PostgreSQL Practice Queries – Detailed Explanations

# 1) Schema Summary (from tables.sql)

CREATE TABLE categories (  
category\_id SERIAL NOT NULL PRIMARY KEY,  
category\_name VARCHAR(255),  
description VARCHAR(255)  
);

CREATE TABLE customers (  
 customer\_id SERIAL NOT NULL PRIMARY KEY,  
 customer\_name VARCHAR(255),  
 contact\_name VARCHAR(255),  
 address VARCHAR(255),  
 city VARCHAR(255),  
 postal\_code VARCHAR(255),  
 country VARCHAR(255)  
);

CREATE TABLE products (  
 product\_id SERIAL NOT NULL PRIMARY KEY,  
 product\_name VARCHAR(255),  
 category\_id INT,  
 unit VARCHAR(255),  
 price DECIMAL(10, 2)  
);

CREATE TABLE orders (  
 order\_id SERIAL NOT NULL PRIMARY KEY,  
 customer\_id INT,  
 order\_date DATE  
);

CREATE TABLE order\_details (  
 order\_detail\_id SERIAL NOT NULL PRIMARY KEY,  
 order\_id INT,  
 product\_id INT,  
 quantity INT  
);

CREATE TABLE testproducts (  
 testproduct\_id SERIAL NOT NULL PRIMARY KEY,  
 product\_name VARCHAR(255),  
 category\_id INT  
);

# 2) Query-by-Query Explanations

## 1. Query

SELECT customer\_name, country FROM customers;

### What it does

Returns two columns (customer\_name, country) from the customers table. Good for projecting only the fields you need.

* No filtering, so it returns all rows.
* Projection reduces bandwidth vs SELECT \*.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 2. Query

SELECT \* FROM customers;

### What it does

Returns every column and row from customers.

* Useful for quick inspection; avoid in production APIs.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 3. Query

SELECT DISTINCT country FROM customers;

### What it does

Lists unique country values present in customers.

* DISTINCT removes duplicates at the end of the SELECT pipeline.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 4. Query

SELECT COUNT(DISTINCT country) FROM customers;

### What it does

Counts the number of unique countries across all customers.

* Equivalent to SELECT COUNT(\*) FROM (SELECT DISTINCT country ...) t.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 5. Query

SELECT \* FROM customers WHERE city = 'London';

### What it does

Returns every column and row from customers.

* Useful for quick inspection; avoid in production APIs.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 6. Query

SELECT \* FROM customers WHERE customer\_id = 19;

### What it does

Returns every column and row from customers.

* Useful for quick inspection; avoid in production APIs.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 7. Query

SELECT \* FROM customers WHERE customer\_id > 80;

### What it does

Returns every column and row from customers.

* Useful for quick inspection; avoid in production APIs.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 8. Query

SELECT \* FROM products ORDER BY price;

### What it does

Orders products by price ascending.

* ORDER BY requires a sort; an index on price can satisfy it.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 9. Query

SELECT \* FROM products ORDER BY price DESC;

### What it does

Orders products by price descending.

* Same as ascending but reversed; index on price DESC can help.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 10. Query

SELECT \* FROM customers LIMIT 20;

### What it does

Returns every column and row from customers.

* Useful for quick inspection; avoid in production APIs.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 11. Query

SELECT \* FROM customers LIMIT 20 OFFSET 40;

### What it does

Returns every column and row from customers.

* Useful for quick inspection; avoid in production APIs.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 12. Query

SELECT MIN(price) FROM products;

### What it does

Returns the minimum product price.

* MIN() ignores NULLs.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 13. Query

SELECT MAX(price) FROM products;

### What it does

Returns the maximum product price.

* MAX() ignores NULLs.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 14. Query

SELECT MIN(price) AS lowest\_price FROM products;

### What it does

Same as MIN(price) but labels the result column as lowest\_price.

* AS controls output column name.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 15. Query

SELECT MAX(price) AS highest\_price FROM products;

### What it does

Same as MAX(price) but labels the result column as highest\_price.

* AS controls output column name.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 16. Query

SELECT COUNT(customer\_id) FROM customers;

### What it does

Counts non-NULL customer\_id values.

