**PostgreSQL Topics**

**1. Introduction to PostgreSQL**

1.1 What is PostgreSQL?  
1.2 Features of PostgreSQL  
1.3 Advantages of PostgreSQL  
1.4 PostgreSQL vs Other Databases  
1.5 Installation and Configuration  
1.6 PostgreSQL Architecture  
  
**2. Database & Schema Management**

2.1 Creating and Dropping Databases (CREATE DATABASE, DROP DATABASE)  
2.2 Managing Schemas (CREATE SCHEMA, DROP SCHEMA)  
2.3 Tablespaces and Storage Management  
2.4 Database Backup and Restore (pg\_dump, pg\_restore)

**3. Data Types in PostgreSQL**

3.1 Numeric Data Types (INTEGER, DECIMAL, FLOAT)  
3.2 String Data Types (CHAR, VARCHAR, TEXT)  
3.3 Date/Time Data Types (DATE, TIMESTAMP, INTERVAL)  
3.4 Boolean Data Type  
3.5 JSON and JSONB Data Types  
3.6 XML Data Type  
3.7 UUID Data Type  
3.8 Array Data Type  
3.9 Composite and Enumerated Data Types

**4. SQL Basics in PostgreSQL**

4.1 Creating and Managing Tables (CREATE TABLE, ALTER TABLE, DROP TABLE)  
4.2 Constraints (PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK, NOT NULL)  
4.3 Inserting Data (INSERT INTO)  
4.4 Updating Data (UPDATE)  
4.5 Deleting Data (DELETE, TRUNCATE)  
4.6 Selecting Data (SELECT, WHERE, ORDER BY, LIMIT, OFFSET)  
4.7 Joins (INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN)  
4.8 Grouping Data (GROUP BY, HAVING, AGGREGATE FUNCTIONS)

**5. Advanced Querying in PostgreSQL**

5.1 Common Table Expressions (CTE) (WITH RECURSIVE)  
5.2 Window Functions (RANK(), DENSE\_RANK(), ROW\_NUMBER(), LAG(), LEAD())  
5.3 Recursive Queries  
5.4 Pivoting Data (CROSSTAB)  
5.5 Full-Text Search (tsvector, tsquery, to\_tsquery())  
5.6 Materialized Views (CREATE MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW)

**6. Indexing & Performance Optimization**

6.1 Types of Indexes (B-Tree, GIN, GiST, BRIN, Hash)  
6.2 Creating and Managing Indexes (CREATE INDEX, DROP INDEX)  
6.3 Query Execution Plan (EXPLAIN ANALYZE)  
6.4 Autovacuum and Manual VACUUM  
6.5 Table Partitioning (LIST, RANGE, HASH, DEFAULT)

**7. Transactions and Locking**

7.1 ACID Properties of Transactions  
7.2 BEGIN, COMMIT, ROLLBACK  
7.3 Savepoints (SAVEPOINT, ROLLBACK TO SAVEPOINT)  
7.4 Transaction Isolation Levels (READ COMMITTED, REPEATABLE READ, SERIALIZABLE)  
7.5 Deadlocks and Locking (pg\_locks, LOCK TABLE)

**8. JSON & XML in PostgreSQL**

8.1 JSON vs JSONB  
8.2 Storing JSON Data  
8.3 Querying JSON Data (->, ->>, jsonb\_path\_query())  
8.4 Modifying JSON Data (jsonb\_set(), jsonb\_insert())  
8.5 XML Handling (XMLTABLE(), xpath())

**9. Security & User Management**

9.1 Creating Users and Roles (CREATE USER, CREATE ROLE)  
9.2 Granting and Revoking Permissions (GRANT, REVOKE)  
9.3 Row-Level Security (RLS) (CREATE POLICY, ALTER POLICY)  
9.4 SSL and Encryption in PostgreSQL

**10. Replication and High Availability**

10.1 Streaming Replication  
10.2 Logical Replication (CREATE PUBLICATION, CREATE SUBSCRIPTION)  
10.3 Hot Standby and Read Replicas  
10.4 Failover and Clustering

**PL/pgSQL (Procedural Language for PostgreSQL)**

**1. Introduction to PL/pgSQL**

1.1 What is PL/pgSQL?  
1.2 Advantages of PL/pgSQL  
1.3 Enabling PL/pgSQL in PostgreSQL  
  
**2. PL/pgSQL Basics**

2.1 Declaring Variables (DECLARE)  
2.2 Assigning Values (:=)  
2.3 Data Types in PL/pgSQL  
2.4 Control Structures (IF-THEN-ELSE, CASE, LOOPS)

**3. Stored Procedures in PL/pgSQL**

3.1 Creating a Stored Procedure (CREATE PROCEDURE)  
3.2 Executing a Stored Procedure (CALL)  
3.3 Dropping a Stored Procedure (DROP PROCEDURE)  
  
**4. Functions in PL/pgSQL**

4.1 Creating a Function (CREATE FUNCTION)  
4.2 Returning Scalar and Table Values (RETURNS TABLE, RETURNS RECORD)  
4.3 Using OUT and INOUT Parameters  
4.4 Using RETURN NEXT for Multiple Rows

