, salary
FROM employees
ORDER BY department ASC, salary DESC;

```

```
-- Force NULLs last (Postgres)
ORDER BY salary DESC NULLS LAST;

-- LIMIT and OFFSET — pagination
SELECT * FROM employees ORDER BY id LIMIT 20 OFFSET 0;    --
page 1
SELECT * FROM employees ORDER BY id LIMIT 20 OFFSET 20;  --
page 2

-- Top-N pattern
SELECT name, salary
FROM   employees
ORDER BY salary DESC
LIMIT 5;  -- top 5 earners
```

Always ORDER BY when using LIMIT. Without it, the database returns rows in undefined order — you'll get inconsistent pages.

Aggregates

Aggregate functions collapse multiple rows into a single value. Combine with `GROUP BY` to compute per-group statistics.

Function	Returns	Ignores NULLs?
<code>COUNT(*)</code>	Total row count	No
<code>COUNT(col)</code>	Rows where col is not NULL	Yes
<code>COUNT(DISTINCT col)</code>	Unique non-NULL values	Yes
<code>SUM(col)</code>	Total	Yes
<code>AVG(col)</code>	Mean	Yes
<code>MIN(col)</code>	Smallest value	Yes
<code>MAX(col)</code>	Largest value	Yes

```
-- Simple aggregates
SELECT COUNT(*) AS total_employees FROM employees;
```

```
SELECT AVG(salary) AS avg_salary, MAX(salary) AS top_salary
FROM employees;
```

```
-- GROUP BY — one row per group
```

```
SELECT
  department,
  COUNT(*)           AS headcount,
  ROUND(AVG(salary), 0) AS avg_salary,
  MAX(salary)        AS top_salary
FROM   employees
GROUP BY department
ORDER BY headcount DESC;
```

```
-- HAVING filters groups (not rows — that's WHERE)
```

```
SELECT department, COUNT(*) AS headcount
FROM   employees
GROUP BY department
HAVING COUNT(*) >= 5;
```

```
-- Combining WHERE and HAVING
```

```
SELECT department, ROUND(AVG(salary), 0) AS avg_sal
FROM   employees
WHERE  active = true                    -- filter rows first
GROUP BY department
HAVING AVG(salary) > 90000             -- then filter groups
ORDER BY avg_sal DESC;
```

WHERE vs HAVING: WHERE runs before grouping (filters raw rows). HAVING runs after (filters groups). You can't use aggregate functions in WHERE.

Joins

Joins combine rows from two or more tables based on a related column.

Join	Returns
INNER JOIN	Rows with a match in both tables
LEFT JOIN	All rows from left; NULLs where right has no match
RIGHT JOIN	All rows from right; NULLs where left has no match

Join	Returns
<code>FULL OUTER JOIN</code>	All rows from both; NULLs where either side has no match
<code>CROSS JOIN</code>	Every row × every row (cartesian product)

```
-- INNER JOIN – only matched rows
SELECT e.name, d.name AS dept_name, d.budget
FROM   employees e
INNER JOIN departments d ON e.department_id = d.id;

-- LEFT JOIN – keep all employees, even if dept is missing
SELECT e.name, d.name AS dept_name
FROM   employees e
LEFT JOIN departments d ON e.department_id = d.id;
-- employees with no department get dept_name = NULL

-- Find orphan rows (no matching parent)
SELECT e.name
FROM   employees e
LEFT JOIN departments d ON e.department_id = d.id
WHERE  d.id IS NULL;

-- FULL OUTER JOIN
SELECT e.name, d.name AS dept_name
FROM   employees e
FULL OUTER JOIN departments d ON e.department_id = d.id;

-- Joining three tables
SELECT e.name, d.name AS dept, p.title AS project
FROM   employees e
JOIN   departments d      ON e.department_id = d.id
JOIN   project_members pm ON pm.employee_id = e.id
JOIN   projects p         ON pm.project_id = p.id;
```

Self-join

```
-- Employee → their manager (same table)
SELECT e.name AS employee, m.name AS manager
```

```
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.id;
```

Subqueries

A query nested inside another query. Can appear in `SELECT`, `FROM`, or `WHERE`.