* COUNT(column) skips NULLs; COUNT(\*) counts rows.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 17. Query

SELECT COUNT(customer\_id) FROM customers WHERE city = 'London';

### What it does

Filters customers whose city equals 'London'.

* Uses an equality predicate; index on city helps.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 18. Query

SELECT SUM(quantity) FROM order\_details;

### What it does

Sums the quantity across all order line items.

* If you want total revenue, multiply quantity \* price then sum.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 19. Query

SELECT AVG(price) FROM products;

### What it does

Computes average product price.

* AVG() ignores NULLs.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 20. Query

SELECT AVG(price)::NUMERIC(10, 2) FROM products;

### What it does

Casts AVG(price) to NUMERIC(10,2) for fixed 2-decimal output.

* Type casting controls result precision.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 21. Query

SELECT \* FROM customers WHERE customer\_name LIKE 'A%';

### What it does

Returns every column and row from customers.

* Useful for quick inspection; avoid in production APIs.

### Tips & Variations

* Speed up prefix searches (e.g., 'A%') with a btree index; for contains ('%x%'), consider pg\_trgm & a GIN index.

## 22. Query

SELECT \* FROM customers WHERE customer\_name LIKE '%A%';

### What it does

Returns every column and row from customers.

* Useful for quick inspection; avoid in production APIs.

### Tips & Variations

* Speed up prefix searches (e.g., 'A%') with a btree index; for contains ('%x%'), consider pg\_trgm & a GIN index.

## 23. Query

SELECT \* FROM customers WHERE customer\_name ILIKE 'a%';

### What it does

Returns every column and row from customers.

* Useful for quick inspection; avoid in production APIs.

### Tips & Variations

* Speed up prefix searches (e.g., 'A%') with a btree index; for contains ('%x%'), consider pg\_trgm & a GIN index.

## 24. Query

SELECT \* FROM customers WHERE customer\_name LIKE '%en';

### What it does

Returns every column and row from customers.

* Useful for quick inspection; avoid in production APIs.

### Tips & Variations

* Speed up prefix searches (e.g., 'A%') with a btree index; for contains ('%x%'), consider pg\_trgm & a GIN index.

## 25. Query

SELECT \* FROM customers WHERE city LIKE 'L\_nd\_\_';

### What it does

Returns every column and row from customers.

* Useful for quick inspection; avoid in production APIs.

### Tips & Variations

* Speed up prefix searches (e.g., 'A%') with a btree index; for contains ('%x%'), consider pg\_trgm & a GIN index.

## 26. Query

SELECT \* FROM customers WHERE country IN('Germany', 'UK', 'France');

### What it does

Returns every column and row from customers.

* Useful for quick inspection; avoid in production APIs.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 27. Query

SELECT \* FROM customers WHERE country NOT IN('Germany', 'UK', 'France');

### What it does

Returns every column and row from customers.

* Useful for quick inspection; avoid in production APIs.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 28. Query

SELECT \* FROM customers WHERE customer\_id IN (SELECT customer\_id FROM orders);

### What it does

Returns every column and row from customers.

* Useful for quick inspection; avoid in production APIs.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 29. Query

SELECT \* FROM customers WHERE customer\_id NOT IN (SELECT customer\_id FROM orders);

### What it does

Returns every column and row from customers.

* Useful for quick inspection; avoid in production APIs.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 30. Query

SELECT \* FROM products WHERE price BETWEEN 10 AND 15;

### What it does

Filters products with price between 10 and 15 inclusive.

* BETWEEN is inclusive at both ends.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 31. Query

SELECT \* FROM products WHERE product\_name BETWEEN 'Pavlova' AND 'Tofu' ORDER BY product\_name;

### What it does

Filters rows where product\_name sorts between 'Pavlova' and 'Tofu' (lexicographic range).

* Works by text sort order; collation affects boundaries.
* ORDER BY product\_name makes output deterministic.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 32. Query

SELECT \* FROM orders WHERE order\_date BETWEEN '2023-04-12' AND '2023-05-05';

### What it does

Filters orders placed between 2023-04-12 and 2023-05-05 inclusive.

* Date literals are YYYY-MM-DD; timezones are not considered.

### Tips & Variations

* Store dates in DATE or TIMESTAMP; indexes enable efficient range scans.