**5. Control Structures in PL/pgSQL**

5.1 Conditional Statements (IF-THEN-ELSE)  
5.2 Case Statements (CASE-WHEN)  
5.3 Loops (LOOP, WHILE, FOR, EXIT)

**6. Exception Handling in PL/pgSQL**

6.1 Handling Errors with EXCEPTION  
6.2 Catching Specific Errors (WHEN OTHERS THEN)  
6.3 Using RAISE NOTICE for Debugging

**7. Cursors in PL/pgSQL**

7.1 Declaring and Opening a Cursor  
7.2 Fetching Data from a Cursor  
7.3 Closing a Cursor

**8. Dynamic SQL in PL/pgSQL**

8.1 Using EXECUTE for Dynamic Queries  
8.2 Using USING Clause in Dynamic SQL  
8.3 Preventing SQL Injection

**9. Triggers in PL/pgSQL**

9.1 Creating Triggers (CREATE TRIGGER)  
9.2 Using BEFORE, AFTER, INSTEAD OF Triggers  
9.3 Trigger Functions (EXECUTE FUNCTION)

**10. Performance Tuning in PL/pgSQL**

10.1 Optimizing Loops and Queries  
10.2 Using PLPGSQL\_OPTIMIZER  
10.3 Analyzing Execution Plans (EXPLAIN ANALYZE)  
  
  
**Main Topics:**

**Cursor**A cursor in PostgreSQL is used to fetch a large number of rows one by one instead of retrieving all rows at once. This improves performance and memory efficiency when dealing with large datasets.  
  
**🛠️ Key Features of Cursors**

**✔ Fetches rows one by one or in batches.  
✔ Helps in looping through query results in stored procedures or functions.  
✔ Reduces memory consumption compared to SELECT \* FROM table;.  
✔ Works inside PL/pgSQL functions, procedures, and DO blocks.**

**Cursor Lifecycle in PostgreSQL**

1️ Declare a Cursor – Define the cursor with a SELECT statement.  
2️ Open the Cursor – Open the cursor for fetching data.  
3️ Fetch Data – Retrieve rows using FETCH.  
4️ Move Cursor (Optional) – Navigate within the result set (MOVE, FETCH NEXT, FETCH PRIOR).  
5️ Close the Cursor – Free up memory by closing the cursor.

Cursor Types in PostgreSQL  
Unbound Cursor (Default)

A cursor that retrieves the entire result set.  
Bound Cursor with Arguments  
 A cursor that accepts parameters dynamically.  
SCROLL Cursor  
 Allows moving forward and backward in the result set.  
NO SCROLL Cursor  
 Allows only **forward movement**.  
  
  
**Basic Cursor Example:**DO $$

DECLARE

my\_cursor CURSOR FOR SELECT id, name FROM employees;

emp\_id INT;

emp\_name TEXT;

BEGIN

-- Open cursor (not required for simple cursors)

FOR emp\_id, emp\_name IN my\_cursor LOOP

RAISE NOTICE 'Employee ID: %, Employee Name: %', emp\_id, emp\_name;

END LOOP;

END $$;  
  
**Explanation**

✔ A cursor named my\_cursor is declared for the employees table.  
✔ We loop through the cursor and fetch each row.  
✔ RAISE NOTICE prints values.

**Cursor with Explicit Open, Fetch & Close  
Example 2: Using OPEN, FETCH, CLOSE Commands**DO $$

DECLARE

my\_cursor CURSOR FOR SELECT id, name FROM employees;

emp\_id INT;

emp\_name TEXT;

BEGIN

-- Open the cursor

OPEN my\_cursor;

LOOP

-- Fetch a row into variables

FETCH my\_cursor INTO emp\_id, emp\_name;

-- Exit loop when no more rows

EXIT WHEN NOT FOUND;

-- Print values

RAISE NOTICE 'Employee ID: %, Employee Name: %', emp\_id, emp\_name;

END LOOP;

-- Close the cursor

CLOSE my\_cursor;

END $$;

**Cursor with Dynamic Query  
Example 3: Dynamic Cursor (Using Variables in Cursors)**DO $$

DECLARE

department\_id INT := 2;

my\_cursor CURSOR FOR SELECT id, name FROM employees WHERE department = department\_id;

emp\_id INT;

emp\_name TEXT;

BEGIN

FOR emp\_id, emp\_name IN my\_cursor LOOP

RAISE NOTICE 'Employee ID: %, Employee Name: %', emp\_id, emp\_name;

END LOOP;

END $$;  
  
Explanation  
The cursor fetches employees from **a specific department** dynamically.  
✔ The department\_id value is used inside the WHERE clause.  
  
📌 Cursor with SCROLL (Fetching Forward & Backward)  
📝 Example 4: SCROLL Cursor  
  
DO $$

DECLARE

my\_cursor CURSOR SCROLL FOR SELECT id, name FROM employees;

emp\_id INT;

emp\_name TEXT;

BEGIN

-- Open cursor

OPEN my\_cursor;

-- Fetch first row

FETCH NEXT FROM my\_cursor INTO emp\_id, emp\_name;

RAISE NOTICE 'NEXT Employee ID: %, Name: %', emp\_id, emp\_name;

-- Fetch previous row (backward movement)

FETCH PRIOR FROM my\_cursor INTO emp\_id, emp\_name;

RAISE NOTICE 'PRIOR Employee ID: %, Name: %', emp\_id, emp\_name;

-- Close cursor

CLOSE my\_cursor;

END $$;  
  
**Explanation**

✔ SCROLL allows **both forward and backward** fetching.  
✔ FETCH NEXT moves forward, and FETCH PRIOR moves backward.