```
-- Scalar subquery – returns a single value
SELECT name, salary,
       salary - (SELECT AVG(salary) FROM employees) AS
diff_from_avg
FROM employees;

-- Subquery in WHERE
SELECT name
FROM employees
WHERE department_id IN (
    SELECT id FROM departments WHERE budget > 500000
);

-- Subquery in FROM (derived table) – must be aliased
SELECT dept, avg_sal
FROM (
    SELECT department AS dept, AVG(salary) AS avg_sal
    FROM employees
    GROUP BY department
) AS dept_stats
WHERE avg_sal > 95000;

-- EXISTS – true if subquery returns any rows
SELECT name FROM employees e
WHERE EXISTS (
    SELECT 1 FROM project_members pm WHERE pm.employee_id =
e.id
);
```

Correlated subquery: the inner query references the outer row. Runs once per outer row — often slow. Usually replaceable with a `JOIN` or `CTE`.

INSERT / UPDATE / DELETE

```
-- INSERT single row
INSERT INTO employees (name, department_id, salary,
hire_date)
VALUES ('Jordan Lee', 3, 95000, CURRENT_DATE);

-- INSERT multiple rows
INSERT INTO employees (name, department_id, salary,
hire_date)
VALUES
    ('Sam Park', 2, 88000, '2024-01-15'),
    ('Riley Chen', 2, 92000, '2024-03-01');

-- INSERT ... SELECT — copy data from another table
INSERT INTO archive_employees
SELECT * FROM employees WHERE active = false;

-- UPDATE — test your WHERE first with a SELECT
UPDATE employees
SET salary = salary * 1.10
WHERE department = 'Engineering' AND active = true;

-- DELETE
DELETE FROM employees WHERE id = 42;

-- RETURNING (Postgres) — see affected rows immediately
UPDATE employees SET salary = salary * 1.10 WHERE id = 5
RETURNING id, name, salary;
```

Before any UPDATE or DELETE: run the equivalent SELECT first to confirm which rows you're about to change. Wrap in a transaction so you can roll back.

CTEs — Common Table Expressions

A CTE (`WITH` clause) names a subquery so you can reference it like a table. Makes complex queries readable and avoids repeated subexpressions.

```
-- Single CTE
WITH high_earners AS (
```

```

    SELECT * FROM employees WHERE salary > 100000
)
SELECT department, COUNT(*) AS count
FROM   high_earners
GROUP BY department;

-- Chained CTEs
WITH
active_employees AS (
    SELECT * FROM employees WHERE active = true
),
dept_stats AS (
    SELECT department_id, AVG(salary) AS avg_sal, COUNT(*) AS
n
    FROM   active_employees
    GROUP BY department_id
)
SELECT d.name, ds.avg_sal, ds.n
FROM   dept_stats ds
JOIN   departments d ON d.id = ds.department_id
ORDER BY ds.avg_sal DESC;

```

Recursive CTEs

Used for hierarchical data — org charts, file trees, graph traversal.

```

WITH RECURSIVE org_chart AS (
    -- Anchor: start at the CEO (no manager)
    SELECT id, name, manager_id, 0 AS depth
    FROM   employees
    WHERE  manager_id IS NULL

    UNION ALL

    -- Recursive step: join each employee to their manager
row
    SELECT e.id, e.name, e.manager_id, oc.depth + 1
    FROM   employees e
    JOIN   org_chart oc ON e.manager_id = oc.id
)
SELECT depth, name FROM org_chart ORDER BY depth, name;

```

CTE vs subquery: CTEs are not automatically faster. Their advantage is readability and reuse within the same query.

Window Functions

Window functions compute a value for each row using a set of related rows (the *window*), without collapsing rows like `GROUP BY`. The row stays; a new column is added.

```
-- Syntax
function_name() OVER (
  PARTITION BY col    -- split into groups (optional)
  ORDER BY col        -- sort within the group
  ROWS BETWEEN ...    -- frame (optional)
)
```

Ranking functions

```
SELECT
  name,
  department,
  salary,
  ROW_NUMBER() OVER (PARTITION BY department ORDER BY
salary DESC) AS row_num,
  RANK()          OVER (PARTITION BY department ORDER BY
salary DESC) AS rank,
  DENSE_RANK() OVER (PARTITION BY department ORDER BY
salary DESC) AS dense_rank
FROM employees;
-- RANK skips numbers after ties; DENSE_RANK does not
```

Offset functions — look at neighboring rows

```
SELECT
  hire_date,
  salary,
  LAG(salary)          OVER (ORDER BY hire_date) AS
prev_salary,
  LEAD(salary)         OVER (ORDER BY hire_date) AS
next_salary,
```

```
FIRST_VALUE(salary) OVER (ORDER BY hire_date) AS  
first_hired_salary  
FROM employees;
```

Running totals and moving averages