## 33. Query

SELECT customer\_id AS id FROM customers;

### What it does

Renames customer\_id column to id in the result set.

* Aliases are presentation only; they don't rename the table column.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 34. Query

SELECT product\_name || unit AS product FROM products;

### What it does

Concatenates product\_name and unit directly.

* Use || for string concatenation in PostgreSQL.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 35. Query

SELECT product\_name || ' ' || unit AS product FROM products;

### What it does

Concatenates product\_name, a space, and unit.

* Readable product label; consider TRIM to avoid double spaces.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 36. Query

SELECT product\_name AS "My Great Products" FROM products;

### What it does

Uses a quoted alias to preserve case and spaces.

* Double quotes keep exact casing in PostgreSQL.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 37. Query

SELECT testproduct\_id, product\_name, category\_name FROM testproducts INNER JOIN categories ON testproducts.category\_id = categories.category\_id;

### What it does

Inner join between testproducts and categories on category\_id.

* Returns only rows with matching category ids in both tables.

### Tips & Variations

* Make sure join keys are indexed on both sides to avoid hash/merge joins on large tables.

## 38. Query

SELECT testproduct\_id, product\_name, category\_name FROM testproducts LEFT JOIN categories ON testproducts.category\_id = categories.category\_id;

### What it does

Left join: all testproducts plus matching categories when available.

* Unmatched categories appear as NULLs.

### Tips & Variations

* Make sure join keys are indexed on both sides to avoid hash/merge joins on large tables.

## 39. Query

SELECT testproduct\_id, product\_name, category\_name FROM testproducts RIGHT JOIN categories ON testproducts.category\_id = categories.category\_id;

### What it does

Right join: all categories plus matching testproducts.

* Opposite perspective of LEFT JOIN.

### Tips & Variations

* Make sure join keys are indexed on both sides to avoid hash/merge joins on large tables.

## 40. Query

SELECT testproduct\_id, product\_name, category\_name FROM testproducts FULL JOIN categories ON testproducts.category\_id = categories.category\_id;

### What it does

Full outer join: union of LEFT and RIGHT (all rows from both sides).

* Unmatched rows are filled with NULLs.

### Tips & Variations

* Make sure join keys are indexed on both sides to avoid hash/merge joins on large tables.

## 41. Query

SELECT testproduct\_id, product\_name, category\_name FROM testproducts CROSS JOIN categories;

### What it does

Cartesian product of testproducts × categories.

* Size equals rows(testproducts) × rows(categories); use carefully.

### Tips & Variations

* Make sure join keys are indexed on both sides to avoid hash/merge joins on large tables.

## 42. Query

SELECT product\_id, product\_name FROM products UNION SELECT testproduct\_id, product\_name FROM testproducts ORDER BY product\_id;

### What it does

UNION merges distinct rows from two compatible SELECTs.

* Eliminates duplicates; incurs a de-dup sort/hash step.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 43. Query

SELECT product\_id, product\_name FROM products UNION ALL SELECT testproduct\_id, product\_name FROM testproducts ORDER BY product\_id;

### What it does

UNION ALL appends rows from both SELECTs without de-duplicating.

* Faster than UNION; duplicates remain.

### Tips & Variations

* If duplicates matter downstream, switch to UNION; if not, UNION ALL is faster.

## 44. Query

SELECT COUNT(customer\_id), country FROM customers GROUP BY country;

### What it does

Counts customers per country.

* GROUP BY country collapses rows into per-country groups.

### Tips & Variations

* Aggregate on selective groups to reduce memory; add HAVING only for post-aggregation filters.

## 45. Query

SELECT customers.customer\_name, COUNT(orders.order\_id) FROM orders LEFT JOIN customers ON orders.customer\_id = customers.customer\_id GROUP BY customer\_name;

### What it does

Counts orders per customer name.

* LEFT JOIN from orders to customers returns only customers who have at least one order.
* To include customers with zero orders, reverse LEFT JOIN (customers LEFT JOIN orders).

### Tips & Variations

* Make sure join keys are indexed on both sides to avoid hash/merge joins on large tables.
* Aggregate on selective groups to reduce memory; add HAVING only for post-aggregation filters.