📌 Cursor Inside a Function  
Example 5: Using a Cursor Inside a Function  
  
CREATE OR REPLACE FUNCTION get\_all\_employees()

RETURNS TABLE(emp\_id INT, emp\_name TEXT) AS $$

DECLARE

my\_cursor CURSOR FOR SELECT id, name FROM employees;

BEGIN

FOR emp\_id, emp\_name IN my\_cursor LOOP

RETURN NEXT; -- Returns each row to the caller

END LOOP;

END $$ LANGUAGE plpgsql;  
  
**Explanation**

✔ Returns **all employees** using a function.  
✔ RETURN NEXT; outputs each row one by one.

📌 Cursor with Parameterized Query  
Example 6: Cursor with Parameters  
  
CREATE OR REPLACE FUNCTION get\_employees\_by\_department(dep\_id INT)

RETURNS TABLE(emp\_id INT, emp\_name TEXT) AS $$

DECLARE

my\_cursor CURSOR FOR SELECT id, name FROM employees WHERE department\_id = dep\_id;

BEGIN

FOR emp\_id, emp\_name IN my\_cursor LOOP

RETURN NEXT;

END LOOP;

END $$ LANGUAGE plpgsql;  
  
**🔍 Explanation**

✔ Takes dep\_id as a parameter.  
✔ Returns employees **only from the given department**.

**📌 Key Takeaways**

✅ **Cursors help process large datasets row by row.**  
✅ **Explicit cursors need OPEN, FETCH, and CLOSE statements.**  
✅ **SCROLL cursors allow both forward and backward fetching.**  
✅ **Parameterized cursors make queries dynamic.**  
✅ **Use cursors wisely as they impact performance.**

------------------------------  
**loops:**  
  
**What are Loops in PostgreSQL?**

Loops are used in **PL/pgSQL** to execute a block of statements **repeatedly** until a specific condition is met. They help in **iterating over rows, processing records, and automating repetitive tasks**.  
  
**📌 Types of Loops in PostgreSQL**

**1️ LOOP (Infinite Loop)**

* Runs **indefinitely** unless explicitly exited using EXIT.

**2️ WHILE Loop**

* Executes as long as the given condition is TRUE.

**3️ FOR Loop (Integer Loop)**

* Iterates over a range of numbers (e.g., 1 to 10).

**4️ FOR Loop (Cursor-Based Loop)**

* Iterates over a result set (each row from a query).

**5️ Nested Loops**

One loop inside another (useful for complex logic).  
  
LOOP (Infinite Loop):  
  
📝 Example 1: Simple Infinite Loop  
DO $$

DECLARE

counter INT := 1;

BEGIN

LOOP

RAISE NOTICE 'Counter: %', counter;

counter := counter + 1;

-- Exit loop after 5 iterations

EXIT WHEN counter > 5;

END LOOP;

END $$;  
  
**🔍 Explanation**

✔ The LOOP runs **indefinitely** unless we **explicitly** use EXIT WHEN counter > 5;.  
✔ Each iteration **increments** the counter by 1.  
✔ Stops execution **after 5 iterations**.

📌 2️ WHILE Loop:  
  
DO $$

DECLARE

counter INT := 1;

BEGIN

WHILE counter <= 5 LOOP

RAISE NOTICE 'Counter: %', counter;

counter := counter + 1;

END LOOP;

END $$;  
  
**🔍 Explanation**

✔ The loop runs **while** the condition counter <= 5 is TRUE.  
✔ It **increments** the counter inside the loop.  
✔ Stops execution **when counter > 5**.

📌 3️ FOR Loop (Integer Loop)  
📝 Example 3: Simple FOR Loop  
  
DO $$

BEGIN

FOR i IN 1..5 LOOP

RAISE NOTICE 'Iteration: %', i;

END LOOP;

END $$;  
**🔍 Explanation**

✔ The FOR loop **automatically** handles counter increments.  
✔ Runs **from 1 to 5**.

📝 Example 4: Reverse FOR Loop  
DO $$

BEGIN

FOR i IN REVERSE 5..1 LOOP

RAISE NOTICE 'Reverse Iteration: %', i;

END LOOP;

END $$;  
  
**🔍 Explanation**

✔ Runs **from 5 to 1** **(decrementing)**.

📌 4️ FOR Loop with Cursor (Iterating Over a Query)  
DO $$

DECLARE

emp RECORD;

BEGIN

FOR emp IN SELECT id, name FROM employees LOOP

RAISE NOTICE 'Employee ID: %, Name: %', emp.id, emp.name;

END LOOP;

END $$;  
  
**🔍 Explanation**

✔ Retrieves **each row** from employees.  
✔ Uses **a loop variable (emp RECORD)** to store row data.  
✔ Runs **until all rows are processed**.