```
SELECT  
    hire_date,  
    salary,  
    SUM(salary) OVER (ORDER BY hire_date) AS running_total,  
    AVG(salary) OVER (  
        ORDER BY hire_date  
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW -- 3-row  
        moving avg  
    ) AS moving_avg_3  
FROM employees  
ORDER BY hire_date;  
  
-- Percent of total  
SELECT  
    name, salary,  
    ROUND(100.0 * salary / SUM(salary) OVER (), 2) AS  
pct_of_total  
FROM employees;
```

Top-N per group

```
WITH ranked AS (  
    SELECT name, department, salary,  
        RANK() OVER (PARTITION BY department ORDER BY salary  
DESC) AS rnk  
    FROM employees  
)  
SELECT name, department, salary  
FROM ranked  
WHERE rnk <= 3;
```

Indexes & Performance

An index speeds up lookups on a column at the cost of write overhead and storage.

```
-- Basic index
CREATE INDEX idx_employees_department ON
employees(department);

-- Composite index — order matters!
-- Efficient for: WHERE department = ? AND salary > ?
-- Not for:      WHERE salary > ? (missing left column)
CREATE INDEX idx_dept_salary ON employees(department,
salary);

-- Unique index (also enforces a constraint)
CREATE UNIQUE INDEX idx_employees_email ON
employees(email);

-- Partial index — index only a subset of rows
CREATE INDEX idx_active_employees ON employees(department)
WHERE active = true;

-- Covering index (Postgres) — avoids heap access
CREATE INDEX idx_emp_cover ON employees(department)
INCLUDE (name, salary);

-- Drop an index
DROP INDEX idx_employees_department;
```

EXPLAIN — read the query plan

```
-- Show estimated cost and plan
EXPLAIN SELECT * FROM employees WHERE department =
'Engineering';

-- Run it and show real timing (Postgres)
EXPLAIN ANALYZE SELECT * FROM employees WHERE department =
'Engineering';

-- Key things to look for:
-- Seq Scan          = full table scan (no index used)
```

```
-- Index Scan      = using an index
-- rows=X actual=Y = estimated vs real row count (large
gap = stale stats)
```

When to index

- Columns frequently used in `WHERE`, `JOIN ON`, or `ORDER BY`
- Foreign key columns (Postgres doesn't auto-index these — MySQL does)
- High-cardinality columns (email, `UUID`) benefit more than low-cardinality (boolean)
- Don't over-index write-heavy tables — every write must update all indexes

Transactions & ACID

A transaction is a sequence of statements that succeeds or fails as a unit. No partial writes survive.

Property	Guarantee
Atomic	All statements commit, or none do
Consistent	The database moves from one valid state to another
Isolated	Concurrent transactions don't see each other's partial work
Durable	Committed data survives crashes

```
-- Transfer $500 between accounts atomically
BEGIN;

UPDATE accounts SET balance = balance - 500 WHERE id = 1;
UPDATE accounts SET balance = balance + 500 WHERE id = 2;

COMMIT;    -- make it permanent
-- or:
ROLLBACK;  -- undo everything since BEGIN

-- Savepoints — partial rollback
BEGIN;
  INSERT INTO orders (...) VALUES (...);
  SAVEPOINT after_order;
```

```

INSERT INTO payments (...) VALUES (...);
-- something failed:
ROLLBACK TO after_order; -- undo payment, keep order

COMMIT;

```

Isolation levels

Level	Dirty Read	Non-repeatable Read	Phantom Read
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	No	Possible	Possible
REPEATABLE READ	No	No	Possible
SERIALIZABLE	No	No	No

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```



Views

A view is a named query stored in the database. It behaves like a table but runs its underlying query each time it's accessed.

```

-- Create a view
CREATE VIEW active_employees AS
SELECT id, name, department, salary
FROM employees
WHERE active = true;

-- Use it like a table
SELECT * FROM active_employees WHERE department =
'Engineering';

-- Replace (update) a view
CREATE OR REPLACE VIEW active_employees AS
SELECT id, name, department, salary, hire_date

```



```

FROM employees
WHERE active = true;

-- Drop
DROP VIEW active_employees;

-- Materialized view (Postgres) – stored on disk, must be
refreshed
CREATE MATERIALIZED VIEW dept_summary AS
SELECT department, COUNT(*) AS n, AVG(salary) AS avg_sal
FROM employees GROUP BY department;