## 46. Query

SELECT COUNT(customer\_id), country FROM customers GROUP BY country HAVING COUNT(customer\_id) > 5;

### What it does

Counts customers per country.

* GROUP BY country collapses rows into per-country groups.

### Tips & Variations

* Aggregate on selective groups to reduce memory; add HAVING only for post-aggregation filters.

## 47. Query

SELECT order\_details.order\_id, SUM(products.price) FROM order\_details LEFT JOIN products ON order\_details.product\_id = products.product\_id GROUP BY order\_id HAVING SUM(products.price) > 400.00;

### What it does

Sums product prices per order, then filters orders with sum > 400.

* Important: summing price alone ignores quantity; prefer SUM(price \* quantity).

### Tips & Variations

* Make sure join keys are indexed on both sides to avoid hash/merge joins on large tables.
* Aggregate on selective groups to reduce memory; add HAVING only for post-aggregation filters.

## 48. Query

SELECT customers.customer\_name, SUM(products.price) FROM order\_details LEFT JOIN products ON order\_details.product\_id = products.product\_id LEFT JOIN orders ON order\_details.order\_id = orders.order\_id LEFT JOIN customers ON orders.customer\_id = customers.customer\_id GROUP BY customer\_name HAVING SUM(products.price) > 1000.00;

### What it does

Sums product prices per customer across their orders.

* Important: multiply price by quantity for true revenue.

### Tips & Variations

* Make sure join keys are indexed on both sides to avoid hash/merge joins on large tables.
* Aggregate on selective groups to reduce memory; add HAVING only for post-aggregation filters.

## 49. Query

SELECT customers.customer\_name FROM customers WHERE EXISTS(SELECT order\_id FROM orders WHERE customer\_id = customers.customer\_id);

### What it does

Returns customer names where at least one related order exists.

* EXISTS stops at first match; very efficient for semi-joins.

### Tips & Variations

* EXISTS/NOT EXISTS are NULL-safe and often outperform IN/NOT IN on large subqueries.

## 50. Query

SELECT customers.customer\_name FROM customers WHERE NOT EXISTS(SELECT order\_id FROM orders WHERE customer\_id = customers.customer\_id);

### What it does

Returns customer names with no orders.

* NOT EXISTS is NULL-safe, unlike NOT IN with a subquery.

### Tips & Variations

* EXISTS/NOT EXISTS are NULL-safe and often outperform IN/NOT IN on large subqueries.

## 51. Query

SELECT product\_name FROM products WHERE product\_id = ANY (SELECT product\_id FROM order\_details WHERE quantity > 120);

### What it does

Selects product\_name where product\_id matches ANY of the product\_ids from order\_details with quantity > 120.

* = ANY(subquery) is equivalent to IN(subquery).

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 52. Query

SELECT product\_name FROM products WHERE product\_id = ALL (SELECT product\_id FROM order\_details WHERE quantity > 10);

### What it does

Selects product\_name where product\_id equals ALL values from the subquery.

* Using '=' with ALL is rarely useful unless the subquery returns one row; more common are comparisons like price > ALL(subquery).

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

## 53. Query

SELECT product\_name, CASE WHEN price < 10 THEN 'Low price product' WHEN price > 50 THEN 'High price product' ELSE 'Normal product' END AS "price category" FROM products;

### What it does

Classifies products by price using a CASE expression into 'Low', 'High', or 'Normal' buckets.

* CASE returns the first matching WHEN branch; ELSE covers remaining rows.

### Tips & Variations

* Validate output with SELECT COUNT(\*) wrappers or small LIMITs while iterating.

# 3) Common SQL Functions – Meanings & Usage

## LIKE

Performs pattern matching on strings. '%' matches any sequence of characters, '\_' matches a single character.

## ILIKE

PostgreSQL-specific, case-insensitive version of LIKE.

## MIN

Aggregate function that returns the minimum value in a set of values (ignores NULLs).

## MAX

Aggregate function that returns the maximum value in a set of values (ignores NULLs).

## COUNT

Counts rows. COUNT(\*) counts all rows; COUNT(column) counts only non-NULL values of that column.

## DISTINCT

Removes duplicate rows from the result set. Often used with SELECT.

## SUM

Adds up all numeric values in a set (ignores NULLs).