📌 5️ Nested Loops (Loop inside Loop)  
DO $$

BEGIN

FOR i IN 1..3 LOOP

FOR j IN 1..2 LOOP

RAISE NOTICE 'Outer: %, Inner: %', i, j;

END LOOP;

END LOOP;

END $$;  
  
**🔍 Explanation**

✔ **Outer loop** runs from 1 to 3.  
✔ **Inner loop** runs from 1 to 2.  
✔ **Output:**

📌 EXIT and CONTINUE Statements  
📝 Example 7: Using EXIT to Stop a Loop  
  
DO $$

DECLARE

counter INT := 1;

BEGIN

LOOP

RAISE NOTICE 'Counter: %', counter;

EXIT WHEN counter >= 3; -- Stops at 3

counter := counter + 1;

END LOOP;

END $$;  
  
📝 Example 8: Using CONTINUE to Skip an Iteration  
DO $$

BEGIN

FOR i IN 1..5 LOOP

IF i = 3 THEN

CONTINUE; -- Skips iteration when i = 3

END IF;

RAISE NOTICE 'Iteration: %', i;

END LOOP;

END $$;  
  
📌 Cursor-Based Loops (Advanced Use Case)  
📝 Example 9: Using LOOP with an Explicit Cursor  
DO $$

DECLARE

my\_cursor CURSOR FOR SELECT id, name FROM employees;

emp\_id INT;

emp\_name TEXT;

BEGIN

-- Open Cursor

OPEN my\_cursor;

LOOP

FETCH my\_cursor INTO emp\_id, emp\_name;

EXIT WHEN NOT FOUND;

RAISE NOTICE 'Employee ID: %, Name: %', emp\_id, emp\_name;

END LOOP;

-- Close Cursor

CLOSE my\_cursor;

END $$;  
  
✔ Uses **explicit cursor** to iterate **row by row**.  
✔ Stops execution when **no more rows are found**.  
  
📌 Performance Considerations  
🔹 **Avoid Infinite Loops** (Always use EXIT WHEN).  
🔹 **Use FOR Loops Instead of LOOP + EXIT** (More efficient).  
🔹 **Use Cursor Loops for Large Datasets** (Handles memory better).  
  
📌 Summary  
✅ **LOOP** – Runs indefinitely unless explicitly stopped.  
✅ **WHILE Loop** – Runs while a condition is TRUE.  
✅ **FOR Loop (Integer Loop)** – Iterates over a range of numbers.  
✅ **FOR Loop (Cursor-Based)** – Iterates over a query result set.  
✅ **Nested Loops** – Loop inside another loop.  
✅ **EXIT** – Stops a loop when a condition is met.  
✅ **CONTINUE** – Skips an iteration and continues with the next one.  
  
-----------------------------  
XML:  
  
**What is XML in PostgreSQL?**

XML (Extensible Markup Language) is a widely used format for storing and exchanging structured data. PostgreSQL provides built-in support for **XML data types and functions**, allowing efficient storage, retrieval, and manipulation of XML data.

📌 XML Data Type in PostgreSQL  
PostgreSQL provides a **native XML data type** to store XML documents or fragments.

**🔍 Creating a Table with an XML Column**

CREATE TABLE employees (

id SERIAL PRIMARY KEY,

name TEXT NOT NULL,

details XML

);  
  
**📌 Inserting XML Data**

You can insert XML data using the xml keyword.

**📝 Example 1: Inserting XML Data**

INSERT INTO employees (name, details)

VALUES ('John Doe',

'<employee>

<id>1</id>

<position>Software Engineer</position>

<department>IT</department>

</employee>');  
  
📌 Retrieving XML Data  
SELECT name, details FROM employees;  
  
✔ Retrieves XML as-is from the database.  
📌 Converting Text to XML (Casting)  
  
SELECT XMLPARSE(DOCUMENT '<root><item>Value</item></root>');  
  
📌 Extracting Data from XML (XPath Queries)  
You can query XML data using the xpath() function.  
📝 Example 4: Extracting Values from XML  
SELECT xpath('/employee/position/text()', details) FROM employees;  
  
✔ Extracts the <position> value from the XML.  
✔ Returns an **array**, e.g., {"Software Engineer"}.  
  
📌 Extracting Multiple XML Elements  
📝 Example 5: Extracting Multiple Values  
SELECT

xpath('/employee/id/text()', details) AS emp\_id,

xpath('/employee/position/text()', details) AS position,

xpath('/employee/department/text()', details) AS department

FROM employees;  
  
**📌 Modifying XML Data**

PostgreSQL does **not support direct updates to XML fields**, so you must **replace** the XML value.

**📝 Example 6: Updating XML Data**

UPDATE employees

SET details = '<employee>

<id>1</id>

<position>Senior Software Engineer</position>

<department>IT</department>

</employee>'

WHERE name = 'John Doe';  
  
📌 XML Aggregation (Combining XML Data)  
You can concatenate multiple XML documents using xmlconcat().  
**Merging XML Documents:**  
SELECT xmlconcat(

XMLPARSE(DOCUMENT '<employee><name>John</name></employee>'),

XMLPARSE(DOCUMENT '<employee><name>Jane</name></employee>')

);  
  
📌 Converting Table Data to XML  
PostgreSQL provides xmlelement(), xmlforest(), and xmlagg() to generate XML from table data.  
  
SELECT xmlelement(NAME employee,

xmlelement(NAME id, id),

xmlelement(NAME name, name)

) AS xml\_output

FROM employees;  
✔ Converts table rows into **structured XML format**.  
  