REFRESH MATERIALIZED VIEW dept_summary;

```

Materialized views are snapshots — fast to read, but stale until refreshed. Use them for expensive aggregations that don't need to be real-time.

Common Pitfalls

Pitfall	Symptom	Fix
<code>= NULL / != NULL</code>	Filter returns nothing	<code>IS NULL / IS NOT NULL</code> ; remember <code>NULL</code> compared with anything is unknown
<code>NOT IN (subquery)</code> where the subquery returns a <code>NULL</code>	Query returns zero rows	Use <code>NOT EXISTS</code> , or filter <code>NULL</code> s out of the subquery
Join fan-out (joining to a table with multiple matches per key)	<code>SUM(amount)</code> is inflated 2×, 3×...	Check key uniqueness first; aggregate before joining; compare <code>COUNT(*)</code> before/after
Filtering the right table of a <code>LEFT JOIN</code> in <code>WHERE</code>	Left join silently becomes an inner join	Put the condition in the <code>ON</code> clause
<code>COUNT(col)</code> vs <code>COUNT(*)</code> confusion	Counts differ unexpectedly	<code>COUNT(col)</code> skips <code>NULL</code> s; <code>COUNT(*)</code> counts rows
Integer division (<code>1 / 2 = 0</code> in Postgres)	Percentages come out as 0	Cast: <code>1.0 * a / b</code> or <code>a::numeric / b</code> ; guard with <code>NULLIF(b, 0)</code>

Pitfall	Symptom	Fix
<code>ROW_NUMBER()</code> without a unique <code>ORDER BY</code>	Dedup keeps a different row each run	Add a tiebreaker column so ordering is deterministic
Functions on indexed/partitioned columns in <code>WHERE (DATE(ts) = ...)</code>	Index or partition pruning not used; full scan	Use a range on the raw column: <code>ts >= '2024-03-15' AND ts < '2024-03-16'</code>
<code>SELECT *</code> in production queries	Breaks when columns are added; scans everything in columnar stores	Name the columns you need
<code>BETWEEN</code> on timestamps	Includes midnight of the end date, or misses the rest of that day	Half-open ranges: <code>>= start AND < end</code>
<code>UNION</code> when you meant <code>UNION ALL</code>	Unnecessary sort/dedup; rows silently disappear	<code>UNION ALL</code> unless you need deduplication

Cheat Sheet

Which construct?

Need	Use
Reusable query, always fresh data	View
Expensive query, staleness OK	Materialized view
Readable multi-step logic in one statement	<code>CTE (WITH ...)</code>
Per-row value that looks at other rows	Window function
Filter on an aggregate	<code>HAVING</code>
"Rows in A with no match in B"	<code>LEFT JOIN ... WHERE b.id IS NULL</code> or <code>NOT EXISTS</code>
Several writes that succeed or fail together	Transaction
Speed up lookups on a column (OLTP)	Index

Need	Use
Insert-or-update	<code>MERGE / INSERT ... ON CONFLICT</code>

Syntax you'll write every day

```
-- Dedupe: keep latest row per key
SELECT * FROM (
  SELECT *, ROW_NUMBER() OVER (PARTITION BY id ORDER BY
updated_at DESC) AS rn
  FROM raw_orders
) t WHERE rn = 1;
--
Snowflake/BigQuery/Databricks: QUALIFY rn = 1

-- Top N per group
... RANK() OVER (PARTITION BY dept ORDER BY salary DESC) <=
3

-- Running total / 7-row moving average
SUM(amount) OVER (ORDER BY day ROWS BETWEEN UNBOUNDED
PRECEDING AND CURRENT ROW)
AVG(amount) OVER (ORDER BY day ROWS BETWEEN 6 PRECEDING AND
CURRENT ROW)

-- Change vs previous row
amount - LAG(amount) OVER (PARTITION BY customer_id ORDER
BY order_date)

-- Conditional aggregation (pivot)
SUM(CASE WHEN status = 'shipped' THEN 1 ELSE 0 END) AS
shipped_count

-- Safe division
revenue / NULLIF(orders, 0)

-- Find duplicates
SELECT id, COUNT(*) FROM t GROUP BY id HAVING COUNT(*) > 1;