## AVG

Computes the arithmetic mean (average) of numeric values (ignores NULLs).

## BETWEEN

Checks if a value falls within a specified inclusive range.

## IN

Checks if a value matches any value in a list or subquery (equivalent to OR of multiple conditions).

## NOT IN

Checks that a value does not match any value in a list or subquery. Beware NULL handling issues.

## EXISTS

Tests whether a subquery returns at least one row. Returns TRUE if it does, FALSE otherwise.

## NOT EXISTS

Tests whether a subquery returns no rows. Returns TRUE if none found.

## ANY

Compares a value to each value returned by a subquery. Equivalent to using IN in many cases.

## ALL

Compares a value against all values in a subquery. The condition must hold true for every value.

## CASE

Conditional expression that returns values based on WHEN...THEN...ELSE logic.

## COALESCE

Returns the first non-NULL value from its arguments.

## NULLIF

Returns NULL if the two arguments are equal; otherwise returns the first argument.

## CAST / ::

Converts a value from one data type to another (e.g., AVG(price)::NUMERIC(10,2)).

## CONCAT / ||

Concatenates (joins) string values into one.

## ROUND

Rounds a numeric value to a specified number of decimal places.

## NOW

Returns the current date and time (timestamp).

## CURRENT\_DATE

Returns the current date without time.

## LENGTH

Returns the number of characters in a string.

## UPPER

Converts a string to uppercase.

## LOWER

Converts a string to lowercase.

## TRIM

Removes leading and trailing spaces (or specified characters) from a string.

**4) General Best Practices & Tips (Common for All Queries)**

* **Avoid SELECT \*** → Always select only the columns you need to reduce overhead.
* **Use indexes wisely** → Create indexes on columns that are often used in WHERE, JOIN, and ORDER BY.
* **Efficient pagination** → Use LIMIT with ORDER BY. Avoid large OFFSET values; instead use **keyset pagination** for better performance.
* **Aggregate carefully** → SUM, AVG, COUNT ignore NULL. Use COALESCE if you need to treat NULL as 0.
* **Text search** → LIKE 'prefix%' can use normal indexes, but '%substring%' needs the pg\_trgm extension + GIN index.
* **EXISTS vs IN** → Use EXISTS / NOT EXISTS instead of IN / NOT IN with subqueries (faster and NULL-safe).
* **Revenue calculations** → Always compute totals as SUM(price \* quantity) instead of just SUM(price).
* **UNION vs UNION ALL** → Use UNION ALL for speed if you don’t need to remove duplicates.
* **WHERE vs HAVING** → Use WHERE to filter rows *before* aggregation, HAVING to filter *after*.
* **Normalize vs Denormalize** → Keep schema normalized to avoid redundancy; denormalize selectively for reporting/performance.
* **Transactions** → Use transactions (BEGIN, COMMIT, ROLLBACK) when modifying multiple related tables.
* **Vacuum & Analyze** → Regularly run VACUUM ANALYZE or enable autovacuum to keep statistics up-to-date

5) JSON / JSONB in PostgreSQL

**1. JSON vs JSONB**

* **JSON** → Stores text exactly as provided (slower for queries, keeps whitespace and formatting).
* **JSONB** → Binary storage (faster for search, removes duplicate keys, ignores whitespace).  
  👉 **Best practice**: Use JSONB unless you specifically need raw JSON formatting.

**2. Creating Tables**

CREATE TABLE products (id SERIAL PRIMARY KEY, name TEXT NOT NULL, details JSONB);

* details column stores flexible metadata for each product.

**3. Inserting Data**

INSERT INTO products (name, details) VALUES

('Laptop', '{"brand": "Dell", "ram": "16GB", "cpu": "i7"}'),

('Phone', '{"brand": "Apple", "ram": "6GB", "camera": "12MP"}');

✔ JSON is written in **single quotes '...'** (because SQL strings) but contains **double quotes inside "..."**.

**4. Querying JSON**

* -> → Get JSON object (still JSON).
* ->> → Get JSON value as **text**.
* #> → Get nested JSON object.
* #>> → Get nested JSON value as text.