📌 Using xmlforest() for Dynamic XML  
xmlforest() generates XML elements dynamically.

**📝 Example 9: Using xmlforest()**

SELECT xmlforest(id, name) FROM employees;  
  
✔ Converts each column into **XML elements**.  
  
📌 Using xmlagg() for Grouping XML  
xmlagg() helps in aggregating multiple rows into a single XML document.

**📝 Example 10: Using xmlagg()**

SELECT xmlelement(NAME employees, xmlagg(xmlelement(NAME employee, name)))

FROM employees;  
  
**📌 Key Takeaways**

✅ **PostgreSQL supports the XML data type** for structured storage.  
✅ **XPath functions (xpath())** allow querying XML content.  
✅ **Aggregation functions (xmlagg())** help in XML data grouping.  
✅ **Text-to-XML conversion (XMLPARSE())** ensures valid XML format.

-----------------------

📌 Types of User-Defined Data Types in PostgreSQL

1️ **Composite Types** – A structure similar to a table with multiple fields.  
2️ **Enumerated Types (ENUM)** – Defines a fixed set of possible values.  
3️ **Domain Types** – Adds constraints to existing types.  
4️ **Range Types** – Represents a range of values.

1️ Composite Types (Like a Struct)  
Composite types allow defining a **custom structure** similar to a table.  
  
CREATE TYPE employee\_type AS (

id INT,

name TEXT,

position TEXT,

salary NUMERIC

);  
  
✔ Defines a **structured type** with multiple attributes.  
  
  
**Using Composite Types**

You can use composite types in **tables, functions, and variables**.

CREATE TABLE company (

department TEXT,

manager employee\_type -- Using the custom type

);  
  
Inserting Data into a Composite Type:  
INSERT INTO company VALUES ('IT', ROW(1, 'John Doe', 'CTO', 200000));  
  
  
📌 2️ ENUM Types (Fixed Set of Values)  
ENUM types restrict a column to **predefined values**.  
  
📝 Example 5: Creating an ENUM Type  
CREATE TYPE job\_role AS ENUM ('Developer', 'Manager', 'HR', 'Admin');  
✔ The job\_role type can only have **one of these values**.

CREATE TABLE employees (

id SERIAL PRIMARY KEY,

name TEXT,

role job\_role -- Using ENUM type

);  
  
Ensures role only contains valid ENUM values  
  
📝 Example 7: Inserting and Querying ENUM Values  
INSERT INTO employees (name, role) VALUES ('Alice', 'Developer');

SELECT \* FROM employees WHERE role = 'Developer';  
  
📌 3️ Domain Types (Adding Constraints to Data Types)  
A **domain** is a type that adds **validation rules** on top of existing types.  
  
CREATE DOMAIN positive\_salary AS NUMERIC CHECK (VALUE > 0);  
  
CREATE TABLE payroll (

employee\_id INT PRIMARY KEY,

salary positive\_salary -- Uses the domain

);  
  
INSERT INTO payroll VALUES (1, -5000); -- ❌ ERROR: Value must be > 0  
  
  
📌 4️ Range Types (Handling Continuous Values)  
PostgreSQL allows defining **range types** for handling continuous values.  
  
CREATE TABLE project\_dates (

project\_name TEXT,

duration DATERANGE -- Stores a range of dates

);  
  
✔ The DATERANGE type stores **start and end dates** together.  
📝 Example 12: Inserting Data with a Range  
  
INSERT INTO project\_dates VALUES ('Project A', '[2025-01-01, 2025-12-31]');  
  
Querying Overlapping Ranges  
SELECT \* FROM project\_dates WHERE duration && '[2025-06-01, 2025-08-01]';  
  
**📌 Summary**

✅ **Composite Types** – Custom structures similar to tables.  
✅ **ENUM Types** – Restricts values to a predefined list.  
✅ **Domain Types** – Adds validation rules to existing data types.  
✅ **Range Types** – Handles continuous value ranges efficiently.

--------------------  
**🔹 What is JSON in PostgreSQL?**

PostgreSQL provides robust support for **JSON (JavaScript Object Notation)**, a lightweight data format used for storing and exchanging structured data. It offers two JSON-related data types:  
1️ **JSON** – Stores JSON as a text field (slower for queries).  
2️ **JSONB** – Stores JSON in a **binary format** (faster for querying and indexing).  
  
**📌 JSON vs JSONB in PostgreSQL**

| **Feature** | **JSON** | **JSONB** |
| --- | --- | --- |
| **Storage** | Text format | Binary format (optimized) |
| **Indexing** | Not indexable | Indexable using GIN indexes |
| **Insertion Speed** | Faster (no conversion) | Slightly slower (conversion required) |
| **Query Speed** | Slower (text operations) | Faster (optimized operations) |
| **Duplication** | Preserves input format | Removes spaces and duplicates |

✔ Use JSONB for better **performance** and indexing.  
✔ Use JSON if you need to **preserve formatting** exactly as inserted.

**📌 Creating a Table with a JSON Column**

You can store JSON data in a table by defining a column as JSON or JSONB.

**📝 Example 1: Creating a Table with JSONB**

CREATE TABLE employees (

id SERIAL PRIMARY KEY,

name TEXT,

details JSONB -- Stores structured JSON data

);

✔ The details column stores **JSON objects**.