-- Upsert (Postgres)
INSERT INTO t (id, v) VALUES (1, 'x')
ON CONFLICT (id) DO UPDATE SET v = EXCLUDED.v;
```

Execution order: FROM → JOIN → WHERE → GROUP BY → HAVING → SELECT → (window functions) → ORDER BY → LIMIT

Ranking: ROW_NUMBER 1,2,3,4 · RANK 1,2,2,4 · DENSE_RANK 1,2,2,3

Interview Questions

Q: What is the difference between WHERE and HAVING ? A: WHERE filters individual rows *before* grouping, so it can't reference aggregates. HAVING filters groups *after* GROUP BY, so it can: HAVING COUNT(*) > 5. Put every condition you can in WHERE — filtering rows before aggregation is cheaper.

Q: Explain the different join types. A: INNER JOIN returns only rows with a match on both sides. LEFT JOIN returns every row from the left table, with NULL s where the right has no match. RIGHT JOIN is the mirror image. FULL OUTER JOIN returns everything from both, matched where possible. CROSS JOIN returns every combination (a Cartesian product). Watch for fan-out: if the join key isn't unique on one side, rows multiply.

Q: What's the difference between ROW_NUMBER, RANK, and DENSE_RANK ? A: All three number rows within a window ordering. On ties, ROW_NUMBER still assigns unique numbers (1, 2, 3, 4) — arbitrarily unless the ordering is unique. RANK gives ties the same number and then skips (1, 2, 2, 4). DENSE_RANK gives ties the same number without gaps (1, 2, 2, 3). Use ROW_NUMBER for deduplication and DENSE_RANK for "top N distinct values".

Q: How would you find the second-highest salary in each department? A: DENSE_RANK() OVER (PARTITION BY department ORDER BY salary DESC) in a CTE or subquery, then filter where the rank equals 2. DENSE_RANK means that if two people tie for the top salary, the next distinct salary is still ranked 2.

Q: How do you remove duplicate rows but keep the latest version of each record? A:

ROW_NUMBER() OVER (PARTITION BY <business key> ORDER BY updated_at DESC, <tiebreaker>) and keep rn = 1 (QUALIFY does this without a subquery in Snowflake/BigQuery/Databricks). The tiebreaker matters: without a deterministic order, reruns can pick different rows.

Q: What is the difference between a CTE and a subquery? Is a CTE faster? A: Both define an intermediate result; a CTE names it up front with WITH, which makes multi-step logic readable and lets you reference the result more than once. Performance is usually identical, because most modern optimizers inline CTEs. Some engines may materialize a CTE that's referenced multiple times, which can help or hurt. Choose CTEs for readability, and check EXPLAIN if performance matters.

Q: What does NULL do in comparisons, joins, and aggregates? A: NULL means "unknown", so `NULL = NULL` is not true — it's unknown, and rows with unknown conditions are filtered out. Joins on NULL keys never match. `COUNT(col)`, `SUM`, and `AVG` ignore NULLs, while `COUNT(*)` counts all rows. Use `IS NULL`, `COALESCE`, and `IS NOT DISTINCT FROM` (NULL-safe equality) to handle them explicitly.

Q: A query that was fast last month is now slow. How do you debug it? A: Run `EXPLAIN ANALYZE` (or check the warehouse query profile) and compare estimated vs actual rows. Common causes: data growth pushing a join into a spill; stale statistics causing a bad plan; a new function on a filtered column preventing index use or partition pruning; join fan-out from new duplicate keys; or, in a warehouse, a too-small warehouse or clustering that has degraded. Fix the cause — add a filter, fix the join key, recluster — rather than just throwing more compute at it.

Q: What are ACID transactions and isolation levels? A: ACID means atomic (all or nothing), consistent (constraints hold), isolated (concurrent transactions don't see each other's partial work), and durable (committed data survives crashes). Isolation levels trade correctness for concurrency: `READ COMMITTED` (the Postgres default) prevents dirty reads; `REPEATABLE READ` also prevents a row changing between two reads; `SERIALIZABLE` behaves as if transactions ran one at a time.

Further Reading

- [PostgreSQL documentation — SQL language](https://www.postgresql.org/docs/current/sql.html) (https://www.postgresql.org/docs/current/sql.html)
- [Use The Index, Luke](https://use-the-index-luke.com/) (https://use-the-index-luke.com/) — free, the best explanation of indexes and query performance
- [Modern SQL](https://modern-sql.com/) (https://modern-sql.com/) — window functions, `FILTER`, `LATERAL` and other features across databases
- [SQLBolt](https://sqlbolt.com/) (https://sqlbolt.com/) — interactive beginner lessons
- [DataLemur](https://datalemur.com/) (https://datalemur.com/) and [LeetCode Database](https://leetcode.com/problemset/database/) (https://leetcode.com/problemset/database/) — interview-style practice problems
- [DuckDB](https://duckdb.org/docs/) (https://duckdb.org/docs/) — run SQL locally over CSV/Parquet files with no server; great for practicing

Previous: [DE Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/) • **Next:** [Python for DE](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 29, 2026