Examples:

-- Full object

SELECT details->'ram' FROM products;

-- Extract text

SELECT details->>'cpu' FROM products;

-- Nested path

SELECT details#>>'{specs,storage}' FROM products;

**5. Filtering with JSON**

-- Products where brand = Dell

SELECT \* FROM products

WHERE details->>'brand' = 'Dell';

-- Products with RAM >= 8GB

SELECT \* FROM products

WHERE (regexp\_replace(details->>'ram','[^0-9]','','g'))::int >= 8;

👉 regexp\_replace strips non-numeric characters so you can cast to integer.

**6. Updating JSON**

-- Add new key

UPDATE products

SET details = jsonb\_set(details, '{warranty}', '"2 years"')

WHERE name = 'Laptop';

-- Update nested value

UPDATE products

SET details = jsonb\_set(details, '{ram}', '"32GB"')

WHERE name = 'Laptop';

**7. JSON Operators**

* ? → check if key exists
* ?| → check if **any key** in list exists
* ?& → check if **all keys** in list exist

Examples:

SELECT \* FROM products WHERE details ? 'brand';

SELECT \* FROM products WHERE details ?& array['brand','ram'];

**8. JSON Functions**

-- Get all keys

SELECT jsonb\_object\_keys(details) FROM products;

-- Pretty print

SELECT jsonb\_pretty(details) FROM products;

-- Expand JSON into rows

SELECT name, key, value FROM products, jsonb\_each(details);

**9. Aggregation with JSON**

-- Build JSON array from rows

SELECT json\_agg(name) FROM products;

-- Build JSON object

SELECT json\_build\_object('name', name, 'brand', details->>'brand')

FROM products;

**10. Indexing JSON**

-- GIN index on JSONB column

CREATE INDEX idx\_products\_details ON products USING gin(details);

-- Now queries are fast:

SELECT \* FROM products WHERE details->>'brand' = 'Dell';

**📝 PostgreSQL Mock Test (with Answers)**

**1. Basic Query**

**Q: Select all columns from a table called customers.**

**A: SELECT \* FROM customers;**

**2. Filtering**

**Q: Get all customers from London.  
A: SELECT \* FROM customers WHERE city = 'London';**

**3. Sorting**

**Q: Show all products sorted by price in descending order.  
A: SELECT \* FROM products ORDER BY price DESC;**

**4. Aggregation**

**Q: Find the total number of orders placed.  
A: SELECT COUNT(\*) FROM orders;**

**5. Grouping**

**Q: Count customers per country.  
A: SELECT country, COUNT(\*) AS total\_customers FROM customers GROUP BY country;**

**6. JOIN**

**Q: Show each order with the customer’s name.  
A: SELECT o.id AS order\_id, c.name AS customer\_name FROM orders o JOIN customers c ON o.customer\_id = c.id;**

**7. LEFT JOIN**

**Q: List all customers and their orders, including those with no orders.  
A: SELECT c.name, o.id AS order\_id FROM customers c LEFT JOIN orders o ON c.id = o.customer\_id;**

**8. Subquery**

**Q: Get all customers who have placed at least one order.  
A: SELECT \* FROM customers WHERE id IN (SELECT customer\_id FROM orders);**

**9. EXISTS**

**Q: Find customers who have not placed any orders.  
A: SELECT \* FROM customers c**

**WHERE NOT EXISTS (**

**SELECT 1 FROM orders o WHERE o.customer\_id = c.id**

**);**

**10. BETWEEN**

**Q: Get all products with price between 100 and 500.  
A: SELECT \* FROM products WHERE price BETWEEN 100 AND 500;**

**11. UNION**

**Q: Combine product IDs from products and testproducts tables (no duplicates).  
A: SELECT id FROM products UNION SELECT testproduct\_id FROM testproducts;**

**12. Window Function**

**Q: Rank products by price (highest first).  
A: SELECT name, price, RANK() OVER (ORDER BY price DESC) AS price\_rank FROM products;**

**13. CTE**

**Q: Show customers who registered in the last 30 days using a CTE.  
A: WITH recent\_customers AS (**

**SELECT \* FROM customers**

**WHERE created\_at > NOW() - interval '30 days'**

**)**

**SELECT \* FROM recent\_customers;**

**14. JSON Query**

**Q: Find all products where the brand = 'Dell' (stored in JSONB column details).  
A: SELECT \* FROM products WHERE details->>'brand' = 'Dell';**

**15. Update JSON**

**Q: Add a "warranty": "2 years" field to all products.  
A: UPDATE products SET details = jsonb\_set(details, '{warranty}', '"2 years"');**

**⚡ Query Optimization Practice Set**

EXPLAIN ANALYZE

SELECT \* FROM products WHERE price = 500;

👉 Create an index:

CREATE INDEX idx\_products\_price ON products(price);

Re-run and compare plan (should switch from Seq Scan → Index Scan).