**📌 Inserting JSON Data**

You can insert JSON data **as a valid JSON object**.

**📝 Example 2: Inserting JSON Data**

INSERT INTO employees (name, details)

VALUES ('John Doe',

'{"age": 30, "position": "Software Engineer", "skills": ["Python", "SQL"]}');

✔ The details column now holds structured JSON.

**📌 Retrieving JSON Data**

You can **fetch** JSON data using a simple SELECT statement.

**📝 Example 3: Selecting JSON Data**

SELECT name, details FROM employees;

✔ Returns the JSON structure as stored.

**📌 Extracting Values from JSON**

PostgreSQL provides operators and functions to query JSON data efficiently.

**🔹 JSON Operators**

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| -> | Extracts JSON object field (as JSON) | details -> 'position' |
| ->> | Extracts JSON field as **text** | details ->> 'position' |
| #> | Extracts JSON sub-element (as JSON) | details #> '{skills, 0}' |
| #>> | Extracts JSON sub-element as **text** | details #>> '{skills, 0}' |

**📝 Example 4: Extracting a JSON Field**

SELECT name, details ->> 'position' AS job\_role FROM employees;

✔ Extracts "position" as **text** (Software Engineer).

**📌 Querying JSON Arrays**

You can retrieve **array elements** from a JSON field.

**📝 Example 5: Extracting an Array Value**

SELECT details #>> '{skills, 0}' AS first\_skill FROM employees;

✔ Retrieves the **first skill** (Python).

**📌 Filtering JSON Data**

You can filter rows based on JSON fields.

**📝 Example 6: Filtering Based on a JSON Field**

SELECT \* FROM employees WHERE details ->> 'position' = 'Software Engineer';

✔ Returns employees with "position": "Software Engineer".

**📌 Updating JSON Data**

PostgreSQL does **not allow partial updates** to JSON fields directly, so you must **replace** the entire JSON object.

**📝 Example 7: Updating JSON Data**

UPDATE employees

SET details = jsonb\_set(details, '{position}', '"Senior Software Engineer"')

WHERE name = 'John Doe';

✔ Updates the "position" field inside JSON.

**📌 Deleting a JSON Field**

You can remove a specific **key** from JSON using the - operator.

**📝 Example 8: Removing a Field from JSON**

UPDATE employees

SET details = details - 'age'

WHERE name = 'John Doe';

✔ Removes the "age" key from JSON.

**📌 Checking If a Key Exists in JSON**

You can check if a **specific key** exists in JSON data.

**📝 Example 9: Checking for Key Existence**

SELECT \* FROM employees WHERE details ? 'skills';

✔ Finds employees who have the "skills" key.

**📌 Indexing JSONB for Faster Queries**

JSONB allows **GIN indexing** for improved performance.

**📝 Example 10: Creating a GIN Index on JSONB**

CREATE INDEX idx\_details ON employees USING GIN (details);

✔ Indexing speeds up JSON queries.

**📌 Searching Inside JSON**

PostgreSQL provides powerful search capabilities inside JSONB.

**📝 Example 11: Searching for JSON Values**

SELECT \* FROM employees WHERE details @> '{"position": "Software Engineer"}';

✔ Finds employees **where "position" is "Software Engineer"**.

**📌 Aggregating JSON Data**

You can **group and aggregate** JSON data.

**📝 Example 12: Aggregating JSON Data**

SELECT json\_agg(details) FROM employees;

✔ Combines multiple JSON rows into **one JSON array**.

**📌 JSON Functions in PostgreSQL**

| **Function** | **Description** | **Example** |
| --- | --- | --- |
| jsonb\_pretty() | Formats JSON for readability | SELECT jsonb\_pretty(details) FROM employees; |
| jsonb\_array\_elements() | Expands JSON array elements into rows | SELECT jsonb\_array\_elements(details -> 'skills') FROM employees; |
| jsonb\_object\_keys() | Lists all keys in a JSON object | SELECT jsonb\_object\_keys(details) FROM employees; |

**📌 Summary**

✅ **JSON** stores structured data in text format.  
✅ **JSONB** is a binary-optimized format, allowing indexing and faster queries.  
✅ **Operators (->, ->>, #>, #>>)** help extract values.  
✅ **Filtering (@>, ?)** allows searching inside JSON data.  
✅ **GIN indexes** make querying JSONB efficient.

----  
JSONB  
**What is JSONB in PostgreSQL?**

JSONB (Binary JSON) is a **binary storage format** for JSON data in PostgreSQL. It is **more efficient** for querying, indexing, and processing than regular JSON.

**🔍 Why Use JSONB Instead of JSON?**

✔ **Faster Queries** – JSONB is optimized for indexing.  
✔ **Efficient Storage** – Stores data in a binary format, eliminating whitespace and duplicate keys.  
✔ **Supports Indexing** – Can use **GIN (Generalized Inverted Index)** for efficient lookups.  
✔ **Better Query Performance** – Optimized for search and retrieval.  
✔ **Supports JSON Functions** – PostgreSQL provides a variety of built-in functions for JSONB.

**📌 Creating a Table with JSONB**

To store JSON data in PostgreSQL, define a column as JSONB.

**📝 Example 1: Creating a Table with JSONB**

CREATE TABLE employees (

id SERIAL PRIMARY KEY,

name TEXT,

details JSONB -- Stores structured JSON data

);

✔ The details column stores **JSONB objects**.

**📌 Inserting Data into JSONB**

You can insert data in **valid JSON format**.

**📝 Example 2: Inserting JSONB Data**

INSERT INTO employees (name, details)

VALUES ('John Doe',

'{"age": 30, "position": "Software Engineer", "skills": ["Python", "SQL"]}');

✔ JSONB automatically removes extra whitespace and reorders keys for **efficient storage**.

**📌 Retrieving JSONB Data**

You can retrieve the entire JSON object using a SELECT statement.

**📝 Example 3: Selecting JSONB Data**

SELECT name, details FROM employees;

✔ Returns the JSON structure stored in the column.

**📌 JSONB Operators**

PostgreSQL provides powerful operators to extract and manipulate JSONB data.

**🔹 JSONB Operators**

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| -> | Extracts JSON field as **JSON** | details -> 'position' |
| ->> | Extracts JSON field as **TEXT** | details ->> 'position' |
| #> | Extracts JSON sub-element as **JSON** | details #> '{skills, 0}' |
| #>> | Extracts JSON sub-element as **TEXT** | details #>> '{skills, 0}' |
| @> | Checks if JSONB contains a given key-value pair | details @> '{"position": "Software Engineer"}' |
| ? | Checks if a key exists | details ? 'skills' |
| - | Removes a key | details - 'age' |

**📌 Extracting Data from JSONB**

You can extract specific fields from JSONB objects.

**📝 Example 4: Extracting a JSON Field**

SELECT name, details ->> 'position' AS job\_role FROM employees;

✔ Retrieves the "position" field as text.

**📌 Querying JSONB Arrays**

You can retrieve **array elements** from a JSONB field.

**📝 Example 5: Extracting an Array Value**

SELECT details #>> '{skills, 0}' AS first\_skill FROM employees;

✔ Retrieves the **first skill** (Python).

**📌 Filtering JSONB Data**

You can filter rows based on JSONB fields.

**📝 Example 6: Filtering JSONB Data**

SELECT \* FROM employees WHERE details @> '{"position": "Software Engineer"}';

✔ Returns employees where "position" is "Software Engineer".

**📌 Updating JSONB Data**

You cannot update a single field directly in JSONB. Instead, use jsonb\_set().

**📝 Example 7: Updating a JSONB Field**

UPDATE employees

SET details = jsonb\_set(details, '{position}', '"Senior Software Engineer"')

WHERE name = 'John Doe';

✔ Updates the "position" field to "Senior Software Engineer".

**📌 Deleting a JSONB Field**

You can remove a specific key from JSONB using the - operator.

**📝 Example 8: Removing a JSONB Key**

UPDATE employees

SET details = details - 'age'

WHERE name = 'John Doe';

✔ Removes the "age" field from JSONB.

**📌 Checking If a Key Exists in JSONB**

You can check if a **specific key** exists in JSONB data.

**📝 Example 9: Checking for Key Existence**

SELECT \* FROM employees WHERE details ? 'skills';

✔ Finds employees who have the "skills" key.

**📌 Indexing JSONB for Faster Queries**

To improve query performance, create a **GIN index** on JSONB columns.

**📝 Example 10: Creating a GIN Index**

CREATE INDEX idx\_details ON employees USING GIN (details);

✔ This speeds up queries that filter JSONB data.

**📌 Searching Inside JSONB**

You can perform advanced searches inside JSONB.

**📝 Example 11: Searching for JSON Values**

SELECT \* FROM employees WHERE details @> '{"skills": ["Python", "SQL"]}';

✔ Finds employees **who have both "Python" and "SQL"** as skills.

**📌 Aggregating JSONB Data**

You can **group and aggregate** JSONB data.

**📝 Example 12: Aggregating JSONB Data**

SELECT jsonb\_agg(details) FROM employees;

✔ Combines multiple JSONB rows into **one JSON array**.

**📌 JSONB Functions in PostgreSQL**

| **Function** | **Description** | **Example** |
| --- | --- | --- |
| jsonb\_pretty() | Formats JSON for readability | SELECT jsonb\_pretty(details) FROM employees; |
| jsonb\_array\_elements() | Expands JSON array elements into rows | SELECT jsonb\_array\_elements(details -> 'skills') FROM employees; |
| jsonb\_object\_keys() | Lists all keys in a JSONB object | SELECT jsonb\_object\_keys(details) FROM employees; |

**📌 Summary**

✅ **JSONB** stores structured data in an optimized **binary format**.  
✅ **Faster than JSON** due to indexing and storage efficiency.  
✅ **Supports GIN indexing** for improved query performance.  
✅ **Operators (->, ->>, @>, ?)** help extract and filter data.  
✅ **Functions (jsonb\_set(), jsonb\_pretty())** allow easy modification and formatting.  
  
-------  
arrays:  
**🔹 What is an Array in PostgreSQL?**

In PostgreSQL, an **array** is a **collection of values** stored within a single column. Unlike traditional relational databases that normalize repeated values into separate tables, PostgreSQL allows **arrays** to store multiple values in a single row, making it easier to handle structured data.