**✅ How to Practice**

1. Run each query with EXPLAIN ANALYZE.
2. Note the execution plan (Seq Scan, Index Scan, Hash Join, etc.).
3. Add suggested indexes or rewrite queries.
4. Re-run and compare costs & timings.

**🎯 Conclusion**

**I’ve successfully explored PostgreSQL from basics to advanced:**

* **✅ Core SQL: CRUD, filtering, ordering, grouping**
* **✅ Joins & Subqueries: Inner/Outer joins, IN/EXISTS**
* **✅ Aggregates & Window Functions**
* **✅ Advanced SQL: CTEs, UNION, HAVING**
* **✅ JSON/JSONB: Flexible storage, querying, indexing**
* **✅ Best Practices: Indexing, query optimization, schema design**

**This POC document now serves as a complete PostgreSQL study + practice guide — covering theory, queries, functions, JSON, tips, and mock questions with answers.**

**⚡ PostgreSQL SQL Cheat Sheet**

**Basics**

-- Select all

SELECT \* FROM table;

-- Select specific columns

SELECT col1, col2 FROM table;

-- Filtering

SELECT \* FROM table WHERE col = 'value';

-- Sorting

SELECT \* FROM table ORDER BY col DESC;

-- Limit & Offset

SELECT \* FROM table LIMIT 10 OFFSET 20;

**Aggregates**

SELECT COUNT(\*), SUM(col), AVG(col), MIN(col), MAX(col)

FROM table

WHERE condition;

**Grouping**

SELECT col, COUNT(\*)

FROM table

GROUP BY col

HAVING COUNT(\*) > 5;

**Joins**

-- Inner join

SELECT a.col, b.col

FROM a

JOIN b ON a.id = b.a\_id;

-- Left join

SELECT a.col, b.col

FROM a

LEFT JOIN b ON a.id = b.a\_id;

-- Right join

SELECT a.col, b.col

FROM a

RIGHT JOIN b ON a.id = b.a\_id;

-- Full join

SELECT a.col, b.col

FROM a

FULL JOIN b ON a.id = b.a\_id;

**Subqueries**

SELECT \* FROM table

WHERE col IN (SELECT col FROM other\_table);

SELECT \* FROM table t

WHERE EXISTS (SELECT 1 FROM other\_table o WHERE o.id = t.id);

**Set Operations**

SELECT col FROM a

UNION

SELECT col FROM b;

SELECT col FROM a

UNION ALL

SELECT col FROM b;

**Window Functions**

SELECT col, RANK() OVER (ORDER BY col DESC) AS rank

FROM table;

CTEs

WITH recent AS (

SELECT \* FROM table WHERE created\_at > NOW() - interval '30 days'

)

SELECT \* FROM recent;

**JSON / JSONB**

-- Insert JSON

INSERT INTO table (details) VALUES ('{"key": "value"}');

-- Access keys

SELECT details->'key', details->>'key' FROM table;

-- Nested path

SELECT details#>>'{nested,key}' FROM table;

-- Update

UPDATE table SET details = jsonb\_set(details, '{newkey}', '"value"');

-- Check key existence

SELECT \* FROM table WHERE details ? 'key';

-- Aggregate JSON

SELECT json\_agg(col) FROM table;

**Indexing**

-- Simple index

CREATE INDEX idx\_col ON table(col);

-- JSONB index

CREATE INDEX idx\_json ON table USING gin(details);

**Admin**

-- Create role

CREATE ROLE username LOGIN PASSWORD 'pass';

-- Grant privileges

GRANT SELECT ON table TO username;

-- Backup

pg\_dump -U user dbname > backup.sql

-- Restore

psql -U user -d dbname -f backup.sql