**🔍 Why Use Arrays?**

✔ **Stores multiple values in one column** – Reduces table complexity.  
✔ **Efficient querying** – Allows advanced operations like searching, filtering, and aggregation.  
✔ **Supports various data types** – Arrays can be of **INTEGER, TEXT, JSONB, UUID**, etc.  
✔ **Built-in functions** – PostgreSQL provides many array functions for easy manipulation.

**📌 Declaring an Array in PostgreSQL**

To define an **array column**, use square brackets [ ] after the data type.

**📝 Example 1: Creating a Table with an Array Column**

CREATE TABLE employees (

id SERIAL PRIMARY KEY,

name TEXT,

skills TEXT[] -- Array of text values

);

✔ The skills column stores **multiple text values** in an array.

**📌 Inserting Data into an Array Column**

You can insert **array values** using ARRAY[] syntax.

**📝 Example 2: Inserting Data into an Array Column**

INSERT INTO employees (name, skills)

VALUES ('John Doe', ARRAY['Python', 'SQL', 'JavaScript']);

✔ PostgreSQL stores the skills as an array.

Alternatively, **curly braces {}** can be used:

INSERT INTO employees (name, skills)

VALUES ('Alice', '{Java, HTML, CSS}');

✔ {} notation works the same way as ARRAY[].

**📌 Retrieving Data from an Array**

To fetch the entire array:

SELECT name, skills FROM employees;

✔ Returns all array values stored for each employee.

**📌 Extracting Specific Elements from an Array**

PostgreSQL **arrays use 1-based indexing**, meaning the first element is at index **1**, not **0**.

**📝 Example 3: Extracting an Array Element**

SELECT name, skills[1] AS first\_skill FROM employees;

✔ Retrieves the **first skill** from the array.

**📌 Searching Within an Array**

PostgreSQL provides operators to search for values within an array.

**🔹 Array Operators**

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| @> | Contains | skills @> ARRAY['Python'] |
| <@ | Is Contained By | ARRAY['Python'] <@ skills |
| && | Overlaps | skills && ARRAY['Python', 'Ruby'] |
| = ANY(array) | Value Exists | 'Python' = ANY(skills) |

**📌 Checking If an Array Contains a Specific Value**

To check if an array **contains a specific value**, use the @> operator.

**📝 Example 4: Searching for Employees Who Know Python**

SELECT \* FROM employees WHERE skills @> ARRAY['Python'];

✔ Returns employees whose skills **include** "Python".

**📌 Filtering Based on Multiple Values**

To check if an array contains **at least one** matching value:

**📝 Example 5: Checking for Overlapping Skills**

SELECT \* FROM employees WHERE skills && ARRAY['Python', 'Ruby'];

✔ Finds employees who have **either** "Python" or "Ruby".

**📌 Updating an Array Column**

To update a **whole array**:

UPDATE employees

SET skills = ARRAY['C++', 'Java']

WHERE name = 'John Doe';

✔ Replaces the **entire array** with ['C++', 'Java'].

To add a new element, use array\_append():

UPDATE employees

SET skills = array\_append(skills, 'Go')

WHERE name = 'John Doe';

✔ Adds "Go" to the **end** of the array.

To remove an element, use array\_remove():

UPDATE employees

SET skills = array\_remove(skills, 'JavaScript')

WHERE name = 'John Doe';

✔ Removes "JavaScript" from the skills array.

**📌 Retrieving the Length of an Array**

You can count the number of elements in an array using array\_length().

**📝 Example 6: Getting the Number of Skills**

SELECT name, array\_length(skills, 1) AS skill\_count FROM employees;

✔ Returns the number of **skills** each employee has.

**📌 Unnesting Arrays (Expanding into Rows)**

To convert array elements into separate rows, use unnest().

**📝 Example 7: Expanding an Array into Rows**

SELECT name, unnest(skills) AS skill FROM employees;

✔ Converts each skill in the array into **a separate row**.

**📌 Aggregating Array Data**

PostgreSQL provides aggregation functions for arrays.

To combine multiple arrays into one:

SELECT array\_agg(skills) FROM employees;

✔ Merges all skills arrays into **one array**.

To find unique values across all employees:

SELECT array\_agg(DISTINCT unnest(skills)) FROM employees;

✔ Returns a **unique list** of all skills.

**📌 Sorting Arrays**

Use array\_sort() (PostgreSQL 16+ required) to sort array elements.

sql

CopyEdit

SELECT array\_sort(ARRAY[3, 1, 2]);

✔ Returns {1,2,3}.

For older versions, use unnest(), ORDER BY, and array\_agg():

SELECT array\_agg(skill ORDER BY skill)

FROM (SELECT unnest(skills) AS skill FROM employees) AS subquery;

✔ Returns a sorted list of all skills.

**📌 Using Arrays in Joins**

You can join **arrays with regular tables** using ANY().

**📝 Example 8: Joining Arrays with Another Table**

SELECT e.name, p.project\_name

FROM employees e

JOIN projects p ON p.required\_skills && e.skills;

✔ Finds **employees** who have **any** of the required skills for a project.

**📌 Summary**

✅ **PostgreSQL arrays** store multiple values in a single column.  
✅ **Operators (@>, <@, &&, ANY())** allow easy filtering.  
✅ **Functions (array\_append(), array\_remove(), unnest())** help manipulate arrays.  
✅ **Arrays can be indexed** for better performance.  
  
--------------------